Resume-Day

Learn. Practise. Build.

THE LEARNING AND PRACTICE WORKBOOK

C++

From Your First Program to Confident Practice

Simple explanations. Familiar examples.

Objects, safe ownership and the standard library.

CHAPTERS 01-10 / SAMPLE

Ten complete chapters with practice answers.

Preview the explanations, diagrams and programs.

The full edition adds the workbook and companion code.

Contents - click a chapter or workbook title

Before you begin 3

01. What is C++ and why learn it? 4

02. Set up your tools and run a C++ file 9

03. Your first C++ program: understand every part 15

04. Print clear output and write useful comments 21

05. Variables, constants and data types 27

06. Read input and check it before using it 36

07. Operators and conversions: calculate without surprises 43

08. Conditions with if, else and else if 51

09. Choose a matching option with switch 58

10. Loops: repeat useful work and know when to stop 65

Before you begin

This book starts with your first C++ program. Read slowly, type the examples and try the questions before checking the answers. You do not need to know C first. A computer will help you practise; reading alone cannot replace running and changing programs.

How to use this book

Read the idea, follow the diagram, predict the output, then run the complete example. A code fragment shows only one part of a program. An intentional error is marked clearly: study the problem, but do not use it as working code. The output shown belongs to the stated input and conditions.

Build and run one program

Chapter 2 explains setup. Use a compiler that supports C++20. Save the source using the filename above the program. Each complete example is a separate program; do not link all examples together.

Build with GCC

```
g++ -std=c++20 -Wall -Wextra -Wpedantic study_group.cpp -o study_group
```

Run

```
./study_group
# Windows PowerShell: .\study_group.exe
```

The companion archive has input files where needed and expected outputs. One type-size example may print different numbers on your computer; its explanation describes why. Chapter 38 also has a three-file project with separate build instructions.

Lessons and the extra workbook

The 40 chapters include 165 complete single-file programs, 240 practice questions with answers, 196 interview answers and two guided projects. The extra workbook adds twelve debugging tasks, eight printable trace sheets with answers, twenty independent tasks with suggested approaches, project extension plans and three mock interviews. Extension plans are tasks for you to build; they are not completed features of the supplied project models.

This sample contains the first ten chapters in full. Later chapters and the extra workbook belong to the full edition.

An error tells you what to check next. It does not mean you cannot learn programming. Change one thing, test it and write down what you learned.

Edition: November 2026. Contents page numbers count the cover as page 1. Click the contents or use PDF bookmarks. Copyright 2026 Resume-Day. Personal learning use; do not redistribute or resell the book or archive. You may adapt the example code for your own learning projects.

YOUR FIRST STEPS

01. What is C++ and why learn it?

A program is a set of instructions that a computer follows. C++ is a language we can use to write those instructions. We will begin with small programs and explain each new word before asking you to use it.

Start with a familiar job

Think about a college library. Someone needs to record books, find a book by its number, and check whether a copy is available. A computer does not understand a request such as "please manage the library" by itself. We must describe the information and the steps clearly.

Programming means writing those steps in a form a computer can work with. **Code** is the text we write. A **programming language** gives that text rules and meaning. C++ is one such language; C, Java and Python are other examples.

You do not need to understand an entire library application today. Our first goal is smaller: make a program show a clear message and understand how that message reaches the screen.

What does the name C++ mean?

C++ grew from work by Bjarne Stroustrup on adding useful ways to organise programs to C. The name is pronounced **C plus plus**. In C and C++, ++ is an operator that increases a value by one. You will practise that operator in a later chapter.

C and C++ share much of their spelling and many basic ideas. However, **they are different languages**. Some valid C programs need changes before a C++ compiler will accept them. Renaming every `.c` file to `.cpp` is not a reliable way to convert a project.

Knowing C can help you with loops, functions, arrays and pointers. It is not a requirement for starting these lessons. If those words are new, we will meet them one at a time.

Why does C++ exist?

As programs grow, keeping related information and actions together becomes useful. For example, a book record has a title, a number and a count of available copies. A class can bring related values and operations together. We will build classes after learning the basic writing rules.

C++ also provides reusable tools. Instead of creating a text container from scratch, we can use `std::string`. Instead of building every resizable sequence ourselves, we can use `std::vector`. These tools belong to the **standard library**, a collection of facilities provided by a C++ implementation.

C++ supports more than one way to organise a solution. We can write small functions, build classes, and later write reusable code with templates. You do not have to put everything inside a class to write a C++ program.

Where can C++ be useful?

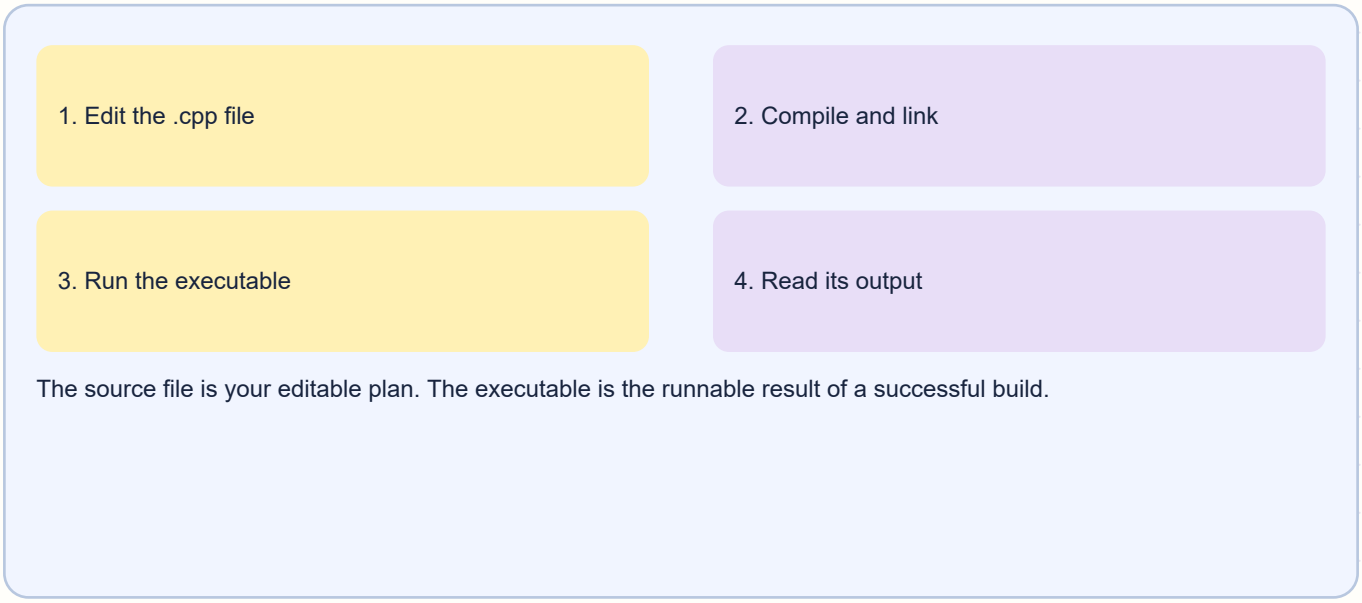
Kind of work	A possible task	Why C++ may suit it
Desktop software	A drawing or editing tool	Work with large amounts of data and respond to user actions
Games and simulations	Update a moving scene	Control how repeated calculations and memory are handled
Device software	Read a sensor and process its values	Work close to the device's limits and available tools
Libraries and services	Build a reusable processing component	Offer carefully designed operations to other parts of a system
Learning and interviews	Implement and explain a solution	Practise types, functions, containers and problem solving

These are examples, not a promise that C++ is the best choice for every task. A small website form, for example, may use a different language and set of tools. Choose a language that fits the job and the team.

From your text to a running program

A **source file** is a file containing your code. We will use names ending in `.cpp`. An **editor** helps you write that file. A **compiler** checks and translates source code. A **linker** connects the translated pieces and needed library parts to make a runnable program.

You usually ask a tool such as g++ to handle compilation and linking together. The **executable** is the resulting runnable file. Opening an editor is not the same as running the program.



When you change the source, save it and build again. Otherwise you might run an older executable and wonder why your new message does not appear.

A first look: a message for a study group

This complete program prints two lines. You may read it now without memorising its punctuation. Chapter 2 prepares the tools; Chapter 3 explains the program structure in detail.

study_group.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "C++ study group\n";
    std::cout << "One small program at a time.\n";
    return 0;
}
```

Output

```
C++ study group
One small program at a time.
```

Output is information a program produces. Here the output is text. `\n` asks for a new line; it is not printed as a backslash followed by the letter n.

Problem: put a useful notice on screen

A lab assistant needs a notice with a room and a start time. Before writing code, decide what the user should see. Our required output has two lines: the room first, then the time.

lab_notice.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Practice room: 204\n";
    std::cout << "Session starts: 10:30\n";
    return 0;
}
```

Output

```
Practice room: 204
Session starts: 10:30
```

The program is deliberately small. It does not read a timetable, accept input or send a notification. Learning to state what a program actually does will help you when the tasks become larger.

How we will learn

We will use **C++20** as the language setting. This is a published edition of C++. The early examples need only a small part of the language. A modern compiler with C++20 support will also support the basic features we use here.

The plan has 40 chapters. The first five are available in this first batch. Later chapters will cover input, decisions, loops, functions, arrays, classes, safe memory handling, library containers, files and projects. DSA remains a separate course; we will still explain how to use C++ containers here.

Read an explanation, predict an example's output, run it, and change one thing. An error is something to investigate, not a judgement about your ability to learn.

Common mistakes and useful tips

- Trying to memorise a large program before understanding a small one. Start with two output lines.

- Assuming C++ means "C with a different file extension". Some rules and preferred tools differ.

- Expecting the editor alone to build a program. Check that a compiler is installed too.

- Changing the source but running an old executable. Save, build, then run.

- Reading answers immediately. Try a short written explanation first, even if it is incomplete.

Practice: try before reading the answers

1. In your own words, what is a program?

2. Which file do you edit: the source file or the executable?

3. Is an editor the same thing as a compiler?

4. What are the two output lines of lab_notice.cpp?

5. Change the study-group message to say "We practise every day." Which part should change?

6. A friend says all C code is automatically valid C++. What would you tell them?

Answer: Check the practice answers

1. A program is a set of instructions for carrying out a task. A useful answer gives an example, such as showing a notice or counting available books.

2. Edit the source file, such as study_group.cpp. Then rebuild to produce a new executable.

3. No. An editor helps you write text. A compiler checks and translates source code. An IDE may bring several tools into one application.

4. The first line is "Practice room: 204". The second is "Session starts: 10:30".

5. Change the text inside the second pair of double quotes. Keep the surrounding program structure and the final newline escape.

6. C and C++ are related but different languages. Some valid C programs are rejected by C++, so conversion needs checking rather than only renaming files.

Interview questions

Answer: What is C++?

C++ is a general-purpose programming language. It lets us describe data and operations, write functions, build classes and use reusable library tools. A good beginner answer should explain what you have built with it rather than only repeat labels.

Answer: Must every C++ program contain a class written by the programmer?

No. A small program can use a main function and ordinary statements without defining a class of its own. Classes become useful when they help organise the problem.

Answer: What is the difference between source code and an executable?

Source code is the text we edit. An executable is the runnable result produced by the build tools for a particular environment. They are different files with different purposes.

Answer: Does learning C++ guarantee a programming job?

No. Language knowledge is one part of preparation. You also need practice, problem solving, projects, testing and the ability to explain your choices. We will work on those skills throughout the course.

Quick revision

- C++ is a language for writing programs, not the name of an editor.

- Source code, build tools and the executable have different jobs.

- C and C++ are related, but their rules are not identical.

- Begin with small tasks and check the actual output.

- The next chapter sets up the tools needed to run these examples.

02. Set up your tools and run a C++ file

You need a place to write code and a compiler to build it. We will take one recommended Windows route slowly, then explain alternatives for other systems. Choose one route; you do not need to install every tool listed here.

Understand the three tools

Tool	What it does	Familiar comparison
Editor	Helps you write and save code as plain text	A notebook for your instructions
Compiler and linker	Check and build the source into a runnable file	A workshop that turns a plan into a working item
Terminal	Lets you type commands to build and run files	A place to give short written requests to your tools

An **IDE**, short for integrated development environment, brings editing, building and debugging tools into one interface. A **debugger** helps you pause a running program and inspect its values; we will use that idea later.

VS Code is an editor. Its C/C++ extension helps with writing and debugging, but **the extension does not install a C++ compiler for you**. That difference explains many first-day setup problems.

Recommended Windows setup: MSYS2 and VS Code

These steps are for a supported 64-bit Windows installation. Download [MSYS2](https://www.msys2.org/) and [VS Code](https://code.visualstudio.com/) from their official sites. If your machine cannot use those current installers, ask your college lab administrator for a supported compiler setup rather than downloading an unknown old package.

