PROGRAMMING BY DESIGN

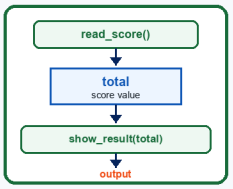
Think Before Coding

Structured Problem Solving with C++ for CS1

THINK DESIGN IMPLEMENT TEST

A design-centered C++17 path: problem analysis, readable code, testing, debugging, documentation, and data-structure thinking.

```
int main()
{
    int total = 0;
    total = read_score();
    show_result(total);
    return 0;
}
```



C++17

Stable. Reliable. Teachable.

KEVIN MESS

Think Before Coding

Structured Problem Solving with C++ for CS1

Kevin Mess

First Edition

Programming by Design series

Published by Kevin Mess

Henderson, Nevada

Think Before Coding

Structured Problem Solving with C++ for CS1

Kevin Mess

Independently published by Kevin Mess
Henderson, Nevada
Programming by Design series
First Edition
First published 2026
ISBN 979-8-9969314-2-2 (PDF edition)

Production

This book was independently produced by Kevin Mess from a Markdown source manuscript. PDF editions were converted with Pandoc and typeset with LuaLaTeX using a custom LaTeX book design. EPUB and HTML editions were produced with Pandoc and a shared stylesheet.

AI-assisted tools supported copyediting, consistency review, and selected production tasks. Except where otherwise credited, the instructional content, code examples, academic class and project designs, pedagogical decisions, and technical judgments are the author's own. The author directed the work, made the final editorial and technical decisions, and takes responsibility for the complete publication.

Disclaimer

This publication and its code examples are provided for educational purposes. Although reasonable care has been taken to ensure accuracy, the author and publisher make no warranties, express or implied, concerning the accuracy, completeness, fitness for a particular purpose, or behavior of the programs in any particular environment. Readers are responsible for testing and validating programs before relying on them. To the fullest extent permitted by law, the author and publisher disclaim liability for loss or damage arising from the use of this publication.

Product and company names mentioned in this publication may be trademarks of their respective owners. Their use is for identification and instruction and does not imply endorsement.

This is an independent publication. References to the author's institutional affiliation are for identification only and do not imply institutional sponsorship or endorsement.

Accessibility was considered throughout the production of the digital editions. Accessibility barriers may be reported to the author.

Acknowledgments

The author gratefully acknowledges the people who assisted with the development, review, testing, classroom use, and production of this book.

Contributors

- D'Andrew Lee Harrington
- Brandon Lingo
- Alvin Singo

Brief Contents

Detailed Contents

INTENTIONALLY BLANK

Preface

Programming is not typing code.

A computer can follow a flawed plan exactly as written. Programming begins earlier: understand the problem, identify the available information and required result, design the steps, express those steps precisely, and test whether the result is trustworthy. *Think Before Coding*, the CS1 book in the *Programming by Design* series, teaches C++ through that process.

At first, C++ may look like a collection of rules about semicolons, braces, headers, variables, decisions, loops, functions, arrays, and files. Those rules matter. Code that violates the language rules will not compile, but code that compiles may still be wrong. The center of this book is therefore not syntax alone. It is the habit of thinking clearly before trusting the code.

Language Standard

This first edition uses C++17 because it is widely supported by common compilers, course systems, online tools, and student computers. C++20 and later standards add useful features, but the CS1 ideas in this book do not require them.

If your environment uses a later standard by default, these programs should still compile. They use C++ features available in C++17 and later. Use the standard selected for your course or project.

Who *Think Before Coding* Is For

Think Before Coding is written for students taking CS1 in C++. In many colleges and universities, **CS1** means the first programming course, often called Computer Science I. Chapter 1 assumes that you may be an absolute beginner. You do not need prior experience with C++, Linux, command-line tools, or computer science.

If you have taken an introductory programming course, a Linux course, or a high-school computing course, some early material may be familiar. Use that familiarity carefully. Prior exposure can help, but it can also hide weak habits. The goal is not merely to recognize code. The goal is to explain it, trace it, test it, and modify it correctly.

Some students take CS1 because it supports another STEM program. Some complete an AA, AS, or similar program and enter technical work. Others continue into *Own the Design*,

upper-division computer science courses, and later data structures and algorithms work. These paths share a foundation: careful reading, precise terminology, sound reasoning about data, disciplined testing, and the ability to explain programming decisions.

If CS1 is your only programming course, this book provides a complete foundation in structured programming. If you continue into CS2, it also prepares the habits that *Own the Design* requires: following how data moves, separating what a program offers from how it works, testing behavior with evidence, and explaining why one way of storing data fits a problem.

The *Think Before Coding* Arc

The twelve chapters move through four stages. Each stage changes what your programs can do and what you must be able to explain.

Stage	Chapters	Development
First programs and control	1-4	Move from fixed output to variables, input, decisions, loops, and repeated processing.
Functions, text, and files	5-8	Divide work into functions, follow values into and out of functions, trace basic recursion, process text, and read or write files.
Collections, algorithms, and records	9-11	Store related values, search and summarize collections, process grids, and model table rows as records.
Complete project	12	Analyze, design, implement, test, document, and explain one structured program.

The arc begins with one statement at a time and ends with a program large enough to require planning. Syntax accumulates along the way, but the deeper progression is from following code to designing it.

How the Two Books Fit Together

The series contains two student books.

Book	Course role	Arc
Think Before Coding	CS1 / Part I	From first programs to a complete structured project.
Own the Design	CS2 / Part II	From classes that protect data to complete, tested data structures.

Taken together, the books form one progression. *Think Before Coding* asks what a program should do and how data should move through functions and collections. *Own the Design* asks how a class can keep its data valid, how a program can manage memory safely, and how design choices affect the work a program performs. Records grow into

classes, related functions become a public interface that other code can use, and familiar collections become data structures that you design and test.

Each book can be used in its own course. The continuing arc makes the transition deliberate without requiring every CS2 student to have used this particular CS1 book.

How to Use the Chapters

Begin each chapter with its purpose, prerequisites, outcomes, and pacing guidance. These tell you what you will learn, what the chapter assumes, and where to pause. New terms appear first in the explanation and are collected in the chapter glossary.

Programming vocabulary is part of the technical content. Terms that sound similar may describe different operations, states, or responsibilities.

Hint

Connect each new term to the current code or diagram, explain it in your own words, and then check your explanation against the glossary.

Typographic Conventions

The typeface helps identify what you are reading. The surrounding explanation remains the final guide to meaning.

Appearance	Meaning
Italic type	book titles and occasional emphasis
Bold type	a new term at its first definition or a short label that needs attention
`Monospaced type`	C++ identifiers, keywords, literals, headers, filenames, commands, and short code
A monospaced display	a program listing, code fragment, command session, program input or output, or pseudocode

These visual cues make technical material easier to scan, but they do not replace the definitions and explanations in the text.

Appendices A-G are searchable, task-oriented support rather than extra chapters to read in sequence. Use the reference that answers the question in front of you:

When you need to...	Use...
configure tools, build programs, or use Git	Appendix A
check code style, documentation, formatting, or static analysis	Appendix B
interpret diagnostics, debug a program, or review error handling	Appendix C
write focused Catch2 tests	Appendix D

When you need to...	Use...
look up C++ syntax, types, operators, or standard-library headers	Appendix E
review memory organization or use Valgrind	Appendix F
compare standard containers, iterators, or algorithms	Appendix G

Study the Guided Example or Guided Project as a process, not merely as a listing. Follow the movement from problem statement to design, implementation, trace, test, and review. If an assignment feels too large, return to that order of work.

Practice sections isolate individual ideas. Programming Exercises ask you to combine them in complete programs. Unmarked exercises are standard chapter work; `**` marks more demanding work, and `***` marks larger, more integrative, or more open-ended work. Later exercises may improve an earlier program, because revising a design is part of learning to program.

Project Connection sections show where the chapter leads. Tool Notes introduce only the commands needed for the current work.

Read, Trace, and Test

Do not read code as a wall of syntax, and do not repair it by guessing. Use the same evidence-based sequence when studying a listing or debugging your own program:

1. State the program's purpose and expected result.
2. Identify its input, process, and output.
3. Read the declarations and their purpose comments.
4. Trace one small path by hand.
5. Save, compile, and read the first diagnostic.
6. Confirm that you are running the current executable.
7. Compare actual behavior with expected behavior, using one focused debugger command or temporary diagnostic when needed.
8. Ask for help with the problem statement, command, input, output, and evidence you have already collected.

Syntax is easier to understand when the purpose is clear. A small, specific question is easier to answer than a large, confusing program.

Style, Tools, and Responsibility

Consistent style helps a reader understand a program and helps you find mistakes when you return to code later. Meaningful names, initialized variables, documented purposes, clear indentation, and consistent braces reduce distractions between the reader and the idea.

Tools support that discipline without replacing it. A compiler reports violated language rules. A debugger lets you inspect a program while it runs. Tests compare expected and actual behavior. Static analysis tools inspect source code for likely defects, and formatting tools help keep its style consistent. None of these tools can decide what the program should mean. Use them to check your reasoning, not to avoid it.

If you are using this book in a course, use AI tools only when your instructor permits them and only within the rules for the assignment. An AI-generated answer is not evidence that a program is correct. If you use AI assistance, you are still responsible for understanding the code, testing it, explaining it, and disclosing the assistance when required.

The instructional content, examples, programming constraints, chapter sequence, and technical judgments in this book are my own. AI tools assisted with copyediting for wording, clarity, and consistency and with generating the book cover. Any errors, omissions, pedagogical choices, and technical judgments are mine.

Central Theme

The central theme remains constant:

> Think before coding. Trace before trusting. Test before submitting. Document the contract. Own the result.

Chapter 1 starts small on purpose.

Students can begin Chapter 1 after this point. The remaining preface material is primarily for instructors, program coordinators, and anyone planning a CS1 course.

For Instructors

Think Before Coding is designed for CS1. It can stand alone or serve as Part I of a two-course sequence with *Own the Design*. Its four-stage arc provides natural instructional units: control flow, modular and persistent-data processing, collection and record reasoning, and a complete project. Students who continue should leave able to trace function calls, design with arrays and vectors, group related data with records, process files, explain basic recursion, and test a multi-function program with evidence.

The following timelines show how the chapter sequence can fit common term lengths. Instructors can adjust coverage, assignments, and project timing to fit local needs.

Term Length	Example Sequence
10 weeks	Chapters 1-2, 3-4, 5, 6, 7-8, 9, 10, 11 and project planning, 12 project, final review
12 weeks	Chapters 1-2, 3, 4, 5, 6, 7-8, 9, 10, 11, 12 planning, 12 implementation, final review

Term Length	Example Sequence
16 weeks	Chapters 1-9 at about one chapter per week, Chapters 10-11 over three weeks, Chapter 12 over three weeks, final review

Basic recursion is deliberately staged. Chapter 6 introduces calls, base and recursive cases, call-stack behavior, and a factorial trace while students are studying functions. Chapter 10 applies the same model to collections and strings after its algorithm and table work. Functions, arrays, records, and the integration project often require additional time because students are moving from recognizing syntax to planning program structure.

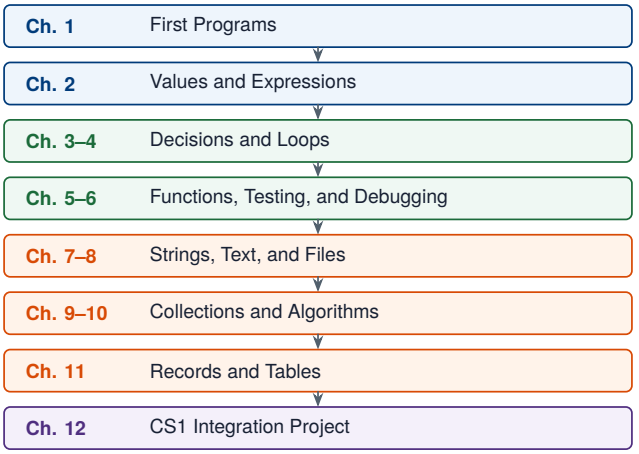
Faculty resources include chapter test banks and lecture slides aligned with the chapter outcomes and sequence. Instructors can download these materials from the Programming by Design instructor support page. The resources support local selection and pacing while remaining separate from the student manuscript.

Part I references Appendices A-H, which must remain available with the book. Appendix H provides the compact curriculum, project, and assessment map for faculty, curriculum reviewers, and adoption committees. In this book, the Part I / CS1 rows are direct evidence; the Part II / CS2 rows describe the continuation in *Own the Design*. This boundary prevents the CS1 book from claiming delivery of the full two-course sequence by itself.

Part I

CS1 Foundations

This part takes a complete beginner from first programs to a complete structured C++ project.

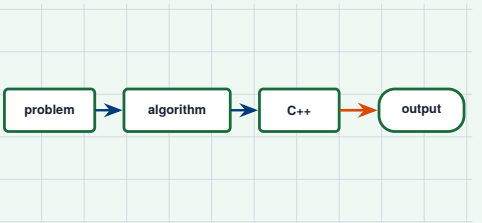


Roadmap: Part I builds from first programs through collections, records, and a CS1 integration project.

INTENTIONALLY BLANK

1 Computing, Algorithms, and First C++ Programs

Think • Design • Implement • Test • Document



A program is not magic.

It may feel that way the first time you type a few lines of C++, run a command, and see text appear on the screen. The computer seems to have understood you. It did not. It followed a precise set of instructions.

This chapter begins with that idea: programming is the work of making instructions exact.

By the end of this chapter, you should be able to read a tiny C++ program, compile it, run it, and trace what each statement contributes. You should also be able to explain the difference between source code, the compiler, the executable program, and the program's output.

Purpose

The purpose of this chapter is to help you read, compile, run, and trace tiny C++ programs.

This chapter assumes no prior programming experience. If you have already taken an introductory programming or Linux course, some tools may look familiar. Use that familiarity to build careful habits: read first, predict first, trace first, then run.

Prerequisites

None.

Chapter Outcomes

After this chapter, you can take a small C++ source file, compile it, run it, observe its output, and explain what each line contributes.

That capability is modest, but it is the foundation for the rest of the book. Chapter 2 will add stored values, input, arithmetic, and formatted output. Those topics only make sense if you already understand what it means to compile and run a program.

To do that, you should be able to:

- explain what computer science studies
- describe the relationship among problems, algorithms, source code, compilation, executable programs, and output
- compile and run a tiny C++17 program in a 64-bit Linux environment
- distinguish compile-time errors from runtime behavior
- trace each statement in a simple output-only program
- identify the roles of the CPU, main memory, storage, input devices, output devices, peripherals, and operating system
- use compiler diagnostics as information instead of changing code randomly
- explain why one successful output does not prove that a program is correct

How to Pace This Chapter

This chapter introduces the programming workflow, the first C++ program shape, and the basic computer model. Do not try to memorize every term in one pass.

A good first stopping point is the end of **Your First C++ Program**. At that point, make sure you can identify `main`, braces, statements, output, and `return 0`.

A good second stopping point is the end of **Source Code, Compilation, and Executable Programs**. At that point, make sure you can explain the difference between source code, a compiler, an executable program, output, and a compiler diagnostic.

Then finish the computer model, trace-before-code activity, guided example, and review sections. The goal is not speed. The goal is a repeatable build-run-trace habit.

1.1. What Computer Science Is Doing Here

Many beginning programming courses look, at first, like courses about typing code. You learn where semicolons go, how to write braces, how to display output, and how to correct compiler errors. A **compiler** is a program that translates a C++ program into a form the computer can execute. Those details matter, but they are not the main subject.

This book is about computer science through C++.

Computer science is the study of problems, algorithms, data, and computation.

A **problem** is a task to be solved.

An **algorithm** is a step-by-step procedure for solving a problem.

Data is the information an algorithm uses, stores, changes, or produces.

Computation is the process of carrying out an algorithm using a computing device.

A C++ program is one way to express an algorithm so that a computer can execute it.

That distinction is important. If you think programming is only about memorizing C++ syntax, then every new program will feel like a guessing game. You will ask, "What code do I type?" before you understand what problem the code is supposed to solve. A stronger habit is to ask these questions first:

1. What information is given?
2. What result is required?
3. What steps transform the given information into the required result?
4. How can those steps be expressed clearly in C++?
5. How can the result be tested?

For example, suppose the problem is:

> Given the price of one item and the number of items purchased, compute the total cost before tax.

