Resume-Day

Learn. Practise. Build.

YOUR FIRST PROGRAMMING WORKBOOK

C Programming

Learn and Practise

Simple English. Small steps. Clear examples.

Start with your first program.

Build your confidence through practice.

40 foundation chapters + a complete workbook

100 extra solved programs | 4 expanded projects

40 debugging exercises | 50 output questions

60 independent exercises | 100 interview answers

Sample contents - complete Chapters 01-10

Before you begin: a friendly computer guide 3

Chapter 01: What is C programming? 5

Chapter 02: Get your tools ready 9

Chapter 03: Your first C program, one line at a time 18

Chapter 04: Writing rules, output and comments 24

Chapter 05: Variables and constants 30

Chapter 06: Data types: choose the right kind of value 39

Chapter 07: Operators: calculate and compare values 48

Chapter 08: Type conversion: keep the value you need 58

Chapter 09: User input: read and check a value 64

Chapter 10: Decisions with if and else 71

Before you begin: a friendly computer guide

You do not need to know programming words before starting. You need a place to save your work, a way to write plain text and a tool that builds C programs. Take these first steps slowly. The first ten chapters are also included in the sample edition.

A file is something you save

A file is a named item on your device. A photo, a PDF and a C source file are all files, but they store different kinds of information. Our source files contain plain text. Do not write C inside a Word document and expect a compiler to read the Word format.

A folder keeps work together

Create a folder such as CPractice. Inside it, give each independent example its own folder. That makes it easier to find the source file, its input and the executable built from it. The code archive supplied with this book already follows that pattern.

Show the real filename

Turn on filename extensions in File Explorer. A modern C source should be saved with lowercase .c, such as hello.c. Check that it is not hello.c.txt. Older DOS tools often display filenames in uppercase; follow the modern lowercase convention when saving files for GCC.

Save before building

Typing changes text in an editor. Saving writes those changes to the file. Building then reads the saved source. If you edit without saving, the compiler may build the previous version. If a build fails, do not assume an older executable contains your changes.

A terminal is a place for commands

A terminal lets you type instructions to tools. PowerShell is one Windows option. gcc --version is a command to a compiler, not a C statement. Do not paste terminal commands inside the main function. The book labels terminal boxes separately from source and output.

Build and run are different steps

The build creates a program for your environment. Running starts that program. The compiler can finish quietly without printing a success sentence. Read and fix reported errors before running the output file. Chapter 2 gives setup choices; Chapter 3 explains the first build.

Use the keyboard without rushing

Ctrl+S commonly saves, Ctrl+C copies selected text and Ctrl+V pastes in editors. Terminal shortcuts can differ. Parentheses (), braces {} and brackets [] are different characters with different roles in C. Type a short example first and check punctuation carefully.

Find a file again

Keep a note of the folder where you saved it. A file dialog and a terminal can be open in different folders. The working folder matters for commands and for files your program opens. If a file is not found, check its location before rewriting the program.

If you only have a shared computer

An online C editor is enough to begin with small examples. Keep your own copies of code and notes because the website may not retain them. Later, practise compiling local files and using multiple source files so your projects are reproducible.

How to use the companion archive

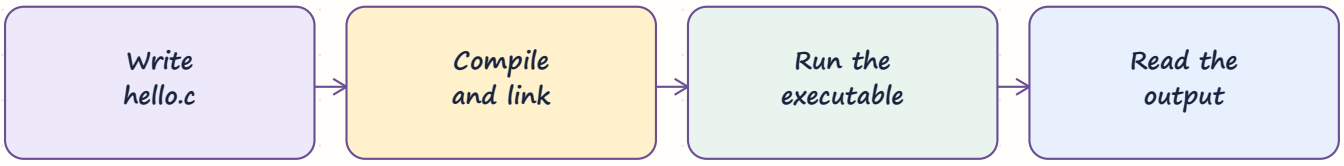
Extract the supplied ZIP into a practice folder. Each program folder contains its source, build instructions and expected output, with input files when needed. Independent examples each have main; do not build all main files together. Project support files belong beside their project main.

How this workbook teaches

Read one explanation, predict a small result, run the program and change one thing. The solved-program section gives complete worked solutions. The independent exercises put hints before solutions so you can try first. Write down a failed case and keep it after you fix the cause.

An honest note about checking

The complete examples were checked with a Windows C17 compiler using strict warnings. That does not mean every installer or operating system was tested. Wrong-code fragments are labelled for study and are not treated as valid programs. A human teacher review and testing with a real beginner remain valuable before the final commercial release.



Follow the idea from left to right. Check the chapter text for the exact C rules.

A small practice record

Date	What I tried	Expected	Actual	What I changed
Write here	Program and input	Predict first	Observe carefully	Explain the reason

Chapter 01: What is C programming?

You do not need to know programming to start this chapter. We will begin with a simple idea: giving a computer instructions.

What is a program?

A **program** gives a computer instructions for a task. The task might be adding marks or showing a class notice.

Think about a calculator. You enter two numbers, choose addition, and see the answer. A program tells the calculator how to do that work.

Programming means writing those instructions. A person who writes programs is called a **programmer**. You can learn this work one small step at a time.

You do not have to remember every new word today. Read the example, try the small task, and come back when you need a reminder.

A familiar example: finding a marks total

Suppose a student gets 35 marks in one test and 40 in another. To find the total, you can follow these steps:

1. Start with 35 and 40.

2. Add the numbers.

3. Show the answer, 75.

A computer needs clear instructions too. It cannot decide what you meant by "do the marks work." You need to say which values to use and what to do with them.

Word	Simple meaning	In this example
Input	Information given to a program	35 and 40
Processing	The work done with that information	Adding the numbers
Output	The result the program gives	75

Input does not always come from a keyboard. A program can also start with values already written in its instructions. We will begin that way.

What is a programming language?

A **programming language** gives us words and writing rules for making programs.

People use languages such as Hindi and English to share ideas. We use programming languages to describe work for a computer. C, Java and Python are different programming languages.

The comparison has a limit: a computer tool follows exact writing rules. It cannot always work out a missing symbol from the meaning of your sentence. That is why a small typing mistake can stop a program from building.

You do not need advanced English to write a small C program. You need to learn a few C words, understand their job and practise using them.

What is C?

C is a programming language used to write computer programs. We can use it to do calculations, show messages, work with files and build many other kinds of software.

Dennis Ritchie developed C at Bell Labs in the early 1970s. C was closely connected with the development of Unix, an operating system.

An **operating system** is the main software that helps a computer run programs and manage things such as files. Windows and Linux are examples. You do not need to build an operating system to learn C.

We will start with much smaller jobs: showing a message, remembering a value and calculating a total.

Why learn C?

C gives you practice with ideas that appear in many languages: values, decisions, repeated work and functions. A function is a named piece of work; we will meet a small one in Chapter 3.

C is also used in software that works closely with devices, such as parts of operating systems and programs inside electronic products. We will leave those larger applications for later.

Learning C does not mean you must learn every programming language afterwards. First, learn to write and explain a small program correctly. That skill is more useful than memorising a long list of language names.

What does C code look like?

Code is the text we write to describe the program. Here is a small complete program. For now, just notice the message inside the quotes. Chapter 3 will explain every line.

welcome.c

```c
#include <stdio.h>

int main(void)
{
    printf("Welcome to your first C lesson.\n");
    return 0;
}
```

Output

```
Welcome to your first C lesson.
```

The `printf` line asks the program to show the message. The output is the text you see after the program runs. The other lines help form a complete C program.

You are not expected to understand all the symbols yet. The next two chapters will help you prepare your tools, run this kind of program and read it slowly.

How should I practise?

1. Read one small explanation.
2. Type its example when you have your tools ready.
3. Run it and compare the result with the shown output.
4. Change one small thing, such as the message.
5. Explain what changed in your own words.

An **error** means something needs attention. It does not mean you cannot learn programming. We will show common errors and how to fix them as we go.

Try not to copy a long program and move on without reading it. A five-line program you understand is a better starting point than a large program you cannot explain.

Common beginner worries

Worry	A useful way to begin
My English is weak.	Learn the small set of words used in each lesson. You can explain the idea aloud in your own language.
I have never written code.	Start with the message example. Change only the words inside its quotes.
I cannot remember all the symbols.	Keep the example beside you. Remember their jobs through practice.
I made an error.	Read the first useful error message and check one line at a time.

Try it on paper

You buy two notebooks for 30 rupees each. Write the information you already know, the work you need to do and the answer you expect. You do not need C code for this task.

Answer: Check the notebook answer

The information is 2 notebooks and 30 rupees per notebook. The work is multiplication: 2 × 30. The result is 60 rupees. This gives us the input, processing and output of a small problem before we write any code.

Questions and quick revision

Answer: What is the difference between a program and programming?

A program is the set of instructions. Programming is the work of writing those instructions. A recipe and the work of writing a recipe are different in a similar way.

Answer: Is C the same as C++?

No. C and C++ are different languages. Some of their writing looks similar, but their rules differ. In these lessons we save files with .c and compile them as C.

Answer: Do I need to understand the whole welcome program now?

No. In this chapter, recognise the message and its output. Chapters 2 and 3 explain the tools and every line of the program.

Remember: a program gives instructions, C is one language for writing them, and output is the result you see.

Chapter 02: Get your tools ready

To write C on your computer, you need a place to type your code and a tool that builds it into a program. We will explain both before installing anything.

The two tools you need

A **text editor** lets you type and save text. We will use it to write C code. Notepad and Visual Studio Code are examples of editors.

A **compiler** checks your C code and helps turn it into a program your computer can run. GCC and Clang are examples of C compilers.

Think of writing a recipe and preparing the food. The recipe is the written plan. It is not the finished dish. In a similar way, saving C code does not by itself create a running program.

Item	Its job
Editor	Write and change the code.
Source file	Keep the code you saved, such as hello.c.
Compiler	Check and build the code.
Built program	Run the instructions and produce a result.

An **IDE** means **Integrated Development Environment**. In simple words, it is an application where you can write code, build it and run it using menus or buttons. It can also help you find mistakes. Code::Blocks and Turbo C++ are examples of IDEs.

The IDE is the place where you work; the compiler does the job of translating your code. Some IDE installers include a compiler, and some do not. Installing an editor or an IDE alone does not always give you everything needed to run C.

Want to try an example before installing tools?

An online C editor lets you type code in a website and run it there. This is useful for trying your first small example, especially on a shared computer.

Open an online C editor, choose **C** if it offers several languages, replace its sample code with the example below, and choose **Run**. Look for a panel called Output or Console.

tools_check.c

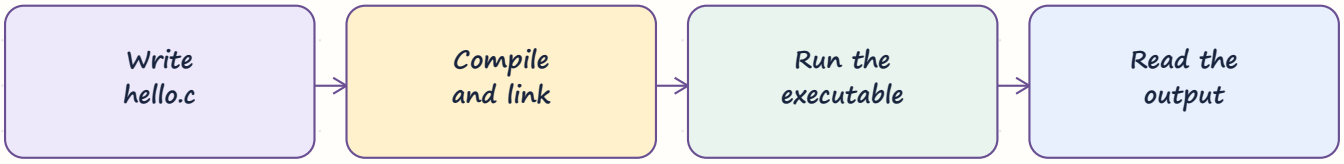
```c
#include <stdio.h>

int main(void)
{
    printf("My C practice starts here.\n");
    return 0;
}
```

Output

```
My C practice starts here.
```

Chapter 3 explains each line. For this first check, the important result is seeing the same message in the output panel. This ebook contains examples to read and copy. Run them in your chosen editor or terminal.

An online editor may not keep your work after you close the page. Save a copy of code you want to keep. For longer practice, a computer with your own files is easier to use.



Follow the idea from left to right. Check the chapter text for the exact C rules.