1. Install MSYS2. Its usual folder is C:\msys64. If you choose another folder, use that folder when following the PATH steps below.
2. Open **MSYS2 UCRT64** from the Start menu. MSYS2 provides more than one terminal; the UCRT64 name matters here.
3. Update its installed packages by typing pacman -Syu and pressing Enter. If it asks you to close the terminal, close it, reopen UCRT64 and run the update again until it finishes normally.
4. In that same UCRT64 terminal, install the toolchain with the command below. When a package-group selection is offered, Enter accepts the default group. Read the confirmation and type Y to proceed when appropriate.

Terminal command

```
pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
```

A **toolchain** is a set of cooperating build tools. This package group includes the C++ compiler we will call as g++. `pacman` is MSYS2's package-management command; run it in the MSYS2 terminal, not in an ordinary PowerShell window.

Add the compiler folder to PATH

PATH is a list of folders Windows searches when you type a command. Adding the compiler's bin folder lets a new PowerShell terminal find g++ without you typing its full location.

1. Open Windows search and look for "Edit environment variables for your account".
2. In the user variables, select Path and choose Edit.
3. Choose New and add C:\msys64\ucrt64\bin. Adjust the beginning if you installed MSYS2 elsewhere.
4. Keep the existing entries. Add this as a new entry rather than replacing the list.
5. Confirm the dialogs, close old terminals, and restart VS Code if it was already open.
6. Open a new PowerShell terminal and run g++ --version.

Terminal command

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

You should see a compiler name and version information. The exact version text varies. If Windows says the command is not recognised, first check whether g++.exe exists in the folder you added. Then check the spelling of the PATH entry and open a fresh terminal.

Create a practice folder and save a real .cpp file

In File Explorer, open Documents and create a folder named cpp-practice. Open that folder in VS Code using File, then Open Folder. Create a file named **tool_check.cpp** and paste the complete program below. Save it with Ctrl+S.

If using another plain-text editor, check that the name is not tool_check.cpp.txt. In File Explorer, turn on file-name extensions so you can see the real ending. A Word document is not a plain-text source file.

tool_check.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Compiler check passed.\n";
    std::cout << "I can build and run a C++ file.\n";
    return 0;
}
```

Output

```
Compiler check passed.
I can build and run a C++ file.
```

Build and run in Windows PowerShell

In VS Code choose Terminal, then New Terminal. Check that the terminal is PowerShell and its current folder is your cpp-practice folder. The command `dir` lists the files there; tool_check.cpp should appear.

Terminal command

```
g++ -std=c++20 -Wall -Wextra -Wpedantic tool_check.cpp -o tool_check.exe
```

Command part	Meaning
g++	Ask the C++ build tool to compile and link
-std=c++20	Use the C++20 language rules
-Wall -Wextra -Wpedantic	Request useful compiler warnings
tool_check.cpp	Read this source file from the current folder
-o tool_check.exe	Give the output executable this filename

After a successful build, run the following as a **separate command**. PowerShell needs the leading dot and backslash to run a file from the current folder.

Terminal command

```
.\tool_check.exe
```

Compare the two output lines with the displayed example. Now change "Compiler check passed." to "My setup is ready." Save, build again, and run again. If the old line remains, you may have skipped saving or rebuilding, or run a file in another folder.

1. Edit
2. Save
3. Build successfully
4. Run and check

After changing a message, repeat the whole cycle. An old executable does not prove the new source built successfully.

Other choices: Code::Blocks, Visual Studio and older lab tools

[Code::Blocks](https://www.codeblocks.org/downloads/binaries/) is an IDE. On Windows, choose its installer that explicitly includes MinGW when you need a bundled compiler; an editor-only installer is different. Create a Console Application project, select C++, choose GNU GCC Compiler if that is your installed compiler, and replace the generated main.cpp content with tool_check.cpp's program.

Set the project compiler option for C++20 support, or add -std=c++20 in its other compiler options. Then use Build and Run and inspect the build log if it fails. Exact menu wording can differ between releases. Do not add a second main.cpp to the same project with another main function.

Visual Studio is another IDE, separate from VS Code. Its C++ support comes through the Desktop development with C++ workload. In a configured Developer Command Prompt, an equivalent command is cl /std:c++20 /EHsc tool_check.cpp. Our step-by-step examples use g++, so you do not need this alternative today.

Some college labs still demonstrate Turbo C++. Older DOS-era Turbo C++ versions do not support the modern standard used by these lessons. You may see old headers such as iostream.h or nonstandard functions such as getch. Do not add them to these examples to make them resemble an old lab sheet. Ask which compiler your lab assessment requires, and keep that requirement separate from this modern C++ practice setup.

Linux and macOS

On Ubuntu or Debian, install the build tools through the system package manager, then check g++ --version. These installation commands require an account allowed to install software.

Terminal command

```
sudo apt update
sudo apt install build-essential
g++ --version
```

On macOS, install Apple's command-line developer tools with xcode-select --install and follow the installation dialog. After installation, check clang++ --version.

Terminal command

```
xcode-select --install
clang++ --version
```

For either system, open a terminal in the folder containing tool_check.cpp. On Linux use the first build command below; on macOS use the second. Run only the command for your system, then run ./tool_check.

Terminal command

```
g++ -std=c++20 -Wall -Wextra -Wpedantic tool_check.cpp -o tool_check
clang++ -std=c++20 -Wall -Wextra -Wpedantic tool_check.cpp -o tool_check
./tool_check
```

Problem: a build failed, but an old program still runs

You first built a correct program. Later you removed a semicolon and the new build failed. An older executable can still remain in the folder. Running it does not prove the changed source is correct.

Solution: read the build result before running. Fix the first reported source error, save, build successfully, and only then run the executable. Do not type terminal commands inside the source file.

Symptom	First thing to check
g++ is not recognised	Compiler installed, correct bin folder in PATH, new terminal opened
Source file not found	Current folder, spelling and hidden .txt extension
iostream not found	Correct C++ compiler and a complete toolchain installation
Permission denied while replacing .exe	Close the previously running program and rebuild
No new output after an edit	Save, rebuild successfully, and run the intended executable

Practice questions

1. Why does installing the VS Code C/C++ extension not finish compiler setup?
2. Which folder should be added to PATH for the default UCRT64 installation?
3. Is pacman a C++ statement?
4. Which command builds tool_check.cpp in PowerShell, and which runs it?
5. Why might a failed build be followed by a program that still opens?
6. What changes in the run command between PowerShell and a Linux terminal?

Answer: Check the practice answers

1. The extension helps the editor understand and debug code. The compiler is another tool and must be installed separately for this setup.
2. C:\msys64\ucrt64\bin, if MSYS2 was installed in C:\msys64. It is the bin folder, not the g++.exe file itself, that belongs in PATH.
3. No. It is a package-management command run in the MSYS2 terminal.
4. Build with g++ -std=c++20 -Wall -Wextra -Wpedantic tool_check.cpp -o tool_check.exe. After success, run .\tool_check.exe.
5. The earlier executable may still exist. Its behaviour belongs to the last successful build, not necessarily to the currently edited source.
6. PowerShell uses .\tool_check.exe for this Windows build. A Linux terminal uses ./tool_check for the executable name shown in the Linux build command.

Interview questions

Answer: What is the difference between compiling and running?

Compiling checks and translates source code; a typical g++ build command also links it into an executable. Running starts that executable and carries out its instructions. A successful build alone does not prove that the program solves the intended task.

Answer: What does PATH do?

It tells the command environment which folders to search for tools. It helps locate the compiler command; it does not compile source files by itself.

Answer: What is an IDE?

An IDE combines tools for writing, building and debugging programs. Some installers include a compiler and some require a separate compiler installation. Check the actual package rather than assuming every IDE includes everything.

Answer: Why use compiler warnings?

Warnings point out code that may be mistaken even when a build can continue. Read and understand them. A warning-free build is useful, but tests and reasoning are still needed.

Quick revision

- Choose one working tool setup before trying several alternatives.
- Keep source files as plain text with a .cpp ending.
- PATH points to a folder containing a tool.
- Save, build successfully, then run.
- Keep terminal commands outside the source file.

03. Your first C++ program: understand every part

In the last chapter you built and ran a file. Now we will explain what its parts mean. Read slowly: recognising why a line exists is more useful than copying it without understanding it.

Start with the required result

A student club wants its program to show a welcome message and a room number. The required result is two readable lines. We do not need input, a loop or a class for this small job.

Save this complete program as first_cpp.cpp. In PowerShell, build with g++ -std=c++20 -Wall -Wextra -Wpedantic first_cpp.cpp -o first_cpp.exe, then run .\first_cpp.exe. The build-and-run pattern is the same as Chapter 2; only the filenames change.

first_cpp.cpp

```
#include <iostream>

int main()
{
    std::cout << "Welcome to the coding club.\n";
    std::cout << "Meet in room 18.\n";
    return 0;
}
```

Output

```
Welcome to the coding club.
Meet in room 18.
```

What is #include <iostream>?

The C++ standard library offers tools that many programs need. A **header** makes declarations for particular tools available to our source file. A declaration tells the compiler about a name and how it can be used.

`#include <iostream>` is an instruction used while preparing the source for compilation. Here it makes the standard stream facilities, including `std::cout`, available. **A stream is a way to send or receive a sequence of characters.** Our program sends characters to standard output, normally the terminal in these examples.

The # begins this instruction. The angle brackets surround the standard header name. Do not change them to braces, and do not add `.h` to `iostream` in this modern C++ example. An include line does not end with a semicolon.

This line does not itself display a message. It prepares access to facilities used by later code. We include the headers that our program needs, rather than treating one header as a magic line required by every possible program.

What is int main()?

A **function** is a named piece of program work. We will learn to write several functions later. For these ordinary console programs, `main` is the function the C++ execution environment calls to begin our program's main work.

`int` means that this function returns a whole-number status. It does not mean the message must contain a number. `main` is the required name of this entry function. The empty parentheses, `()`, mean this form takes no parameters. A parameter is information a function receives through its call; we do not need any here.

Use lowercase `main`. `Main` is a different name. Use `int main()`, not `void main()`. There are other allowed forms of main, but this is the simplest suitable form for our examples.

What do the braces mean?

The opening brace { begins the function's **body**, the instructions belonging to that function. The closing brace } ends it. Read the two as a pair.

Indented lines make that boundary easier for a person to see. Spaces at the beginning of a normal statement do not change what this example prints. Missing braces can stop the program from compiling; untidy indentation makes errors harder to notice.

1. #include <iostream>
2. int main() {
3. Output statements; return 0;
4. }

The include is outside main. The statements between the matching braces form its body.

The includes are outside main. The output statements and the return statement are inside it. We will keep this arrangement consistent while learning.

What is std::cout?

`std::cout` is the standard library's output stream object. At this stage, think of it as a ready-to-use destination for the information we want to print. We will explain objects more fully in the classes chapter.

`std` is a **namespace**: a named group that helps organise names. `::` selects a name from that group. Thus `std::cout` identifies cout from the standard-library namespace, rather than another name that might appear in a larger program.

Some tutorials write `using namespace std;` and then use only cout. That lets the compiler look for names such as cout in std when you omit the std:: prefix. It is not a required line in a C++ program. We write the `std::` prefix explicitly so you can see where a library name comes from.

What does << do here?

In an output expression such as `std::cout << "Welcome"`, `<<` sends the value on its right to the output stream on its left. It is often called the **insertion operator** in this context. Later you will see that the same symbol can mean something else when used with other kinds of values.

The double quotes mark a **string literal**: text written directly in the program. The quotation marks tell C++ where that text starts and ends; the marks themselves are not part of the output.

The characters `\n` inside the literal represent a newline. Without an explicit space or newline, separate output operations can appear directly beside one another.

Why is there a semicolon?

A **statement** is a unit of program work. Each output expression statement here ends with `;`. Think of it as the closing punctuation for that statement, rather than a mark that belongs at the end of every physical line.

The include directive does not use that punctuation. The line `int main()` introduces a function definition, so placing a semicolon immediately after it would break the intended definition. The closing brace of this main function also does not need one.

Two output statements run in the order shown. Writing two statements on separate lines is easier to read, although line breaks alone do not replace their semicolons.

What does return 0 mean?

`return` ends the current function and gives back its result. Returning zero from main reports successful completion to the environment that started the program. It does not print zero.

In C++, reaching the closing brace of main without an explicit return has the same effect as returning zero. We include `return 0;` in these early programs so the successful finish is visible. This special rule for main should not be copied as a general rule for every function that returns a value.

Follow the program without a computer

Step	What happens	Visible output so far
1	The execution environment calls main	Nothing yet
2	The first output statement runs	Welcome to the coding club. followed by a newline
3	The second output statement runs	The first line, then Meet in room 18.
4	main returns zero	Output stays the same; the program finishes

The compiler does its work before this run. Reading the include directive during preparation is not an extra runtime message or a line shown to the user.

Problem: make the notice clear

The club now needs three labelled lines: an event, a day and a place. Write the desired lines first, then use one output statement for each line.

club_notice.cpp

```
#include <iostream>

int main()
{
    std::cout << "Event: Beginner practice\n";
    std::cout << "Day: Saturday\n";
    std::cout << "Place: Computer lab\n";
    return 0;
}
```

Output