Before writing C++, identify the information and the required result.

Part	Meaning
Input	item price and quantity
Process	multiply item price by quantity
Output	total cost before tax

The algorithm can be stated in ordinary language:

```
Get the item price.
Get the quantity.
Multiply the item price by the quantity.
Display the total cost.
```

This algorithm is not yet a C++ program, but it is already a solution plan. Once the plan is clear, the program has a job to do: represent the plan precisely enough that the computer can carry it out.

A short C++ version might look like this:

```
#include <iostream>

int main()
{
    double price    = 12.50;  // item price
    int    quantity = 4;      // items purchased
    double total    = 0.0;    // total cost

    total = price * quantity;

    std::cout << "Total: $" << total << "\n";

    return 0;
}
```

Try it: Run the code

This program previews details that later study will explain more carefully. Chapter 2 teaches stored values, input, and expressions. Chapter 3 and Chapter 4 teach decisions and loops. Chapter 5 teaches functions, and Chapter 9 teaches collections. Later CS2 work returns to memory, addresses, and ownership in more detail.

Do not memorize the variable syntax yet. For now, focus on the relationship between the problem, the algorithm, and the program.

The statement

```
double price = 12.50;  // item price
```

creates a variable named `price` that stores a number with a decimal part. The statement

```
int quantity = 4;  // items purchased
```

creates a variable named `quantity` that stores a whole number. The statement

```
double total = 0.0;  // total cost
```

creates a variable named `total` that stores the computed cost. The assignment statement

```
total = price * quantity;
```

computes the product of `price` and `quantity`, then stores that result in `total`.

The program carries out the algorithm instead of only displaying text.

A useful way to read this program is to trace the changing values.

Statement	Value after the statement executes
`double price = 12.50;`	`price` is `12.50`
`int quantity = 4;`	`quantity` is `4`
`double total = 0.0;`	`total` is `0.0`
`total = price * quantity;`	`total` is `50.0`
`std::cout << "Total: $" << total << "\n";`	output is displayed

The output is:

```
Total: $50
```

The exact formatting of decimal numbers can be controlled, but that is not the main point yet. The main point is that the program follows a precise sequence of operations.

This is one of the first habits of computer science: separate the solution from the notation used to express it.

C++ is the notation used in this book. It is a powerful, widely used programming language that supports procedural programming, object-oriented programming, and low-level control over memory and performance. Those features make C++ valuable, but they also mean that the language must be learned carefully. Small details can matter.

Beginners should not confuse detail with mystery. A compiler error is not a personal failure. A wrong answer is not proof that programming is random. These are signals that the program, as written, does not yet match the intended algorithm.

The work of programming is the work of making ideas exact.

1.2. From Problem to Program

Most programs in this book will follow a basic development pattern:

1. Understand the problem.
2. Identify the inputs.
3. Identify the outputs.
4. Develop an algorithm.
5. Translate the algorithm into C++.
6. Compile and run the program.
7. Test the result.
8. Revise when needed.

The process can be pictured this way:

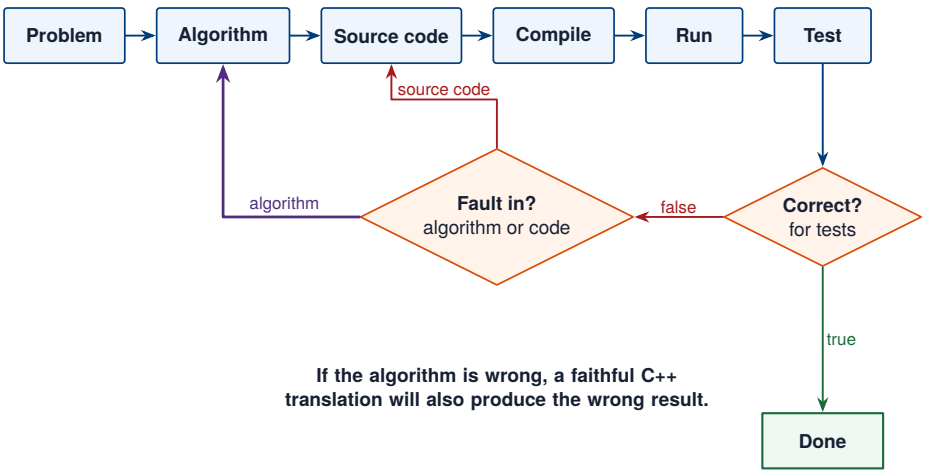


Figure 1.1: A programming task moves from problem statement to algorithm, source code, compilation, execution, testing, and revision.

For small programs, this process may take only a few minutes. For larger programs, the same pattern still applies, but each step requires more care.

Consider this problem:

> Given the number of hours worked and the hourly pay rate, compute gross pay.

The **input** is the data needed by the program. Here, the inputs are hours worked and hourly pay rate.

The **output** is the result produced by the program. Here, the output is gross pay.

The **process** is the computation that connects the input to the output. Here, the process is multiplication:

```
gross pay = hours worked * hourly pay rate
```

A first algorithm might be:

```
Get hours worked.
Get hourly pay rate.
Multiply hours worked by hourly pay rate.
Display gross pay.
```

This may seem almost too simple to write down. Write it down anyway. Clear algorithms help on difficult problems and ordinary ones. They are how you train yourself to think before coding.

Later in the book, the problems will involve decisions, loops, functions, arrays, classes, files, recursion, and dynamic memory. The syntax will become more substantial, but the underlying questions will remain familiar:

1. What is the program supposed to do?
2. What information does it need?
3. What steps must happen?
4. What can go wrong?
5. How will I know the program is correct?

Quick Check: Problem Analysis

Suppose a program must convert inches to feet. There are 12 inches in 1 foot.

Identify the input, process, and output.

A reasonable answer is:

Part	Answer
Input	number of inches
Process	divide inches by 12
Output	equivalent number of feet

The algorithm could be:

```
Get the number of inches.
Divide the number of inches by 12.
Display the number of feet.
```

Before writing code, test the idea with a simple value.

If the input is `36`, then the output should be `3`.

That quick mental test is part of programming. You are not waiting for the compiler to think for you. You are using the compiler to check a solution that you already understand.

1.3. Your First C++ Program

The first C++ program in this book displays one line of output.

```
#include <iostream>

int main()
{
    std::cout << "C++ is running.\n";
    return 0;
}
```

Try it: Run the code

The output is:

```
C++ is running.
```

Read the program slowly. Do not worry if every symbol is not clear yet. The goal is to connect each part of the program to its role.

Program part	What it contributes
`#include <iostream>`	Gives the program access to standard input and output tools, including `std::cout`.
`int main()`	Begins the function named `main`, where program execution starts.
`{` and `}`	Mark the beginning and end of the function body.
`std::cout << "C++ is running.\n";`	Sends text to standard output.
`return 0;`	Ends the program and reports successful completion to the operating system.

A **function** is a named block of code that performs a task. Every complete C++ program must have a function named `main`. When the program runs, execution begins in `main`.

A **statement** is an instruction in a program. In C++, many statements end with a semicolon. These two lines are statements:

```
std::cout << "C++ is running.\n";
return 0;
```

The text `"C++ is running.\n"` is a **string literal**. A string literal is text written directly in a program and enclosed in double quotation marks. The characters `\n` form an escape sequence that means newline. It moves the output cursor to the next line.

The expression `std::cout` names the standard output stream. The operator `<<` is the stream insertion operator. In this chapter, you can read it as "send this to the output stream."

So this line means:

```
Send the string "C++ is running.\n" to standard output.
```

The line

```
return 0;
```

ends `main` and returns the integer value `0` to the operating system. Returning `0` from `main` conventionally means the program ended successfully.

Comments and Whitespace

C++ programs often include **comments**. A comment is text written for human readers and ignored by the compiler.

This program contains a single-line comment:

```
#include <iostream>

int main()
{
    // display a message
    std::cout << "C++ is running.\n";
    return 0;
}
```

The comment begins with `//` and continues to the end of the line.

Comments should explain intent, clarify a decision, or mark a section of code. They should not repeat what the next line already says.

This comment is weak:

```
// print hello
std::cout << "Hello\n";
```

This comment is more useful:

```
// show setup success
std::cout << "Setup complete.\n";
```

Whitespace means spaces, tabs, and blank lines. In many places, C++ ignores extra whitespace, but readers do not. Use whitespace to make programs readable.

These two statements have the same meaning to the compiler:

```
std::cout<<"Hello\n";
```

```
std::cout << "Hello\n";
```

The first version is easier to read. Prefer readable code.

1.4. Source Code, Compilation, and Executable Programs

A C++ program begins as **source code**. Source code is the text a programmer writes in a programming language. The short program from earlier is source code:

```
#include <iostream>

int main()
{
    std::cout << "C++ is running.\n";
    return 0;
}
```

A computer cannot directly execute C++ source code as written. Before the program can run, the source code must be translated into a form the operating system can load and execute.

The earlier description can now be stated more precisely. A compiler translates source code into machine instructions. Machine instructions are low-level instructions that the computer's processor can carry out. The result of successful compilation is an **executable program**, often called an executable.

The basic workflow is:

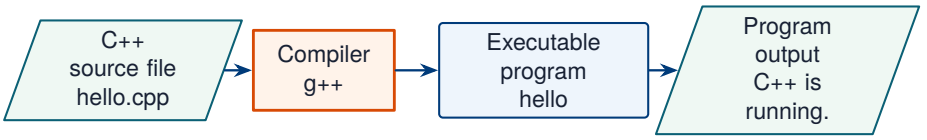


Figure 1.2: Compilation translates a C++ source file into an executable program that can produce output.

The exact command depends on the compiler and operating system. In this book, the assumed working environment is a 64-bit Linux system with a C++17 compiler. A common compiler is `g++`.

If the source file is named `hello.cpp`, this command compiles it:

```
g++ -std=c++17 hello.cpp -o hello
```

This command has several parts.

Command part	Meaning
`g++`	Run the GNU C++ compiler.
`-std=c++17`	Compile using the C++17 language standard.
`hello.cpp`	Use `hello.cpp` as the source file.
`-o hello`	Name the executable program `hello`.

If compilation succeeds, the compiler creates an executable program named `hello`.

On Linux, the program can be run from the command line like this:

```
./hello
```

The `./` tells the shell to run the executable named `hello` in the current directory. A **shell** is a command-line program that accepts commands from the user and asks the operating system to carry them out.

The output should be:

```
C++ is running.
```

At this point, there are two different files involved:

File	Purpose
`hello.cpp`	Source code written by the programmer.
`hello`	Executable program produced by the compiler.

Warning

Editing source code does not update the executable. Recompile before testing the change.

Compile-Time Errors

Sometimes the compiler cannot translate the source code. This is usually because the source code violates a rule of the C++ language.

For example, this program is missing a semicolon:

```
#include <iostream>

int main()
{
    std::cout << "C++ is running.\n"
    return 0;
}
```

The statement that displays output should end with a semicolon:

```
std::cout << "C++ is running.\n";
```

Without that semicolon, the compiler will report a **compile-time error**. A compile-time error is an error found while the compiler is trying to translate the source code.

A compiler diagnostic may look long and unfriendly at first. Do not try to understand every word immediately. Start with three questions:

1. What file is mentioned?
2. What line number is mentioned?
3. What rule does the compiler seem to be complaining about?

Compiler diagnostics are not always written in beginner-friendly language, but they are useful. Read the first diagnostic first. Later diagnostics may be side effects of the first problem.

Note

A diagnostic reports where the compiler became confused, not always where the mistake began. Check that line and the line before it.

Fixing the first real error may make several later messages disappear.

Runtime Behavior

A program that compiles successfully can still behave incorrectly.

Runtime behavior is something that happens while the executable program is running. Displaying output is runtime behavior. Waiting for input is runtime behavior. Computing a value is runtime behavior. Crashing is runtime behavior.

For example, this program compiles and runs:

```
#include <iostream>

int main()
{
    std::cout << "The answer is 5.\n";
    return 0;
}
```

Try it: Run the code

If the program was supposed to display `The answer is 6.`, then the program is logically wrong even though it compiled successfully.

This gives us an important distinction.

Situation	Meaning
The program does not compile.	The compiler found a translation problem.
The program compiles but gives the wrong result.	The program has incorrect runtime behavior or incorrect logic.
The program compiles and gives the expected result for one test.	The program passed one test, but more testing may still be needed.

Compilation checks whether the program follows the language rules well enough to be translated. Compilation does not prove that the program solves the intended problem.

Source Code, Executable Code, and the Operating System

The operating system manages the computer's resources and provides services that programs need. It loads executable programs, manages files, coordinates input and output devices, and controls access to memory and hardware.

When you type:

```
./hello
```

The shell asks the operating system to run the executable file named `hello`. The operating system loads the executable into memory, starts the program, and connects the program to standard input and standard output.

Standard output is the normal destination for a program's text output. In a terminal window, standard output usually appears on the screen. When a program uses `std::cout`, it sends characters to standard output.

The name `cout` is pronounced "see-out" by many programmers. It stands for character output. The prefix `std::` means that `cout` belongs to the C++ standard namespace. A **namespace** is a named region used to organize program names and avoid name conflicts.

In older examples or other textbooks, you may see this line:

```
using namespace std;
```

When that line appears, the program can write `cout` instead of `std::cout`. This book uses the clearer form:

```
std::cout
```

The prefix keeps the namespace visible. You do not need to master namespaces in this chapter. For now, read `std::cout` as "the standard output stream named `cout`."

Quick Check: Compile and Run

Suppose you write this source file and save it as `message.cpp`:

```
#include <iostream>

int main()
{
    std::cout << "First line\n";
    std::cout << "Second line\n";
    return 0;
}
```

Try it: Run the code

You compile it with:

```
g++ -std=c++17 message.cpp -o message
```

Then you run:

```
./message
```

What output should appear?

The answer is:

```
First line
Second line
```

Now suppose you edit `message.cpp` so that the first output statement says:

```
std::cout << "Changed line\n";
```

If you run `./message` again without recompiling, the old executable may still run. To see the new behavior, compile again:

```
g++ -std=c++17 message.cpp -o message
```

Then run:

```
./message
```

The edit changes the source code. Recompilation changes the executable.

1.5. A Simple Model of the Computer

A program runs on a physical machine. You do not need to know all the details of computer hardware to begin programming, but you do need a simple working model.

A computer system includes hardware and software.

Hardware is the physical equipment: the processor, memory chips, storage devices, keyboard, monitor, network hardware, and other devices.

Software is the collection of programs that run on the hardware. The operating system, compiler, shell, editor, browser, and your own C++ programs are all software.

For this book, the most important parts of the computer are the CPU, main memory, storage, input devices, output devices, and operating system.

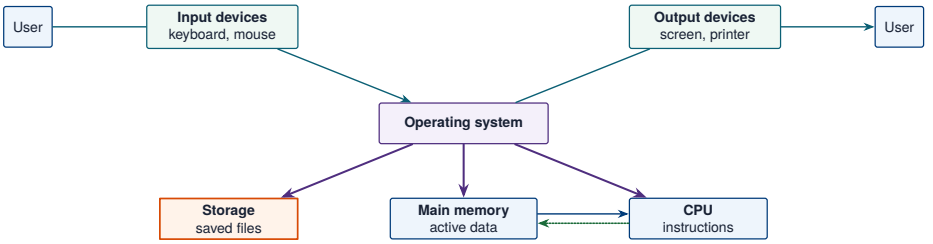


Figure 1.3: The operating system coordinates input, output, storage, memory, and the CPU while a program runs.

The **CPU**, or central processing unit, executes program instructions. When a program runs, the CPU carries out instructions such as moving data, performing arithmetic, comparing values, and jumping to another part of the program.

Main memory, also called RAM, temporarily stores programs and data while they are being used. When your C++ program runs, its instructions and data are placed in main memory. Variables are stored in memory while the program is running.

Storage keeps information even when the computer is turned off. Source files, executable files, documents, images, and installed programs are stored on devices such as solid-state drives or hard drives.

An **input device** sends data into the computer. A keyboard is an input device. So are a mouse, touchscreen, scanner, microphone, and network connection.

An **output device** receives data from the computer. A monitor is an output device. So are speakers, printers, and network connections used to send data elsewhere.

A **peripheral** is a device connected to the computer system but not part of the central processor or main memory. Keyboards, monitors, printers, external drives, and many network devices are peripherals.