Other ways to work: choose one setup

You may hear different tool names from friends or teachers. They are different ways to write and run programs, not different versions of the C language itself.

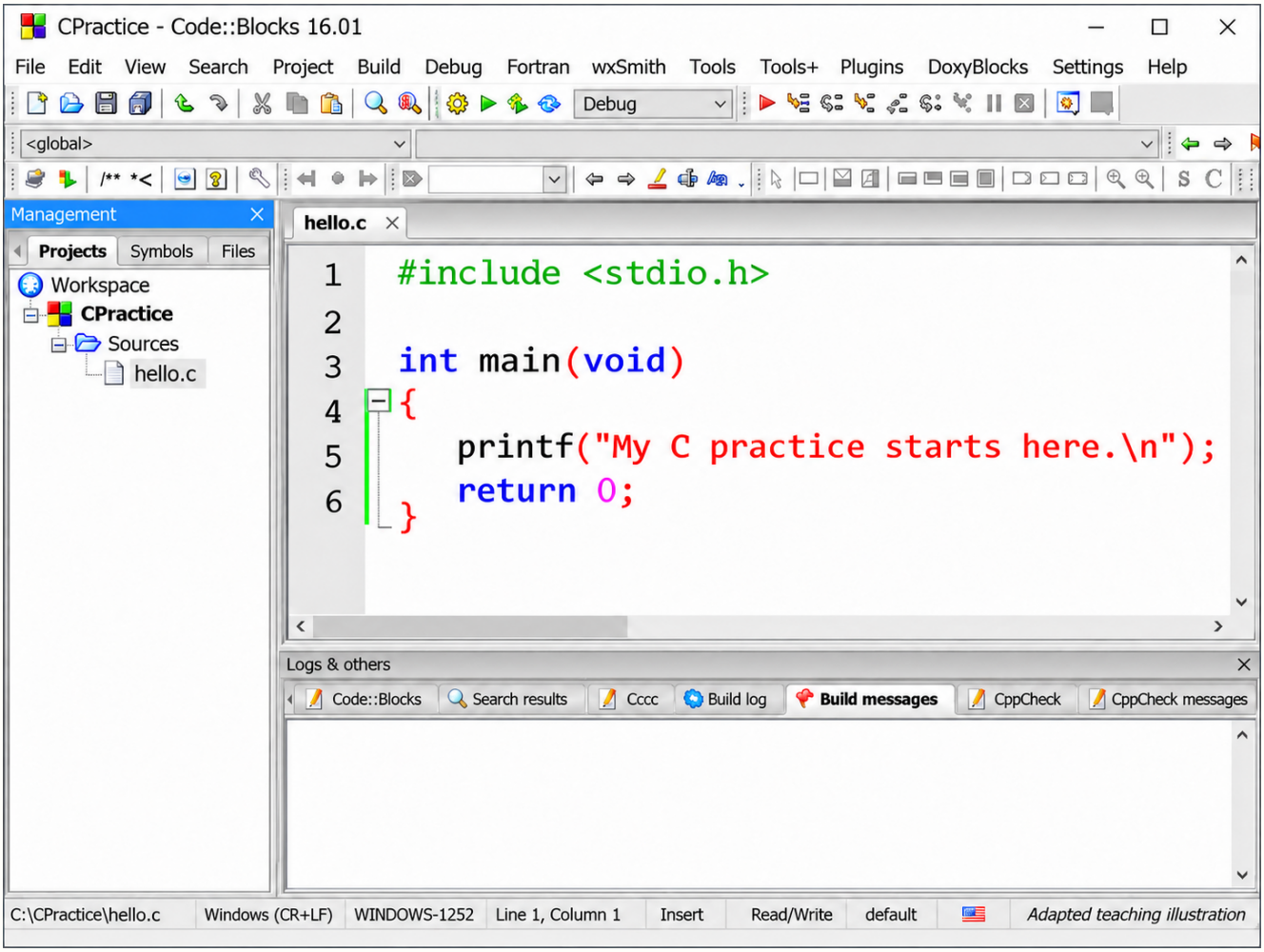
Tool	What it gives you	When to consider it
Code::Blocks with GCC	An IDE with menus for writing, building and running C	You want a desktop application with build buttons
Visual Studio Code with GCC or Clang	An editor with extra programming features when configured	You want an editor you can also use for other languages; the compiler needs separate setup
Older Turbo C++	An older IDE and compiler that can also work with C source files	Your college lab specifically requires that older setup
An online C editor	A website that builds and runs your code	You want to try small examples before installing tools

For this course, choose a compiler that supports C17, the C version used by our examples. On Windows, Code::Blocks with a suitable GCC compiler is a useful option if you prefer buttons. The MSYS2 instructions later in this chapter are another option if you prefer to follow our terminal commands. You do not need to install both.

If your present setup already builds and runs the example correctly with a C17 compiler, keep using it. Learning C matters more than repeatedly changing your editor.

Code::Blocks: build and run with buttons

Code::Blocks lets you keep your source files in a **project**. A project groups the files and settings needed to build one program. For your first exercise, that can be just one C source file.



Adapted teaching illustration of Code::Blocks with hello.c open. The left panel lists files, the middle shows our C program and the lower panel shows build messages. This is an AI-assisted illustration of an older interface, not a screenshot of a live build. Newer versions may look different.

Original screenshot: unknown author, via Softpedia / Wikimedia Commons / CC BY 4.0. AI-assisted adaptation by Resume-Day: replaced source text with our C example, changed filename to hello.c and rebuilt the illustration.

1. Open the official download page at https://www.codeblocks.org/downloads/binaries/ and choose a package for your computer.

2. On Windows, choose an installer whose name includes **mingw-setup.exe** if you need the included compiler. A package without that compiler needs a separately installed and configured compiler. Read the package description rather than choosing only by its version number.

3. Install and open Code::Blocks. If it asks you to select a detected compiler, choose the installed GNU GCC compiler.

4. Choose **File -> New -> Project**, then **Console application**. A console application reads and prints text in a terminal window, like our examples.

5. When asked for the language, select **C**, not C++. Give the project a name, such as CPractice, and choose its folder. Select the GCC compiler for the project and finish the wizard.

6. Open the generated **main.c** file under the project's Sources group. Replace its sample code with the complete tools_check.c program above. Keep this file named main.c; the program does not depend on that particular filename.

7. Save, then choose **Build -> Build and run**. Building checks and translates the saved code; running starts the built program. You should see My C practice starts here.

For this tutorial's language settings, open **Project -> Build options**, select the project, and add `-std=c17 -Wall -Wextra -Wpedantic` under the compiler's **Other compiler options**. These options select C17 and request useful warnings. Menu wording can vary by version. If the compiler rejects the C17 option, check which compiler the project is using and install a suitable newer one.

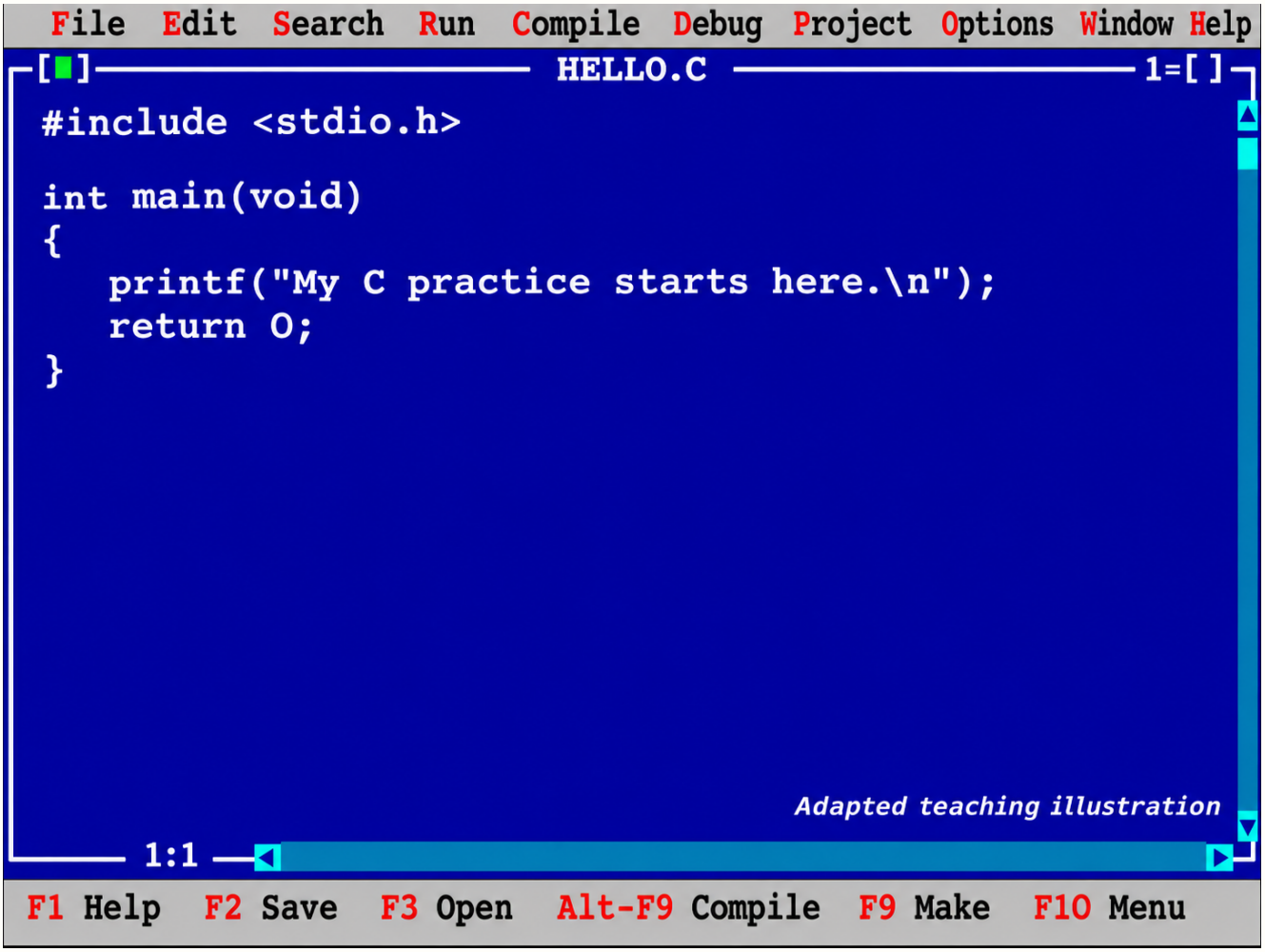
If Code::Blocks cannot find a compiler, check **Settings -> Compiler -> Toolchain executables** and the selected compiler's installation folder. The IDE needs the actual compiler files, not just an empty folder with the right name.

Start a separate project for each independent example. Do not add many examples that each contain main to the same project: one built program should have one main function. Save C files with **.c**, not .cpp; choosing a C++ project changes how the source may be compiled.

You can use this route instead of the Windows terminal setup below. You do not need to change PowerShell's Path just to use an included compiler that Code::Blocks has already found. If you later want to run gcc directly in PowerShell, that terminal needs its own working compiler setup too.

Turbo C++: what if your college uses it?

You may see an older Turbo C++ installation in a college lab or an old video lesson. Despite the C++ in its name, it can also compile C source files. C and C++ are different languages, so keep C work in a .c file and follow your lab's instructions for its C mode.



Adapted teaching illustration of older Turbo C++ with HELLO.C open. The blue editor shows the same C program as our Code::Blocks illustration. Menus are above and keyboard shortcuts below. This is an AI-assisted illustration, not a screenshot of a live build; use a C17 compiler for the full course.

Original screenshot: SAMNAD.S, via Wikimedia Commons / CC BY-SA 3.0. AI-assisted adaptation by Resume-Day: replaced source text with our C example, changed filename to HELLO.C and rebuilt the illustration.