```
Event: Beginner practice
Day: Saturday
Place: Computer lab
```

No keyword needs to change to add another notice line. We add another output statement inside the existing braces. A complete source file should contain one main definition for the single program being built.

Common mistakes: read the first useful error

The following is an intentionally broken complete program. The output statement has no semicolon. Do not use it as a working model.

Intentional error for study - missing_semicolon.cpp

```
#include <iostream>

int main()
{
    std::cout << "Check the punctuation.\n"
    return 0;
}
```

A compiler may point at `return`, even though the missing punctuation belongs to the previous line. Exact error wording varies. Look at the reported location and the line immediately before it, add the missing semicolon, and rebuild.

- Spell iostream and std::cout exactly, including lowercase letters.

- Use ordinary straight quotes in source code, not decorative quotes pasted from a document.

- Keep output statements inside the function body.

- Build one example file at a time; combining files with separate main definitions causes another kind of build error.

- Check both punctuation and spelling before deleting large parts of your program.

Practice questions

1. Which part of std::cout selects a name from a namespace?

2. Does #include <iostream> print text by itself?

3. What does the empty pair of parentheses in int main() mean?

4. Does return 0 print a zero on screen?

5. Add "Bring a notebook." as the final line of club_notice.cpp.

6. Explain why a compiler pointing at return might still be reporting a problem on the preceding line.

Answer: Check the practice answers

1. The double colon, ::, selects a name from a namespace or another suitable scope. Here it selects cout from std.

2. No. It makes the relevant library declarations available. The output statements perform the visible printing during execution.

3. This form of main takes no parameters. We will study parameters when we write functions with inputs.

4. No. It reports successful completion to the starting environment. Printing requires an output operation.

5. Insert std::cout << "Bring a notebook.\n"; inside main after the place line and before return. Save, build and run again.

6. The parser may reach the next statement before it can tell that the previous statement was unfinished. Check the earlier line's semicolon, quotes and brackets too.

Interview questions

Answer: Is using namespace std required?

No. Writing names such as std::cout explicitly works without it. The std prefix identifies the standard-library namespace.

Answer: Why should main return int rather than void?

The standard forms used for these hosted C++ programs require main to have an int return type. Its return value provides a completion status. Use int main() for the simple no-argument form taught here.

Answer: Must we write return 0 at the end of main?

An explicit return is optional when control reaches the end of main: C++ treats that as returning zero. Writing it is still valid and makes the intended successful finish clear in these beginner examples.

Answer: What is the difference between a syntax error and a wrong result?

A syntax error breaks a writing rule and prevents a successful build. A program can also build successfully while showing the wrong room or computing the wrong answer. Building and checking behaviour are separate steps.

Quick revision

- iostream provides declarations for standard stream facilities.

- main has a body inside a matching pair of braces.

- std::cout sends output; double quotes enclose literal text.

- The output statements need semicolons; not every source line does.

- Returning zero from main signals success rather than printing a value.

04. Print clear output and write useful comments

A program's result should be easy for another person to read. In this chapter we will control spaces and new lines, print numbers, write notes beside code and recognise common writing mistakes.

Output does not invent spaces for us

`std::cout` prints the values we send to it. It does not automatically add a space between two separate messages. Look at this complete example before running it.

joined_messages.cpp

```cpp
#include <iostream>

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

Output

```
Practiceroom
```

The result is not two words because neither literal supplies a space. To show "Practice room", put a space at the end of the first literal, or print a single literal containing both words.

Print text and numbers together

Numbers written directly in code are **numeric literals**. Unlike literal text, they do not need quotation marks. We can chain several insertions to show a label and a number in the same statement.

library_board.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Desk number: " << 6 << '\n';
    std::cout << "Open seats: " << 14 << '\n';
    std::cout << "Study time: " << 1.5 << " hours\n";
    return 0;
}
```

Output

```
Desk number: 6
Open seats: 14
Study time: 1.5 hours
```

Read the first statement from left to right: send the label, send the number 6, then send a newline. `6` is a numeric value. `"6"` is text. They look the same when printed here, but their types and permitted operations differ. Types are the next chapter's topic.

Newlines, tabs, quotation marks and backslashes

An **escape sequence** is a backslash spelling that represents a particular character inside a character or string literal. For example, `\n` represents one newline character. It is not two ordinary printed characters.

Spelling inside a literal	Meaning	When it helps
\n	Newline	Begin the next output line
\t	Horizontal tab	Separate simple fields; alignment depends on the terminal
\"	Double quote	Include a quote inside a double-quoted literal
\\	One backslash	Show a Windows-style path or the backslash character

escape_notes.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Say \"Ready\" before starting.\n";
    std::cout << "Folder: C:\\cpp-practice\n";
    std::cout << "First line\nSecond line\n";
    return 0;
}
```

Output

```
Say "Ready" before starting.
Folder: C:\cpp-practice
First line
Second line
```

In source text, two backslashes produce one backslash in that output path. Copying a path straight into a string without considering escapes can change its meaning or produce a diagnostic. Tabs are useful for simple experiments, but they are not a reliable replacement for proper table formatting.

'\n', "\n" and std::endl

`'\n'` is a character literal containing a newline. `"\n"` is a string literal containing a newline. Both can add a newline when inserted into cout.

`std::endl` also writes a newline, and then **flushes** the stream. Flushing asks buffered output to be sent onward immediately. A buffer is temporary storage used while output is collected. For ordinary output lines here, `\n` is sufficient; we do not need to request a flush after every line.

You do not need to add a waiting command such as getch to every program. Running the executable from an already open terminal lets you read its output after it finishes.

A comment is a note for the reader

A **comment** explains something in the source. Its words are not executed as program instructions and are not printed just because they appear beside code. Comments help another student, a teammate or your future self understand a decision.

A **single-line comment** begins with `//` and continues to the end of that source line. A **block comment** begins with `/*` and ends at the next `*/`; it can cover several lines.

commented_notice.cpp

```
#include <iostream>

int main()
{
    // Keep the room visible so students can find the session.
    std::cout << "Room: 12\n";

    /* The time is printed on its own line
       to make the notice easier to read. */
    std::cout << "Start: 09:45\n";
    return 0;
}
```

Output

```
Room: 12
Start: 09:45
```

Notice that neither explanatory comment appears in the output. However, `std::cout << "// read this\n";` would print the slashes: inside a string literal they are text, not a comment opener.

Write comments that explain a reason

Compare "print the room" with "keep the room visible so students can find the session". The second note tells us why the line matters. Do not describe code as correct merely because a comment says so; comments can become outdated when the code changes.

Do not nest block comments. An inner-looking `/*` does not create another nested level; the next `*/` closes the comment. Use separate single-line comments when temporarily explaining several lines. Comments are also not a safe place to keep passwords or private information in shared code.

1. // a source note

2. Not printed

3. std::cout << text;

4. Text goes to output

A comment explains the source. Printing needs an output operation. Comment markers inside a quoted string are just text.

Writing rules: names, punctuation and layout

C++ is **case-sensitive**: changing a letter's case can change the name. `std::cout` and `std::Cout` do not identify the same name. Keywords such as int and return have required spellings.

Indent code inside braces and put one main action on each line while learning. Spaces between tokens often improve readability. Do not insert spaces inside a keyword or inside the << operator. Inside quoted text, spaces are part of the output, so changing them may change the result.

Mistake	Why it causes trouble	Repair
cout without std:: in these examples	No declaration introduced that unqualified name	Use std::cout
A missing closing quote	The text literal is not properly closed	Match both quotes
A missing semicolon	The expression statement is unfinished	Add ; at the statement's end
Expecting a comment to print	A comment is for source readers	Use an output statement for visible text
Expecting a tab to align every display	Tab positions depend on the output environment	Inspect the result; use appropriate formatting for real tables

Problem: a notice has everything on one line

The required result is two labelled lines: "Help desk: Room 7" and "Open: 11:00". The current fragment below does not supply a separator between the messages.

Code fragment - Code fragment

```
std::cout << "Help desk: Room 7";
std::cout << "Open: 11:00";
```

The repair is to add a newline at the end of each message. This fragment belongs inside main in a program that includes iostream; it is not a complete source file by itself.

Code fragment - Code fragment

```
std::cout << "Help desk: Room 7\n";
std::cout << "Open: 11:00\n";
```

The number of source lines was never the reason for the output layout. The characters sent to cout determine that layout.

Practice questions

1. Why did joined_messages.cpp print Practiceroom as one word?
2. Print "Batch: " followed by the numeric value 3 and a newline.
3. How do you include double quotes inside a double-quoted string literal?
4. Will the words in // Check the room be printed automatically?
5. What extra action does std::endl perform compared with inserting '\n'?
6. Explain why a comment containing "capacity is ten" must be reviewed after the program's limit changes.

Answer: Check the practice answers

1. Neither output operation supplied a separating space. A source line break does not automatically become an output line break.
2. A suitable statement is std::cout << "Batch: " << 3 << '\n'; inside a complete program.
3. Use the escape sequence backslash-double-quote, as in the complete escape_notes.cpp example.
4. No. The comment is a source note. To show those words, put them in a string passed to an output operation.
5. It requests a stream flush after writing the newline. The ordinary line output here does not require that extra request each time.
6. A comment can become misleading. Update the explanation to match the actual rule, or explain the rule by a meaningful named constant later rather than duplicating a changing number in several places.

Interview questions

Answer: What is the purpose of a comment?

It helps a source reader understand code, especially a reason or assumption that is not obvious. It is not a substitute for clear code, and the compiler does not check that the explanation remains true.

Answer: Can we nest block comments in C++?

No. A block comment ends at the next closing marker. Use non-nested comments or separate single-line comments instead.

Answer: Is an output line the same thing as a source line?

No. A source line is how we arrange code in the file. Output lines depend on the characters written during execution, including explicit newline characters.

Answer: Does code that compiles always have correct output?

No. It might print a wrong label, omit a space, or use the wrong data. Compare actual output with the stated requirement as well as checking the build.

Quick revision

- Output needs explicit separators such as spaces and newlines.
- Chained insertions can print text, numbers and characters together.
- Escape sequences represent characters inside literals.
- Comments explain source code; they do not automatically appear in output.
- Good layout helps readers, while the language's punctuation and naming rules still apply.

05. Variables, constants and data types

A message written directly in quotes is useful, but most programs must also keep values that can change. We will give values meaningful names, choose suitable types, and protect values that should remain fixed.

First see the common choices together

A **data type** tells C++ what kind of value a variable can hold and which operations make sense for it. Start with the common choices below. Unlike C's printf format strings, cout does not need a separate %d or %f placeholder for these values.

Need	Type	Example declaration	Show the value
Whole-number count	int	int seats{24};	std::cout << seats;
Larger whole number	long long	long long visits{5000000000LL};	std::cout << visits;
Measurement with a fractional part	double	double distance{2.5};	std::cout << distance;
A floating-point value with float type	float	float height{1.75f};	std::cout << height;
A simple character	char	char section{'B'};	std::cout << section;
Yes/no state	bool	bool ready{true};	std::cout << std::boolalpha << ready;
Text	std::string	std::string name{"Isha"};	std::cout << name;

std::string is a library type, declared through the string header. It is not a built-in keyword like int. A char holds one character-sized code unit; it is suitable for the simple English characters used here, not automatically every written character in every language.

A variable is a named place for a value

Think of a desk card labelled "available seats". The label stays the same while the written number changes. A variable lets us use a name such as availableSeats rather than repeat a number in every relevant statement.

Code fragment - Code fragment

```
int availableSeats{24};
```

Read this as: create an int named availableSeats and start it with 24. The type is int, the name is availableSeats, and 24 is its **initial value**. Giving a variable its first value when it is created is called **initialization**.

1. int

2. availableSeats

3. {24}

4. ;

Type: int. Name: availableSeats. Starting value: 24. The semicolon ends the declaration.

The braces in this declaration hold the starting value. They do not define a function body here. Punctuation gets its meaning from the surrounding construct.