The **operating system** is system software that manages the hardware and provides services for programs. Linux, Windows, and macOS are operating systems. In this book, the assumed working environment is 64-bit Linux.

When you run a C++ executable, the operating system loads the program into memory, gives it time on the CPU, connects it to input and output, and manages its access to files and devices.

You do not usually see all of that work. From the command line, you type:

```
./hello
```

The program runs and displays output. Behind that simple command, the operating system coordinates the hardware resources needed to make the program execute.

What Happens When a Program Runs

The process can be described in a simplified sequence:

```
The user runs the executable program.
The operating system loads the program into main memory.
The CPU begins executing the program's instructions.
The program sends output to standard output.
The operating system displays the output in the terminal.
The program ends and returns control to the operating system.
```

This is not a complete description of every technical detail. It is a useful starting model.

The key idea is that source code, executable programs, memory, and output are different things.

Item	What it is
Source code	Text written by the programmer.
Compiler	Program that translates source code.
Executable program	Translated program that the operating system can run.
Main memory	Temporary workspace used while programs run.
CPU	Hardware that executes instructions.
Storage	Long-term location for files and programs.
Standard output	Usual destination for text output.

As you write larger programs, this model will become more important. Variables are stored in memory. Files are stored on storage devices. Input comes from outside the program. Output is sent somewhere. Functions use memory while they run. Arrays occupy sequences of memory locations. Objects have state stored in memory. Dynamic memory is requested while the program runs and must be managed carefully.

Those later topics will be easier if you begin with a clear mental picture of what is happening now.

A First Look at Memory and Variables

You will study variables in detail in Chapter 2. For now, a **variable** is a named memory location used to store a value while a program runs.

Consider these statements:

```
int count = 3;  // current count
int next  = 0;  // next value

next = count + 1;
```

A simplified memory picture is:

```
+--------+-------+
| Name   | Value |
+--------+-------+
| count  |   3   |
+--------+-------+
| next   |   4   |
+--------+-------+
```

The program uses the name `count`, but the actual value is stored in memory. The program uses the name `next`, but that value is also stored in memory.

The table is not showing actual hardware addresses. It is a beginner's model. Later, when you study arrays, pointers, references, classes, and dynamic memory, the memory model will become more detailed.

Bits, Bytes, and Hexadecimal Notation

Computers store information using bits. A **bit** is a value that can be either 0 or 1. A **byte** is a group of 8 bits.

Beginning C++ programs usually work with values such as integers, floating-point numbers, characters, and strings. Underneath those values, the machine stores patterns of bits. You do not need to write those bit patterns by hand in this chapter, but you should know why programmers talk about binary and hexadecimal notation.

The number system you use most often is **decimal notation**, or base 10. Base 10 uses ten digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9`. The value of a digit depends on its position.

```
number:       427
hundreds:     4 hundreds
tens:         2 tens
ones:         7 ones
value:        400 + 20 + 7
```

The word **base** tells you how many digit values the notation uses before the next position is needed. In base 10, the digits run from 0 through 9, and the next value is written as `10`. Other bases follow the same positional idea but use a different number of digit values.

Binary notation is base 2. It uses only the digits 0 and 1. For example:

```
binary  decimal
0       0
1       1
10      2
11      3
100     4
```

Binary matches the hardware idea of bits, but long binary values are hard for people to read. **Hexadecimal notation** is base 16. It uses the digits `0` through `9` and the letters `A` through `F`.

```
hexadecimal  decimal
0            0
1            1
9            9
A            10
B            11
F            15
10           16
```

A group of 4 bits is sometimes called a **nibble**. This gives programmers an old pun and a useful memory aid: two nibbles make one byte. One nibble also corresponds exactly to one hexadecimal digit. Four bits can form 16 different patterns, from `0000` through `1111`, so those 16 patterns match the 16 hexadecimal digits. Two hexadecimal digits therefore represent 8 bits, or one byte.

```
binary      hexadecimal
0000        0
0001        1
1010        A
1111        F
11111111    FF
```

This is why memory addresses are often displayed in hexadecimal. A long address is still a number, but hexadecimal makes the value shorter and easier to group.

For now, treat binary and hexadecimal as notation systems. They are different ways to write numbers. Chapter 7 connects numeric codes to characters, and Chapter 9 and Chapter 10 connect indexes to arrays and vectors. Later CS2 work connects addresses to memory and resource ownership.

For now, remember this:

1. Source code is stored in a file.
2. The compiler translates source code into an executable.
3. The operating system runs the executable.
4. The executable's instructions and data use main memory while the program runs.
5. The CPU executes the program's instructions.
6. Output appears through an output device, often the terminal.

Quick Check: Computer Organization

Match each description to the correct term.

Description	Term
Executes program instructions	CPU
Temporarily stores running programs and data	main memory
Keeps files when the computer is off	storage
Translates C++ source code	compiler
Manages hardware and runs programs	operating system
Normal destination for terminal text output	standard output

If these terms feel new, do not try to memorize them as isolated vocabulary. Connect each term to the program workflow:

```
source code -> compiler -> executable
executable -> operating system -> memory and CPU -> output
```

That chain is the first practical map of programming.

1.6. Trace-Before-Code Activity

Before writing new code, practice reading existing code.

To **trace** a program is to follow it step by step and record what happens. Tracing slows you down on purpose. It trains you to see what the program actually says instead of what you assume it says.

Consider this program:

```
#include <iostream>

int main()
{
    std::cout << "Name: Ada\n";
    std::cout << "Course: CS1\n";
```

```
    std::cout << "Status: beginning C++\n";
    return 0;
}
```

Try it: Run the code

Trace each executable statement in `main`.

1. `std::cout << "Name: Ada\n";`

   Displays `Name: Ada` and moves to the next line.

2. `std::cout << "Course: CS1\n";`

   Displays `Course: CS1` and moves to the next line.

3. `std::cout << "Status: beginning C++\n";`

   Displays `Status: beginning C++` and moves to the next line.

4. `return 0;`

   Ends the program successfully.

The output is:

```
Name: Ada
Course: CS1
Status: beginning C++
```

The `#include <iostream>` line is necessary for this program, but it is not a statement executed inside `main`. It is a preprocessor directive. Use Appendix E when you need to confirm which header declares a standard-library facility. For now, remember that `#include <iostream>` gives the program access to standard input and output declarations.

Trace output programs by paying attention to three details:

1. The order of the output statements.
2. The exact text inside each string literal.
3. Whether each string literal includes `\n`.

Small differences matter.

Compare these two statements:

```
std::cout << "One\n";
std::cout << "Two\n";
```

The output is:

```
One
Two
```

Now compare:

```
std::cout << "One";
std::cout << "Two\n";
```

The output is:

```
OneTwo
```

The second program did not include a newline after `One`, so the next output appears on the same line. The computer did not choose that formatting. The program specified it.

Trace Before You Run

When you receive a small program, predict its output before running it.

For example:

```
#include <iostream>

int main()
{
    std::cout << "A";
    std::cout << "B\n";
    std::cout << "C\nD\n";
    return 0;
}
```

Try it: Run the code

Trace it:

1. `std::cout << "A";`

   Produces `A` with no newline.

2. `std::cout << "B\n";`

   Produces `B`, then a newline.

3. `std::cout << "C\nD\n";`

   Produces `C`, a newline, `D`, and a final newline.

The full output is:

```
AB
C
D
```

When your prediction and the actual output disagree, the computer is not being mysterious. It is following the program you gave it. Your job is to find the difference between what you thought the program said and what it actually says.

1.7. Guided Example: Build, Run, and Trace a Tiny Program

This guided example builds one small program from problem statement to test run.

Problem Statement

Write a C++ program that displays a short course banner:

```
C++ Programming I
Chapter 1
Status: ready
```

Step 1: Identify the Required Output

This program has no user input yet. The required output is fixed text.

Part	Description
Input	none
Process	none beyond displaying fixed text
Output	three lines of text

The algorithm is:

```
Display C++ Programming I.
Display Chapter 1.
Display Status: ready.
End the program successfully.
```

Step 2: Write the Source Code

Save the following program as `banner.cpp`.

```
#include <iostream>

int main()
{
    std::cout << "C++ Programming I\n";
    std::cout << "Chapter 1\n";
    std::cout << "Status: ready\n";

    return 0;
}
```

Try it: Run the code

Step 3: Compile the Program

From the directory containing `banner.cpp`, run:

```
g++ -std=c++17 banner.cpp -o banner
```

If the command succeeds, it creates an executable named `banner`.

If the command reports an error, read the first diagnostic. Check the file name and line number. Look for common early mistakes:

- missing semicolon
- missing quotation mark
- misspelled `iostream`
- misspelled `std::cout`
- missing brace

Step 4: Run the Executable

Run the program:

```
./banner
```

The expected output is:

```
C++ Programming I
Chapter 1
Status: ready
```

Step 5: Compare Expected and Actual Output

Do not stop at "it ran." Compare the output carefully.

Expected	Actual
`C++ Programming I`	Does the first line match exactly?
`Chapter 1`	Does the second line match exactly?
`Status: ready`	Does the third line match exactly?

Output-only programs are small, but they teach a professional habit: know what you expected before deciding whether the program worked.

Practice

Use these short practice tasks to build the first habits correctly: predict, trace, compile, run, and compare.

The practice set is deliberately small. Work it as a chapter problem set: predict first, debug second, write third, then check style and output evidence.

Predict: Output Statements

Predict the output before running the program.

```
#include <iostream>

int main()
{
    std::cout << "Red\n";
    std::cout << "Green";
    std::cout << "Blue\n";
    return 0;
}
```

Try it: Run the code

Check your answer by tracing each output statement. Pay attention to the missing newline after `Green`.

Debug: Fix the Syntax Errors

The following program has syntax errors. A **syntax error** is a violation of the grammar rules of the programming language.

```
#include <iostream>

int main()
{
    std::cout << "Practice program\n"
              std::cout << "Fix me\n";
    return 0
}
```

Try it: Run the code

Compile the program, read the first diagnostic, and fix the errors. Do not change lines randomly. Use the compiler's information.

Write: Build and Run a Tiny Program

Write a program named `profile.cpp` that displays three lines:

```
Name: your name
Course: CS1
Goal: compile and run a C++ program
```

Try it: Run the code

Compile it with:

```
g++ -std=c++17 profile.cpp -o profile
```

Run it with:

```
./profile
```

Compare the actual output with the required output.

Syntax Pattern: A Minimal Program

A minimal C++ program in this book has this shape:

```
#include <iostream>

int main()
{
    // statements go here
    return 0;
}
```

Copy the pattern by hand once. Label the preprocessor directive, the function header, the function body, and the return statement.

Style Check: Output Formatting

Review your `profile.cpp` program.

- Does each output line end intentionally, either with `"\n"` or by continuing on the same line?
- Are the displayed words spelled exactly as the required output shows them?
- Is the program indented so the body of `main` is easy to see?

Selected Checks

Use these answers to check your reasoning after you try the practice tasks.

- In the predict exercise, the missing newline after `Green` makes the second output line display `GreenBlue`.
- In the syntax-error exercise, the first output statement is missing a semicolon before the next `std::cout` statement.
- A successful first program is not only one that compiles. It should also display the exact required output.

Tool Note: Compilers, Diagnostics, and Debuggers

This chapter uses the compiler and the command line just enough to build and run small programs. Full tool tutorials belong in the appendices.

For now, you need these basics:

- Use `g++ -std=c++17 source.cpp -o program` to compile a simple C++17 program.
- Use `./program` to run a program in the current Linux directory.
- Read the first compiler diagnostic before editing code.
- Recompile after changing source code.

Compiler diagnostics help find translation problems. A debugger helps inspect a program while it runs. GDB is a common debugger on Linux. You do not need full debugger commands in this chapter, but you should know the purpose: a debugger lets you pause a running program, step through statements, and inspect values.

In early chapters, trace tables are your first debugger. They train the same habit: watch the program one step at a time.

Use Appendix A when you need the complete setup and command-line workflow. Use Appendix C's GDB workflow when you are ready to inspect a running program, or use its compiler warnings and assertions reference when you need a fuller guide to compiler diagnostics.

Additional Notes

If you already have programming or Linux experience, use this chapter to tighten your workflow. Compile from the command line, read the first diagnostic before changing code, and explain the path from source code to executable program without skipping steps.

Common Errors

Beginning programmers often make the same early mistakes. That is normal. The goal is to respond with a method instead of panic.

Error	Why it matters	What to do
Confusing source code with the executable	Editing the source file does not change the already-built executable.	Recompile after every source change.
Ignoring compiler diagnostics	The compiler is giving information about the first place it became confused.	Read the first diagnostic carefully.
Changing code randomly	Random edits can add new errors while hiding the original one.	Make one reasoned change, then recompile.
Assuming output proves the whole program is correct	One output can match by accident or cover only one case.	Compare against expected output and test more cases as programs grow.

Error	Why it matters	What to do
Misplacing braces	Braces mark blocks of code. A missing brace changes the program structure.	Match each `{` with a corresponding `}`.
Forgetting semicolons	Many C++ statements must end with `;`.	Check the statement before the line mentioned by the compiler.
Treating formatting as irrelevant	Readable code is easier to debug and maintain.	Use indentation and spaces consistently.

One practical rule: after a compiler error, check the line mentioned and the line immediately before it. A missing semicolon or quotation mark often causes the compiler to complain on the next line.

Debugging Checklist

Use this checklist when one of your first programs does not compile or does not display the expected output.

- Did you save the source file before compiling?
- Are you compiling the file you intended to compile?
- Does the compile command use `-std=c++17`?
- Did compilation succeed before you tried to run the executable?
- Did you recompile after editing the source file?
- Are you running the executable in the current directory with `./program`?
- Did you read the first compiler diagnostic before changing code?
- Does every output statement end with a semicolon?
- Does every string literal begin and end with a double quotation mark?
- Does each `{` have a matching `}`?
- Did you compare the actual output with the expected output line by line?

The checklist is not a substitute for understanding. It is a way to slow down when a small mistake makes the program look more confusing than it is.

Project Connection

This chapter begins a running habit:

1. Save source files with meaningful names.
2. Compile from the command line.
3. Run the executable.
4. Record the test run.
5. Explain what the program does.

For this chapter, the project artifact can be small: one output-only program, one compile command, one run command, and one short explanation.

A sample explanation might be:

```
The program named banner.cpp displays three fixed lines of course inf rmation.
It uses std::cout to send each line to standard output.
I compiled it with g++ using the C++17 standard and ran the executabl
from the Linux command line.
```

That explanation is not busywork. It forces you to name the source file, the executable, the output mechanism, and the toolchain.

Before You Continue

At this point, a tiny C++ program should no longer be mysterious. You should understand how source code becomes an executable, what its output statements do, and which parts of a computer store and execute the program.

Before moving to Chapter 2, make sure you can:

- edit, compile, and run a small C++17 program
- predict a program's output by tracing its statements
- distinguish a compile-time error from runtime behavior
- explain how bits, bytes, memory, storage, input, output, and the CPU fit into the basic computing model

If one of these checks is uncertain, return to the corresponding example or debugging procedure before adding input and calculations.

Bridge Forward

The programs in this chapter displayed fixed output. They did not read values, store changing information, or compute results from user input.

You will continue the same compile-run-trace workflow in Chapter 2 while adding values, variables, input, arithmetic expressions, named constants, and formatted output. The next step is the first useful calculator-style program: input comes in, a calculation transforms it, and a result comes out.

Glossary

This glossary summarizes the chapter's key terms. Each term was introduced earlier in context.

algorithm A finite, ordered step-by-step procedure for solving a problem.

binary notation Base-2 notation that writes numbers using only 0 and 1.

bit A value that can be either 0 or 1.

byte A group of 8 bits.

compile-time error An error found while the compiler is translating source code.

compiler A program that translates source code into a form that can become an executable program.

comment Text written for human readers and ignored by the compiler.

computation The process of carrying out an algorithm using a computing device.