The older DOS-era Turbo C++ compiler does **not** support the C17 version used in this course. Some later examples may fail there even when the C code is correct for C17. Its screen, menus and assumptions about type sizes can also differ from a modern GCC setup. An old tool's behaviour is not a rule for every C compiler.

Older lessons may contain names such as conio.h, clrscr and getch. These are not part of the standard C library used in this course. clrscr clears the screen in certain older environments, and getch reads a key in those environments. You do not need to add either one to our examples. Keep **int main(void)** rather than changing it to void main to match an old example.

If your teacher requires Turbo C++, use the lab's approved setup for those exercises and ask which compiler version is expected. For working through this whole course, also have access to a C17 compiler, either locally or through a suitable online editor. **You can learn the same basic ideas without copying old compiler-specific code into every program.**

Windows setup: install GCC using MSYS2

The following path is for a **64-bit Intel or AMD computer running Windows 10 version 1809 or newer**, including Windows 11. If that does not match your computer, start with the online option instead of following these installation steps.

MSYS2 is a tool we will use to install GCC. You do not need to learn all its other features.

1. Open the official MSYS2 website: https://www.msys2.org/.

2. Download and run the installer for your supported Windows computer.

3. Keep the usual installation folder, `C:\msys64`, unless you have a reason to choose another one. If you change it, keep a note of the location.

4. Finish the installation. Open **MSYS2 UCRT64** from the Windows Start menu if it does not open automatically.

5. Copy the next command into that window and press Enter. It installs the C compiler and the tools it needs.

Terminal command - type this in your terminal

```
pacman -S mingw-w64-ucrt-x86_64-gcc
```

If it asks whether to continue with the installation, read the question and enter `Y`. Wait for the installation to finish. You need an internet connection for the download.

Next, check that GCC is available in that window:

Terminal command - type this in your terminal

```
gcc --version
```

You should see GCC version information. Its numbers may differ from someone else's computer. You do not need to match a particular version number to recognise this successful check.

Make GCC available in Windows PowerShell

PowerShell is a Windows application where you can type commands. A command is a written instruction to a tool. We will use PowerShell to build and run our saved C files.

Path is a Windows setting that lists folders where Windows looks for tools. Adding the compiler's folder lets PowerShell find `gcc` when you type its name.

1. Open the Windows Start menu and search for **Edit environment variables for your account**.

2. In your user variables, select **Path** and choose **Edit**.

3. Choose **New**. Add `C:\msys64\ucrt64\bin`. If you installed MSYS2 elsewhere, use that installation's `ucrt64\bin` folder instead.

4. Save the change with **OK**. Keep the other Path entries; add this folder as a new entry.

5. Close any PowerShell windows that were already open. Open a new PowerShell window.

6. Run `gcc --version` there too.

The Path change is a setup task, not a C statement. Do not paste the folder path or the installation command into a `.c` source file.

Create a folder and save a C file

A **folder** groups files. A **filename extension** is the ending after the last dot in a filename. In `hello.c`, `.c` identifies a C source file.

1. Create a folder named `CPractice` inside your Documents folder.

2. Open Notepad and type the complete `tools_check.c` example shown above.

3. Choose **Save As** and open your CPractice folder.

4. For the filename, enter `tools_check.c`. In Notepad, choose **All files** as the file type so it does not add `.txt`.

5. Save the file. In File Explorer, turn on **File name extensions** so you can check that its name is `tools_check.c`, not `tools_check.c.txt`.

You can use another plain text editor if you already have one. Save code as text, not as a Word document.

Open PowerShell in the correct folder

Open your CPractice folder in File Explorer. Select its address bar, type `powershell`, and press Enter. This opens PowerShell in that folder.

You can check the files in the current folder with this command:

Terminal command - type this in your terminal

```
dir
```

Look for `tools_check.c` in the list. If it is missing, you may be in a different folder or may have saved it with another name.

Now build and run it:

Terminal command - type this in your terminal

```
gcc -std=c17 -Wall -Wextra -Wpedantic tools_check.c -o tools_check.exe
.\tools_check.exe
```

Run the second line only after the first line finishes without errors. The first line builds the program. The second line runs it. If the build succeeds, GCC may show no message at all. Chapter 3 explains the command and its parts.

Using macOS or Linux instead?

You only need the setup for the computer you use. The C source examples stay the same, but installation and run commands can differ.

Computer	Starting point
macOS	Install Apple's Command Line Tools with `xcode-select --install`, then follow its on-screen instructions. Check `clang --version` afterwards.
Ubuntu or Debian Linux	Use `sudo apt update`, then `sudo apt install build-essential`. These commands may ask for your administrator password. Check `gcc --version` afterwards.
Another Linux distribution	Follow that distribution's instructions for installing a C compiler. Its installation command may be different.

After opening a terminal in the folder containing your file, use these commands on macOS:

Terminal command - type this in your terminal

```
clang -std=c17 -Wall -Wextra -Wpedantic tools_check.c -o tools_check
./tools_check
```

On Linux with GCC installed:

Terminal command - type this in your terminal

```
gcc -std=c17 -Wall -Wextra -Wpedantic tools_check.c -o tools_check
./tools_check
```

`./` means "run the program in this folder" in these shells. In the Windows PowerShell examples, we use `.\` and an `.exe` filename instead.

If the setup does not work

What you see	What to check first
gcc is not recognised	Check that GCC is installed, the Path entry points to its bin folder, and you opened a new PowerShell window.
No such file or directory for tools_check.c	Run dir and check the folder and exact filename.
The file is tools_check.c.txt	Save or rename it with the real .c ending, then try again.
Source-code errors with line numbers	Open the source file and compare it with the complete example. Fix the first useful error first.
The program shows an older message	Save the source file and build again before running.

If a build fails, do not assume that an older `.exe` file contains your new changes. Fix the error and rebuild first.

Questions and quick revision

Answer: Does installing VS Code alone give me GCC?

No. VS Code is an editor. You still need a compiler unless your chosen setup installs one separately. In this chapter, MSYS2 installs GCC and an editor saves the code.

Answer: Why do we use hello.c rather than hello.docx?

A C compiler expects source text. The .c ending identifies a C source file. A Word document is a different file format and is not our C source file.

Answer: What is the sign that I can move to the next chapter?

You can run the small tools_check program and see its message. If you used the online option, that is enough to begin Chapter 3. If you chose local setup, also check that you can find, save, build and run the file in your own folder.

Remember the order: write, save, build, then run. Editing the source and running an older built program are different actions.

Chapter 03: Your first C program, one line at a time

You will now write a small program that displays a message. We will read every part, including the symbols. You do not need to guess what a new word means.

Write and run hello.c

Open your editor, create a file named `hello.c`, and type the following program. You can also paste it into the online C editor you used in Chapter 2.

hello.c

```c
#include <stdio.h>

int main(void)
{
    printf("I am learning C.\n");
    return 0;
}
```

Output

```
I am learning C.
```

This is a **complete program**. You do not need to add other code around it. Do not type the filename or the Output heading into your source file.

On Windows, save the file in CPractice, open PowerShell in that folder, and enter:

Terminal command - type this in your terminal

```
gcc -std=c17 -Wall -Wextra -Wpedantic hello.c -o hello.exe
.\hello.exe
```

Run the second command only if the build succeeds. On Linux, use the same build command with `-o hello` and then run `./hello`. On macOS, use `clang` in place of `gcc`, build with `-o hello`, and then run `./hello`.

What does #include <stdio.h> mean?

Our program uses a ready-made function named `printf` to show text. A **function** is a named piece of work that a program can ask to do a job.

A **header** supplies information about functions and other items we can use. `stdio.h` is the standard input and output header. "Input" means information going into a program. "Output" means information the program gives out.

`#include <stdio.h>` tells a build tool to include that header's information. This gives the compiler the information it needs about `printf`.

Read the line as: "include the standard input and output header." Keep the `#`, the spelling `stdio.h`, and the angle brackets `< >` as shown. There is no semicolon at the end of this line.

We do not need to learn other headers yet. Also, this line does not mean that all C programs must print something. We include this header here because these examples use `printf`.

What does int main(void) mean?

`main` is the starting function for the ordinary programs we run in these lessons. When we run `hello`, its work begins in `main`.

Part	Its job here
int	Says that main returns a whole number when it finishes.
main	The name of the starting function.
(void)	Says that this form of main takes no arguments.

An **argument** is a value supplied to a function when you call it. You will see text supplied to `printf` on the next line. This form of `main` does not receive such values.

"Returns a number" does not mean "prints a number." A returned value goes back to whatever called the function. We will connect `int` with `return 0;` shortly.

What do the braces mean?

The symbols { and } are called **braces**. They group the instructions that belong to `main`.

The opening brace starts the function's body. The closing brace ends it. Here, **body** just means the code inside the braces.

Code fragment - use within a complete program

```
{
    printf("I am learning C.\n");
    return 0;
}
```

This smaller piece is an **example fragment**, not another complete program. It shows the body so you can recognise its boundaries.

The spaces before the two lines make them easier to read. This spacing is called **indentation**. It helps your eyes follow the code; it does not replace either brace.

What does printf do?

`printf` writes output. In our example, that output appears in a terminal or the online editor's output panel. Here, "print" means show text; it does not mean use a paper printer.

Code fragment - use within a complete program

```
printf("I am learning C.\n");
```

The parentheses ( ) hold what we supply to the function. The text between the double quotes is a **string**: a sequence of characters, such as letters, spaces and punctuation.

The quotes tell C where the text starts and ends. They are not shown around the message in the output.

`printf` is a library function: ready-made work provided by the C library. It is not a keyword that you are learning to invent or rename.

What does backslash n do?

`\n` inside the quoted text means **start a new output line**. It is a backslash followed by the letter n.

The program does not print the visible characters `\n` in this example. It moves to the next line. This makes the output finish neatly before another message or the next terminal prompt.

A **prompt** is the text a terminal shows when it is ready for your next command. It is separate from your program's message.

Do not replace the backslash with a forward slash. `/n` is ordinary text and does not request a new line.

Why is there a semicolon?

The symbol `;` is a **semicolon**. It ends a simple statement, such as this printing statement.

A **statement** is a piece of code that describes an action. In this program, asking `printf` to write the message is one statement. `return 0;` is another.

Keep the semicolon after the closing parenthesis, outside the quotes. Do not add a semicolon after the opening brace to replace a missing one later.

As we learn other kinds of statements, we will see other shapes. For now, check the semicolons on the `printf` and `return` lines shown here.

What does return 0 mean?

`return 0;` finishes `main` and returns the number zero. For these programs, zero reports **successful completion** to the environment that ran the program.

It does not display a zero on the screen. We see only the message because the `printf` line writes that message.

Returning zero also does not prove that every calculation in a larger program is correct. It is the success status chosen by the program. We still need to check its results.

Read the whole program in order

1. The build uses the header information about printf.
2. Running the built program starts main.
3. The opening brace starts main's body.
4. printf writes the message and a new line.

5. return 0 finishes main with a success status.

6. The closing brace marks the end of the body in the source.

The closing brace is a boundary in the written code, not an extra action that prints something after return.

Understand the build command

Build means prepare a program that can run from the source you saved. In these small examples, the compiler command handles the needed build steps for us.

Part of the command	Meaning
gcc	Start the GCC compiler tool.
-std=c17	Use the C17 language version for these lessons.
-Wall -Wextra -Wpedantic	Ask for useful warnings about possible problems and nonstandard code.
hello.c	Read this source file.
-o hello.exe	Give the built Windows program this filename.

A **warning** is a message about something worth checking. An **error** is a problem that can stop this build. We aim to fix both in our examples.

Building involves several steps. You may later see the names preprocessing, compiling, assembling and linking. They prepare the source, translate it, and join the needed parts into a program. You do not need to run these steps separately for `hello.c`.