A complete program with different kinds of values

student_values.cpp

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name{"Isha"};
    int availableSeats{24};
    long long yearlyVisits{5000000000LL};
    double walkDistance{2.5};
    float deskHeight{1.75f};
    char section{'B'};
    bool ready{true};

    std::cout << "Student: " << name << '\n';
    std::cout << "Seats: " << availableSeats << '\n';
    std::cout << "Visits: " << yearlyVisits << '\n';
    std::cout << "Walk in km: " << walkDistance << '\n';
    std::cout << "Desk height in m: " << deskHeight << '\n';
    std::cout << "Section: " << section << '\n';
    std::cout << "Ready: " << std::boolalpha << ready << '\n';
    return 0;
}
```

Output

```
Student: Isha
Seats: 24
Visits: 5000000000
Walk in km: 2.5
Desk height in m: 1.75
Section: B
Ready: true
```

`LL` gives the large integer literal a long long type. `f` gives 1.75f a float type; an unsuffixed literal such as 1.75 has double type. Use decimal points, not commas, inside these numeric literals.

`std::boolalpha` is an output setting that makes bool values appear as true or false. Without it, ordinary cout output shows false as 0 and true as 1. This setting affects later Boolean output on that stream until changed back with std::noboolalpha.

Choose meaningful, legal names

Use names such as availableSeats, studentName and isReady. Letters and digits can appear in ordinary beginner names, but a name must not start with a digit. An underscore can also appear, although starting your own names with an ordinary letter avoids C++'s reserved-identifier rules.

Do not use keywords such as int, return or class as variable names. Do not put spaces or a hyphen inside a name. availableSeats and availableseats are different names because case matters.

Proposed name	Suitable?	Reason
studentName	Yes	A readable name starting with a letter
room2	Yes	A digit is allowed after the first character
2room	No	Starts with a digit
student name	No	Contains a space
class	No	A reserved keyword
seat_count	Yes	An underscore separates the words

Initialization and later assignment are different steps

stock_update.cpp

```cpp
#include <iostream>

int main()
{
    int availableSeats{24};
    int earlierCount{availableSeats};
    availableSeats = 19;
    std::cout << "Earlier: " << earlierCount << '\n';
    std::cout << "Now: " << availableSeats << '\n';
    return 0;
}
```

Output

```
Earlier: 24
Now: 19
```

The first line creates availableSeats with a starting value. The later `availableSeats = 19;` assigns a new value to that existing variable. For these int variables, earlierCount receives a copy of the value 24. It

is not a live link that changes when availableSeats changes. References, which can act as another name for an existing object, are a later topic.

Keep one variable declaration per line while learning. It makes the type, name and starting value easier to check.

Start values safely and notice narrowing

Use `int count{};` when zero is the right starting value, or provide another explicit starting value. An ordinary local `int count;` does not provide a value you should read. Do not assume it starts at zero just because one run appeared to show zero.

These forms all create an int with value 8: `int seats = 8;`, `int seats(8);` and `int seats{8};`. Brace initialization is useful because it rejects certain **narrowing conversions**: conversions that could lose information in ways this form does not permit.

For example, the following complete program is intentionally invalid. A whole-number int cannot preserve the fractional part of 4.5, and this brace initializer is rejected. We will study conversion rules in Chapter 7.

Intentional error for study - narrowing_value.cpp

```cpp
#include <iostream>

int main()
{
    int hours{4.5};
    std::cout << hours << '\n';
    return 0;
}
```

The repair depends on the meaning: use double if 4.5 hours is allowed, or supply a genuine whole-number value if the count must be integral. Do not hide a warning by changing punctuation before deciding what the value should mean.

Integer types, signed and unsigned

The standard signed integer families are signed char, short, int, long and long long. Each has an unsigned counterpart. For example, unsigned int represents a nonnegative range, while int also represents negative values.

Choose int for small ordinary counts in these lessons. Use a suitable larger type when the allowed input range requires it. **Unsigned does not mean a subtraction can never go wrong.** Unsigned arithmetic wraps within its range, so subtracting one from zero does not give a useful negative count.

Family	Typical use or important rule
signed char / unsigned char	Small integer or byte-oriented work; streaming these directly usually prints a character
short / unsigned short	A smaller-range integer family; check the actual limits
int / unsigned int	General whole-number work within documented limits

Family	Typical use or important rule
long / unsigned long	Width can differ between platforms; do not assume it is 64 bits
long long / unsigned long long	At least 64 bits of value width for the unsigned type, and the corresponding signed range

Signed integer overflow is not a reliable wrap-around operation in C++. Choose a type that can hold the task's possible results. We will practise checking arithmetic before it overflows when solving larger problems.

Floating-point, character and other type families

`float`, `double` and `long double` represent floating-point numbers. They have limited precision. Many decimal fractions cannot be stored exactly, so do not assume every decimal calculation is exact. Double is a useful default for the simple measurements here. We will discuss conversion and rounding when doing calculations.

`char`, signed char and unsigned char are distinct types. C++ also has wchar_t, char8_t, char16_t and char32_t for particular character and text-processing needs. The last three help represent UTF code units; one code unit is not always a complete displayed character. These are names to recognise now, not a requirement to learn Unicode processing before writing your first program.

`bool` has the two values false and true. `void` describes the absence of a value in appropriate contexts, such as a function returning no result; it is not a type for a normal variable holding data. `std::nullptr_t` is the type of the null-pointer literal nullptr. We will use nullptr in the pointers chapter.

Arrays, pointers and references build on types, while classes and enumerations let us describe further kinds of values. Their detailed rules belong to their own later chapters. This overview is not a claim that int, double, char and bool are the only C++ types.

sizeof: check storage, do not guess

`sizeof` reports a size in **C++ bytes**. A byte is a unit of storage; it has at least eight bits. sizeof(char) is always 1, but the sizes of types such as int and double depend on the compiler and the system it targets. A size table showing 4 for int is a common example, not a universal rule.

type_sizes.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "char bytes: " << sizeof(char) << '\n';
    std::cout << "int bytes: " << sizeof(int) << '\n';
    std::cout << "double bytes: " << sizeof(double) << '\n';
    return 0;
}
```

Output

```
char bytes: 1
int bytes: 4
```

```
double bytes: 8
```

The output shown is from a setup with a four-byte int and an eight-byte double. Your compiler and target system may give different sizes. `sizeof` itself has an unsigned result type called std::size_t; cout can print that result directly. Do not replace observations from your compiler with memorised sizes from another machine.

const: protect a value that should stay fixed

Use `const` for a value that your program should not change after its initialization. For example, once a room's capacity has been chosen for this run, changing the current occupancy should not also change the capacity.

fixed_values.cpp

```cpp
#include <iostream>

int main()
{
    const int roomCapacity{28};
    int peoplePresent{11};
    const int openingCount{peoplePresent};
    peoplePresent = 16;
    constexpr int daysPerWeek{7};

    std::cout << "Capacity: " << roomCapacity << '\n';
    std::cout << "Opening count: " << openingCount << '\n';
    std::cout << "Present now: " << peoplePresent << '\n';
    std::cout << "Days per week: " << daysPerWeek << '\n';
    return 0;
}
```

Output

```
Capacity: 28
Opening count: 11
Present now: 16
Days per week: 7
```

roomCapacity cannot be assigned a new value. peoplePresent can change. openingCount is a fixed copy of the value obtained at that point; its const declaration does not make peoplePresent itself constant.

Here is an intentionally invalid attempt to change a constant. The compiler should reject it. A const declaration protects this simple value even when the later assignment looks like an ordinary variable update.

Intentional error for study - change_constant.cpp

```cpp
int main()
{
    const int roomCapacity{28};
    roomCapacity = 30;
    return 0;
}
```

constexpr and const are related, but not identical

A **constant expression** is an expression that satisfies the language's rules for working out its value while building the program, without depending on values first obtained during its run. For a variable such as daysPerWeek, constexpr requires an appropriate constant initializer and makes the object const.

A plain const value need not come from a constant expression. In fixed_values.cpp, openingCount copies the current value of the mutable peoplePresent variable. That is allowed for const; trying to declare it constexpr from that mutable variable would not be allowed.

Use const when a value must stay fixed after it is obtained. Use constexpr for a simple constant you intend to make available under compile-time rules. Do not replace every const with constexpr without checking where its initial value comes from. Named typed constants are preferable to introducing a macro just to name a number in these beginner examples.

auto: let the initializer tell the compiler the type

`auto` asks the compiler to deduce a type from the starting expression. It does not make a variable free to change its type later. Use it when the resulting type is clear; explicit int and double declarations are still useful while learning.

deduced_values.cpp

```cpp
#include <iostream>

int main()
{
    auto notebooks = 4;
    auto travelDistance = 12.5;
    notebooks = 6;
    std::cout << "Notebooks: " << notebooks << '\n';
    std::cout << "Distance: " << travelDistance << '\n';
    return 0;
}
```

Output

```
Notebooks: 6
Distance: 12.5
```

notebooks has type int because 4 is an int literal. travelDistance has type double because 12.5 is a double literal. In this basic variable form, auto needs an initializer. `auto notebooks;` does not give the compiler enough information.

Problem: choose values for a lab sign-in record

The record needs a student's name, a small whole-number desk ID, a section letter and a yes/no attendance state. Choose std::string, int, char and bool respectively. Include string as well as iostream when using std::string.

Use student_values.cpp as a complete model, replacing the relevant field names and values. If the desk ID can contain leading zeros or letters, reconsider int: an identifier such as "004A" is text, not a number to calculate with. **Choose a type from the meaning of the data, not only from how it looks.**

Common mistakes and tips

- Reading an ordinary local variable before giving it a starting value. Initialize it deliberately.
- Using 'Isha' as if it were ordinary text. Use double quotes with std::string; a simple char example uses one character such as 'B'.
- Assuming every int has four bytes. Check the implementation when size or range matters.
- Treating a floating-point measurement as an exact decimal money system. Plan representation and rounding for the real task.
- Trying to assign to a const value. Decide what should change and what should remain fixed.
- Assuming auto allows a variable's type to change. The compiler deduces one type for this declaration.

Practice questions

1. Choose a type for a small seat count, a section letter, an attendance flag and a full name.
2. After int first{8}; int second{first}; first = 3;, what are the two values?
3. Explain the difference between int count{}; and an ordinary local int count;.
4. Why is int length{2.75}; rejected, and what repair could fit a measured length?
5. Can a later assignment change roomCapacity in fixed_values.cpp?
6. What types are deduced for auto a = 9; and auto b = 9.0;? Must sizeof(int) be 4?

Answer: Check the practice answers

1. Suitable choices for the stated data are int, char, bool and std::string. The full-name field requires the string header. Reconsider these choices if the task's input rules change.
2. first becomes 3. second stays 8 because it received a copied int value, not a live link to first.
3. The empty braces initialize count to zero. An ordinary local int declared without an initializer must not be read as though it had a valid starting value.
4. The brace initialization would narrow a floating-point value to int. A declaration such as double length{2.75}; fits a fractional measurement, if that is the task's meaning.
5. No. It is a const int. Keep capacity fixed and update a separate variable such as peoplePresent.
6. a has type int and b has type double. Their types are fixed for these variables. sizeof(int) is implementation-dependent, so it need not be 4.

Interview questions

Answer: What is the difference between initialization and assignment?

Initialization gives an object its starting state when it is created. Assignment changes the value of an existing assignable object. int seats{8}; initializes; seats = 5; later assigns.

Answer: What is the difference between const and constexpr for these variables?

Both protect the declared simple object from later assignment. constexpr also requires the initializer to satisfy the relevant constant-expression rules. A const value can instead be obtained during execution.

Answer: Is std::string a built-in C++ data type?

It is a standard-library class type, not a fundamental type or keyword. Include the string header and refer to it as std::string. We can use it for simple text before learning how classes are implemented.

Answer: Does auto remove type checking?

No. The compiler deduces a type and still checks later operations against that type. It does not mean that the variable can hold any unrelated kind of value at any time.

Quick revision

- Choose types from the meaning and range of the data.
- Give variables starting values and use readable names.
- Copying an int value does not connect two variables.
- Sizes and ranges must not be guessed from one machine's output.
- const protects a fixed value; constexpr adds compile-time requirements.
- auto deduces a type from an initializer; it does not remove the type.

06. Read input and check it before using it

Until now, our programs supplied their own values. Input lets the person running a program supply information. We will read numbers and names, understand failed reads, and check a whole line when accepting only part of it would be misleading.

Input, a prompt and the value we read

Input is information supplied to a program. In these console examples, it normally comes from typing in the terminal. A **prompt** is a message explaining what to enter. A prompt does not read anything by itself.

`std::cin` is the standard input stream. Like cout, it is available through iostream. With cin, `>>` tries to extract a value of the destination variable's type. Read `std::cin >> seats` as "try to read an integer into seats" when seats has type int.

Input can fail. The user may type a word, provide a number too large for int, or end the input. **Check the read before using the value.** Giving seats a starting value is sensible, but that starting value does not prove a later read succeeded.

A first checked number

This program asks for a seat count from 1 through 60. It uses a small if statement to stop after a failed check. For now, read `if (condition)` as "do the following block when the condition is true". Chapter 8 explains decisions in full.

read_seat_count.cpp

```cpp
#include <iostream>

int main()
{
    int seats{};
    std::cout << "Enter a seat count from 1 to 60:\n";
    if (!(std::cin >> seats)) {
        std::cout << "Please enter a whole number.\n";
        return 1;
    }
    if (seats < 1 || seats > 60) {
        std::cout << "The count must be from 1 to 60.\n";
        return 1;
    }
    std::cout << "Seats requested: " << seats << '\n';
    return 0;
}
```

Walkthrough

```
18
```

Output

```
Enter a seat count from 1 to 60:
Seats requested: 18
```

The input box shows what to type, followed by Enter. The output box contains only what the program prints. Your terminal may also display the characters you type; that keyboard display is not another cout statement.

`!(std::cin >> seats)` tests whether the attempted read failed. The `!` means "not". The range check uses two vertical bars, `||`, meaning "or": a count below 1 or above 60 is rejected. Returning 1 stops this example with a failure status; returning 0 reports successful completion.