CPU The central processing unit, the hardware component that executes program instructions.

data Information an algorithm uses, stores, changes, or produces.

decimal notation Base-10 notation that writes numbers using the digits 0 through 9.

escape sequence A sequence of characters that represents a special character, such as `\n` for a newline.

executable program A translated program that the operating system can load and run.

function A named block of code that performs a task.

hardware The physical parts of a computer system.

hexadecimal notation Base-16 notation that writes numbers using `0` through `9` and `A` through `F`.

input device A device that sends data into the computer.

machine instructions Low-level instructions that a computer's processor can carry out.

main memory Temporary storage, often called RAM, used for programs and data while they are running.

namespace A named region used to organize program names and avoid name conflicts.

nibble A group of 4 bits, corresponding to one hexadecimal digit.

operating system System software that manages hardware and provides services for programs.

output device A device that receives data from the computer.

peripheral A device connected to the computer system but not part of the CPU or main memory.

problem A task to be solved.

program A set of instructions written so a computer can carry them out.

runtime behavior What happens while an executable program is running.

shell A command-line program that accepts commands from the user and asks the operating system to carry them out.

software Programs that run on computer hardware.

source code Program text written by a programmer in a programming language.

standard output The normal destination for a program's text output, usually the terminal window.

statement An instruction in a program.

storage Long-term storage for files and programs, such as a solid-state drive.

string literal A sequence of characters written directly in a program and enclosed in double quotation marks, such as `"A"`.

syntax error A violation of the grammar rules of a programming language.

trace A step-by-step record of how a program or algorithm executes and how relevant values change.

variable A named memory location used to store a value while a program runs.

whitespace Spaces, tabs, and blank lines used to separate and arrange program text.

Chapter Review

This chapter introduced the first working model of programming in C++.

A program begins as source code, which is text written by a programmer. A compiler translates source code into an executable program. The operating system loads and runs the executable. While the program runs, the CPU executes instructions, main memory stores active programs and data, and output appears through standard output or another output device.

You also saw the first complete C++ program:

```
#include <iostream>

int main()
{
    std::cout << "C++ is running.\n";
    return 0;
}
```

The symbols matter, but the roles matter more:

- `#include <iostream>` makes standard input and output declarations available.
- `main` is where execution begins.
- Braces mark the function body.
- Statements give instructions.
- `std::cout` sends text to standard output.
- String literals provide exact characters to display.
- `\n` moves output to the next line.
- `return 0;` ends the program successfully.

You practiced tracing output statements before running a program. That habit matters. When your prediction and the actual output disagree, the difference points to what you misunderstood or what the program actually says.

You also saw the first connection between memory and notation. Decimal notation writes values in base 10. A bit stores 0 or 1, a byte groups 8 bits, binary notation writes values in base 2, and hexadecimal notation writes values in base 16.

Finally, you learned the difference between compile-time errors and runtime behavior. A program must compile before it can run, but successful compilation does not prove that the program is correct.

Programming Exercises

1. Write a program that displays your name, your course, and one sentence about what you expect to learn in programming. Use at least three separate output statements and make the output easy to read.

2. Write a program that displays a small receipt for three school supplies. The program should print the item names, prices, and a total that you calculate by hand before writing the program.

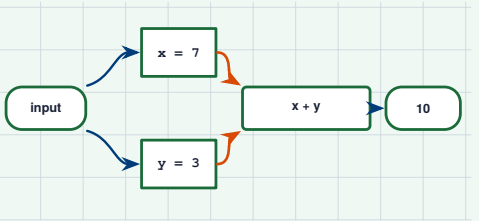
3. Write a program that prints a simple weekly schedule with one line for each class meeting. Use newline characters so that each day or meeting appears on its own line.

4. Write a program that displays a short “build and run checklist” for a beginner C++ programmer. Include the source file name, compile command, executable name, and the expected output.

5. Write a program that intentionally contains one missing semicolon, compile it, record the first compiler diagnostic, then fix the program and submit the corrected source file with a one-paragraph explanation of what the compiler reported.

2 Values, Variables, Input, and Expressions

Think • Design • Implement • Test • Document



The first programs in Chapter 1 displayed fixed text. If the program said `C++ is running.`, it always said `C++ is running.` If the program displayed three lines of course information, those three lines were already written into the source code.

That is useful for learning the compile-run-trace workflow, but it is not enough for most real programs.

A useful program usually works with values that may change. A grade calculator should work for different students. A bill calculator should work for different prices and quantities. A unit converter should work for different measurements.

To do that, a program must be able to store values, read input, compute results, and display output in a predictable form.

This chapter adds those abilities.

Purpose

The purpose of this chapter is to help you write a calculator-style program that reads values, computes a result, and formats the output.

Prerequisites

Before studying this chapter, you should be able to:

- edit and save a C++ source file, then compile and run it from the command line
- trace output statements before running the program
- distinguish source code from the executable produced by the compiler
- read the first compiler diagnostic before changing code

Chapter Outcomes

After this chapter, you can write a small input-process-output program.

An **input-process-output program** reads data, performs a calculation or transformation, and displays the result. Many useful programs begin with this pattern.

For example, a grade percentage calculator can read points earned and points possible, compute a percentage, and display the result.

```
percentage = points earned / points possible * 100
```

This small pattern introduces ideas used throughout the book: values have types, variables have state, expressions follow conversion rules, and output must communicate the intended result.

The chapter develops these abilities:

- explain the relationship among values, types, variables, initialization, and assignment
- declare and initialize variables and named constants using the book's naming conventions
- distinguish basic uses of `const` and `constexpr`
- read numeric input and trace changes in variable values
- write and evaluate integer, floating-point, and mixed arithmetic expressions
- use `static_cast` when a calculation requires a controlled conversion
- format numeric output predictably with `<iomanip>`
- design, run, and test a small calculator-style program

How to Pace This Chapter

This chapter has many small syntax details because calculator-style programs need values, variables, input, expressions, conversions, constants, casts, and formatted output. Read it in stages.

A good first stopping point is the end of **Values, Types, Variables, and Initialization**. At that point, make sure you can declare a variable, initialize it, name its type, and distinguish assignment from initialization.

A good second stopping point is the end of **Arithmetic Expressions**. At that point, make sure you can read numeric input, trace assignment, and evaluate an arithmetic expression using C++ operator precedence.

A good third stopping point is the end of **Integer Division, Floating-Point Values, and Conversions**. At that point, make sure you can predict when division keeps a fractional part and when it does not. Treat the discussion of overflow and underflow as an early warning about numeric limits rather than a complete treatment.

Then finish constants, formatted output, the trace-before-code activity, and the guided calculator example. The chapter is successful when you can trace the values before trusting the final output.

2.1. From Fixed Output to Input, Process, Output

A fixed-output program always displays the same result.

```
#include <iostream>

int main()
{
    std::cout << "Total: $50\n";
    return 0;
}
```

Try it: Run the code

This program runs, and it displays output. But it does not calculate anything. The total has already been typed into the source code.

If the item price changes, the program must be edited. If the quantity changes, the program must be edited. If a different customer buys a different number of items, the program still says the same thing.

A calculator-style program should not work that way. It should read values, compute a result, and display the result.

That pattern is called **input, process, output**, often shortened to **IPO**.

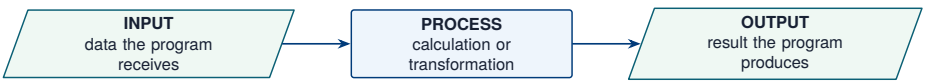


Figure 2.1: Input, process, and output separate the data a program receives from the calculation it performs and the result it displays.

Input is data read by a program. In this chapter, input will usually come from the keyboard through `std::cin`.

Process is the computation or transformation that converts input into output.

Output is data displayed, printed, stored, or otherwise produced by a program. In this chapter, output will usually appear on standard output through `std::cout`.

Consider this problem:

> Given the number of points a student earned and the number of points possible, compute the student's percentage grade.

Before writing C++, identify the IPO parts.

Part	Meaning in this problem
Input	points earned and points possible
Process	divide points earned by points possible, then multiply by 100
Output	percentage grade

The algorithm can be written in ordinary language:

```
Get the points earned.
Get the points possible.
Divide points earned by points possible.
Multiply the result by 100.
Display the percentage grade.
```

At this point, the solution is still not C++ code. That is fine. The goal is to understand the problem before choosing syntax.

A first design table might look like this:

Program name	`grade_percentage.cpp`
Input value 1	`points_earned`
Input value 2	`points_possible`
Computed value	`percentage`
Formula	`percentage = points_earned / points_possible * 100`
Output	percentage grade

The names in this table are meaningful names. A **meaningful name** tells the reader what a value represents. The name `points_earned` is more useful than `x`. The name `points_possible` is more useful than `y`.

The formula also gives us something to test before writing code.

Suppose the student earned `45` points out of `50`.

```
percentage = 45 / 50 * 100
           = 0.9 * 100
           = 90
```

This hand calculation uses ordinary arithmetic. The C++ rules for integer division appear later in this chapter.

The expected result is `90`.

That test case is small, but it matters. A **test case** is a specific set of input values and the expected result. You should know at least one expected result before you trust a program.

Here are two useful test cases.

Points earned	Points possible	Expected percentage
45	50	90
37	40	92.5

The program has not been written yet, but the design is already useful. You know what data must be read. You know what calculation must be performed. You know what output should appear for at least two cases.

That is the habit from Chapter 1, extended to programs that compute results:

1. Understand the problem.
2. Identify the input.
3. Identify the process.
4. Identify the output.
5. Work at least one test case by hand.
6. Then write C++.

The compiler can check whether your source code follows the rules of C++. It cannot decide whether your formula matches the problem you meant to solve. That part starts with you.

A First Look at the Complete Program

The complete grade percentage program will eventually look like this:

```
#include <iomanip>
#include <iostream>

int main()
{
    constexpr double PERCENT_FACTOR = 100.0;  // percent conversion

    int    points_earned   = 0;    // earned points
    int    points_possible = 0;    // possible points
    double percentage      = 0.0;  // percentage grade

    // input
    std::cout << "Points earned: ";
    std::cin >> points_earned;

    std::cout << "Points possible: ";
    std::cin >> points_possible;

    // process
    percentage =
        static_cast<double>(points_earned) / points_possible *
        PERCENT_FACTOR;

    // output
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Percentage: " << percentage << "%\n";

    return 0;
}
```

These phase comments are learning guides. Each label describes the purpose of a group, so the prompts belong to input even though they use `std::cout`. Later programs use more specific comments when validation, repetition, and functions make the work more detailed.

Try it: Run the code

Do not try to master every line at once. This chapter will build the program piece by piece.

For now, notice the shape:

Program part	Purpose
`#include <iomanip>` and `#include <iostream>`	Make input, output, and formatting tools available.

Program part	Purpose
`constexpr double PERCENT_FACTOR = 100.0;`	Names the conversion factor used to turn a ratio into a percent.
`int points_earned = 0;`	Creates and initializes a variable for the first input value.
`int points_possible = 0;`	Creates and initializes a variable for the second input value.
`double percentage = 0.0;`	Creates and initializes a variable to store the calculated percentage.
`std::cin >> points_earned;`	Reads a value from standard input.
`std::cin >> points_possible;`	Reads another value from standard input.
`static_cast<double>(points_earned)`	Converts one integer value so the division can keep a fractional result.
`std::fixed << std::setprecision(2)`	Formats the numeric output with two digits after the decimal point.

This program also shows why these foundational ideas deserve careful attention. A small calculator uses many ideas at once. If those ideas are rushed, the code may compile but still produce the wrong answer.

The rest of the chapter separates the ideas, explains each one, and then puts them back together in the guided example.

2.2. Values, Types, Variables, and Initialization

A **value** is a piece of data. The following are values:

```
42
3.75
"Hello"
```

The value `42` is a whole number. The value `3.75` has a fractional part. The value `"Hello"` is text.

C++ does not treat all values the same way. Each value has a **type**. A type tells the compiler what kind of value is being used and what operations are valid for that value.

This chapter uses four common types:

Type	Kind of value	Example
`int`	signed integer type for whole-number values	`42`
`double`	floating-point type for approximate fractional values	`3.75`
`bool`	Boolean value: either `true` or `false`	`true`

Type	Kind of value	Example
`std::string`	text value	initialized from `"Ada"`

An **integer** is a whole number with no fractional part. It can be negative, zero, or positive. The type `int` stores ordinary signed integer values, such as `-7`, `0`, and `42`.

A **floating-point value** can represent a fractional part. The type `double` is the usual floating-point type for beginning C++ programs. A `double` represents an approximation, so later in this chapter you will see that some decimal values cannot be stored exactly.

Note

`42` and `42.0` are not the same kind of value in C++. `42` is an `int`, while `42.0` is a `double`. Adding `.0` can change a calculation from integer arithmetic to floating-point arithmetic.

The number lines below show the basic distinction. Integer values occupy whole-number positions. Floating-point values can represent selected values between those positions.

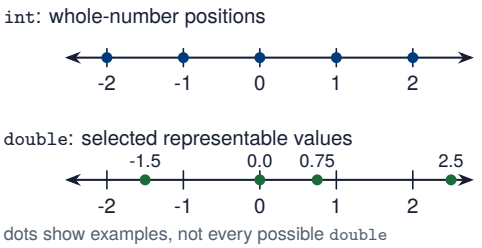


Figure 2.2: An `int` represents signed whole-number values, including zero; the marked `double` values are selected representable values between whole numbers.

The `int` type is signed, so it can represent negative as well as positive whole numbers. An **unsigned integer** type starts at zero and cannot represent a negative value. Appendix E provides the fuller signed and unsigned type reference; this chapter uses `int` for ordinary calculator input and calculations.

A **Boolean value** is either `true` or `false`. The type `bool` stores a Boolean value. Chapter 3 uses Boolean values to express program decisions.

The quoted text `"Ada"` is a **string literal**, a form introduced in Chapter 1. A string literal can initialize a `std::string` object, but the literal and the object are not the same type. The type `std::string` requires the `<string>` header. It appears here as a preview; text input is introduced later. You do not need `<string>` for the numeric programs in this chapter.

Every type has a **representable range**: the set of values that the type can store. For `bool`, that set contains the two Boolean values `false` and `true`. For `int`, it contains a finite range of signed whole numbers. For `double`, it contains a finite set of approximate floating-point values, including many values between whole numbers.

A program also has a **valid input domain**: the values that make sense for its particular problem. The valid domain is often much smaller than the type's representable range. For example, an `int` can represent negative values, but a program that reads the number of students in a section should accept only a nonnegative count. Chapter 3 uses selection statements to check such requirements.

A **variable** is a named memory location used to store a value while a program runs. The name lets the program refer to the stored value later.

You can picture a variable as a labeled storage box inside the running program:

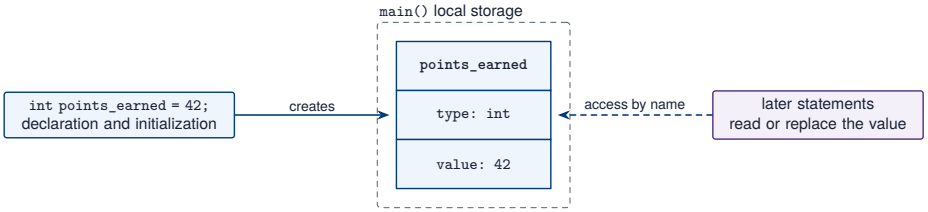


Figure 2.3: A variable gives a named location in local storage a type and a current value.

This statement creates a variable:

```
int points_earned = 42;
```

The statement has three important parts.

Part	Meaning
`int`	The variable stores an integer value.
`points_earned`	The variable's name is `points_earned`.
`= 42`	The variable is given the initial value `42`.

A **declaration** introduces a name and its type to the compiler. In the local variable examples in this book, a declaration also creates the variable, so it is also a **definition**. The statement above declares and defines `points_earned`.

Initialization gives a variable its first value when it is created. In this book, variables are initialized in their declarations unless there is a specific reason not to.

In a trace table, write *not declared yet* before execution reaches a local variable's declaration. At that point, the local variable does not exist and cannot be used. Reserve *not defined*

yet for later situations in which a name has been declared but its required definition or implementation is missing.

This is good:

```
int    points_earned = 0;      // earned points
double price         = 0.0;    // item price
bool   is_valid      = false;  // validation state
```

This is potentially dangerous:

```
int points_earned;
```

The second statement declares a variable but does not give it a known starting value. It also violates the style used in this book. Reading the value of an uninitialized local variable is an error. It can produce unpredictable behavior.

Beginners sometimes assume that every numeric variable automatically starts at `0` or that every Boolean variable starts as `false`. Do not depend on that. Initialize variables yourself.

Initialization Syntax

C++ supports more than one initialization syntax.

A **scalar variable** stores one value at a time. The `int`, `double`, and `bool` variables in this chapter are scalar variables.