The run command `.\hello.exe` does a different job from the build command: it starts the program that was built. If you edit the message, save and build again before running.

Solve a small problem: a college notice

Problem: show a notice telling students where and when to meet. Put the place and time on separate lines.

Plan: choose the two messages, then use one printing statement for each line. We do not need input or calculations for this job.

college_notice.c

```c
#include <stdio.h>

int main(void)
{
    printf("Meet in Room 12.\n");
    printf("Time: 10 AM.\n");
    return 0;
}
```

Output

```
Meet in Room 12.
Time: 10 AM.
```

The first `printf` shows the place. Its `\n` moves to the next line. The second shows the time. Both run before `return 0;` finishes the program.

Save it as `college_notice.c`. In the build and run commands, replace the name `hello` with `college_notice`.

Common mistakes

Mistake	Fix
Typing Printf with a capital P	Use printf in lowercase. C treats these names differently.
Forgetting a closing quote	Check that the message has one opening and one closing double quote.
Leaving out the semicolon after printf	Put ; after its closing parenthesis.
Leaving out the last brace	Check that main has both { and }.
Editing the message but seeing old output	Save the source, build successfully again, then run.

You do not have to fix all mistakes at once. Begin with the first useful error message and compare that area with the complete example.

Practice with explained answers

Change the first program so it shows your own name. Then write a program that shows a name and a course on two separate lines. Try it before opening the answer.

Answer: One complete answer: a student introduction

Use two printing statements, each ending its message with \n. This example uses Asha and BCA. You can change those words inside the quotes.

student_intro.c

```c
#include <stdio.h>

int main(void)
{
    printf("Name: Asha\n");
    printf("Course: BCA\n");
    return 0;
}
```

Output

```
Name: Asha
Course: BCA
```

Answer: Does return 0 print 0?

No. It finishes main with the success status zero. printf is the function writing our visible output in these examples.

Answer: What is the difference between hello.c and hello.exe?

hello.c is the source text you can edit. hello.exe is the Windows program built from that source. After editing hello.c, save and rebuild to update hello.exe.

Before moving on, explain the header, main, braces, printf, quotes, \n, semicolons and return 0 in your own words. Keep the example beside you if needed.

Chapter 04: Writing rules, output and comments

You have run a small program. Now we will look at the writing rules that help C understand your code, and the notes that help a person read it.

What is syntax?

Syntax means the rules for writing code.

For example, the text given to `printf` goes inside double quotes. Its parentheses must close, and its printing statement ends with a semicolon. If we miss a required symbol, the compiler may not be able to build the program.

Think about a form that needs a date in a particular order. The information matters, but so does how you write it. C also has a required form for its instructions.

Correct writing rules are only one check. A program can build and still calculate the wrong total. We will check both the code and the result of our examples.

Recognise the small pieces of code

Look at this printing statement:

Code fragment - use within a complete program

```
printf("Class starts soon.\n");
```

Piece	What it does
printf	Names the function we want to use.
(and)	Enclose what we supply to the function.
"Class starts soon.\n"	Supplies the text, including a new-line request.
;	Ends this printing statement.

The small meaningful pieces of C code are called **tokens**. You do not need to memorise every token category now. Start by recognising the name, text and symbols in this one line.

A **keyword** is a word with a special meaning in C, such as `int`, `void`, `return` or `const`. We cannot use a keyword as the name of our own variable. Chapter 5 explains variable names.

`printf` and `main` are function names, not C keywords. They still have particular jobs and rules in our examples. Calling something "not a keyword" does not mean we can change its spelling and expect the same result.

Capital letters, spaces and line breaks

C is **case-sensitive**. This means capital and small letters matter. `printf` is not the same name as `Printf`.

Spaces help separate words. For example, `int main` has two words. Joining them into `intmain` changes what the compiler sees.

We use blank lines and indentation to make a program easier to read. Extra spacing between many pieces of code is allowed, but spacing inside quoted text is part of that text. Compare `"Class starts"` and `"Classstarts"`: the output messages are different.

Keep each `#include` line in the form shown in the examples. A line starting with `#` has its own writing rules; do not join the whole program onto that line.

Print one line or several lines

`printf` does **not automatically add a new line** after every call. We choose where to put `\n`.

output_lines.c

```c
#include <stdio.h>

int main(void)
{
    printf("Good ");
    printf("morning!\n");
    printf("Bring your notebook.\n");
    return 0;
}
```

Output

```
Good morning!
Bring your notebook.
```

The first call writes `Good ` , including a space at the end. It does not move to a new line. The second call continues on that line, then ends it. The third call starts after that new line.

Without the space after `Good`, the first two words would join. Without the first `\n`, the next message would continue on the same line too.

Show quotes, a backslash or a percent sign

Sometimes we want a symbol that already has a job in C's writing rules. An **escape sequence** uses a backslash inside text to represent a character in a special way.

Write inside a C string	Meaning
\n	Move to a new line.
\t	Insert a tab gap; its displayed width depends on the output window.
\"	Include a double quote as part of the message.

Write inside a C string	Meaning
\\	Include one backslash.

For a percent sign in a printf format, write %%. This is a printf rule, not a backslash escape sequence. We will learn more about percent formats when we print variable values in Chapter 5.

special_text.c

```c
#include <stdio.h>

int main(void)
{
    printf("She said, \"Practise today.\"\n");
    printf("Folder: C:\\CPractice\n");
    printf("Attendance: 100%%\n");
    return 0;
}
```

Output

```
She said, "Practise today."
Folder: C:\CPractice
Attendance: 100%
```

The escaped quotes belong to the message. Each \\ in the source produces one visible backslash. The %% produces one percent sign. The final \n on each line moves the output to a new line.

We are only displaying the folder's name. This program does not create that folder or check whether it exists.

What is a comment?

When you want to leave a note beside your code, write a **comment**. It helps someone reading the file understand your choices. The note is not a step that the program runs.

For example, a comment can explain why a message is shown. A useful comment adds information. It does not need to repeat every obvious word in the next line.

There are two comment forms in the C version used by these lessons: a line comment and a block comment.

Single-line comments

Start a line comment with //. Outside quoted text, the rest of that source line becomes a comment. The next line can contain code again.

Code fragment - use within a complete program

```c
// Tell students what they need for the lesson.
printf("Bring your notebook.\n");
```

You can also place the comment after a statement on the same line:

Code fragment - use within a complete program

```c
printf("Room 12\n"); // Display the meeting place.
```

Do not put code after // on the same line and expect it to run. It is part of the comment.

Block comments

Start a block comment with /* and finish it with */. It can fit on one line or extend over several lines.

comments_demo.c

```c
#include <stdio.h>

int main(void)
{
    /* This notice is shown before a practical class.
       Keep the two messages easy to read. */
    printf("C practical today.\n");
    // Remind students about the material they need.
    printf("Bring your notebook.\n");
    return 0;
}
```

Output

```
C practical today.
Bring your notebook.
```

Neither comment is printed. The two printf statements produce the output.

Always close a block comment. Also, do not start another block comment inside one that is already open. C block comments do not nest: an inner-looking /* does not create a separate comment to close later.

Comment text and printed text are different

Comment symbols inside a string are just text in that string. The following program prints them:

comment_symbols.c

```c
#include <stdio.h>

int main(void)
{
    printf("// is written inside quotes here.\n");
    printf("/* this is printed text */\n");
    return 0;
}
```

Output

```
// is written inside quotes here.
/* this is printed text */
```

Look at whether the symbols are inside or outside the quotes. That difference tells you whether they belong to printed text or begin an actual source comment.

Read and fix a small mistake

The following complete example is intentionally incorrect. Look at the `printf` line before reading the answer.

Intentional error for study - missing_semicolon.c

```c
#include <stdio.h>
int main(void)
{
    printf("Remember your notebook.\n")
    return 0;
}
```

Answer: What needs fixing?

The printf statement is missing its final semicolon. Add ; after the closing parenthesis. The compiler's message may point near return because that is where it notices it cannot continue with the expected form. Check the line above an error location too.

There is no expected program output for the incorrect source. First fix the build error, then run the corrected program.

Common mistakes

Mistake	Better habit
Expecting each printf call to start a new line	Put \n where you want the line to end.
Writing /n instead of \n	Use a backslash before n.
Putting ordinary quotes inside a string without escaping them	Use \" for a quote that belongs to the message.
Leaving a block comment open	Check for its matching */.
Adding so many comments that the code is hard to follow	Explain a useful reason, not every obvious action.

Practice with explained answers

Write a complete program that prints `Class: C` on one line and `Room: 12` on the next. Add a comment explaining that it is a class notice. The comment must not appear in the output.

Answer: One complete answer: a class notice

The comment explains the purpose. Each printing statement ends its message with a new-line request.

class_notice.c

```c
#include <stdio.h>

int main(void)
{
    // Show students the class and room.
    printf("Class: C\n");
    printf("Room: 12\n");
    return 0;
}
```

Output

```
Class: C
Room: 12
```

Answer: Are syntax and comments the same thing?

No. Syntax means the writing rules for code. A comment is a note in the source. The symbols that start and end a comment must still follow C's writing rules.

Answer: Does a program that builds always give the right answer?

No. A program can follow the writing rules and still do the wrong work. For example, it might add numbers when the problem needs multiplication. Compare the actual result with the result you expected.

Remember: spelling and symbols matter, printf needs your new-line requests, and comments explain the source without becoming output.

Chapter 05: Variables and constants

Programs need values such as a student's age, a mark or a ticket price. In this chapter, you will give values useful names, print them and decide which ones should be allowed to change.

What is a variable?

A program can keep a value under a name so we can use it later. That named place is a **variable**.

Think of a box with a label on it. The label tells you what belongs inside. In a program, the name `age` can help us remember that the stored number is a person's age.

Code fragment - use within a complete program

```
int age = 18;
```

Here, the variable is named `age`, and its starting value is `18`. We use `int` because an age in this example is a whole number.

The box picture is a way to understand the idea. You do not need to learn computer memory addresses to use this first variable.

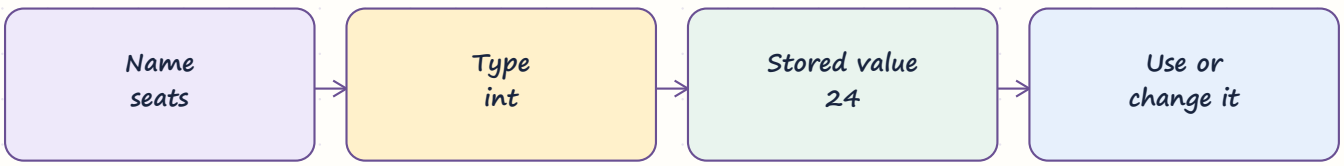
Read a declaration one part at a time

Declaring a variable means writing its type and name. A **data type** tells C what kind of value it can store.

Part of int age = 18;	Meaning
int	Use a whole-number type.
age	Give the variable this name.
= 18	Start it with the value 18.
;	End the declaration.

Giving a variable its starting value when you create it is called **initialisation**. In `int age = 18;`, the variable is declared and initialised on the same line.

The general pattern is `data_type name = starting_value;`. Those three words are placeholders for your choices; do not type them unchanged as if they were a working declaration.



Follow the idea from left to right. Check the chapter text for the exact C rules.

Print a variable's value

When we printed a fixed message, the words were inside quotes. To display the current value of an `int`, we use `%d` inside the quoted text and supply the variable after a comma.

student_age.c

```c
#include <stdio.h>

int main(void)
{
    int age = 18;
    printf("Age: %d\n", age);
    return 0;
}
```

Output

```
Age: 18
```

`%d` is a **format specifier**. That means it marks a place where `printf` should display a value in a chosen way. Here it asks for a whole number of type `int`.