Reading a number is not the same as checking an entire line

Formatted input, such as cin >> seats, usually skips leading spaces and then reads a value. It can stop after a valid numeric prefix. With input `18x`, an int read can obtain 18 and leave x unread. Similarly, reading an int from `3.5` can obtain 3 and leave the fractional text behind.

The first example checks a numeric extraction and its range. **It is not a strict whole-line form validator.** Later in this chapter we will reject leftover non-space characters too. This difference matters whenever the task says "one whole number and nothing else".

Input situation	What must we check?
A word where an integer is expected	Did the extraction succeed?
An integer outside the destination type's range	Did the extraction fail due to the conversion?
A valid integer outside the task's allowed range	Does the value obey the task's limits?
A valid prefix followed by extra text	Does the rest of the line contain anything except whitespace?
No more input	Can we stop instead of asking forever?

1. Read the input
2. Did conversion succeed?
3. Is the format complete?
4. Is the value allowed?

A value such as 61 may be a valid integer but outside a 1-60 rule. A numeric prefix such as 18 in 18x does not validate the whole line.

Read a name containing spaces with getline

Reading a string with `cin >> name` stops at whitespace. **Whitespace** includes separators such as spaces, tabs and newlines. To read a full name such as "Nila Joseph", use `std::getline(std::cin, name)`.

getline reads up to the newline and removes that delimiter from the input. It does not store the newline in name. A blank line can be read successfully as an empty string, so successful reading and a nonempty name are different checks.

student_checkin.cpp

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

int main()
{
    std::string name;
    int year{};
    std::cout << "Enter your full name:\n";
    if (!std::getline(std::cin, name) || name.empty()) {
        std::cout << "A name is required.\n";
        return 1;
    }
    std::cout << "Enter your study year, 1 to 6:\n";
    if (!(std::cin >> year) || year < 1 || year > 6) {
        std::cout << "That study year is not accepted.\n";
        return 1;
    }
    std::cout << "Student: " << name << '\n';
    std::cout << "Year: " << year << '\n';
    return 0;
}
```

Walkthrough

```
Nila Joseph
2
```

Output

```
Enter your full name:
Enter your study year, 1 to 6:
Student: Nila Joseph
Year: 2
```

`name.empty()` asks whether the string has no characters. This small example preserves spaces in the name; it does not trim them or claim to validate every real person's name. The year uses the earlier numeric-extraction approach, with the same prefix-reading limitation.

Mixing >> and getline: the leftover newline

Suppose you reverse those operations: read a year using >> and then a name using getline. The numeric read usually leaves the line's newline behind. getline may immediately consume it and return an empty string instead of waiting for a name.

If the policy is to discard the rest of the numeric line, explicitly ignore through its newline before reading the next line. Include limits for the maximum count used here.

Code fragment - Code fragment

```
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
std::getline(std::cin, name);
```

This is a repair fragment for a successful prior numeric read, not a complete program. The long expression supplies the stream's largest supported count rather than an arbitrary limit such as 100 characters. `ignore` consumes characters, including the newline if found. It does not validate discarded text.

Another common spelling is getline(cin >> ws, name). `std::ws` skips all leading whitespace, not only one newline. It can remove intentional leading spaces and skip blank lines. Choose it only when that matches the input rules. For line-based forms, consistently using getline is often clearer.

Failure flags: clear is not the same as discard

A stream keeps state flags describing events such as a failed conversion or reaching the end. While its failure state remains set, a later numeric extraction will not simply start afresh.

`clear()` resets stream state flags. It does not remove the bad characters. `ignore()` discards characters. For a recoverable bad numeric line, clear the flags first, then discard through the newline. **End-of-input is different:** stop cleanly instead of clearing it and retrying forever.

The next complete program permits at most one retry. It is intentionally about recovery, not the seat-range policy. Chapter 10 will use a loop to repeat input checks when appropriate.

retry_count_once.cpp

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

int main()
{
    int count{};
    std::cout << "Enter a count:\n";
    if (!(std::cin >> count)) {
        if (std::cin.eof() || std::cin.bad()) {
            std::cout << "Input ended or is unavailable.\n";
            return 1;
        }
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Try once more:\n";
        if (!(std::cin >> count)) {
            std::cout << "No valid count was read.\n";
            return 1;
        }
    }
    std::cout << "Count read: " << count << '\n';
    return 0;
}
```

Walkthrough

```
wrong
6
```

Output

```
Enter a count:
Try once more:
Count read: 6
```

`eof()` reports the end-of-input flag. `bad()` reports a serious stream error. A stream can encounter more than one state at once. We do not need to memorise every flag today, but we must not build a retry rule that ignores the possibility of input ending.

Problem: accept exactly one valid seat count per line

A booking form must accept a whole number from 1 through 60. It should reject "18x", "18 20", an empty line and numbers outside the range. Leading and trailing spaces around a valid number are allowed.

First read the whole line. Then use `std::istringstream`, from sstream, as a small reader whose input is that saved text. This is a library tool for reading a string as input; you do not need to write a class yourself to use it.

full_line_count.cpp

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::cout << "Enter one seat count, 1 to 60:\n";
    std::string line;
    if (!std::getline(std::cin, line)) {
        std::cout << "No input line was read.\n";
        return 1;
    }
    std::istringstream reader{line};
    int seats{};
    char extra{};
    if (!(reader >> seats) || (reader >> extra)) {
        std::cout << "Enter one whole number only.\n";
        return 1;
    }
    if (seats < 1 || seats > 60) {
        std::cout << "The count must be from 1 to 60.\n";
        return 1;
    }
    std::cout << "Accepted seats: " << seats << '\n';
    return 0;
}
```

Walkthrough

```
24
```

Output

```
Enter one seat count, 1 to 60:
Accepted seats: 24
```

The first extraction must succeed. The attempted read into extra then skips remaining whitespace: if another character exists, the line contains unwanted text. If no character remains, that local reader's extraction fails, which is the expected result of this particular check. It does not damage std::cin.

The logical OR operator stops evaluating once it knows the condition is true. If the number read fails, the extra-character check is not needed. We will practise this behaviour in Chapter 7.

This reads a whole line into memory. It is suitable for these small local exercises, not an unbounded-input policy for a public service. Real interfaces also need appropriate input-size limits.

Common mistakes and tips

- Reversing << and >>. cout sends output; cin extracts input in these examples.

- Checking only the range after a failed read. First establish that a value was read.

- Calling clear without removing the offending input. The next read can fail on the same characters.

- Using ws without considering whether blank lines or leading spaces matter.

- Promising strict validation after checking only a numeric prefix.

- Retrying forever after the input has ended. Every retry design needs a stopping rule.

Practice questions

1. What is the difference between a prompt and a read operation?

2. Why can cin >> name fail to read a complete name containing spaces?

3. Can reading int from "24x" successfully obtain 24? Does that validate the entire line?

4. Why are clear and ignore often used together after a recoverable failed numeric read?

5. Test full_line_count.cpp with "0", "60", "61", "18x" and an empty line. Predict each result first.

6. What should a retrying program do when no more input is available?

Answer: Check the practice answers

1. A prompt tells the user what to enter. A read operation actually attempts to receive the input. Printing a prompt does not change the variable by itself.

2. Extraction into a string stops at whitespace. getline is suitable when the task needs the rest of a line, including internal spaces.

3. Yes, numeric extraction can accept the valid prefix. No, the remaining x must also be checked when the whole line is required to contain only one integer.

4. clear resets the failure state; ignore removes unwanted characters. Doing only one of these does not necessarily prepare the next intended read.

5. Zero and 61 are outside the allowed range. Sixty is accepted. 18x is rejected as extra non-space text. An empty line fails to provide a whole number. A number surrounded only by spaces is accepted if it is in range.

6. Stop or return a clear cancellation/failure result. Do not keep clearing the stream and asking for input that cannot arrive.

Interview questions

Answer: How are cin extraction and getline different?

Formatted extraction interprets a value and normally skips leading whitespace. getline reads through a line delimiter, preserves the line's other characters and removes the delimiter. Choose based on the input format.

Answer: Why does getline sometimes return an empty string after a number read?

The earlier numeric extraction can leave a newline. getline then sees that newline immediately. Use a deliberate transition policy, or consistently read whole lines and parse them.

Answer: Is stream success enough to prove a value is acceptable?

No. A value can be correctly read but outside the task's permitted range. A numeric extraction can also accept only a prefix of the line. Conversion, range and full-format checks are separate concerns.

Answer: Why should an input loop test end-of-input?

Without a way to stop, a recovery loop may repeatedly reset and retry even though no more data exists. A robust program distinguishes a replaceable bad line from input ending or becoming unavailable.

Quick revision

- Prompt, read, then check before using a value.

- Use getline when the line, including spaces, matters.

- Clear flags and discard bad input for different reasons.

- Validate the whole line when a valid prefix is not enough.

- Every retry strategy needs to handle the end of input.

07. Operators and conversions: calculate without surprises

An operator asks C++ to do a particular job with values: add them, compare them or combine conditions. We will group the common operators first, then explain how the value types affect the result.

The operator families at a glance

An **expression** is code that computes a value or performs an operation. In seats + visitors, + is the operator and the two named values are its **operands**. These words describe the parts; you do not need to memorise a complicated definition.

Family	Job	Examples
Arithmetic	Calculate	+, -, *, /, %
Assignment	Store or update a value	=, +=, -=, *=, /=, %=
Increment and decrement	Change by one	++, --
Comparison	Ask how two values compare	==, !=, <, <=, >, >=
Logical	Combine or reverse true/false conditions	AND, OR, NOT
Conditional	Choose one of two expression results	condition ? first : second
Bitwise	Work with individual integer bits	AND, OR, XOR, complement and shifts
Other operators used elsewhere	Sizes, access and other specialised work	sizeof here; pointer and member access in later chapters

Use the topic navigation to jump to each family. The same spelling can have different meanings in different contexts: << writes to cout, but can also shift an integer's bits. We explain the meaning used by each example rather than treating the symbol alone as the whole rule.

Arithmetic: addition, subtraction and multiplication

Use + to add, - to subtract and * to multiply numeric values. A minus can also appear before one value, as in -temperature, to negate it. Parentheses can group a calculation.

arithmetic_values.cpp

```
#include <iostream>

int main()
{
    int sheets{17};
    int sheetsPerPack{5};
    std::cout << "After adding 3: " << sheets + 3 << '\n';
    std::cout << "After using 4: " << sheets - 4 << '\n';
    std::cout << "Three sets: " << sheets * 3 << '\n';
```

```
    std::cout << "Full packs: " << sheets / sheetsPerPack << '\n';
    std::cout << "Loose sheets: " << sheets % sheetsPerPack << '\n';
    return 0;
}
```

Output

```
After adding 3: 20
After using 4: 13
Three sets: 51
Full packs: 3
Loose sheets: 2
```

These expressions compute results; they do not change sheets. Assign a result back when the program needs to keep the new value.

Division and remainder depend on the types

When both operands are integers, / performs integer division. The fractional part is discarded toward zero: 17 / 5 is 3, and -17 / 5 is -3. % gives the integer remainder; 17 % 5 is 2 and -17 % 5 is -2. Do not assume the result is always nonnegative.

Do not divide or take a remainder by zero. Also ensure the division result is representable: the smallest signed integer divided by -1 is a special overflowing case. The small positive values in our example avoid both issues.

To get a fractional result, make an operand floating-point **before dividing**. Converting the integer result afterward cannot restore a fraction already discarded.

conversion_examples.cpp

```
#include <iostream>

int main()
{
    int sheets{17};
    int packs{5};
    double afterDivision = sheets / packs;
    double beforeDivision = static_cast<double>(sheets) / packs;
    double measurement{8.75};
    int wholePart = static_cast<int>(measurement);
    std::cout << "Integer division stored in double: " << afterDivision << '\n';
    std::cout << "Floating-point division: " << beforeDivision << '\n';
    std::cout << "Whole part: " << wholePart << '\n';
    return 0;
}
```

Output

```
Integer division stored in double: 3
Floating-point division: 3.4
Whole part: 8
```

Automatic conversion and static_cast

A **conversion** produces a value of another type. Some conversions happen automatically, such as converting the small int value 17 to double during a mixed calculation. This does not turn the original int variable into a double variable.

`static_cast<double>(sheets)` explicitly requests a double value from sheets. Read the angle-bracket part as the requested type and the parentheses as the expression being converted. For these numbers, static_cast makes the intended arithmetic clear.

Converting a finite floating-point value to an integer discards its fractional part toward zero; it does not round to the nearest integer. The resulting integer must be representable in the destination type. A cast is **not a range check** and does not make an out-of-range conversion safe. Large integers may also lose precision when converted to floating point.

Brace initialization rejects certain narrowing conversions, as Chapter 5 showed. A C-style cast such as (int)measurement exists, but we prefer the named cast for these explicit numeric conversions. Other C++ casts serve different purposes and will be explained where their types and uses become relevant.

1. 17 / 5 gives int 3
2. Then convert: double 3
3. Convert 17 first
4. 17.0 / 5 gives 3.4

The destination variable does not change how an earlier division was performed. A lost fractional part cannot be recovered by casting afterward.

Assignment and compound assignment

= stores a value in an assignable variable. It does not ask whether two values are equal. `seats += 2` adds two and stores the result back; similarly -=, *=, /= and %= combine an operation with an update.

Code fragment - Code fragment