This book begins with = initialization for simple scalar variables because it is easy to read while learning variables, assignment, expressions, input, and tracing.

```
int    points_earned = 42;     // earned points
double price         = 9.95;   // item price
bool   is_valid      = false;  // validation state
```

Modern C++ also commonly uses **brace initialization**:

```
int    points_earned {42};     // earned points
double price         {9.95};   // item price
bool   is_valid      {false};  // validation state
```

Both examples create variables and give them initial values. The braces are not a different kind of variable. They are a different initialization form.

Brace initialization is useful because it helps prevent **narrowing conversions**. A narrowing conversion changes a value to a type that cannot be guaranteed to preserve

the original value exactly. It may discard a fractional part, fail to represent the source value's full range, or lose precision.

For example:

```
int value = 3.7;  // value becomes 3
int other {3.7};  // error: narrowing conversion
```

An `int` stores whole numbers only. To store `3.7` in an `int`, C++ must convert the floating-point value to a whole-number value. This conversion discards the fractional part. The result stored in `value` is `3`.

This rule is called **truncation toward zero**. For example, `3.7` becomes `3`, and `-3.7` becomes `-3`. In both cases, the result moves toward zero.

The = form permits this implicit conversion, even though information is lost. Brace initialization rejects the initialization because `3.7` cannot be represented as an `int` without discarding `.7`. That rejection helps you notice a type choice that may be wrong for the problem.

You may also see empty braces:

```
int    count {};  // current count
double total {};  // running total
bool   found {};  // search result
```

This form is called **value-initialization**. It gives simple numeric and Boolean variables a zero-like value. It is valid modern C++, but the early examples in this book usually write the value explicitly:

```
int    count = 0;      // current count
double total = 0.0;    // running total
bool   found = false;  // search result
```

The explicit values are easier to trace when you are first learning. The array, vector, struct, and class chapters use brace initialization more often because braces become the natural form for those topics.

Assignment

Assignment stores a value in a variable after the variable has been created.

```
int count = 0;  // current count
count = 5;
```

The first statement declares and initializes `count`. The second statement assigns a new value to `count`.

Read the assignment statement from right to left:

```
Store the value 5 in count.
```

The assignment operator = does not mean mathematical equality. It means "store the value on the right into the variable on the left."

This statement is valid:

```
count = count + 1;
```

In algebra, `count = count + 1` would be impossible. In C++, it means:

1. Get the current value of `count`.
2. Add `1`.
3. Store the result back into `count`.

If `count` starts as `5`, the trace is:

Statement	Value of `count` after the statement
`int count = 5;`	5
`count = count + 1;`	6

The expanded form is useful here because it makes assignment semantics visible. Chapter 4 introduces prefix increment and compound assignment, then uses those shorter forms for ordinary updates.

Do not confuse assignment with equality. Chapter 3 introduces the equality operator ==, which is used in Boolean expressions.

A Variable Trace Table

A trace table records variable values as a program runs. Consider this short program:

```
#include <iostream>

int main()
{
    int points_earned   = 45;  // earned points
    int points_possible = 50;  // possible points
    int difference      = 0;   // points missed

    difference = points_possible - points_earned;

    std::cout << "Points missed: " << difference << "\n";

    return 0;
}
```

Try it: Run the code

Trace the variables. The Operation column names the effect of each statement without repeating the complete source line. In the table, *not declared yet* means that the variable does not yet exist at that step.

Step	Operation	`points_earned`	`points_possible`	`difference`
1	initialize earned	45	not declared yet	not declared yet
2	initialize possible	45	50	not declared yet
3	initialize difference	45	50	0
4	assign difference	45	50	5

The output is:

```
Points missed: 5
```

The phrase `not declared yet` means the variable does not exist at that step. Once a variable is declared and initialized, the table records its value.

Trace tables help you find mistakes before the compiler or debugger gets involved. They also make expression evaluation visible.

Quick Check: Values and Variables

Before moving to input, make sure you can answer these questions without running the program:

- What is the difference between declaring a variable and assigning to it?
- What value does each variable have immediately after it is initialized?
- Which expression is evaluated before the assignment takes place?
- What line of output should the final `std::cout` statement produce?

2.3. Reading Input with `std::cin`

You used `std::cout` to send text to standard output in Chapter 1. This chapter adds `std::cin`, which reads data from standard input.

In a terminal, **standard input** usually comes from the keyboard. It can also be redirected from a file; Appendix A introduces input redirection for repeatable tests. The object `std::cin` belongs to the C++ standard library and is declared by the `<iostream>` header.

This program reads one integer and displays it back:

```
#include <iostream>

int main()
{
    int points_earned = 0;  // earned points

    std::cout << "Points earned: ";
    std::cin >> points_earned;

    std::cout << "You entered " << points_earned << "\n";

    return 0;
}
```

Try it: Run the code

The operator `>>` is the **stream extraction operator**. In this chapter, read it as "read from the input stream into this variable."

This statement:

```
std::cin >> points_earned;
```

means:

```
Read a value from standard input and store it in points_earned.
```

The variable must already exist before input can be stored in it. If the name has not been declared, compilation fails. Depending on the compiler, the diagnostic may say that the name "was not declared in this scope" or that the program used an "undeclared identifier."

Prompts

A **prompt** is output that tells the user what to enter.

Hint

Display a clear prompt immediately before every interactive input. Do not leave the user at a blinking cursor wondering what value to enter or which units to use.

```
std::cout << "Points earned: ";
std::cin >> points_earned;
```

The first statement displays the prompt. The second statement reads the input.

The prompt does not read anything by itself. This is a common beginner mistake:

```
std::cout << "Points earned: ";
```

That statement only displays text. Without `std::cin`, the program has not read a value.

Reading More Than One Value

A program can read multiple values:

```
int points_earned   = 0;  // earned points
int points_possible = 0;  // possible points

std::cout << "Points earned: ";
std::cin >> points_earned;

std::cout << "Points possible: ";
std::cin >> points_possible;
```

A sample run might look like this:

```
Points earned: 45
Points possible: 50
```

The user types `45` and presses Enter. Then the user types `50` and presses Enter.

The stream extraction operator can also be chained when related values belong in one clear prompt:

```
std::cout << "Points earned and possible: ";
std::cin >> points_earned >> points_possible;
```

The first `>>` reads a value into `points_earned`, and the second reads the next value into `points_possible`. The user may enter `45 50` on one line or enter the values on separate lines. Whitespace separates the input values. Use a combined prompt only when the requested values and their order are clear.

After the input statements run, the trace table is:

Statement	`points_earned`	`points_possible`
`int points_earned = 0;`	0	not declared yet
`int points_possible = 0;`	0	0
input `45` into `points_earned`	45	0
input `50` into `points_possible`	45	50

The initial values `0` are replaced by the values read from input.

This chapter assumes the user enters a value of the expected type. For example, if the program asks for an integer, the user enters an integer. Chapter 3 introduces validation and begins separating ordinary user input mistakes from programmer errors.

2.4. Arithmetic Expressions

An **expression** is a combination of values, variables, operators, and function calls that produces a value.

These are expressions:

```
points_earned
points_possible - points_earned
points_earned / points_possible * 100
```

C++ uses arithmetic operators for common calculations.

Operator	Meaning	Example
`+`	addition	`a + b`
`-`	subtraction	`a - b`
`*`	multiplication	`a * b`
`/`	division	`a / b`
`%`	remainder after integer division	`a % b`

The `%` operator works with integer operands. For example:

```
int remainder = 0;  // remainder

remainder = 17 % 5;
```

The value of `remainder` is `2`, because `17` divided by `5` is `3` with a remainder of `2`.

The grade percentage program does not need `%`, but many early programs do. A time conversion program might use `/` and `%` to convert total minutes into hours and leftover minutes.

```
int total_minutes = 135;  // total minutes
int hours         = 0;    // whole hours
int minutes       = 0;    // leftover minutes

hours   = total_minutes / 60;
minutes = total_minutes % 60;
```

The trace is:

Variable	Value
`total_minutes`	135
`hours`	2
`minutes`	15

An **integer expression** is an expression whose operands and result are integer values. In the time conversion above, `total_minutes / 60` and `total_minutes % 60` are integer expressions. They answer whole-number questions:

```
How many whole hours are in 135 minutes?
How many minutes are left over?
```

Integer expressions are useful when the problem asks for counts, indexes, menu choices, whole units, or remainders. They are not appropriate when the problem needs a fractional result.

Operator Precedence

Operator precedence determines which operators are evaluated first when an expression contains more than one operator.

Many math courses use the mnemonic **PEMDAS**, short for Parentheses, Exponents, Multiplication and Division, Addition and Subtraction. It is a useful starting point: parentheses control grouping, and multiplication or division happens before addition or subtraction. C++ uses related rules, but the mnemonic does not completely describe C++ expression evaluation.

Multiplication, division, and remainder are evaluated before addition and subtraction.

Multiplication, division, and remainder have the same precedence, so C++ evaluates them from left to right. Addition and subtraction also have the same precedence and are evaluated from left to right. C++ has no exponentiation operator. Do not use `^` to mean "raise to a power"; C++ gives `^` a different meaning that is outside this chapter.

```
int result = 0;  // computed result

result = 2 + 3 * 4;
```

The value of `result` is `14`, not `20`, because `3 * 4` is evaluated first.

Use parentheses when they make the expression clearer:

```
int result = 0;  // computed result

result = (2 + 3) * 4;
```

Now `result` is `20`.

Do not use parentheses as decoration around everything. Use them to show the intended grouping.

Appendix E gives a compact operator precedence summary. In early programs, if you are unsure, use parentheses and then check the expression by hand.

Expression Tracing

Suppose a program contains:

```
int points_earned   = 45;  // earned points
int points_possible = 50;  // possible points
int points_missed   = 0;   // missed points

points_missed = points_possible - points_earned;
```

Trace the third statement:

```
points_missed = points_possible - points_earned
              = 50 - 45
              = 5
```

Then store `5` in `points_missed`.

When an expression appears on the right side of an assignment, C++ evaluates the expression first. Then it stores the result in the variable on the left.

This statement:

```
points_earned = points_earned + 5;
```

does not mean both sides are equal forever. It means:

```
Take the current value of points_earned.
Add 5.
Store the result back into points_earned.
```

That is assignment, not algebraic equality.

2.5. Integer Division, Floating-Point Values, and Conversions

The grade percentage formula looks simple:

```
percentage = points_earned / points_possible * 100
```

But C++ has an important rule: when both operands of `/` are integers, C++ performs integer division.

Integer division discards the fractional part of the result.

Expression	Result	Reason
`45 / 50`	0	integer division discards `.9`
`50 / 50`	1	exact integer result
`37 / 40`	0	integer division discards `.925`
`7 / 2`	3	integer division discards `.5`

The following diagram shows why the percentage calculation can go wrong:

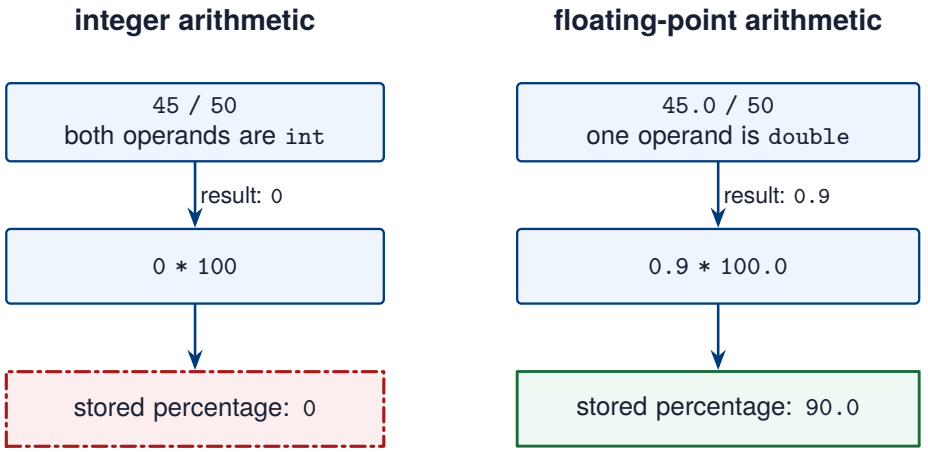


Figure 2.4: Integer division truncates before multiplication, while floating-point division preserves the fractional part of the percentage.

This is a serious problem for the grade percentage calculator.

```
int points_earned   = 45;  // earned points
int points_possible = 50;  // possible points
int percentage      = 0;   // percentage grade

percentage = points_earned / points_possible * 100;
```

Trace the expression:

```
percentage = points_earned / points_possible * 100
           = 45 / 50 * 100
           = 0 * 100
           = 0
```

The mathematically expected answer is `90`, but the program computes `0`!

The program is following the C++ rules. The error is in the type choices and expression.

Hint

When a numeric result is surprising, write the types above the expression before changing the formula. In C++, `45 / 50` and `45.0 / 50` are different calculations.

Floating-Point Division

A **floating-point value** can represent a fractional part. The type `double` is the usual floating-point type for beginning C++ programs.

If at least one operand of `/` is a floating-point value, C++ performs floating-point division.

Expression	Result
`45.0 / 50`	0.9
`45 / 50.0`	0.9
`45.0 / 50.0`	0.9

The corrected calculation should store the result in a `double`:

```
double percentage = 0.0;  // percentage grade

percentage = 45.0 / 50 * 100;
```

This produces `90.0`.

But the program reads `points_earned` and `points_possible` as integers. That is reasonable because points are often whole numbers. The program needs a way to perform floating-point division without changing the meaning of the input values.

Mixed Numeric Expressions and Implicit Conversions

A **floating-point expression** is an expression whose result is a floating-point value. The expression `45.0 / 50.0` is a floating-point expression because both operands are `double` values.

A **mixed numeric expression** contains operands of different numeric types. These expressions are mixed:

```
45.0 / 50
45 / 50.0
```

In each expression, one operand is an integer and the other is a floating-point value.

C++ usually evaluates a binary arithmetic operator, such as `+`, `-`, `*`, or `/`, by first finding a common type for the two operands. For the simple numeric cases in this chapter, if one operand is `int` and the other is `double`, C++ converts the `int` value to `double` before performing the operation.

That automatic conversion is an **implicit conversion**. The programmer does not write a cast. The compiler applies the conversion because the operation needs compatible operand types.

Trace this expression:

```
45 / 50.0 * 100.0
```

The value `45` is an `int`. The value `50.0` is a `double`. Before the division, C++ converts `45` to `45.0`.

```
45 / 50.0 * 100.0
= 45.0 / 50.0 * 100.0
= 0.9 * 100.0
= 90.0
```

The conversion changes the type used in the calculation. It does not change the original variable. If `points_earned` is an `int`, it remains an `int` after the expression is evaluated.

Implicit conversions are convenient, but they can hide an important design choice. When a calculation depends on preserving a fractional part, this book usually makes the conversion visible with `static_cast<double>`.

Hint

Convert an operand before integer division occurs. Converting the result afterward cannot restore the discarded fractional part.

Casts

A **cast** is an explicit type conversion requested by the programmer. In modern C++, prefer named casts such as `static_cast`.

This expression converts `points_earned` to `double` for the division:

```
static_cast<double>(points_earned)
```

Use it in the percentage calculation:

```
double percentage = 0.0;  // percentage grade

percentage =
    static_cast<double>(points_earned) / points_possible * 100.0;
```

Trace it with `points_earned` as `45` and `points_possible` as `50`:

```
percentage = static_cast<double>(45) / 50 * 100.0
           = 45.0 / 50 * 100.0
           = 0.9 * 100.0
           = 90.0
```

Only one operand has to become `double` to make the division floating-point division.

This version is also acceptable:

```
double percentage = 0.0;  // percentage grade

percentage =
    points_earned / static_cast<double>(points_possible) * 100.0;
```

In this book, the first version is used because it reads naturally: convert the earned points to a floating-point value before dividing.

Comparison Table

The following table shows how a small type difference changes the result.

Code	Result
`45 / 50 * 100`	0
`45 / 50 * 100.0`	0.0
`45.0 / 50 * 100`	90.0
`45 / 50.0 * 100`	90.0
`static_cast<double>(45) / 50 * 100`	90.0

The expression `45 / 50 * 100.0` is not a fix. Division and multiplication have the same precedence, so C++ evaluates them from left to right. It first evaluates `45 / 50` with integer operands, producing `0`. Only then does it multiply by `100.0`, producing `0.0`.