The comma separates the format text from the value we supply. `printf` uses the value of `age`, which is `18`, to fill the `%d` place. The `\n` still ends the output line.

Writing `printf("age\n");` would print the letters age. It would not look up the variable, because those letters are inside the quoted text.

Change an ordinary variable

An ordinary variable can receive a new value. Giving it a value using `=` is called **assignment**.

available_seats.c

```c
#include <stdio.h>

int main(void)
{
    int seats = 30;
    printf("Before booking: %d\n", seats);
    seats = 29;
    printf("After booking: %d\n", seats);
    return 0;
}
```

Output

```
Before booking: 30
After booking: 29
```

The first line with `int` creates `seats`. The later line `seats = 29;` gives that existing variable a different value. We do not write `int` again when changing it.

After the assignment, the current value is `29`. The variable does not keep a list of all its earlier values for you.

Give a value before reading it

You can declare an ordinary variable and assign its first value on a later line, as long as you do so **before using its value**.

Code fragment - use within a complete program

```
int age;
age = 18;
printf("Age: %d\n", age);
```

For your early programs, prefer `int age = 18;`. It makes the starting value easy to see.

Do not declare `int age;` inside `main` and immediately print it. You cannot rely on it starting at zero or giving a useful value. Do not run such code and treat one observed number as a rule.

Choose useful names

The name you choose for a variable is also called an **identifier**. `student_age` explains a value better than a name such as `x` when the purpose is a student's age.

For the names we write in these lessons:

1. Use letters, digits and underscores.

2. Start with a letter. C allows an underscore too, but some underscore names are reserved, so start your own beginner names with a letter.

3. Do not put spaces or a hyphen in the name.

4. Do not use a C keyword, such as `int`, as the name.

5. Keep capital letters consistent. `age` and `Age` are different names.

Name	Suitable?	Reason
student_age	Yes	Describes the value and uses an underscore.
mark2	Yes	A digit is allowed after the first letter.
2mark	No	Starts with a digit.
student age	No	Contains a space.
student-age	No	A hyphen has a different job in C.
int	No	Already a C keyword.

Use one name for one variable in the same block. We will explain blocks and where names can be used in more detail later.

A first look at other data types

`int` is not the only choice. Here are four common types to recognise now. The later data-types chapter will explain the wider list, sizes, ranges and more examples.

Type	Kind of value	Example declaration	printf format here
int	Whole number	int students = 28;	%d
float	Number that can have a decimal part	float distance = 2.5f;	%f
double	Common choice for decimal calculations	double temperature = 24.5;	%f
char	One character-sized value	char section = 'B';	%c

Use single quotes around a character such as `'B'`. Use double quotes around a text string such as `"Section"`. One `char` is not a complete student name.

The `f` after `2.5` makes that written value a float. Without that suffix, a value such as `24.5` is a double. A double usually keeps more useful digits than a float. We call this **precision**: how much number detail a type can keep.

Both types can involve rounding. A displayed decimal number does not promise that every such value is stored exactly. The data-types chapter will explain their differences with more examples.

For `printf`, float values are passed as double values, so we use `%f` for both in this table. Input functions have different rules. This chapter is only showing output.

student_values.c

```c
#include <stdio.h>

int main(void)
{
    int students = 28;
    float distance = 2.5f;
    double temperature = 24.5;
    char section = 'B';
    printf("Students: %d\n", students);
    printf("Distance: %.1f\n", distance);
    printf("Temperature: %.1f\n", temperature);
    printf("Section: %c\n", section);
    return 0;
}
```

Output

```
Students: 28
Distance: 2.5
Temperature: 24.5
Section: B
```

The `.1` in `%.1f` asks for one digit after the decimal point in the output. It controls the display. It does not change the variable's type. Each printing line uses the format that matches its supplied value.

More than one variable

You can declare several variables of the same type on one line, separated by commas. Give each one its own starting value if it needs one.

Code fragment - use within a complete program

```
int first_mark = 35, second_mark = 40;
```

Writing each declaration on its own line is also correct and is often easier to read. Choose that form when you are unsure.

Copying an ordinary integer value gives the other variable its own value. It does not make the two names stay linked.

copy_a_value.c

```
#include <stdio.h>

int main(void)
{
    int first = 20;
    int second = first;
    first = 25;
    printf("First: %d\n", first);
    printf("Second: %d\n", second);
    return 0;
}
```

Output

```
First: 25
Second: 20
```

When `second` is created, it receives the current value of `first`: 20. The next assignment changes only `first`. This is like copying a number into a second notebook, then editing only the first notebook.

What is a constant?

A **constant** is a value that you do not want to change while a program runs. In C, use `const` when you declare a named value that must not change.

For example, an event's ticket price may be fixed at 50 rupees. The number of tickets can change, but the price stays the same for this program.

Code fragment - use within a complete program

```
const int ticket_price = 50;
```

The program can use `ticket_price` to work out a bill or show the price on the screen. After this declaration, an assignment such as `ticket_price = 60;` is not allowed. We call the value **read-only** because the program may use it but must leave it unchanged.

Part	Meaning
const	This value must not be changed.
int	It holds a whole number.
ticket_price	The name we chose.
= 50	Give it the starting value 50 now.

Part	Meaning
;	End the declaration.

Give this constant its starting value when you declare it. Do not leave the value out and try to fill it later with a separate assignment.

Capital names such as `TICKET_PRICE` are a common style, but lowercase `ticket_price` is allowed too. Capital letters alone do not protect a value. `const` provides that protection in this declaration.

Solve a problem using a constant

Problem: a college event charges 50 rupees per ticket. Calculate the total for three tickets.

Plan: keep the price fixed, store the quantity in an ordinary variable, then multiply them. The * symbol means multiplication in C.

ticket_total.c

```c
#include <stdio.h>

int main(void)
{
    const int ticket_price = 50;
    int tickets = 3;
    int total = ticket_price * tickets;
    printf("Ticket price: %d\n", ticket_price);
    printf("Tickets: %d\n", tickets);
    printf("Total: %d\n", total);
    return 0;
}
```

Output

```
Ticket price: 50
Tickets: 3
Total: 150
```

`ticket_price` starts at 50 and is protected by `const`. `tickets` starts at 3. The calculation `50 * 3` gives 150, which becomes the starting value of `total`.

Try changing the starting quantity to 4, then save, build and run again. The total should be 200. Change the quantity, not the constant price.

What if we try to change a constant?

This complete program is intentionally incorrect. It tries to change the fixed price after declaring it.

Intentional error for study - change_constant.c

```c
#include <stdio.h>

int main(void)
{
    const int ticket_price = 50;
    ticket_price = 60;
    printf("Price: %d\n", ticket_price);
    return 0;
}
```

The compiler reports an error on the assignment to `ticket_price`. It may describe the value as read-only. Different compilers can use different words for the same mistake.

There is no expected output of 60 for this incorrect example. Remove the assignment that changes the constant, then build again.

If a value really needs to change while the program runs, use an ordinary variable. You can also edit a constant's starting value in your source file for a new version of the program. Editing source and changing a value during a run are different things.

Another form you may see: #define

Some C code uses `#define` to give a fixed value a name. It replaces the name with the written value before the C code is compiled.

fixed_pass_mark.c

```c
#include <stdio.h>
#define PASS_MARK 40

int main(void)
{
    printf("Pass mark: %d\n", PASS_MARK);
    return 0;
}
```

Output

```
Pass mark: 40
```

Before compilation, `PASS_MARK` in the printing line is replaced with `40`. In this simple definition, there is no equals sign and no semicolon on the `#define` line.

`#define` replaces source text; it does not declare a variable. In our early exercises, use `const` for a named value that should stay fixed. We will explain more `#define` rules in a later chapter.

A value written directly in code, such as `40`, is often called a **literal**. The name `PASS_MARK` helps explain what that value is for.

Common mistakes

Mistake	What to do
Printing age before giving it a value	Give it a starting value first.
Writing printf("age") and expecting the number	Use printf("%d", age) for this int value.
Writing Age after declaring age	Keep the name's capital letters the same.
Expecting int first, second = 0; to set both to zero	Give each its own initial value, or use two clear lines.
Trying to assign a new value to a const value	Keep it fixed, or choose an ordinary variable if change is required.
Thinking an uppercase name is automatically constant	Use const in the declaration when you need read-only protection.

Practice with explained answers

A bus charges a fixed fare of 25 rupees. Write a program that finds the fare for four journeys. Use a constant for the fare and an ordinary variable for the number of journeys.

Answer: One complete answer: a bus fare total

The fare is fixed in this example. The number of journeys is allowed to change. Multiplying them gives the total.

bus_fare.c

```c
#include <stdio.h>

int main(void)
{
    const int fare = 25;
    int journeys = 4;
    int total = fare * journeys;
    printf("Total fare: %d\n", total);
    return 0;
}
```

Output

```
Total fare: 100
```

Answer: What is the difference between initialisation and assignment?

Initialisation gives a variable its starting value when it is created, as in int age = 18;. Assignment puts a value into an existing variable, as in age = 19;. Both use = in these examples, but happen at different points.

Answer: Should every value be constant?

No. Use const for a value that this program should not change, such as a fixed ticket price. Use an ordinary variable for a value that may change, such as the number of tickets.

Answer: Are variables only for values typed by a user?

No. Our examples start with values written in the source and values calculated by the program. A later chapter will teach how to ask a user for input and check it.

Remember: choose a useful name, give a value before reading it, match the printing format to its type, and use const for a named value that must stay fixed.

Chapter 06: Data types: choose the right kind of value

A class size, a distance and a section letter are different kinds of information. In this chapter, we will choose a C type for each one and learn how to display its value correctly.

Quick reference: C data types and how to print them

A **data type** tells C what kind of value a variable holds. Start with this table to see the choices together. You do not need to memorise it. The sections below explain the choices one at a time.

In the last column, the part beginning with % tells `printf` how to display the value. The table shows individual lines to use inside `main`, not separate complete programs. Include `stdio.h` above `main` for printing.

Type	Simple use	Example declaration	Print the value
`int`	An ordinary whole number	`int students = 32;`	`printf("%d\n", students);`
`float`	A number that may have a decimal part	`float walk = 1.25f;`	`printf("%.2f\n", walk);`
`double`	Our usual choice for decimal calculations	`double distance = 6.75;`	`printf("%.2f\n", distance);`
`long double`	Another decimal type; at least as much precision as double	`long double reading = 6.75L;`	`printf("%.2Lf\n", reading);`
`char`	One character-sized value	`char section = 'C';`	`printf("%c\n", section);`
`bool`	A true or false value	`bool open = true;`	`printf("%d\n", open);`
`signed char`	A small whole number that may be negative	`signed char change = -4;`	`printf("%hhd\n", change);`
`unsigned char`	A small nonnegative whole number	`unsigned char level = 200;`	`printf("%hhu\n", level);`
`short int`	A whole-number type for smaller ranges	`short int rooms = 24;`	`printf("%hd\n", rooms);`
`unsigned short int`	Nonnegative short integer	`unsigned short int seats = 120;`	`printf("%hu\n", seats);`
`unsigned int`	Zero or a positive whole number	`unsigned int pages = 600U;`	`printf("%u\n", pages);`
`long int`	A whole-number type with at least the range of int	`long int visitors = 90000L;`	`printf("%ld\n", visitors);`
`unsigned long int`	Nonnegative long integer	`unsigned long int files = 90000UL;`	`printf("%lu\n", files);`

Type	Simple use	Example declaration	Print the value
`long long int`	A whole-number type with at least the range of long	`long long int total = 5000000000LL;`	`printf("%lld\n", total);`
`unsigned long long int`	Nonnegative long long integer	`unsigned long long int count = 5000000000ULL;`	`printf("%llu\n", count);`
`void`	No value, such as a function with no returned result	`void show_notice(void);` declares a function	No value to print.