```
int seats{12};
seats += 3;  // now 15
seats -= 2;  // now 13
```

For these ordinary int variables, seats += 3 has the same value effect as seats = seats + 3. This does not justify repeating a complex expression with side effects on both sides in more advanced code.

Assignments can convert a result to the variable's existing type. For example, an int variable does not gain fractional storage because you assigned a floating-point result to it. Choose the destination type intentionally and avoid narrowing a value by accident.

Increment and decrement: one step at a time

++count increases count and yields the updated value. count++ increases count but yields the previous value. The -- forms similarly decrease by one. If you only need a separate statement that increments an ordinary int, both count++ and ++count have the same final effect on count.

change_counters.cpp

```
#include <iostream>

int main()
{
    int pending{8};
    int oldValue{pending++};
    int newValue{++pending};
    std::cout << "Value kept by postfix: " << oldValue << '\n';
    std::cout << "Value kept by prefix: " << newValue << '\n';
    std::cout << "Pending now: " << pending << '\n';
    return 0;
}
```

Output

```
Value kept by postfix: 8
Value kept by prefix: 10
Pending now: 10
```

After the first update pending is 9. The second update changes it to 10 before giving that value to newValue. Keep updates in separate statements while learning. Complicated expressions that modify the same value several times can be unclear or even have undefined behaviour; they are not useful memory puzzles to copy into real programs.

Comparison: questions with Boolean answers

Use == for equality and != for inequality. The operators <, <=, > and >= compare order. Each of these built-in comparisons produces bool: true or false.

Code fragment - Code fragment

```
int seats{12};
bool enough = seats >= 10;
bool exactlyTen = seats == 10;
```

enough is true and exactlyTen is false. Do not write 1 <= year <= 6 to express a range. C++ treats the first comparison's Boolean result as the left value of the next comparison. Write year >= 1 && year <= 6 instead.

Avoid accidental assignment inside a condition. An expression such as count = 5 changes count; count == 5 compares it. Good names, parentheses and compiler warnings help reveal the difference.

Logical AND, OR and NOT

&& means logical AND: both conditions must be true. || means logical OR: at least one must be true. ! means NOT: reverse a Boolean condition. Parentheses can make combined rules easier to read.

A	B	A AND B	A OR B	NOT A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

These built-in logical operations **short-circuit**. AND skips its right side when its left side is false. OR skips its right side when its left side is true. NOT evaluates just its one operand.

For a small nonnegative total, count > 0 && total / count >= 2 avoids evaluating the division when count is zero. The guard must appear first. A single & is a different, bitwise operation and is not a replacement for this safety rule.

Conditional operator: choose one expression result

The spelling is `condition ? valueWhenTrue : valueWhenFalse`. Only the selected second or third expression is evaluated. This can express a small choice clearly; a long collection of nested choices is usually easier to read as if/else blocks.

Code fragment - Code fragment

```
bool hasPass{true};
int entryTokens = hasPass ? 1 : 2;
```

entryTokens becomes 1. This illustration is a classroom rule, not a real admission policy. Chapter 8 will practise choosing actions rather than only choosing one value.

Bitwise operators: work with bits, not whole conditions

A **bit** is a binary digit, 0 or 1. Integer values have a bit representation. Bitwise &, | and ^ combine corresponding bits using AND, OR and XOR. XOR sets a result bit when the two input bits differ. ~ complements bits; << and >> shift them.

bit_flags.cpp

```
#include <iostream>

int main()
{
    unsigned int flags{5u};
    std::cout << "Test low bit: " << (flags & 1u) << '\n';
    std::cout << "Set bit one: " << (flags | 2u) << '\n';
    std::cout << "Toggle low bit: " << (flags ^ 1u) << '\n';
    std::cout << "Shift left once: " << (flags << 1) << '\n';
    std::cout << "Shift right once: " << (flags >> 1) << '\n';
```

```
    return 0;
}
```

Output

```
Test low bit: 1
Set bit one: 7
Toggle low bit: 4
Shift left once: 10
Shift right once: 2
```

The low bits of 5 are 101 in binary. The u suffix gives these small literals unsigned type. We print computed results without assigning them back, so flags stays 5 for each line. A bitwise complement affects the type's full width, not only the three bits we drew.

Use unsigned values for these introductory bit examples. A shift count must be nonnegative and smaller than the width of the promoted left operand. Do not assume shifting is a universally safe way to multiply arbitrary signed values. Later pointer and container chapters will explain other operator uses in their proper context.

Bitwise operations also have compound assignment forms: &=, |=, ^=, <<= and >>=. They store the computed result back in the left variable. For example, flags |= 2u sets the selected bit and updates flags, whereas the earlier printed expression flags | 2u leaves flags unchanged. The same valid-type and shift-count requirements still apply.

Precedence: how an expression groups

Precedence tells us which operators bind more tightly. It does not, by itself, specify the order in which every operand is evaluated. Avoid side effects in complicated expressions instead of guessing a universal left-to-right execution rule.

Common group	Example	Meaning
Multiplication, division, remainder before addition/subtraction	2 + 3 * 4	14
Parentheses make a chosen group explicit	(2 + 3) * 4	20
Comparisons before logical AND/OR	year >= 1 && year <= 6	Two comparisons joined by AND
Logical AND before logical OR	a OR (b AND c)	Use parentheses when combining a real rule

This is a useful starting guide, not the entire language's precedence table. Write readable parentheses around bitwise tests and mixed logical groups. A statement can be valid C++ and still group differently from what you intended.

Problem: divide a reading task fairly

Seventeen pages are shared among five students. Each can receive three whole pages, with two pages left. If the task asks for the average pages per student, use a floating-point calculation and get 3.4 instead.

The arithmetic and conversion programs above solve both interpretations. **First decide what answer the task wants.** A quotient, a remainder and an average are different outputs; changing the variable name to average does not change integer division.

Common mistakes and tips

- Using = when you meant ==.

- Casting the result after integer division and expecting the missing fraction to reappear.

- Using % with a double; the built-in remainder operator requires integral operands.

- Treating && and & as interchangeable.

- Assuming every cast checks that its result fits.

- Relying on undefined signed overflow or on an invalid shift count.

- Memorising precedence puzzles rather than writing readable expressions.

Practice questions

1. What are 19 / 4 and 19 % 4 with int operands?

2. Why does double answer = 19 / 4; still store a whole-number result?

3. Write a calculation for the fractional average of two small integer values total and people, assuming people is positive.

4. Rewrite 1 <= year <= 6 as a correct C++ range check.

5. In change_counters.cpp, what is pending immediately after oldValue is initialized?

6. What is the difference between logical OR and bitwise OR? Why does the distinction matter for a guard?

Answer: Check the practice answers

1. The quotient is 4 and the remainder is 3. Both operations use integer rules here.

2. The integer division happens before the result is converted to double. The value 4 is converted; no fractional part is recovered.

3. static_cast<double>(total) / people converts before division. Check that people is nonzero and that the input ranges and desired precision suit the problem.

4. year >= 1 && year <= 6. Each comparison asks one question, and AND requires both answers to be true.

5. pending is 9, while oldValue is 8. The next prefix increment makes pending and newValue 10.

6. Logical OR combines conditions and short-circuits when the first is true. Bitwise OR combines integer bits and does not provide that conditional skipping rule. Use the logical operation for a Boolean guard.

Interview questions

Answer: Does assigning an integer division result to double make the division floating-point?

No. Operand types determine the arithmetic operation before assignment. Convert an operand before division when a fractional quotient is needed.

Answer: What is short-circuit evaluation?

Built-in logical AND and OR may skip their right operand when the left result already determines the answer. This supports readable conditions and guards, provided the guard is placed first.

Answer: Does static_cast guarantee a numeric value is safe?

No. It requests a permitted explicit conversion, but the program must still respect range and meaning. In particular, a floating-to-integer conversion must have a representable truncated result.

Answer: Are precedence and evaluation order the same thing?

No. Precedence describes expression grouping. Evaluation order concerns when the parts are evaluated. Do not infer a universal order from a precedence table or combine unnecessary side effects in one expression.

Quick revision

- Identify the operator family and the operand types.
- Integer division discards a fraction before later assignment.
- A cast expresses a conversion, not automatic range validation.
- Comparison and logical operations help build clear conditions.
- Parentheses improve grouping clarity; they do not cure invalid arithmetic.

08. Conditions with if, else and else if

A program often needs to choose. A study room may be open or closed; a mark may belong to one of several bands. We will write those rules clearly and check the values at the boundaries, not only one convenient example.

A condition is a question with a true or false answer

An **if statement** runs its associated statement or block when its condition is true. If an else is present, that alternative runs when the condition is false. The condition is tested when execution reaches the if statement.

Code fragment - Code fragment

```
if (roomOpen) {
    std::cout << "You may enter.\n";
} else {
    std::cout << "Please wait outside.\n";
}
```

roomOpen should be a previously defined bool here. The parentheses surround the condition. The braces group the instructions for each choice. The fragment is not a complete program; the examples below include the required declarations and main.

1. Test the condition
2. True: run the if body
3. False: run the else body
4. Continue after the choice

Only one branch of a reached if/else is selected. Independent if statements instead allow several actions to run.

Start with a simple if

An if does not require an else. When its condition is false, its body is skipped and execution continues afterward. A college notice may need an extra reminder only when an assessment is approaching.

study_reminder.cpp

```
#include <iostream>

int main()
```

```
{
    int daysLeft{2};
    if (daysLeft <= 3) {
        std::cout << "Plan a short revision session.\n";
    }
    std::cout << "Keep your notes ready.\n";
    return 0;
}
```

Output

```
Plan a short revision session.
Keep your notes ready.
```

Change daysLeft to 5. The reminder is skipped, but "Keep your notes ready." still prints because that statement is outside the if body. The example assumes a nonnegative number of days; a real calendar calculation needs to define how past dates are handled too.

if and else: exactly one of two paths

For a reached if/else with an ordinary Boolean condition, exactly one branch is selected. Do not put a condition after else; it is the alternative when the preceding if condition is false.

Code fragment - Code fragment

```
if (hasPass) {
    std::cout << "Pass checked.\n";
} else {
    std::cout << "Please visit the help desk.\n";
}
```

Use an explicit bool condition or a clear comparison. Although C++ allows integer conditions, with zero meaning false and nonzero meaning true, a spelling such as count > 0 usually explains the intended rule better than simply count.

An else-if ladder chooses the first matching branch

An **else-if ladder** checks alternatives in order. Once one condition is true, its branch runs and the later alternatives in that ladder are skipped. Order therefore matters.

Suppose a practice assessment uses four illustrative bands: 80-100, 60-79, 40-59 and below 40. These are exercise rules, not a claim about a university's grading policy. First reject values outside 0-100, then check the highest threshold first.

assessment_band.cpp

```
#include <iostream>

int main()
{
    int mark{};
    std::cout << "Enter a practice mark, 0 to 100:\n";
    if (!(std::cin >> mark)) {
        std::cout << "A whole-number mark is required.\n";
        return 1;
    }
```

```
    if (mark < 0 || mark > 100) {
        std::cout << "The mark must be from 0 to 100.\n";
        return 1;
    }
    if (mark >= 80) {
        std::cout << "Band: Strong\n";
    } else if (mark >= 60) {
        std::cout << "Band: Good progress\n";
    } else if (mark >= 40) {
        std::cout << "Band: Developing\n";
    } else {
        std::cout << "Band: More practice\n";
    }
    return 0;
}
```

Walkthrough

```
73
```

Output

```
Enter a practice mark, 0 to 100:
Band: Good progress
```

The input is a formatted numeric extraction as described in Chapter 6. If this is a strict form where extra text must be rejected, use the whole-line parsing approach from that chapter too. Here we are concentrating on the decision after reading a mark.

Input mark	First matching band condition	Result
0 or 39	None of the three thresholds	More practice
40 or 59	mark >= 40	Developing
60 or 79	mark >= 60	Good progress
80 or 100	mark >= 80	Strong
-1 or 101	Invalid range checked first	Reject before assigning a band

If mark >= 40 were checked first in the same ladder, a mark of 90 would also match it. The program would never reach the intended higher-band branch for that value.

Separate if statements can perform several actions

Use a ladder when the alternatives are exclusive. Use independent if statements when several reminders may all apply. For example, a new student may need both a room map and an account-help note.

independent_checks.cpp

```
#include <iostream>

int main()
{
    bool needsMap{true};
    bool needsAccountHelp{true};
```

```
    if (needsMap) {
        std::cout << "Collect a room map.\n";
    }
    if (needsAccountHelp) {
        std::cout << "Visit the account help desk.\n";
    }
    return 0;
}
```

Output

```
Collect a room map.
Visit the account help desk.
```

Changing the second if into else if would make the second reminder conditional on the first test being false. That would change the rule, not merely the appearance of the code.

Combine conditions and use nested decisions carefully

Logical AND requires both conditions; OR requires at least one. For example, a library-room rule may require both an open room and a valid pass. Write roomOpen && hasPass when only the final yes/no answer matters.

A **nested if** is an if inside another branch. It can be useful when we want different reasons for rejection.

entry_decision.cpp