This is not a formatting issue. It is a computation issue. If the arithmetic is wrong, no output formatting command can repair it.

Overflow and Underflow

Numeric types have limits. An `int` can store only a certain range of whole-number values, and a `double` can represent only a limited range and precision of floating-point values.

The goal here is to recognize that arithmetic can exceed those limits. A complete treatment is not needed to write and test the calculator programs in this chapter.

An **overflow** occurs when a calculation produces a result outside the representable range of its type. An **underflow** occurs when a nonzero floating-point result is too close to zero to represent normally.

Note

C++17 supports **digit separators**. In a number written directly in source code, apostrophes group digits for human readers. They are not quotation marks and do not change the value, so `2'000'000'000` has the same value as `2000000000`.

For beginning programs, the practical lesson is simple: C++ arithmetic is not unlimited mathematics. The type matters. The following unsafe calculation illustrates the problem:

```
int score_total = 0;  // total score

score_total = 2'000'000'000 + 2'000'000'000;
```

The mathematical result is `4'000'000'000`. On a system with a 32-bit `int`, that value is too large for `int`. The addition overflows before its result can be stored in `score_total`. Signed integer overflow has undefined behavior in C++17. Do not depend on it to wrap around. For large counts, money, measurements, or totals, choose an appropriate type and test boundary values.

Floating-point underflow occurs at the other end of the scale. In the literal `1.0e-200`, the exponent notation `e-200` means "times 10 to the power -200." This calculation starts with that very small positive value and makes it smaller:

```
double tiny = 1.0e-200;  // small positive value

tiny = tiny * tiny;
```

The mathematical result is `1.0e-400`. On a typical 64-bit Linux system, that value is too close to zero for `double` to represent, so the stored result becomes `0.0`. Floating-point formats vary by implementation, so this example illustrates the condition rather than defining a portable numeric limit.

Underflow is different from ordinary floating-point approximation. Floating-point values can represent fractions, but not every decimal value exactly. They can also lose useful precision when very large and very small values are combined.

```
double amount = 0.0;  // amount to compute

amount = 0.1 + 0.2;
```

The result may be very close to `0.3` without being represented internally as exactly `0.3`.

Hint

When a numeric program seems correct for small examples, test boundary cases too: zero, one, the largest reasonable input, and values near any stated limit.

Appendix E summarizes fundamental numeric type families and the headers used in these books. Appendix F explains bytes and addresses and binary, hexadecimal, and integer representation.

2.6. Constants, Magic Numbers, and Meaningful Names

Programs become easier to read when important values have names.

The percentage calculation uses the value `100.0`:

```
double percentage = 0.0;  // percentage grade

percentage =
    static_cast<double>(points_earned) / points_possible * 100.0;
```

The value `100.0` has a meaning: it converts a ratio into a percent. A reader can probably infer that meaning here, but many literal values are less obvious.

A **magic number** is an unnamed literal value whose purpose is not clear from context.

This code works, but it hides meaning:

```
double total = 0.0;  // total cost

total = subtotal + subtotal * 0.0825;
```

What is `0.0825`? A tax rate? A service fee? A discount? The code does not say.

A **constant** is a named value that cannot be changed after it is initialized.

A **compile-time constant** is a named value that can be evaluated before the program runs. In C++, the keyword `constexpr` creates a value that must be known at compile time. That early knowledge lets the compiler reject an initializer that depends on runtime input and allows the constant to be used where C++ requires a value known at compile time.

```
constexpr double SALES_TAX_RATE = 0.0825;  // sales tax rate
double           total          = 0.0;     // total cost

total = subtotal + subtotal * SALES_TAX_RATE;
```

Now the reader knows what the value represents.

This book uses `constexpr` for fixed values known when the program is written, such as conversion factors, tax rates, menu choices, and sentinel values.

The keyword `const` is also important. Use `const` when a value should not change after initialization, but the value is not necessarily known at compile time. The function chapters use `const` for read-only parameters and `const` references; the class chapters in *Own the Design* use `const` member functions for operations that promise not to modify an object.

The short rule is:

Situation	Prefer
Fixed value known when the program is written	`constexpr`
Value should not change, but may be set at runtime	`const`

Both tools protect values from accidental change. `constexpr` is the stronger choice only when the value is intended to be a compile-time constant.

For the grade percentage program, write:

```
constexpr double PERCENT_FACTOR = 100.0;  // percent conversion
```

Then use it in the expression:

```
percentage =
    static_cast<double>(points_earned) / points_possible *
    PERCENT_FACTOR;
```

This is longer than using `100.0`, but it is clearer. Clarity matters more as programs grow.

Naming Variables and Constants

An **identifier** is a name that a programmer gives to a variable, constant, function, type, or other named program entity. C++ uses identifiers to distinguish one named entity from another. The identifiers used in this book contain letters, digits, and underscores, and they begin with a letter. Spaces and punctuation marks such as hyphens are not part of an identifier.

C++ identifiers are case-sensitive. For example, `score`, `Score`, and `SCORE` are three different identifiers. Using capitalization consistently prevents subtle naming errors.

A **keyword** is a word that C++ reserves for a language purpose. Names such as `int`, `double`, `if`, and `constexpr` cannot be used as identifiers. Also avoid identifiers that begin with an underscore because C++ reserves many underscore patterns for the implementation. Appendix E provides the complete C++17 keyword list and a more detailed reserved-identifier reference.

These rules apply to both variable identifiers and constant identifiers. After choosing a valid identifier, use a naming convention that communicates the identifier's role.

Use names that describe the role of a value.

Weak name	Better name
`x`	`points_earned`
`y`	`points_possible`
`z`	`percentage`
`n`	`total_minutes`
`r`	`remainder`

Short names are sometimes appropriate in narrow contexts, such as loop counters introduced later. In calculator-style programs, descriptive names are better.

This book uses `snake_case` for ordinary variables and for `const` variables whose values are obtained at run time:

```
int    points_earned = 0;    // earned points
double percentage    = 0.0;  // percentage grade
```

This book uses uppercase names with underscores for fixed named constants declared with `constexpr`:

```
constexpr double PERCENT_FACTOR   = 100.0;  // percent conversion
constexpr int    MINUTES_PER_HOUR = 60;     // minutes per hour
```

These conventions define the consistent style used in this book; they are not the only valid C++ style. Other developers and projects may choose different conventions. An instructor may require a course style, and a development team normally adopts a project style. Follow the applicable style and apply it consistently. Constant identifiers deserve the same care as variable identifiers.

2.7. Formatted Output with `<iomanip>`

If a program displays a percentage, price, or measurement, the output should be predictable. **Formatted output** is output arranged in a controlled form, such as a fixed number of digits after the decimal point.

Consider this program fragment:

```
double percentage = 92.5;  // percentage grade
std::cout << "Percentage: " << percentage << "%\n";
```

The output might be acceptable:

```
Percentage: 92.5%
```

But many reports should show a fixed number of digits after the decimal point:

```
Percentage: 92.50%
```

C++ provides output formatting tools in the `<iomanip>` header.

```
#include <iomanip>
#include <iostream>
```

The formatting tools `std::fixed` and `std::setprecision(2)` are commonly used together. They display a floating-point value with two digits after the decimal point. `std::fixed` selects decimal-style output, and `std::setprecision(2)` sets the number of displayed digits after the decimal point.

```
std::cout << std::fixed << std::setprecision(2);
std::cout << "Percentage: " << percentage << "%\n";
```

If `percentage` is `92.5`, the output is:

```
Percentage: 92.50%
```

The formatting command changes how later floating-point values are displayed on that output stream. It does not change the stored value of the variable.

Trace the idea:

Step	Action	Stored value of `percentage`	Displayed form
1	assign `percentage`	92.5	not displayed yet
2	set fixed precision	92.5	future floating-point output uses two digits
3	display `percentage`	92.5	`92.50`

Formatted output is not the same as rounding the stored value. It controls the printed representation.

Use Appendix E when you need to confirm which header declares an output manipulator.

2.8. Trace-Before-Code Activity

Trace the following program before running it.

```
#include <iomanip>
#include <iostream>

int main()
{
    constexpr double PERCENT_FACTOR = 100.0;  // percent conversion

    int    points_earned   = 37;   // earned points
    int    points_possible = 40;   // possible points
    double percentage      = 0.0;  // percentage grade

    percentage =
        static_cast<double>(points_earned) / points_possible *
        PERCENT_FACTOR;

    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Percentage: " << percentage << "%\n";

    return 0;
}
```

Try it: Run the code

Complete the trace table. The last three columns record the current values of `points_earned`, `points_possible`, and `percentage` after each step.

Step	Statement	Earned	Possible	Percentage
1	`constexpr double PERCENT_FACTOR = 100.0;`	not declared yet	not declared yet	not declared yet
2	`int points_earned = 37;`	37	not declared yet	not declared yet
3	`int points_possible = 40;`	37	40	not declared yet
4	`double percentage = 0.0;`	37	40	0.0
5	compute `percentage`	37	40	92.5
6	format and display output	37	40	92.5

The expected output is:

```
Percentage: 92.50%
```

Now change only the calculation to the following incorrect version:

```
percentage = points_earned / points_possible * PERCENT_FACTOR;
```

Trace it again:

```
percentage = 37 / 40 * 100.0
           = 0 * 100.0
           = 0.0
```

The output becomes:

```
Percentage: 0.00%
```

This is why tracing matters. The program compiles. The output is formatted. The answer is still wrong because integer division happened before the multiplication.

2.9. Guided Example: Grade Percentage Calculator

Now put the chapter's ideas together.

Problem Statement

Write a C++ program that reads the number of points a student earned and the number of points possible, then displays the student's percentage grade with two digits after the decimal point.

Step 1: Identify Input, Process, and Output

IPO part	Description
Input	points earned and points possible
Process	divide earned points by possible points, then multiply by 100
Output	percentage grade displayed with two digits after the decimal point

Step 2: Choose Test Cases

Use at least two test cases.

Test case	Points earned	Points possible	Expected output
1	45	50	`Percentage: 90.00%`
2	37	40	`Percentage: 92.50%`

These test cases are chosen because the second one requires a fractional result. It helps catch integer division errors.

This chapter does not yet validate impossible input such as `0` points possible or negative points. Chapter 3 adds decisions and validation.

Step 3: Write the Program

```
#include <iomanip>
#include <iostream>

int main()
{
    constexpr double PERCENT_FACTOR = 100.0;  // percent conversion

    int    points_earned   = 0;    // earned points
    int    points_possible = 0;    // possible points
    double percentage      = 0.0;  // percentage grade

    // input
    std::cout << "Points earned: ";
    std::cin >> points_earned;
```

```
    std::cout << "Points possible: ";
    std::cin >> points_possible;

    // process
    percentage =
        static_cast<double>(points_earned) / points_possible *
        PERCENT_FACTOR;

    // output
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Percentage: " << percentage << "%\n";

    return 0;
}
```

Try it: Run the code

Step 4: Compile and Run

If the file is named `grade_percentage.cpp`, compile it with:

```
g++ -std=c++17 grade_percentage.cpp -o grade_percentage
```

Run it with:

```
./grade_percentage
```

Sample run:

```
Points earned: 45
Points possible: 50
Percentage: 90.00%
```

Second sample run:

```
Points earned: 37
Points possible: 40
Percentage: 92.50%
```

Step 5: Explain the Program

The program starts by including two headers.

```
#include <iomanip>
#include <iostream>
```

The `<iostream>` header provides `std::cout` and `std::cin`. The `<iomanip>` header provides `std::setprecision`.

The program then defines a named constant:

```
constexpr double PERCENT_FACTOR = 100.0;  // percent conversion
```

This avoids hiding the purpose of `100.0` inside the calculation.

The variables are declared and initialized near the beginning of `main`:

```
int    points_earned   = 0;    // earned points
int    points_possible = 0;    // possible points
double percentage      = 0.0;  // percentage grade
```

The short comments to the right of these declarations identify each variable's purpose. Chapter 1 introduced comments as text for human readers that the compiler ignores. Here, the comments make the declaration block easier to scan.

The program displays prompts and reads input:

```
std::cout << "Points earned: ";
std::cin >> points_earned;

std::cout << "Points possible: ";
std::cin >> points_possible;
```

The program computes the percentage:

```
percentage =
    static_cast<double>(points_earned) / points_possible *
    PERCENT_FACTOR;
```

The cast forces floating-point division. The result is stored in a `double`.

The program formats and displays the result:

```
std::cout << std::fixed << std::setprecision(2);
std::cout << "Percentage: " << percentage << "%\n";
```

The output is predictable and easy to compare against expected results.

Practice

Use these practice tasks to build small calculator-style programs. The goal is not to memorize a reference table. The goal is to predict, trace, calculate, compile, run, and compare.

Work the tasks in order. Start with hand calculation, then trace storage, then write programs, and finally compare actual output with expected output.

Concept Check: Evaluate Expressions by Hand

For each expression, compute the result before using a compiler. Write your result in the table, then check your reasoning against the selected checks at the end of this practice set.

Expression	Your result
`7 + 3 * 2`	
`(7 + 3) * 2`	
`17 / 5`	
`17 % 5`	
`17.0 / 5`	
`static_cast<double>(17) / 5`	

If any result surprises you, identify which rule explains it.

Trace: Variable Values

Trace the following program.

```
#include <iostream>

int main()
{
    int total_minutes = 135;  // total minutes
    int hours         = 0;    // whole hours
    int minutes       = 0;    // leftover minutes

    hours   = total_minutes / 60;
```

```
    minutes = total_minutes % 60;

    std::cout << hours << " hour(s) and "
              << minutes << " minute(s)\n";

    return 0;
}
```

Try it: Run the code

Expected output:

```
2 hour(s) and 15 minute(s)
```

Explain why `/` and `%` are both useful in this program.

Write: Bill Calculator

Write a program named `bill_calculator.cpp` that reads:

- item price
- quantity

Then it displays the subtotal.

Use `double` for the price and subtotal. Use `int` for the quantity.

Sample run:

```
Item price: 12.50
Quantity: 4
Subtotal: $50.00
```

Use `std::fixed` and `std::setprecision(2)` so the subtotal displays with two digits after the decimal point.

Style Check: Use a Named Constant

Write a program named `inches_to_feet.cpp` that reads a number of inches and displays the equivalent number of feet.

Use a named constant:

```
constexpr double INCHES_PER_FOOT = 12.0;  // inches per foot
```

Sample run:

```
Inches: 36
Feet: 3.00
```

Test Plan: Compare Expected and Actual Output

For each program you write in this chapter:

1. Choose at least two test cases.
2. Compute the expected result by hand.
3. Run the program.
4. Compare the actual output with the expected output.

Do not skip the hand calculation. It is how you know whether the program solved the intended problem.

Bug Prevention: Integer and Floating-Point Division

Before using `/`, ask what kind of result the problem requires.

- If both operands are integers, integer division produces an integer result.
- If at least one operand is a floating-point value, the result can include a fractional part.
- If the problem needs a percentage or average, make the conversion visible with a floating-point literal or `static_cast<double>`.

Syntax Pattern: A Cast for Division

Use this pattern when an integer count must participate in floating-point division:

```
average = static_cast<double>(total) / count;
```

The cast applies to `total`. It does not change the type of `total`; it produces a converted value for this expression.

Further Practice: Fractional Percentages

Test the same calculator with several input values that produce fractional percentages. Explain which results require floating-point division, which would be damaged by integer division, and which rely on an implicit conversion.

Selected Checks

Use these answers after you try the tasks.

- `7 + 3 * 2` produces `13` because multiplication happens before addition.
- `(7 + 3) * 2` produces `20` because the parentheses are evaluated first.
- `17 / 5` produces `3` because both operands are integers.
- `17 % 5` produces `2` because 17 is 3 groups of 5 with 2 left over.
- `17.0 / 5` and `static_cast<double>(17) / 5` both produce `3.4`.
- In the time trace, `135 / 60` gives `2` whole hours and `135 % 60` gives `15` leftover minutes.

Tool Note: Trace Tables Before Debugger Commands

A trace table is the first debugger in this book.

Later, GDB can pause a running program and inspect variables such as `points_earned`, `points_possible`, and `percentage`. That is useful, but the habit begins now. Before using a tool to inspect a value, learn to predict the value yourself.

For this chapter, use this workflow:

1. Work a test case by hand.
2. Trace the variables.
3. Write or edit the program.
4. Compile the program.
5. Run the program with the test case.
6. Compare actual output with expected output.