For `bool`, also include `stdbool.h` in these C17 programs. It supplies the names `bool`, `true` and `false`. The underlying type is `_Bool`. In this printing example, true appears as `1` and false as `0`.

`short`, `long` and `long long` are shorter spellings of the corresponding `int` types. Writing `signed int` means the same type as `int`; the word `signed` can also be written explicitly with short, long and long long. Plain `char` remains distinct from `signed char` and `unsigned char`.

***Signed** means the type can hold negative numbers as well as zero and positive numbers. **Unsigned** means it holds zero and positive numbers. Exact sizes and limits depend on the system; the names alone do not promise a particular number of bytes.*

*The `.2` in `%.2f` and `%.2Lf` requests two digits after the decimal point. Use a capital `L` for `long double` printing. These are **output formats for printf**. Input with `scanf` has different rules, explained in Chapter 9.*

Other type names you will meet later

The table above covers the ordinary number and character types, the true/false type and `void`. C also lets us build other types. Here is a first map, so you know where they belong. Their declarations and printing rules will be taught in their own chapters.

Type or form	What it is for	How we display its information
Array, such as `int marks[3]`	Several values of one element type	Print each element with that element's format; there is no single array format.
String stored in a `char` array	Text with an end marker; not a separate built-in string type	Use `%s` with a valid string that has its end marker.
Pointer, such as `int *place`	An address identifying where something is stored	For an object pointer's address, use `%p` and pass it as `(void *) place`.
Structure, declared with `struct`	Related values grouped together, such as a student's details	Print the individual members with their matching formats.
Union, declared with `union`	Members that share the same storage	Print the member whose value the program currently stores, using its matching format.
Enumeration, declared with `enum`	Named integer choices	In C17, convert an enum value to `int` for `%d` only when it fits in int. A name is not printed automatically.
Function type	Describes a function's result and parameters	Call the function and print its returned value if it has one. There is no direct function print format.

Type or form	What it is for	How we display its information
Complex floating-point types	Mathematical values with real and imaginary parts	Use the functions from `complex.h` to obtain and print the two parts. No single complex-number format exists.

The complex types are `float _Complex`, `double _Complex` and `long double _Complex`. They are a specialist topic; support can be optional in C17 implementations. You do not need them for our first exercises.

`typedef` gives an existing type another name; it does not create a new kind of value. For example, `size_t` is the unsigned integer type used for sizes, and `printf` uses `%zu` for it. We will use it with `sizeof` below.

Words such as `const` and `volatile` add rules to a type rather than introducing a new number family. C also has specialist atomic and optional imaginary types, which are outside these beginner exercises. Start with the ordinary types above, then learn the later forms when we reach them.

Why do we need different types?

A **data type** tells C what kind of value a variable can hold and which rules apply to it. We used `int` for whole numbers in Chapter 5. Now we will look at the other choices more carefully.

Imagine a college form with separate spaces for the number of students, the distance to college and the section letter. You would not write all three in the same way. A program also needs to know which kind of information it is working with.

For example, 32 students is a whole-number count. A distance of 3.75 kilometres needs a decimal part. A section such as `'C'` is a character.

Whole numbers with int

An **integer** is a whole number, such as 0, 32 or -6. Use `int` for ordinary whole-number values that fit its range. A **range** is the lowest through highest value a type can hold.

classroom_count.c

```c
#include <stdio.h>

int main(void)
{
    int registered = 32;
    int absent = 3;
    int present = registered - absent;
    printf("Registered: %d\n", registered);
    printf("Present: %d\n", present);
    return 0;
}
```

Output

```
Registered: 32
Present: 29
```

The minus sign in the calculation means subtract. We start with 32 and take away 3. `%d` displays each `int` value. We do not put single or double quotes around the numbers in their declarations.

An `int` cannot keep a decimal part. Do not choose it for a distance when the part after the decimal point matters. Chapter 8 explains what happens when a decimal value is converted to an integer.

Decimal values with float and double

A **floating-point type** can represent values with a decimal part. `float`, `double` and `long double` are C's real floating-point types. They have limits on size and precision; they cannot keep every possible decimal number exactly.

Precision means how much detail a number type can keep. `double` provides at least as much precision as `float`, and on common computers it provides more. We will usually choose `double` for decimal calculations in these lessons.

travel_details.c

```c
#include <stdio.h>

int main(void)
{
    float walk_km = 1.25f;
    double bus_km = 6.75;
    printf("Walk: %.2f km\n", walk_km);
    printf("Bus: %.2f km\n", bus_km);
    return 0;
}
```

Output

```
Walk: 1.25 km
Bus: 6.75 km
```

The `f` in `1.25f` makes that written value a `float`. The value `6.75`, without an `f`, is a `double`. The `.2` in `%.2f` asks for two digits after the decimal point when printing.

For `printf`, `%f` works with both examples: a `float` is passed to this function as a `double`. For a `long double`, use `%Lf`, with a capital `L`.

Printing two decimal places does not give a type extra storage precision. It only changes how the output looks. For money that must be exact to the paise, whole-number paise can be a better starting choice than an approximate decimal amount.

One character with char

A **character** is one item of text, such as a letter, digit or punctuation mark. A plain `char` holds one character-sized value. Put a character constant inside single quotes.

section_card.c

```c
#include <stdio.h>

int main(void)
{
    char section = 'C';
    char desk_label = '7';
    printf("Section: %c\n", section);
    printf("Desk label: %c\n", desk_label);
    return 0;
}
```

Output

```
Section: C
Desk label: 7
```

%c shows a character. '7' is the character used to write the digit seven; 7 without quotes is the number seven. Do not treat them as interchangeable in a calculation.

"C" uses double quotes and is a string. A **string** is text kept as a sequence of characters with an end marker. A single char is not a place for a full name. Some written characters, including many Indian-language characters, need more than one byte in common text encodings. We will return to text in the strings chapters.

True and false with bool

Sometimes a program needs to remember whether something is true or false: has a student paid a fee, for example?

For the C17 version used here, include stdbool.h to use bool, true and false. The underlying C type is named _Bool; the header supplies these easier names.

fee_status.c

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    bool fee_paid = true;
    bool form_submitted = false;
    printf("Fee paid: %d\n", fee_paid);
    printf("Form submitted: %d\n", form_submitted);
    return 0;
}
```

Output

```
Fee paid: 1
Form submitted: 0
```

Here, %d displays true as 1 and false as 0. It does not automatically print the English words. In Chapter 10, we will choose a message based on a true or false result.

Newer C versions change how some of these names are provided. Keep using the -std=c17 build command for these lessons so the examples follow one set of rules.

The integer family: short, long and unsigned

`int` is one member of a larger family. The following table gives the standard integer choices and small declarations you can recognise. The sample values fit the types; the table does not claim a particular byte size on every computer.

Type	Example declaration	printf format
signed char	signed char change = -4;	%hhd
unsigned char	unsigned char level = 200;	%hhu
short int	short int rooms = 24;	%hd
unsigned short int	unsigned short int seats = 120;	%hu
int	int students = 32;	%d
unsigned int	unsigned int pages = 600U;	%u
long int	long int visitors = 90000L;	%ld
unsigned long int	unsigned long int files = 90000UL;	%lu
long long int	long long int total = 5000000000LL;	%lld
unsigned long long int	unsigned long long int count = 5000000000ULL;	%llu

Signed types can hold negative and nonnegative numbers. **Unsigned** types hold zero and positive numbers. `int` means `signed int`; `short` means `short int`; `long` means `long int`. Plain `char` is a separate type and may behave as signed or unsigned on your compiler.

The letters at the end of constants help choose their types: `U` requests unsigned, `L` requests long, and `LL` requests long long. The combinations in the table match the declarations.

`long` is not guaranteed to be larger than `int` on every system. Also, an unsigned type does not reject a negative calculation with a helpful message. Mixing signed and unsigned values can give surprising results. Use ordinary `int` counts in our small exercises unless there is a clear reason for another choice.

Sizes, bytes and limits

A **byte** is a unit used to measure storage. A **bit** is a smaller unit that can have one of two values, usually written as 0 and 1. On common computers a byte has eight bits, but C does not require that exact number everywhere.

`sizeof` tells us the size of a type or value in C bytes. `sizeof(char)` is always 1. Other sizes depend on the compiler and system. Use `%zu` to print the result of `sizeof`; its result type is named `size_t`.

Code fragment - use within a complete program

```
printf("int uses %zu bytes here.\n", sizeof(int));
printf("double uses %zu bytes here.\n", sizeof(double));
```

This is a fragment to place inside `main` in a program that includes `stdio.h`. Run it on your own computer; do not expect all students to see the same sizes.

`limits.h` supplies integer limits such as `INT_MIN`, `INT_MAX` and `CHAR_BIT`. `float.h` supplies floating-point information such as `FLT_MAX` and `DBL_DIG`. The uppercase names are provided by the headers, not names you must invent.

Code fragment - use within a complete program

```
/* Add #include <limits.h> above main before using this fragment. */
printf("Smallest int here: %d\n", INT_MIN);
printf("Largest int here: %d\n", INT_MAX);
```

If a signed integer calculation goes outside its allowed range, the C rules do not promise a useful result. This is called **undefined behaviour**: we cannot rely on what the program will do. Choosing a larger type afterwards does not repair a calculation that has already gone wrong.

What about void, arrays and other types?

`void` expresses the absence of a value. In `main(void)`, it says that this form of `main` takes no arguments. Later, a function returning `void` will do work without returning a value. You cannot create an ordinary value-holding variable with `void item;`.

C also has arrays, pointers, structures, unions, enumeration types and function types. These let us work with groups of values, addresses and other arrangements. They each need their own explanation and will have separate chapters.

For completeness, C17 also defines complex floating-point types, such as `double _Complex`, for mathematical work with real and imaginary parts. Optional imaginary types are a further specialist area. We do not need these for the beginner exercises. `const` changes whether a value may be modified; it is not a separate number family.

Choose a type for a real problem

Suppose you are making a small practical-class card. You need a room number, a walking distance, a section and a yes/no flag for whether the lab is open.

Choose `int` for the room, `double` for the distance, `char` for the section, and `bool` for the flag. The reason matters more than memorising the choices: each type matches the information we want to keep.

Try writing these four declarations yourself. Choose your own small values, then display each one using the matching format. For now, display the flag as `0` or `1`.

Answer: One complete answer: a practical-class card

The two headers provide printing and the C17 Boolean names. Each declaration gives one value a clear name.

practical_card.c

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    int room = 16;
    double walk_km = 0.75;
    char section = 'D';
    bool lab_open = true;
    printf("Room: %d\n", room);
    printf("Walk: %.2f km\n", walk_km);
    printf("Section: %c\n", section);
    printf("Lab open: %d\n", lab_open);
    return 0;
}
```

Output

```
Room: 16
Walk: 0.75 km
Section: D
Lab open: 1
```

Common mistakes

Mistake	Better choice
Using int when a decimal part is needed	Choose a floating-point type and check the calculation too.
Using double quotes for a char value	Use single quotes for one character constant, such as 'C'.
Printing a long long with %d	Use %lld for that type.
Assuming every int has the same size	Use sizeof and the limits header on the actual system.
Expecting %.2f to make storage exact	It controls the displayed decimal places only.

Questions and quick revision

Answer: Is double always exactly eight bytes?

No. Eight bytes is common, but it is not a rule for every C system. Check sizeof(double) when you need the actual size. Do not choose program rules from one machine's output alone.

Answer: Can char store a whole student name?

One char cannot store a whole name. Later we will use an array of char for a text string. For now, a section letter is a suitable small example.

Answer: Which type should I learn first?

Start with int, double, char and bool, and recognise float. Learn why each example uses its type. Keep the larger integer table as a reference rather than trying to memorise every row today.