```
#include <iostream>

int main()
{
    bool roomOpen{true};
    bool hasPass{false};
    if (roomOpen) {
        if (hasPass) {
            std::cout << "Entry allowed.\n";
        } else {
            std::cout << "Visit the pass desk.\n";
        }
    } else {
        std::cout << "The room is closed.\n";
    }
    return 0;
}
```

Output

```
Visit the pass desk.
```

Trace the outer question first: the room is open. Then check the inner question: the student has no pass. Only that inner else runs. If the room were closed, the program would not need to inspect hasPass in this nested version.

Braces prevent the dangling-else surprise

Without braces, a following else belongs to the nearest suitable unmatched if, not necessarily the one suggested by indentation. This is often called the **dangling else** problem. Braces make the intended grouping clear.

Always use braces in these beginner examples, even when a body contains only one statement. Adding a second statement later should not quietly place it outside the intended branch. An extra semicolon immediately after if(condition) creates an empty statement and can make the next block run independently.

Code fragment - Code fragment

```
if (hasPass) {  // no semicolon immediately after the condition
    std::cout << "Pass checked.\n";
    std::cout << "Continue to the room.\n";
}
```

Choose a short value with ?: when it helps

The conditional operator from Chapter 7 can choose a short result, such as an output word. Only the chosen alternative is evaluated.

Code fragment - Code fragment

```
std::cout << (roomOpen ? "Open" : "Closed") << '\n';
```

Do not force a long sequence of actions into nested conditional expressions just to reduce the line count. A readable if/else is usually better for several statements or different explanations.

Problem-solving checklist and common mistakes

Before coding a rule, write one example for each branch. For threshold rules, include the value just below, exactly at, and just above each boundary. Check invalid input before allowing it into a meaningful category.

- Use == for comparison, not an accidental assignment with =.

- Keep braces and indentation consistent; indentation alone does not group statements.

- Put higher overlapping thresholds first in a descending-band ladder.

- Choose independent if statements when multiple actions are allowed.

- State whether an endpoint is included: >= 60 includes 60; > 60 does not.

- Avoid deeply nested rules when a few early validity checks would make the main choices clearer.

Practice questions

1. What prints from study_reminder.cpp after changing daysLeft to 5?

2. Which band should assessment_band.cpp report for 40, 60 and 80?

3. Why would checking mark >= 40 first in that ladder be wrong?
4. When should two independent if statements be preferred over if/else if?
5. What message does entry_decision.cpp show if roomOpen is false and hasPass is true?
6. Why should you test -1 and 101 as well as normal marks?

Answer: Check the practice answers

1. Only "Keep your notes ready." prints. That statement is outside the if block.
2. Forty is Developing, 60 is Good progress and 80 is Strong. Each threshold is included by >=.
3. It would also match higher valid marks and stop the ladder before their more specific branch could run.
4. Use independent statements when both actions may be needed. A ladder selects only the first matching alternative.
5. "The room is closed." The outer condition is false, so the nested pass test is skipped.
6. They are outside the permitted input range. Without validation, an invalid value could receive a misleading band instead of being rejected.

Interview questions

Answer: What is the difference between if/else and two independent if statements?

An if/else selects between two alternatives. Independent if statements test their own conditions separately, so both bodies may run. Choose according to the problem's rule.

Answer: Why does else-if ordering matter?

The first true condition wins. When conditions overlap, a broad condition placed too early can hide the more specific branch you intended to select.

Answer: Which if owns an else when braces are omitted?

An else is associated with the nearest suitable unmatched if according to the language's grammar. Indentation does not change that association. Braces are a clearer way to show the intended structure.

Answer: How do you test a decision with numeric thresholds?

Test every branch and the values around each boundary, plus the allowed minimum, maximum and rejected values. Explain the expected result before comparing it with the program.

Quick revision

- A condition chooses whether a branch runs.
- An else-if ladder selects its first matching alternative.
- Independent if statements allow independent actions.
- Braces show grouping and prevent accidental else association.
- Boundary cases are part of the rule, not optional extra tests.

09. Choose a matching option with switch

A menu often asks the user to choose one numbered or lettered option. switch expresses that kind of exact-value choice. We will explain case labels, default handling, break, grouped cases and intentional fallthrough.

When does switch fit the problem?

Use switch when one value should select among known individual choices, such as option 1, 2 or 3. Use if/else for ranges or unrelated conditions, such as a mark from 60 to 79 or a room that is open and has available seats.

A switch condition can use integral or enumeration types, and certain class values that convert appropriately. An **enumeration** is a type describing named choices; we will study such types with later language features. For now, int and char are suitable examples. A plain double or std::string is not a valid ordinary switch condition.

1. Evaluate one value
2. Enter matching case
3. Run its statements
4. Break or deliberately continue

Without a match, default is selected if present. Labels do not automatically stop execution from continuing into later statements.

Read the structure first

Code fragment - Code fragment

```
switch (option) {
    case 1:
        std::cout << "Open the notice.\n";
        break;
    case 2:
        std::cout << "Visit the help desk.\n";
        break;
    default:
        std::cout << "Unknown option.\n";
        break;
}
```

The switch evaluates its controlling value once when reached, then transfers execution to the matching case label. Each case uses a colon, not a semicolon. If no case matches, execution enters default if one exists. Without a matching case or a default, the switch body is skipped.

The labels mark places to enter the body. They are not automatically separate if/else blocks. **After entering a case, execution continues through following statements unless something stops or transfers it.** That is why break matters.

A complete help-desk menu

desk_menu.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Choose 1 for timings, 2 for accounts, 3 for books:\n";
    int option{};
    if (!(std::cin >> option)) {
        std::cout << "Enter a whole-number option.\n";
        return 1;
    }
    switch (option) {
        case 1:
            std::cout << "Timings desk: Room 3\n";
            break;
        case 2:
            std::cout << "Accounts desk: Room 8\n";
            break;
        case 3:
            std::cout << "Books desk: Room 11\n";
            break;
        default:
            std::cout << "Choose one of the listed options.\n";
            break;
    }
    std::cout << "Menu visit finished.\n";
    return 0;
}
```

Walkthrough

```
2
```

Output

```
Choose 1 for timings, 2 for accounts, 3 for books:
Accounts desk: Room 8
Menu visit finished.
```

The break leaves the switch, not main. The final message still prints. In this example an unknown integer is reported and the menu visit ends normally; a failed numeric read returns a failure status instead. It is a one-choice demonstration, not a repeating menu. Loops come next.

As in the earlier decision examples, >> reads a numeric prefix. Use Chapter 6's whole-line parsing approach too if your menu must reject extra characters on the input line.

Group several labels for one action

You can place labels together when several exact choices need the same action. There is no action to accidentally skip between those adjacent labels.

shared_choices.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Choose P for practice or R for revision:\n";
    char choice{};
    if (!(std::cin >> choice)) {
        std::cout << "No choice was read.\n";
        return 1;
    }
    switch (choice) {
        case 'P':
        case 'p':
            std::cout << "Read one practice question.\n";
            break;
        case 'R':
        case 'r':
            std::cout << "Review your last solution.\n";
            break;
        default:
            std::cout << "That choice is not listed.\n";
            break;
    }
    return 0;
}
```

Walkthrough

```
p
```

Output

```
Choose P for practice or R for revision:
Read one practice question.
```

Use single quotes for these char case values. 'P' and 'p' are different values, so we list both. Reading a char consumes one non-whitespace character in this formatted-input example; it is not a check that the user typed a one-character line and nothing else.

Fallthrough: continuing into the next case

Fallthrough means execution continues from one case into the statements following another label. It happens naturally when no break, return or other transfer stops it. The switch does not recheck each following label before allowing those statements to run.

Accidentally omitting break often prints several menu actions. If continuing is intentional, `[[fallthrough]];` documents that intention and helps with compiler diagnostics. This attribute is available in our C++20 setting; it does not itself jump or stop execution.

staged_practice.cpp

```
#include <iostream>

int main()
{
    int startStage{1};
    switch (startStage) {
        case 1:
            std::cout << "Prepare the materials.\n";
            [[fallthrough]];
        case 2:
            std::cout << "Practise one problem.\n";
            break;
        default:
            std::cout << "Unknown starting stage.\n";
            break;
    }
    return 0;
}
```

Output

```
Prepare the materials.
Practise one problem.
```

Starting at stage 1 deliberately does both actions. Starting at stage 2 does only the practice action. For an ordinary menu whose options are independent, prefer a break after each finished action instead.

Case values must be suitable constants

Case values must be converted constant expressions of the required controlling type, and they must not duplicate another case value in the same switch after the relevant conversion. A constexpr int initialized with a small literal can provide a meaningful case name.

A normal mutable variable is not a valid case value merely because it currently contains a number. The following complete program is intentionally invalid.

Intentional error for study - variable_case.cpp

```
#include <iostream>

int main()
{
    int option{2};
    int selectedLabel{2};
    switch (option) {
        case selectedLabel:
            std::cout << "Selected.\n";
            break;
    }
    return 0;
}
```

For this simple rule, a repair is constexpr int selectedLabel{2};. If the compared value genuinely changes during execution, use an if comparison instead. A literal zero and another expression evaluating to zero are still duplicate labels; different spelling does not make their values distinct.

Case labels do not create separate variable scopes

When a case needs its own local variable, give that case's work braces. A jump to another label must not bypass an initialization that the language requires. Braces make the variable's intended lifetime and visibility clear.

Code fragment - Code fragment

```
case 1: {
    int tokens{3};
    std::cout << tokens << '\n';
    break;
}
```

This is a fragment for a switch body, not a complete program. It is different from expecting the case label alone to create a new scope. We will revisit scope and lifetime in their own chapter.

Problem: a small copy-desk calculator

The exercise copy desk charges 2 tokens per page for B/b mode and 5 tokens per page for C/c mode. Accept 0 through 50 pages and reject other counts. The exact rates are invented for this classroom problem. Validate the input before multiplying or reporting a total.

copy_cost.cpp

```
#include <iostream>

int main()
{
    char mode{};
    int pages{};
    std::cout << "Enter mode B or C, then pages from 0 to 50:\n";
    if (!(std::cin >> mode >> pages)) {
        std::cout << "A mode and whole-number page count are required.\n";
        return 1;
    }
    if (pages < 0 || pages > 50) {
        std::cout << "The page count is outside the allowed range.\n";
        return 1;
    }
    int tokensPerPage{};
    switch (mode) {
        case 'B':
        case 'b':
            tokensPerPage = 2;
            break;
        case 'C':
        case 'c':
            tokensPerPage = 5;
            break;
        default:
            std::cout << "Unknown copy mode.\n";
            return 1;
    }
    std::cout << "Total tokens: " << tokensPerPage * pages << '\n';
    return 0;
}
```

Walkthrough

```
B 12
```

Output

```
Enter mode B or C, then pages from 0 to 50:
Total tokens: 24
```

The largest permitted total is 250, so these int calculations fit even the minimum required int range. Notice that the unknown-mode branch returns before the total is printed. The program does not accidentally treat an invalid mode as a zero-cost valid order.

Common mistakes and choosing between switch and if

Situation	Prefer	Reason
A menu with exact integer choices	switch	One value selects among known options
A grade band or numeric interval	if/else	Range conditions are needed
Several unrelated Boolean questions	if statements	Each question has its own condition
A plain string command such as "help"	String comparisons with if/else	std::string is not an ordinary switch condition
Several letter spellings with one action	Grouped case labels	Each exact character can share the action

Keep default last for readability, although C++ does not require that position. Avoid accidental fallthrough, duplicate labels and missing colons. Remember that break exits the nearest enclosing loop or switch; when a switch is inside a loop, breaking the switch does not necessarily end the loop.

Practice questions

1. What prints from desk_menu.cpp for choice 3? Does break prevent the final message?
2. Why do both P and p reach the same action in shared_choices.cpp?
3. What would happen to staged_practice.cpp for startStage 2?
4. Can a plain std::string be the controlling value in an ordinary switch?
5. What does copy_cost.cpp report for C 50, B 0 and X 12?
6. Why is a normal variable unsuitable as a case label? What should you use for a runtime comparison instead?

Answer: Check the practice answers

1. It prints the menu prompt, "Books desk: Room 11" and "Menu visit finished." The break leaves the switch only.
2. Both character values have labels immediately before the same statements. No separate action is needed between them.

3. Only "Practise one problem." prints. Execution enters at case 2 rather than running the earlier case's statements.

4. No. Use suitable string comparisons with if/else, or deliberately translate validated commands into another representation first.

5. C 50 reports 250 tokens; B 0 reports zero tokens. X 12 reports "Unknown copy mode." and returns failure without printing a valid total.

6. A case requires a suitable constant expression, not an arbitrary value obtained or changed during execution. Use an if comparison when that is the real rule.

Interview questions

Answer: Is break compulsory after every case?

No, but execution can continue into later case statements without it. Use deliberate grouped cases or clearly documented fallthrough when that behaviour is intended; otherwise end a completed independent case appropriately.

Answer: What happens when no case matches and no default exists?

No part of the switch body is selected, and execution continues after the switch. A default is useful when unlisted values need an explicit response.

Answer: Does default have to appear last?

No. Its position is a readability choice, not a requirement. Execution can still fall through into or out of its statements depending on the surrounding structure, so control flow must be deliberate.

Answer: Why put braces around a case's local declarations?

The labels alone do not create independent scopes. A braced block limits a variable's scope and avoids inappropriate jumps across its initialization. It also makes the case's work easier to read.

Quick revision

- switch chooses an entry label from one controlling value.

- Cases need suitable unique constant values.

- break leaves the switch; return leaves the function.

- Grouped labels and fallthrough must be deliberate.

- Validate input and avoid reporting a result for an unaccepted choice.

10. Loops: repeat useful work and know when to stop

A loop repeats a piece of work without requiring us to copy the same statements many times. The important questions are what repeats, what changes, and why the repetition eventually stops.

Three common loop shapes

An **iteration** is one pass through a loop's body. A **counter** is a value used to track progress, often the number of passes or a position. Choose the loop shape that makes the task's start, condition and progress easy to read.

Shape	When the condition is tested	Useful starting point
while	Before a pass	Repeat while some condition remains true
do/while	After a pass	Perform the body once before deciding whether to repeat
for	Before a pass, with start and update together	Count through a known sequence of positions
Range-based for	Visits the elements of a range	Read each character or container element in turn

These are useful habits, not restrictions. A while can count fixed repetitions too. A for can use a condition that changes with input. Prefer whichever clearly expresses the actual work.

1. Initialize once
2. Test before the body
3. Run the body
4. Update, then test again

A false condition ends the loop. break exits early; continue in a for still leads to its update and condition test.

for: start, test, body, update

A normal for header has three parts separated by semicolons: initialization, condition and update. The initialization happens once. The condition is checked before each pass. The update happens after a normally completed body, then the condition is checked again.

seat_labels.cpp