If the output differs, do not change code randomly. Trace the expression that produced the wrong value.

Trace first. If the trace and actual behavior disagree, Appendix C gives the next-step GDB workflow for inspecting expressions and variable values.

Common Errors

The next two sections serve different purposes. Use the common-error table while writing or revising a program. Use the debugging checklist when the actual output differs from the result you expected.

Beginning programmers often make these mistakes in calculator-style programs.

Error	Example	Correction
Using an uninitialized variable	`int total; std::cout << total;`	Initialize variables before use.
Confusing assignment with equality	`x = x + 1` read as algebra	Read assignment as “store the right side in the left side.”

Error	Example	Correction
Forgetting integer division	`45 / 50 * 100` gives `0`	Use `double`, a decimal literal, or `static_cast<double>`.
Hiding a needed conversion	`earned / possible * 100.0`	Convert before the division.
Hiding meaning in a magic number	`subtotal * 0.0825`	Use a named constant such as `SALES_TAX_RATE`.
Combining input, process, and output too tightly	one long statement does everything	Use named variables for important intermediate results.
Producing inconsistent output	sometimes `92.5`, sometimes `92.50`	Use `std::fixed` and `std::setprecision`.
Forgetting a required header	using `std::setprecision` without `<iomanip>`	Include `<iomanip>`.
Expecting the prompt to read input	only writing `std::cout << "Value: ";`	Use `std::cin >> variable;` after the prompt.

One more common error is dividing by zero. For the grade calculator, `points_possible` should not be `0`. This chapter does not yet have the decision statements needed to reject that input cleanly. Chapter 3 adds that ability.

Debugging Checklist

Use this checklist when a calculator-style program gives the wrong result.

- Did you initialize every variable before using it?
- Did you recompile after editing the source file?
- Are you reading input into the correct variables?
- Did you type the test input values in the expected order?
- Does each variable have the correct type?
- Does the expression use integer division when you expected floating-point division?
- Does a mixed expression rely on an implicit conversion?
- Should one operand be converted with `static_cast<double>`?
- Are the operations grouped correctly with parentheses?
- Did you use a named constant for fixed values with meaning?
- Did you include `<iomanip>` before using `std::setprecision`?
- Did you set `std::fixed` before expecting two digits after the decimal point?
- Did you compare actual output with expected output exactly?

When the answer is wrong, trace the expression. If the displayed form is wrong, check formatting. These are different problems.

Project Connection

For your first calculator-style project, build a small useful program.

Choose one calculator-style program:

- grade percentage calculator
- bill subtotal calculator
- unit converter
- one-record report calculator

Your program should:

- read at least two input values
- initialize every variable
- use meaningful variable names
- use at least one named constant when a fixed value has meaning
- perform at least one arithmetic calculation
- use formatted output for a numeric result
- include at least two test cases with expected and actual output

A short project report can use this format:

```
Program: grade_percentage.cpp
Purpose: Compute a student's percentage grade.
Input: points earned, points possible
Process: earned / possible * 100
Output: percentage displayed with two decimal places

Test 1:
Input: 45, 50
Expected: Percentage: 90.00%
Actual:   Percentage: 90.00%

Test 2:
Input: 37, 40
Expected: Percentage: 92.50%
Actual:   Percentage: 92.50%
```

This is a small program, but it is a real programming workflow: design, code, compile, run, test, and explain.

Before You Continue

Variables and expressions give a program a data-processing path: receive values, transform them according to their types, and display a result. The important skill is not merely writing an expression, but predicting how C++ will evaluate it.

Before moving to Chapter 3, make sure you can:

- trace declaration, initialization, assignment, and numeric input
- predict integer, floating-point, and mixed-type arithmetic, including division and conversion
- use named constants, controlled casts, and numeric formatting deliberately
- design and test a small input-process-output program

If one of these checks is uncertain, revisit the calculator examples and compare predicted output with actual output before adding decisions.

Bridge Forward

The grade percentage calculator can compute a result, but it trusts the input.

If the user enters `0` for points possible, the program attempts to divide by zero. If the user enters negative points, the program still computes a percentage. If the program needs to display a letter grade, it has no way yet to choose among `A`, `B`, `C`, `D`, and `F`.

Decisions come next in Chapter 3. With Boolean expressions, `if` statements, validation checks, and named states, programs can choose among alternatives and reject data that does not make sense for the problem.

Glossary

This glossary summarizes the chapter's key terms. Each term was introduced earlier in context. Use the glossary to review a term when you encounter it in an explanation or exercise; do not try to memorize the entire list in one sitting.

assignment The operation of storing a value in a variable.

brace initialization Initialization syntax that uses braces, such as `int count {0};`.

Boolean value A value that is either `true` or `false`.

cast An explicit type conversion requested by the programmer.

compile-time constant A named value that can be evaluated before the program runs.

constant A named value that cannot be changed after it is initialized.

`constexpr` A C++ keyword used to declare a value that must be known at compile time.

declaration A statement that introduces a name and its type to the compiler.

digit separator An apostrophe placed between digits in a number written in source code to improve readability, such as `2'000'000'000`.

definition A declaration that creates an object or provides an implementation. For the local variables in this chapter, the declaration is also the definition.

`double` A C++ floating-point type used for approximate values that may include a fractional part.

expression A combination of values, variables, operators, and function calls that produces a value.

exponent notation Numeric notation in which `e` indicates a power of 10, such as `1.0e-200` for 1.0 times 10 to the power -200.

floating-point value A numeric value that can represent a fractional part, such as `3.75`.

floating-point expression An expression whose result is a floating-point value.

formatted output Output arranged in a controlled form, such as a fixed number of digits after the decimal point.

implicit conversion An automatic type conversion applied by the compiler.

identifier A name that a programmer gives to a variable, constant, function, type, or other named program entity.

initialization Giving a variable or constant its first value when it is created.

input Data read by a program.

input-process-output program A program that reads data, performs a calculation or transformation, and displays the result.

`int` A C++ signed integer type used for whole-number values.

integer A whole number with no fractional part, such as `-7`, `0`, or `42`.

integer division Division performed with integer operands, where the fractional part is discarded.

representable range The complete set of values that a type can store on a particular implementation.

integer expression An expression whose operands and result are integer values.

keyword A word reserved by C++ for a language purpose that cannot be used as an identifier.

magic number An unnamed literal value whose purpose is not clear from context.

meaningful name An identifier that tells the reader what a value represents.

mixed numeric expression An expression that contains operands of different numeric types, such as an `int` and a `double`.

named constant A constant with an identifier that explains the value's purpose.

narrowing conversion A conversion to a type that cannot guarantee an exact representation of every source value, such as converting `3.7` to an `int`.

operator precedence The rules that determine which operators are evaluated first in an expression.

output Data displayed, printed, stored, or otherwise produced by a program.

overflow A condition in which a calculation produces a result outside the representable range of its type.

process The computation or transformation that converts input into output.

prompt Output that tells the user what to enter.

scalar variable A variable that stores one value at a time, such as an `int`, `double`, or `bool`.

signed integer An integer type that can represent negative, zero, and positive whole-number values. The type `int` is signed.

standard input The normal input stream for a program, usually the keyboard in a terminal.

stream extraction operator The `>>` operator used with `std::cin` to read a value into a variable.

test case A specific set of input values and the expected result.

type A classification that tells the compiler what kind of value a name or expression can hold or produce.

underflow A floating-point condition in which a nonzero result is too close to zero to represent normally and may lose precision or become zero.

unsigned integer An integer type that can represent zero and positive whole-number values, but not negative values.

valid input domain The values that make sense and are accepted for a particular program problem.

value A piece of data, such as `42`, `3.75`, or `"Hello"`.

value-initialization Initialization with empty braces that gives simple numeric and Boolean variables a zero-like value.

variable A named memory location used to store a value while a program runs.

Chapter Review

This chapter extended the Chapter 1 compile-run-trace workflow to useful calculator-style programs.

You learned that a value is a piece of data and that a type tells the compiler what kind of value is being used. A variable is a named memory location that stores a value while the program runs. A declaration introduces a name and type. Initialization gives a variable its first value. Assignment stores a new value in an existing variable.

You used `std::cin` to read input and `std::cout` to display output. You wrote arithmetic expressions and traced how those expressions produce values. You also saw that C++ assignment is not algebraic equality. The statement

```
count = count + 1;
```

means "take the current value of `count`, add `1`, and store the result back into `count`."

The chapter's major arithmetic warning was integer division. If both operands of `/` are integers, C++ discards the fractional part. For percentage calculations, that can turn `45 / 50 * 100` into `0`. A floating-point expression or mixed numeric expression can preserve the fractional part. C++ may apply an implicit conversion in a mixed expression, but an explicit `static_cast<double>` makes the conversion visible when the calculation requires it.

You also learned to avoid magic numbers by using named constants:

```
constexpr double PERCENT_FACTOR = 100.0;  // percent conversion
```

Variable and constant names are identifiers. A valid identifier cannot be a C++ keyword, and the examples in this book use names that begin with a letter and contain only letters, digits, and underscores. Ordinary variables and runtime-initialized `const` variables use `snake_case`. Fixed `constexpr` constants use uppercase names with underscores.

Finally, you used `<iomanip>` tools such as `std::fixed` and `std::setprecision(2)` to produce predictable numeric output.

The complete skill is not any one of those details. The complete skill is the workflow:

1. Identify input, process, and output.
2. Choose meaningful names and appropriate types.
3. Initialize variables.
4. Write the expression carefully.
5. Trace at least one test case by hand.
6. Compile and run the program.
7. Compare actual output with expected output.

Programming Exercises

1. Write a program that reads the number of points earned and the number of points possible for one assignment, then displays the percentage score. Use floating-point division and display the result with one digit after the decimal point.

2. Write a program that reads a length and width in inches, computes the rectangle area and perimeter, and displays both results with clear labels.

3. Write a program that reads the price of one item, the quantity purchased, and a sales-tax rate as a percentage. Display the subtotal, tax amount, and final total with clear labels. Use `std::fixed` and `std::setprecision(2)` so every monetary value has exactly two digits after the decimal point. **

4. Write a program that reads a temperature in Fahrenheit and converts it to Celsius. Use a named constant for the fractional conversion factor and display the result with one digit after the decimal point.

5. Write a program that reads hours worked and hourly pay rate, then displays gross pay. Use meaningful variable names and purpose comments for every variable declaration.

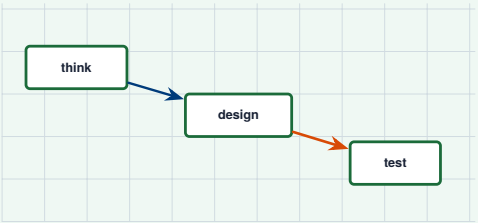
6. Write a program that reads the number of minutes a student spent studying on three different days, then displays the total minutes and the average minutes per day. Make sure the average is not computed with integer division. **

7. Extend one earlier fixed-output program from Chapter 1 so that it reads at least two values from standard input and computes one result. The output should include the input values and the computed result.

8. Write a program that reads a whole number of miles and a whole number of minutes, then displays the average speed in kilometers per hour. Use brace initialization for the variables, a `constexpr` conversion factor, and `static_cast<double>` so the calculation does not use integer division. Display the result with one digit after the decimal point. **

H Curriculum Alignment, Project Sequence, and Assessment Map

Think • Design • Implement • Test • Document



Purpose

This appendix is a compact planning map for curriculum review, adoption review, course planning, and assessment alignment. It summarizes how the two-book series supports CS1 and CS2 outcomes without claiming to cover an entire undergraduate computer science curriculum.

This appendix is written primarily for faculty, curriculum reviewers, program coordinators, and adoption committees. Students may use it to see the course path, but it is not intended to be assigned as ordinary chapter reading.

The series contains two student books. *Think Before Coding* directly supports the Part I / CS1 rows. *Own the Design* directly supports the Part II / CS2 rows. When this appendix appears in either book, rows outside that book refer to the companion book, not to coverage contained in the current book alone.

This appendix is not an official articulation document. Recheck the current institutional, state, ACM, IEEE, and program guidance before using it for adoption, transfer, or formal program-review decisions.

H.1. Coverage Labels

Label	Meaning
Core	Required in the book's CS1-CS2 sequence with reading, examples, practice, and assessment support.
Core / Partial	Core for the CS1/CS2 subset taught in the series, but only partial coverage of the broader CS2023 knowledge area.
Partial	Introduced or supported, but not complete for the knowledge area.
Extension	Additional preparation beyond the required CS1/CS2 sequence.
Out of Scope	Not taught except for incidental references.

No topic should be marked Core unless the book gives students direct reading, examples, practice, and assessment support for that topic.

H.2. CS2023 Knowledge Area Scope

Knowledge Area	Coverage	Notes
Artificial Intelligence	Out of Scope	AI-use disclosure appears in project practice, but AI concepts and systems are not taught.
Algorithmic Foundations	Core / Partial	Searching, sorting, tracing, basic recursion, introductory Big-O, simple invariants, informal correctness checks, and container cost are included. Trees, maps, hash tables, graphs, and advanced algorithms are deferred.
Architecture and Organization	Partial	Basic computer organization, memory, addresses, arrays, pointers, and 64-bit Linux context are included. Digital logic and architecture depth are not core.
Data Management	Partial	Files, records, tables, structured text, persistent data, data contracts, malformed-data policy, input-to-report data flow, and project data models are included. Databases and SQL are not core.
Foundations of Programming Languages	Core / Partial	Types, expressions, control flow, functions, scope, classes, templates, references, pointers, value/reference behavior, and checked-operation exceptions are included. Formal semantics and language implementation are not complete.
Graphics and Interactive Techniques	Out of Scope	Not a core topic.
Human-Computer Interaction	Out of Scope	Accessibility is a production goal for the manuscript, not a student learning outcome. HCI theory and usability testing are not taught.

Knowledge Area	Coverage	Notes
Mathematical and Statistical Foundations	Partial	Boolean logic, truth tables, informal algorithmic reasoning, recursion traces, and basic growth reasoning appear. Proof, counting, probability, and graph theory need other course support.
Networking and Communication	Out of Scope	Not a core topic.
Operating Systems	Partial	Command-line workflow, files, executable programs, memory errors, and tools appear. Scheduling, virtual memory, concurrency, and OS design are not core.
Parallel and Distributed Computing	Out of Scope	Parallel arrays are a data-representation topic; parallel and distributed algorithms and systems are not taught.
Security	Partial	Input validation, trust boundaries, defensive programming, assertions, checked-operation exceptions, memory safety, testing, and code review habits appear. Cryptography, web security, threat modeling, and network security are not complete.
Society, Ethics, and the Profession	Partial	Academic integrity, AI-use disclosure, correctness, documentation, reliability, evidence-based project claims, and professional habits appear. Privacy, law, sustainability, and professional codes need additional support.
Software Development Fundamentals	Core	Introductory programming, decomposition, testing, debugging, records, data structures, and project workflow are central.
Software Engineering	Core / Partial	Contracts, documentation, testing, debugging, modular design, requirements-to-acceptance-criteria practice, requirement-to-test matrices, ADT design, and project workflow are included. Team-scale lifecycle process is not complete.
Specialized Platform Development	Out of Scope	Not a core topic.
Systems Fundamentals	Partial	Memory representation, address-level thinking, command-line tools, pointers, dynamic memory, and systems preparation appear. Broader systems design is not complete.

H.3. ACM CCECC Alignment Notes

For CS1/CS2 sequences that prepare students for upper-division CS work, this book supports the programming and data structures portion most directly. Its strongest areas are programming fundamentals, structured problem solving, functions and modularity, basic recursion and algorithm-cost reasoning, arrays and vectors, records, file processing, testing, debugging, classes, ADTs, checked-operation exception handling, templates, pointers, dynamic memory, ownership, linked structures, stacks, and queues.

The book deliberately does not try to complete areas normally distributed across other courses, including discrete mathematics, architecture depth, operating systems, databases, networking, cybersecurity, HCI, graphics, platform development, professional codes, and broader social context.