Choose the type for the information you need, use the matching print format, and do not assume all computers give a type the same size.

Chapter 07: Operators: calculate and compare values

You can already store values. Now we will use them to work out totals, find what is left, compare numbers and combine simple yes/no questions.

Types of operators: choose a topic

C has several groups of operators. Start with their jobs, then choose a heading to jump to its explanation. Read them in order if this is your first visit.

Arithmetic operators: calculate a value

Assignment operators: store or update a value

Increment and decrement: change a value by one

Comparison operators: compare two values

Logical operators: combine yes/no tests

Bitwise operators: work with individual bits

Conditional operator: choose one of two values

sizeof operator: find a storage size

Other operators: addresses, arrays and grouped values

Some books group these slightly differently. For example, ++ and -- may appear under arithmetic operators. The grouping helps us learn their jobs; it does not change how C works.

What is an operator?

In `8 + 4`, the `+` asks for addition. A symbol that requests work on values is called an **operator**. The values it works on are called **operands**. Here, `8` and `4` are the operands.

An **expression** is code that produces a value. `8 + 4` produces `12`. We can store that result in a variable or give it to `printf`.

Think about preparing a canteen bill. You may multiply the price by the quantity, add another item and compare the total with the money you have. These are different jobs, so C gives us different operators.

Arithmetic operators

Operator	Job	Small example	Result
+	Add	14 + 3	17
-	Subtract	14 - 3	11
*	Multiply	14 * 3	42
/	Divide	14 / 3	4 when both values are integers
%	Find the integer remainder	14 % 3	2

The remainder is what is left after making complete groups. If we put 14 notebooks into groups of 3, we make 4 complete groups and have 2 notebooks left.

notebook_groups.c

```c
#include <stdio.h>

int main(void)
{
    int notebooks = 14;
    int group_size = 3;
    int full_groups = notebooks / group_size;
    int left_over = notebooks % group_size;
    printf("Full groups: %d\n", full_groups);
    printf("Left over: %d\n", left_over);
    return 0;
}
```

Output

```
Full groups: 4
Left over: 2
```

Both values are integers, so / gives an integer result. It drops the fractional part toward zero: -14 / 3 is -4, and -14 % 3 is -2. For a nonzero remainder, its sign follows the first operand. We will mostly use nonnegative counts in our first grouping problems.

% needs integer operands. It is not a percentage operator. For a percentage calculation, multiply and divide using the required values.

Never divide by zero or use zero as the second value of %. Check a value before using it as a divisor. We will write those checks in Chapter 10.

Decimal division and a canteen bill

If one operand of / is a `double`, these small calculations use decimal division. For example, `14.0 / 3` gives a result close to 4.6667. Chapter 8 explains how to request this when your starting variables are integers.

canteen_bill.c

```c
#include <stdio.h>

int main(void)
{
    int snack_price = 18;
    int snacks = 3;
    int drink_price = 12;
    int total = snack_price * snacks + drink_price;
    printf("Bill: %d rupees\n", total);
    printf("Share for two: %.2f rupees\n", total / 2.0);
    return 0;
}
```

Output

```
Bill: 66 rupees
Share for two: 33.00 rupees
```

First, `18 * 3` gives 54. Adding 12 gives 66. The `2.0` in the final calculation asks for decimal division. The original `total` variable still holds an integer.

Assignment operators

`=` puts a value into a variable. It is **assignment**, not a question about equality.

Code fragment - use within a complete program

```c
int balance = 100;
balance = balance - 18;
```

After these lines, `balance` holds 82. The right side is calculated using the previous value, then the result is stored on the left.

For a simple variable such as `balance`, we can shorten this update to `balance -= 18;`.

Update	Same update written out for an ordinary int variable
balance += 5;	balance = balance + 5;
balance -= 5;	balance = balance - 5;
balance *= 2;	balance = balance * 2;
balance /= 2;	balance = balance / 2;
balance %= 3;	balance = balance % 3;

The variable's type still matters. `/=` on an `int` does not preserve a fractional part. Shorter writing does not make a calculation safer or change its number limits.

Increment and decrement operators

`++` increases a variable by one. `--` decreases it by one. Used as their own statements, `count++;` and `++count;` both increase the stored count.

When the expression's value is used, their timing differs:

counter_steps.c

```c
#include <stdio.h>

int main(void)
{
    int count = 6;
    int old_count = count++;
    int new_count = ++count;
    printf("Old count: %d\n", old_count);
    printf("New count: %d\n", new_count);
    printf("Stored count: %d\n", count);
    return 0;
}
```

Output

```
Old count: 6
New count: 8
Stored count: 8
```

`count++` supplies `6`, then updates `count` to `7` by the end of that statement. `++count` updates it to `8` and supplies `8`. The words **postfix** and **prefix** refer to placing the operator after or before the variable.

For early programs, keep an update on a separate line. Do not combine changes to the same variable in expressions such as `count++ + ++count`. C does not define a reliable result for that expression.

Comparison operators

The symbols `<`, `>`, `<=` and `>=` are also called **relational operators**. The symbols `==` and `!=` are **equality operators**. We learn them together because they all compare values.

A **comparison** asks a question about values. The result is the integer `1` for true or `0` for false. It does not change the compared variables.

Operator	Question
==	Are the values equal?
!=	Are they different?
>	Is the left value greater?
<	Is the left value smaller?
>=	Is the left value greater or equal?
<=	Is the left value smaller or equal?

compare_marks.c

```c
#include <stdio.h>

int main(void)
{
    int mark = 46;
    printf("At least 40: %d\n", mark >= 40);
    printf("Exactly 50: %d\n", mark == 50);
    printf("Different from 50: %d\n", mark != 50);
    return 0;
}
```

Output

```
At least 40: 1
Exactly 50: 0
Different from 50: 1
```

The mark remains 46. Be especially careful with = and ==: `mark = 50` changes a value; `mark == 50` compares it.

Logical operators

Logical operators combine or reverse true/false tests.

Operator	Read it as	When the result is true
&&	AND	Both tests are true.
two vertical bars	OR	At least one test is true.
!	NOT	The test after it is false.

For OR, type two vertical bars: ||.

lab_entry_checks.c

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    bool has_id = true;
    bool has_booking = false;
    printf("Both ready: %d\n", has_id && has_booking);
    printf("At least one ready: %d\n", has_id || has_booking);
    printf("Booking missing: %d\n", !has_booking);
    return 0;
}
```

Output

```
Both ready: 0
At least one ready: 1
Booking missing: 1
```

The last line reverses false to true. Logical tests treat zero as false and a nonzero number as true. Their result is still 0 or 1.

With &&, C skips the right test if the left test is false. With ||, it skips the right test if the left test is true. This is called **short-circuit evaluation**: once the answer is known, the other test is not needed. Put ordinary tests in conditions; keep surprising updates out of them.

Which calculation happens first?

Precedence describes how an expression is grouped when it contains different operators. Multiplication and division group more tightly than addition and subtraction.

`6 + 4 * 2` means `6 + (4 * 2)`, which is 14. `(6 + 4) * 2` is 20. Parentheses make the intended groups clear.

For arithmetic at the same precedence, such as `18 / 3 * 2`, grouping goes from left to right: `(18 / 3) * 2`, giving 12.

Comparison operators group more tightly than &&, and && more tightly than ||. Rather than memorising a large chart now, add parentheses around each question when combining several tests.

Grouping is not a promise that every function call in a larger expression runs left to right. We will use separate statements when the order of work matters.

Do not write `0 <= mark <= 100` to test a range in C. Write `mark >= 0 && mark <= 100`. The first form compares 0 or 1 with 100 after doing its first comparison, so it does not ask the intended question.

Bitwise operators

A **bit** is one small unit of information that can be 0 or 1. A whole number can be represented using a row of bits. **Binary** means writing numbers using only these two digits.

For example, the last four bits of 12 are `1100`. The last four bits of 10 are `1010`. Bitwise operators work on these positions, rather than treating the whole number as one yes/no answer.

Operator	Job	Example with unsigned values	Result
&	AND: keep a 1 where both positions are 1	`12U & 10U`	8
one vertical bar	OR: keep a 1 where either position is 1	See the program below.	14
^	XOR: keep a 1 where the positions differ	`12U ^ 10U`	6
~	Reverse each bit; our example uses unsigned int	See the low-four-bits example below.	Depends on the full type width.
<<	Shift bits to the left	`12U << 1`	24
>>	Shift bits to the right	`12U >> 1`	6

The OR symbol is one vertical bar: `|`. XOR means **exclusive OR**: one input bit is 1 and the other is 0. The `U` suffix in these examples requests an unsigned integer value.

For AND, line the last four bits up on paper: `1100` above `1010`. Only the leftmost of those four positions has a 1 in both rows. The result is `1000`, which represents 8.

bit_positions.c

```c
#include <stdio.h>

int main(void)
{
    unsigned int first = 12U;
    unsigned int second = 10U;
    printf("AND: %u\n", first & second);
    printf("OR: %u\n", first | second);
    printf("XOR: %u\n", first ^ second);
    printf("Left by one: %u\n", first << 1);
    printf("Right by one: %u\n", first >> 1);
    printf("Reversed low four bits: %u\n", (~first) & 15U);
    return 0;
}
```

Output

```
AND: 8
OR: 14
XOR: 6
Left by one: 24
Right by one: 6
Reversed low four bits: 3
```

`~first` reverses all the bits of the unsigned integer, not just four. The `& 15U` keeps only the last four positions, because 15 is written as `1111` in binary. This keeps the printed result small and independent of the unsigned integer's full width.

A **shift count** says how many positions to move. It must be nonnegative and smaller than the number of bits in the type used for the left operand. For beginner bit exercises, use unsigned values and small valid counts. Signed shifts have extra rules and can be unsafe when used without checking.

`&` and `&&` are different: `12U & 10U` gives `8`, while `12U && 10U` gives `1` because both numbers are nonzero. Similarly, `|` is bitwise OR and `||` is logical OR. Bitwise AND and OR do not provide the short-circuit behaviour of the logical forms.

Bitwise updates also have assignment forms: `&=`, `|=`, `^=`, `<<=` and `>>=`. For a simple unsigned variable, `flags &= 3U;` stores the result of `flags & 3U` back in `flags`. Chapter 34 will use larger binary diagrams and explain how these operators can store several small yes/no flags in one value.

Conditional operator

The **conditional operator** chooses one of two values. Its shape is `condition ? value_if_true : value_if_false`. It is also called the **ternary operator** because it works with three operands.

Think of choosing a worksheet pack. If a student has more than two subjects to revise, give a larger pack; otherwise give a smaller one. We can express that value choice without writing the same assignment twice.

worksheet_pack.c

```c
#include <stdio.h>

int main(void)
{
    int subjects = 3;
    int sheets = (subjects > 2) ? 8 : 4;
    printf("Sheets in the pack: %d\n", sheets);
    return 0;
}
```

Output

```
Sheets in the pack: 8
```

The condition `subjects > 2` is true, so the expression supplies `8`. If it were false, it would supply `4`. Only the chosen value expression is evaluated. `sheets` receives the chosen result.

Use this form for a short value choice. A decision with several statements is usually clearer with `if` and `else`, which Chapter 10 explains. Do not squeeze several nested choices into one line while learning.

sizeof operator

`sizeof` tells us how much storage a type or value occupies in **C bytes**. It is an operator, even though `sizeof(int)` looks like a function call. A byte is a unit of storage; its number of bits depends on the system.

character_size.c

```c
#include <stdio.h>

int main(void)
{
    char section = 'D';
    printf("Size of char: %zu C byte\n", sizeof(char));
    printf("Size of section: %zu C byte\n", sizeof section);
    return 0;
}
```

Output