```
#include <iostream>

int main()
{
    for (int seat = 1; seat <= 4; ++seat) {
        std::cout << "Seat " << seat << '\n';
    }
    return 0;
}
```

Output

```
Seat 1
Seat 2
Seat 3
Seat 4
```

Read the header as "start seat at 1; continue while seat is at most 4; increase seat after each pass". The next value after the final update is 5, which fails the condition. No "Seat 5" line is printed.

Before condition	Condition seat <= 4	Body	Value after update
1	true	Print Seat 1	2
2	true	Print Seat 2	3
3	true	Print Seat 3	4
4	true	Print Seat 4	5
5	false	Do not enter	No update from another pass

The loop variable declared in the header belongs to that for statement; do not expect to use it afterward. If the task needs a result after the loop, keep that result in a suitable variable declared outside it.

while: test before entering

A while loop checks its condition first. If that condition is false at the beginning, the body runs zero times. Unlike the usual for header, a while does not have a separate built-in place for an update; the program must make progress through its body or another deliberate mechanism.

remaining_tasks.cpp

```
#include <iostream>

int main()
{
    int remaining{2};
```

```
    while (remaining > 0) {
        std::cout << "Tasks remaining: " << remaining << '\n';
        --remaining;
    }
    std::cout << "All tasks checked.\n";
    return 0;
}
```

Output

```
Tasks remaining: 2
Tasks remaining: 1
All tasks checked.
```

After the second pass remaining becomes zero. The next check fails and the final message prints. If remaining started at zero, only that final message would print. If we removed --remaining, this particular loop would keep seeing the same positive value instead of completing its task.

do/while: perform the body before checking

A do/while is useful when the first attempt must happen before deciding whether another attempt is needed. Its spelling includes a semicolon after the final while(condition).

first_batch_check.cpp

```
#include <iostream>

int main()
{
    int batch{5};
    do {
        std::cout << "Checked batch: " << batch << '\n';
        ++batch;
    } while (batch < 3);
    return 0;
}
```

Output

```
Checked batch: 5
```

Although batch < 3 would have been false initially, it is not tested before the first body in this form. The body prints 5 and changes batch to 6, then the actual condition check is false. This deliberately small example makes the difference from while visible; it is not a recommendation for a real batch-number policy.

Choose correct boundaries and updates

To visit four positions numbered 0 through 3, use index < 4 with a start of zero. To print labels 1 through 4, use seat <= 4 with a start of one. Neither < nor <= is universally right; the intended range decides.

An **off-by-one error** visits one too many or one too few positions. Write down the first value, last included value and first excluded value before coding. Later, array accesses will make these boundaries especially important.

A for update need not be an increment by one. For example, value += 2 can step through every second value, and --value can count downward. Ensure each update can move toward termination and remains in the variable's range. An unsigned countdown tested with value >= 0 never becomes negative, so that is not a valid stopping rule for it.

Do not put a stray semicolon after for(...) or while(condition). It forms an empty body rather than attaching the following braced block as you intended. The final semicolon in do/while is different: it is required by that loop's syntax.

Nested loops: repeat an inner job for each outer item

A **nested loop** is a loop inside another loop. To label places in a small room, let an outer loop select a row and an inner loop select each position in that row.

room_positions.cpp

```
#include <iostream>

int main()
{
    for (int row = 1; row <= 2; ++row) {
        for (int position = 1; position <= 3; ++position) {
            if (position > 1) {
                std::cout << ' ';
            }
            std::cout << row << ',' << position;
        }
        std::cout << '\n';
    }
    return 0;
}
```

Output

```
1,1 1,2 1,3
2,1 2,2 2,3
```

The inner loop starts again at position 1 for each row. Its body prints a total of 2 times 3, or six, positions. The newline belongs after the inner loop, so each row appears on its own output line.

For each outer iteration, the inner loop executes as a whole; it is not restricted to only one inner body pass. With larger sizes, nested repetition can produce a large amount of work. Count the passes when judging whether a solution suits the task.

break and continue have different jobs

`break` immediately leaves the nearest enclosing loop or switch. `continue` skips the rest of the current loop body and proceeds with that loop's continuation behaviour. In a normal for, continue still leads to the update and then the next condition test. In while and do/while, it leads to the condition test.

skip_and_stop.cpp

```
#include <iostream>

int main()
```

```
{
    for (int slot = 1; slot <= 6; ++slot) {
        if (slot == 3) {
            continue;
        }
        if (slot == 5) {
            break;
        }
        std::cout << "Checked slot: " << slot << '\n';
    }
    std::cout << "Scanning finished.\n";
    return 0;
}
```

Output

```
Checked slot: 1
Checked slot: 2
Checked slot: 4
Scanning finished.
```

Slot 3 is skipped, but the loop continues with 4. At 5, break ends the loop before printing that slot, so 6 is never reached. The statement after the loop still runs; return would instead leave the function.

In a nested loop, break does not automatically leave every surrounding loop. Also remember that a break in a switch inside a loop exits that switch, not the loop. Avoid placing a manual while-loop update after a continue that will always skip it; that can prevent progress.

Range-based for: read each element

When you need each element rather than a numeric position, a range-based for can reduce counting mistakes. We can demonstrate it using std::string, which you already know from Chapter 5.

word_letters.cpp

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

int main()
{
    std::string word{"LAB"};
    for (char letter : word) {
        std::cout << letter << '\n';
    }
    return 0;
}
```

Output

```
L
A
B
```

Read the colon as "for each element in". Each pass initializes letter with a copy of the next char value. Changing this copy would not change word. References, taught later, let some range-loop variables refer to the original elements instead.

The simple English word uses one char per letter. Iterating over std::string visits its char elements, not necessarily complete displayed characters in every text encoding. Arrays and standard-library containers will provide more range examples in later chapters.

Problem: accumulate a bounded total

An exercise asks for 1 + 2 + ... + n, accepting n from 0 through 100. For zero, the required total is zero. Keep a total outside the loop and add each included value. The small input limit makes every counter update and result safe for the types shown.

bounded_total.cpp

```
#include <iostream>

int main()
{
    std::cout << "Enter a final number from 0 to 100:\n";
    int finalNumber{};
    if (!(std::cin >> finalNumber) || finalNumber < 0 || finalNumber > 100) {
        std::cout << "The final number is not accepted.\n";
        return 1;
    }
    long long total{};
    for (int number = 1; number <= finalNumber; ++number) {
        total += number;
    }
    std::cout << "Total: " << total << '\n';
    return 0;
}
```

Walkthrough

```
4
```

Output

```
Enter a final number from 0 to 100:
Total: 10
```

The totals after successive passes are 1, 3, 6 and 10. Initializing total inside the body would reset it each pass and lose the earlier additions. The example uses formatted numeric input; use Chapter 6's strict line check too when a whole-line form is required.

Repeat validation, but stop when input ends

Now we can extend Chapter 6's single-line validation into a retrying program. Accept a rating from 1 through 5, allow surrounding whitespace, reject extra text, and stop without saving if no further line can be read. Use zero internally to mean "not yet accepted" because zero is outside the valid rating range.

rating_retry.cpp

```
#include <iostream>
#include <sstream>
#include <string>

int main()
```

```
{
    int rating{};
    do {
        std::cout << "Enter a rating from 1 to 5:\n";
        std::string line;
        if (!std::getline(std::cin, line)) {
            std::cout << "No more input. Nothing saved.\n";
            return 0;
        }
        std::istringstream reader{line};
        char extra{};
        if (!(reader >> rating) || (reader >> extra) || rating < 1 || rating > 5) {
            std::cout << "That rating is not accepted. Try again.\n";
            rating = 0;
        }
    } while (rating == 0);
    std::cout << "Saved rating: " << rating << '\n';
    return 0;
}
```

Walkthrough

```
9
3
```

Output

```
Enter a rating from 1 to 5:
That rating is not accepted. Try again.
Enter a rating from 1 to 5:
Saved rating: 3
```

Each attempt has a fresh local string reader, so a failed conversion there does not leave std::cin stuck in a numeric failure state. A blank line, extra characters and an out-of-range integer all lead to another attempt. A failed getline exits instead of retrying endlessly. Here ending input is treated as a normal cancellation, hence return 0 with "Nothing saved."

This is still a local teaching example that reads a whole line into memory. A public application needs suitable line-size and resource limits as well as these content checks.

Common mistakes and a dry-run habit

Before running a loop, write a table with the counter, condition, changed values and output after each pass. Include a case with no body passes when the loop permits it, a single-pass case and its maximum allowed size.

- Forgetting to change the value used by a while condition.

- Using <= when the intended final index is excluded.

- Resetting an accumulated result inside every iteration.

- Assuming continue performs a manual update located later in a while body.

- Assuming break leaves every surrounding loop or the whole function.

- Retrying failed input without recognising when input has ended.

- Increasing an input limit without reviewing counter and total ranges.

Practice questions

1. How many lines does seat_labels.cpp print, and why does it not print Seat 5?

2. What happens in remaining_tasks.cpp if remaining starts at zero?

3. Why does first_batch_check.cpp still print one line?

4. How many room-position pairs does room_positions.cpp print? Where should its row-ending newline belong?

5. What is the difference between the effect of continue at slot 3 and break at slot 5?

6. Predict bounded_total.cpp for n equal to 0, 1 and 100. Predict rating_retry.cpp after invalid input followed by end-of-input.

Answer: Check the practice answers

1. Four seat lines. The fifth condition check sees seat equal to 5 and is false, so there is no fifth body pass.

2. The while condition is false before its first pass. Only "All tasks checked." prints.

3. The do body runs before its first condition test. That test is false afterward, so there is no second pass.

4. Six pairs. Put the row-ending newline after the inner loop and inside the outer loop, so all positions in one row stay together.

5. Continue skips printing slot 3 but permits the for update and later passes. Break at 5 ends the loop immediately; neither 5 nor 6 prints.

6. The totals are 0, 1 and 5050. After invalid input and then end-of-input, the rating program reports that nothing was saved and stops; it does not keep asking forever.

Interview questions

Answer: What is the key difference between while and do/while?

while tests before entering its body, so zero passes are possible. do/while performs its body before the first test when reached, then repeats only while the condition is true. Choose based on whether that first action is required.

Answer: What happens after continue in a for loop?

The rest of the current body is skipped, then the for update expression is executed and the condition is tested for another pass. This differs from relying on a manual update placed later in a while body.

Answer: Does a nested loop always take quadratic time?

No. The work depends on how many times each body runs and what it does. Two independent input-sized loops nested in the usual way may do a product of their sizes. Fixed limits or dependent ranges can give different counts, so explain the actual bounds.

Answer: How do you make an input-retry loop terminate safely?

Define what is accepted, what causes a retry and what stops input altogether. Check the read result and handle end-of-input or an unavailable stream. Do not reset a failure and retry forever without new input.

Quick revision

- Know the starting state, condition, progress and stopping point.
- A for update is separate from its body; a while needs deliberate progress too.
- do/while checks after its body and ends with a semicolon.
- break, continue and return transfer control in different ways.
- Keep totals outside loops and test the endpoints.
- Input ending must be a recognised outcome, not an infinite retry.