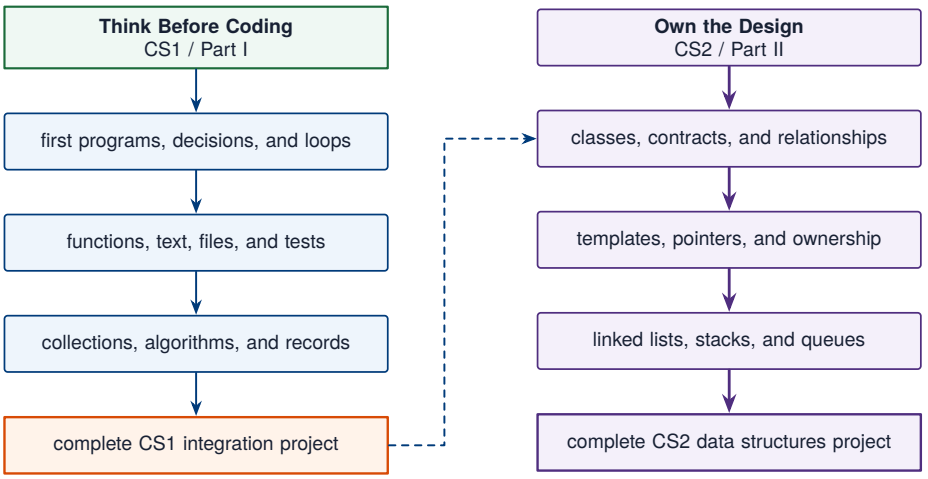


Figure H.1: The CS1 book builds structured-programming evidence through a complete project, and equivalent preparation leads into the CS2 book's class, ownership, linked-structure, and integration-project evidence.

H.4. Project Progression

The alignment map describes scope. The project progression describes evidence: what students should be able to produce as they move through the book.

CS1 Project Progression

Stage	Typical Evidence
Chapters 1-2	Compile and run small programs; trace statements, values, and output.
Chapters 3-4	Use selection, loops, tracing, validation, loop invariants, and concise correctness checks.
Chapters 5-6	Write functions, contracts, reference parameters, and focused function tests; identify deliberate function side effects; trace ordinary and recursive function calls.
Chapter 7	Process strings and characters safely.
Chapter 8	Read files, validate stored data, use command-line filenames, and carry a data contract from raw rows to reports.

Stage	Typical Evidence
Chapter 9	Store file-driven data sets in arrays and vectors; distinguish raw-array copying from standard-container copying.
Chapter 10	Search, summarize, sort, reason about cost, and process raw-array or vector-based grids; apply basic recursion to collections and strings.
Chapter 11	Use records and vectors of records to represent related data.
Chapter 12	Complete a programming fundamentals integration project with a data contract, design, requirement-based tests, and evidence.

Think Before Coding Assessable Skills Map

Chapter	Assessable Skills
Chapter 1	Explain the relationship among a problem, algorithm, source code, compilation, and execution; compile and run a program; trace statement order and output; read basic compiler diagnostics.
Chapter 2	Declare and initialize variables; use expressions and assignment; read keyboard input; format simple output; trace stored values.
Chapter 3	Write Boolean expressions; use `if`, `if/else`, and nested selection; trace branches; validate simple input.
Chapter 4	Use `while` and `for` loops; trace loop variables; design sentinel-controlled and counter-controlled loops; state loop invariants and concise correctness checks; avoid off-by-one errors.
Chapter 5	Decompose programs into functions; design function interfaces; write value-returning and `void` functions; document basic function contracts with Doxygen; test small functional units; call selected `<cmath>` functions; compare calculated floating-point values with a documented tolerance; identify the seed source, engine, and distribution in a modern random-number setup.
Chapter 6	Distinguish pass-by-value and pass-by-reference; extend function contracts to document reference parameters and deliberate side effects; derive normal, boundary, and invalid tests from contracts; debug function calls; identify base and recursive cases; trace recursive calls and returns.
Chapter 7	Process `std::string` and characters; use indexing safely; validate text input; apply character classification and conversion.
Chapter 8	Open and validate files; read token and line input; handle malformed data; use command-line filenames; produce simple reports.
Chapter 9	Store collections in arrays and vectors; copy raw-array elements into independent storage; explain standard-container copying; process file-driven data sets; use parallel arrays only when appropriate; trace indexed access.
Chapter 10	Search, summarize, sort, and trace collections; explain basic Big-O vocabulary; process two-dimensional raw arrays and vectors; apply recursion to collection and string ranges; justify collection algorithms with preconditions, invariants, progress, and postconditions.

Chapter	Assessable Skills
Chapter 11	Define records with `struct`; store vectors of records; search and summarize structured data; separate data representation from reporting.
Chapter 12	Integrate CS1 skills in a complete project; carry a data contract from file rows to records and reports; derive tests from requirements, equivalence classes, and boundary values; document and reflect on the result.

CS2 Project Progression

Part II uses a repeatable ADT development cycle: understand the behavior, specify the public contract, use the interface through client code and public-behavior tests, choose a representation, implement in checkpoints, then verify and analyze the result. The cycle may appear within one chapter or span several staged-container checkpoints. Each pass should leave evidence that the contract, representation, tests, ownership rules, and operation costs agree.

Stage	Typical Evidence
Classes	Classes, invariants, constructors, object-specific and static members, member functions, and separate files.
ADTs	ADT contracts, UML, public behavior, and representation independence.
Composition	Composition, class relationships, and limited inheritance.
Operator overloading and checked access	Conventional operator behavior, checked ADT operations, and class design refinement.
Templates	Function templates, class templates, generic ADT patterns, comparison policies, lambdas, and selected standard algorithms.
Pointer mechanics	Pointers, addresses, dynamic memory, arrays, and Valgrind evidence.
Ownership	Ownership, copying, moving, destruction, and dynamic `Vector<T>`.
Linked nodes	Node chains, traversal, insertion, removal, and cleanup.
Applied recursion	Recursive partitioning over half-open ranges and maze backtracking through generation and solving traces.
Linked lists	Linked-list ADT design, invariants, tests, and copy control.
Stacks	Stack ADT design and LIFO behavior.
Queues	Queue ADT design and FIFO behavior.
Integration project	Complete data structures project with design, tests, memory evidence, and reflection.

Staged Container Project Options

For a CS2 course that wants students to implement substantial data structures rather than only use supplied containers, the academic `Vector<T>` and one or both linked-list implementations form a coherent staged container project. A course may assign

the singly linked `ForwardList<T>`, the bidirectional `LList<T>`, or both. When both are assigned, `ForwardList<T>` is the recommended first experience because it makes singly linked ownership and after-position operations concrete. The `LList<T>` implementation remains independent: it begins from its own documented header, selected initial definitions, public tests, and client-program prompts rather than extending a completed `ForwardList<T>` class.

The student textbook provides the contracts, invariants, diagrams, traces, tests, and selected representative definitions needed to understand each mechanism. Student starter packages remain intentionally incomplete. Complete solutions belong in instructor-controlled resources and should be released only according to local course policy.

Typical time	Implementation emphasis	Expected evidence
four weeks	academic dynamic `Vector<T>`	four cumulative implementation checkpoints, focused tests, memory evidence, and four client programs
about two weeks per selected list	`ForwardList<T>` or independent `LList<T>`	ordered list milestones, link traces, memory evidence, and a new client at each deliverable
one week	`Stack<T>` or `Queue<T>` adapter	completed delegation methods or a supplied adapter, LIFO or FIFO tests, and one application client

The calendar is a pacing model rather than a reduction in requirements. Smaller milestones should still be compiled and tested in order even when several are submitted together as one weekly deliverable. Assigning both linked-list classes adds another implementation sequence; one is not a subclass or extension of the other.

Checkpoint	Chapter Basis	Student Implementation Focus	Required Evidence	New Client Program
fixed-capacity foundation	Chapters 13-14	private array, count, simple observers, insertion, checked access, ADT contract	invariant, public-behavior tests, trace	store and report a small collection
generic preparation	Chapter 17	class template and type aliases	tests with more than one element type	use one selected element type through the public interface
raw-storage preparation	Chapter 18	pointer states, dynamic arrays, allocation and release	pointer diagram and memory-checking evidence	process a run-time-sized collection
dynamic vector behavior and cleanup	Chapter 19	capacity, growth, insertion, access, removal, clear, and destruction	normal, boundary, empty, growth, and memory tests	process an unknown-length input collection

Checkpoint	Chapter Basis	Student Implementation Focus	Required Evidence	New Client Program
dynamic vector copying and moving	Chapter 19	deep copy, copy assignment, move construction, and move assignment	copy-independence, self-assignment, move, and memory tests	create an independent backup or transfer a collection
node preparation	Chapter 20	raw singly linked traversal, insertion, removal, and cleanup	link diagrams and memory-checking evidence	display or process a short node chain
forward-list implementation option	Chapter 21	head-only representation, forward traversal, after-position operations, clear, and destruction	empty, one-node, multi-node, and memory tests	maintain a domain collection through the forward-list interface
bidirectional-list implementation option	Chapter 21	previous links, tail access, bidirectional iteration, copy control, and positional modifiers	iterator, copy-independence, swap, and memory tests	compare or process records through the `LList<T>` interface
ADT application	Chapters 22-24	one stack or queue adapter plus an integration project	LIFO or FIFO tests, design explanation, project evidence	solve an undo, scheduling, simulation, or domain problem

The checkpoints deliberately assess both implementation and client use. A container is not complete merely because its unit tests pass; students should also use its public interface to solve a new problem without depending on private representation.

Own the Design Assessable Skills Map

Chapter	Assessable Skills
Chapter 13	Define simple classes; protect object state with private data members; use constructors, accessors, mutators, `const` member functions, static members, separate files, and value semantics.
Chapter 14	Explain an ADT from the client view; state public contracts; separate interface from representation; test behavior without depending on private data members.
Chapter 15	Use composition to model has-a relationships; protect class invariants across contained objects; distinguish composition, dependency, aggregation, and limited inheritance.
Chapter 16	Overload conventional operators when they clarify client code; explain `this` and `*this`; preserve class invariants; distinguish validation, assertions, and checked-operation exceptions; test equality, output, assignment, self-assignment, and checked access.

Chapter	Assessable Skills
Chapter 17	Write function and class templates; explain template instantiation; use STL-style type aliases, iterators, and selective `auto`; generalize list-like code without changing the ADT contract; sort records with named comparison functions and lambdas.
Chapter 18	Trace pointer values, addresses, null pointers, dangling pointers, pointer arithmetic, dynamic allocation, dynamic arrays, and deallocation.
Chapter 19	Explain ownership; implement copying, moving, destruction, and resizing for an academic dynamic `Vector<T>` through staged checkpoints; explain temporary ownership and basic exception safety; test resource-owning behavior and use the public interface in a client program.
Chapter 20	Build and trace linked nodes; explain head pointers, tail pointers, traversal, insertion, removal, and dynamic cleanup; trace recursive quicksort and explain maze-generation and maze-solving backtracking.
Chapter 21	Implement and test `ForwardList<T>`, `LList<T>`, or both through staged checkpoints; explain how their links, iterators, and operation costs differ; use each assigned public interface in a client program.
Chapter 22	Implement and test a stack ADT; enforce LIFO behavior; distinguish `top` from `pop`; compare vector-backed and linked-list-backed representations.
Chapter 23	Implement and test a queue ADT; enforce FIFO behavior; distinguish `front` from `pop`; compare vector-backed, circular-array-backed, and linked-list-backed representations.
Chapter 24	Integrate CS2 skills in a complete data structures project; select representations using behavior, cost, and ownership; document ADT contracts; derive public-behavior tests from requirements; provide memory-checking and reflection evidence.

H.5. CS2 Assignment Tracks

Part II includes recurring assignment tracks that an instructor may use across several chapters. A track gives students continuity while still leaving room to rotate domains across semesters.

Track	Early Focus	Later Focus
Score collection	Fixed-capacity score storage.	Dynamic vector, linked list, stack or queue application, and grade-processing project.
Course roster	Student records and roster operations.	Roster ADT, waitlist queue, and course-management project.
Inventory	Item records and inventory storage.	Inventory ADT, restock queue, and inventory-management project.
Sensor log	Validated sensor readings.	Dynamic logs, alert stack, event queue, and sensor-monitor project.

Track	Early Focus	Later Focus
Task or ticket	Task or ticket records.	Undo stack, service queue, and help-desk or project-board simulation.
Text processing	Word or token storage.	Linked word list, delimiter stack, token queue, and text-analysis project.

The score collection track is the closest match to the academic `Vector<T>` and linked-list progression used in the text. The other tracks provide variation for faculty who want the same data-structure outcomes with different application contexts.

H.6. Assessment Evidence

Category	Evidence
Correctness	Program meets stated requirements.
Requirements	Requirements are translated into observable acceptance criteria.
Design	Data model and function/class responsibilities are clear.
Style	Code follows Appendix B.
Testing	Normal, boundary, empty, and invalid cases are checked.
Debugging	Evidence shows investigation or verification.
Documentation	Comments and Doxygen notes explain interfaces.
Reflection	Student can explain choices and limits.
Integrity	Collaboration and AI use follow policy.

Chapter 12 projects should usually include source code, a design document, an implementation plan, input files, expected and actual test results, concise debugging evidence, basic function documentation, reflection, and AI-use disclosure when required.

Chapter 24 projects should usually include source code separated into appropriate files, class or ADT design notes, public interface documentation, public-behavior tests, memory-checking evidence when dynamic memory is used, copy-control tests when resources are owned, reflection, and AI-use disclosure when required.

A rubric should reward working behavior and understandable design. Small, correct, tested, explainable programs are better than large programs that students cannot reason about.

H.7. Gap Statement

This book is a CS1-CS2 textbook, not a full undergraduate curriculum. It prepares students for upper-division and later coursework by building strong programming,

testing, debugging, data structure, and design foundations. It does not claim complete coverage of CS2023, complete ACM CCECC pathway guidance, or all topics normally distributed across an undergraduate computer science program.

The appendices provide compact reference support for selected adjacent topics, including standard-library headers, numeric type families, mathematical functions, modern pseudo-random number generation, elapsed-time measurement, C-string functions, default arguments, namespaces, initializer lists, utility facilities, bounded exception propagation, bitwise operators, smart pointers, RAII, lambdas, iterator invalidation, and standard algorithms. Those topics are included to support reading, transfer, and upper-division preparation. They are not treated as full replacement units for discrete structures, computer organization, systems programming, or a complete algorithms course.

This appendix was checked against public CS2023 and ACM CCECC guidance on July 3, 2026.

Primary sources:

- CS2023 Knowledge Areas
- ACM CCECC
- ACM CCECC Computer Science guidance

Sources and Further Reading

This textbook's explanations, examples, exercises, diagrams, and academic container designs were written for this CS1-CS2 sequence unless a nearby note identifies an adaptation. The sources below provide standards context, document tools used by the book, or acknowledge identifiable teaching material that informed a particular presentation.

C++ Language and Library

- ISO/IEC 14882:2017, *Programming Languages — C++*, is the language standard used for this edition. The ISO C++ site explains how the standard is published and obtained.
- cppreference.com provides a detailed working reference for C++ language and standard-library facilities. It is useful after this book introduces a facility, but its reference pages are not written as a first-course tutorial.
- The Carnegie Mellon University Software Engineering Institute publishes the SEI CERT C++ Coding Standard, which documents rules and recommendations for secure, reliable C++ programs. It informs the book's technical guidance, but the textbook does not claim complete CERT conformance.

Teaching Examples and Design Guidance

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition, Prentice Hall, 1988. Chapter 20 of *Own the Design* adapts the compact recursive `qsort` teaching structure to C++17 vector iterators, half-open ranges, and `std::swap`; it does not reproduce the C implementation unchanged.
- Caltech CS 11, C++ Operator Overloading Guidelines. Chapter 16 of *Own the Design* follows the source's general emphasis on conventional operator meaning, compound assignment, and non-mutating binary operators while using original examples fitted to this book's class sequence.

Development and Testing Tools

- The GNU GDB documentation is the full reference for the debugger commands introduced just in time in chapters and summarized in Appendix C.
- The Valgrind User Manual documents the memory-checking tools introduced with pointers, ownership, and linked structures.

- The Catch2 documentation provides current setup and reference information for the testing patterns summarized in Appendix D.
- The Doxygen manual documents the interface-comment conventions introduced in the function chapters and summarized in Appendix B.

Curriculum Sources

Appendix H identifies the ACM and CS2023 curriculum sources used for the faculty-facing alignment map. Those mappings are planning aids rather than official articulation claims and should be checked against current program and institutional requirements.

Index

S

T

U

V

W