```
Size of char: 1 C byte
Size of section: 1 C byte
```

`sizeof(char)` is always `1` in C. `sizeof section` reports the size of that `char` variable, not the size of the printed letter on your screen. Use `%zu` because the result has type `size_t`, an unsigned integer type used for sizes.

With a type name, use parentheses as in `sizeof(int)`. With a simple variable name, either `sizeof section` or `sizeof(section)` is allowed. For our ordinary variables, the expression inside `sizeof` is not run; `sizeof` does not mean print the variable or change it.

Other type sizes, such as `sizeof(int)`, can differ between systems. We will discuss an advanced exception involving variable-length array types when we reach arrays. For now, use sizeof to ask about storage rather than memorising one computer's sizes.

Other operators

Some operators belong to topics we have not learned yet. Here is a map of their jobs. The later chapters will explain complete examples, so you do not have to understand every declaration now.

Symbol or form	Job	Where we will learn it
Unary + and -	Apply a sign to one number, such as `-change`	Ordinary arithmetic examples in this chapter.
`(type)`	Request a type conversion, such as `(double) total`	Chapter 8: type conversion.
`&name`	Obtain the address of an object such as a variable	Chapter 9 introduces it for input; Chapter 23 explains pointers.
`*pointer`	Access the value identified by a pointer	Chapter 23: pointers.
`array[index]`	Choose one element of an array	Chapter 19: arrays.
`function(arguments)`	Call a function	We call printf already; Chapters 15-16 explain our own functions.
`.` and `->`	Select a member of a structure or union, directly or through a pointer	Chapters 27-29: grouped values.
Comma operator	Evaluate its left expression, then its right; supply the right result	Later flow and expression examples; prefer separate statements first.

The commas separating arguments in `printf("%d", count)` are separators, not examples of the comma operator. Likewise, the same symbol can have different jobs: `&` between integers does bitwise AND, while `&name` asks for an address. Check the surrounding code before deciding what a symbol means.

Some references call this last group **special** or **miscellaneous operators**. Those words mean "other jobs collected together," not a new rule that you have to memorise.

Practice: pack revision sheets

A class has 23 revision sheets. Each student pack needs 4 sheets. Find the number of complete packs and the sheets left over. Then print whether there are any sheets left.

Answer: One complete answer: revision packs

Divide for complete packs, use the remainder for leftover sheets, and compare the remainder with zero.

revision_packs.c

```c
#include <stdio.h>

int main(void)
{
    int sheets = 23;
    int per_pack = 4;
    int packs = sheets / per_pack;
    int remaining = sheets % per_pack;
    printf("Complete packs: %d\n", packs);
    printf("Sheets left: %d\n", remaining);
    printf("Any left: %d\n", remaining != 0);
    return 0;
}
```

Output

```
Complete packs: 5
Sheets left: 3
Any left: 1
```

Common mistakes and quick questions

Mistake	What to remember
Using % to calculate a percentage	It finds an integer remainder.
Writing = when asking whether values match	Use == for that question.
Using & in place of &&	Bit operations and logical tests are different.
Expecting 14 / 3 to keep decimals	Both operands are integers, so the result is 4.
Changing the same variable several times in one expression	Split the work into clear statements.

Answer: Does ++ always mean the same as adding one?

It always requests an increase of one for a suitable variable, but the value supplied by the expression depends on whether ++ is before or after the variable. Both forms must also stay within the type's limits.

Answer: Is zero the only false integer value?

Yes, for a condition. Both positive and negative nonzero integers count as true. Comparison and logical operators themselves produce 0 or 1.

Answer: Why do we keep using small numbers?

They let us check the result by hand and keep calculations within the chosen types. Later we will check ranges before doing work with values supplied from outside the program.

Before running a calculation, predict its answer. Check the operand types, grouping and divisor, then compare the actual output with your prediction.

Chapter 08: Type conversion: keep the value you need

Sometimes a calculation needs a different number type from the values we started with. We will use a shared travel cost to see when C converts a value for us and when we should ask for a conversion ourselves.

What does conversion mean?

Type conversion means taking a value of one type and producing a value of another type. For example, the integer 12 can become the `double` value 12.0.

Think about recording a whole-number count in a form that allows decimals. Twelve is still twelve, but the new form can also represent values such as 12.5. A conversion does not always keep all the information, so we need to check the destination type.

There are two useful names to learn. **Implicit conversion** happens under C's rules without a cast written in the code. **Explicit conversion** is one we request with a cast. A **cast** is a type name in parentheses placed before a value or expression.

A conversion C does for us

whole_to_decimal.c

```c
#include <stdio.h>

int main(void)
{
    int pages = 12;
    double pages_as_decimal = pages;
    printf("Whole-number value: %d\n", pages);
    printf("Decimal value: %.1f\n", pages_as_decimal);
    return 0;
}
```

Output

```
Whole-number value: 12
Decimal value: 12.0
```

`pages` is an `int`. When its value is used to initialise a `double`, C converts this small value automatically. The original variable is unchanged: `pages` is still an `int` holding 12.

Do not conclude that every integer fits exactly in every floating-point type. Large integer values may lose detail during conversion. Our example uses a small value that both types represent exactly.

Why a decimal result variable is not enough

Four students share a travel charge of 37 rupees. Each share is 9.25 rupees. Look carefully at this program:

share_without_cast.c

```c
#include <stdio.h>

int main(void)
{
    int charge = 37;
    int students = 4;
    double share = charge / students;
    printf("Calculated share: %.2f\n", share);
    return 0;
}
```

Output

```
Calculated share: 9.00
```

This program builds, but the result is not the share we wanted. First, C calculates `charge / students`. Both operands are `int`, so integer division gives `9`. Only afterwards is that `9` converted to `double` for `share`.

The decimal part was lost before the value reached `share`. Giving the result more decimal places with `%.2f` cannot bring it back.

Ask for decimal division with a cast

Convert one operand before dividing:

share_with_cast.c

```c
#include <stdio.h>

int main(void)
{
    int charge = 37;
    int students = 4;
    double share = (double)charge / students;
    printf("Each share: %.2f rupees\n", share);
    printf("Original charge: %d\n", charge);
    return 0;
}
```

Output

```
Each share: 9.25 rupees
Original charge: 37
```

Read `(double) charge` as "use this value as a double here." In this calculation, `students` is then converted to `double` too, and the division produces `9.25`.

The cast does not permanently change `charge` or its declared type. That is why we still use `%d` to print `charge` afterwards.

For a small fixed denominator you can also write `charge / 4.0`. That literal is already a `double`. With a denominator held in a variable, the cast makes our intention clear.

The position of a cast matters

Compare these three forms for `charge` equal to `37` and `students` equal to `4`:

Expression	What happens	Value
charge / students	Integer division	9
(double) (charge / students)	Integer division first, then conversion	9.0
(double) charge / students	Convert before division	9.25

The parentheses in the second row group the division before the cast. The third row converts the first operand before dividing. They are different calculations even though both use the word `double`.

A cast applied after information has been lost cannot recover it. Ask for the type you need before the operation that would lose the information.

The denominator must still be valid. A cast does not make division by zero acceptable. Use a nonzero number of students; Chapter 10 shows how to check this in code.

From a decimal value to an integer

Converting a finite floating-point value to an integer removes the fractional part **toward zero**. It does not round to the nearest whole number.

keep_whole_part.c

```c
#include <stdio.h>

int main(void)
{
    double reading = 6.8;
    double correction = -6.8;
    int whole_reading = (int)reading;
    int whole_correction = (int)correction;
    printf("Reading whole part: %d\n", whole_reading);
    printf("Correction whole part: %d\n", whole_correction);
    return 0;
}
```

Output

```
Reading whole part: 6
Correction whole part: -6
```

We choose casts here to show that removing the fractions is intentional. Assigning these values to `int` without casts would also perform a conversion, but that can hide a mistake. A compiler may warn about such a conversion.

For `-6.8`, moving toward zero gives `-6`, not `-7`. For `6.8`, the result is `6`, not `7`. If the problem needs rounding, that is a different job from a cast. We will study the appropriate mathematical functions later.

A cast is not a range check

An `int` has a limited range. If the whole-number part of a floating-point value is outside that range, casting it to `int` does not make it valid. The C rules do not define a reliable result for that operation.

Likewise, converting a large value to a smaller integer type can lose information. Signed and unsigned conversions have their own rules; unsigned values can wrap into their allowed range instead of reporting a friendly error.

For example, do not use `(unsigned int) negative_count` as a way to check that a count is nonnegative. Check the original value first. We will practise those checks with `if`.

Keep the starting value until you have checked it. A cast is a request for conversion, not proof that the conversion preserves what your problem needs.

Mixed types in a calculation

When a small `int` and a `double` are added, C converts the integer for that calculation:

mixed_calculation.c

```c
#include <stdio.h>

int main(void)
{
    int full_km = 3;
    double extra_km = 0.5;
    double distance = full_km + extra_km;
    printf("Distance: %.1f km\n", distance);
    return 0;
}
```

Output

```
Distance: 3.5 km
```

The addition uses a converted value of `full_km`. The variable itself remains `int`. This example does not mean that C simply converts every pair of types to whichever looks "bigger." The full rules also consider signedness and the kinds of types involved.

For our early calculations, keep related whole-number values as `int` and related decimal values as `double`. Avoid mixing signed and unsigned values without understanding the conversion. Later lessons will revisit those cases.

A cast does not read a number from text

The character `'8'` is not the integer `8`. A cast such as `(int) '8'` gives the character's numeric code; it does not turn typed text into the number eight by understanding its meaning.

Similarly, a string such as `"37"` cannot be converted to a numeric value by putting `(int)` in front of it. Reading and checking numbers written as text needs an input or parsing function. **Parsing** means reading text according to a format to work out its value.

Chapter 9 introduces numeric input. Chapter 36 will teach stronger whole-line checking when input may be invalid or too large.

Practice: an average for three quizzes

A student scores 18, 21 and 23 in three small quizzes. Calculate the total and the average. Keep the marks and total as integers, but preserve the average's decimal part.

Answer: One complete answer: quiz average

The total is 62. We convert it before dividing by 3, so the average is approximately 20.6667. Printing two decimal places shows 20.67.

quiz_average.c

```c
#include <stdio.h>

int main(void)
{
    int first = 18;
    int second = 21;
    int third = 23;
    int total = first + second + third;
    double average = (double)total / 3;
    printf("Total: %d\n", total);
    printf("Average: %.2f\n", average);
    return 0;
}
```

Output

```
Total: 62
Average: 20.67
```

Common mistakes and quick questions

Mistake	Why it matters
Using a double result variable after integer division	The fraction is already lost.
Casting the completed integer division	It converts the truncated result.
Expecting an int cast to round 6.8 to 7	It removes the fractional part toward zero.
Using a cast to silence every warning	The warning may describe real information loss.
Assuming a cast changes a variable's declaration	It changes the value used at that point, not the original type.

Answer: Do I need to cast both operands for decimal division?

Not for our int and double examples. Converting one operand to double makes the other integer operand convert for that division too. Make sure the conversion happens before the division and that the divisor is not zero.

Answer: Does conversion always lose information?

No. Converting the small integer 12 to double preserves its value. Other conversions may lose a fraction, precision or range. Check the actual types and the values you expect.

Answer: Why do we still check output if the compiler accepts the code?

The share_without_cast program is valid C, but it does not solve our sharing problem correctly. A successful build checks writing and type rules; we still have to check that the calculation matches the problem.

Decide whether the answer needs a fraction. Check the types before calculating, place casts before information is lost, and never use a cast as a substitute for checking a value.

Chapter 09: User input: read and check a value

Until now, our values were written in the source file. We will now let someone type a value while the program is running. We will also check whether reading it worked before using it.

What changes when a user enters a value?

Suppose a study-room program needs today's number of visitors. If the number is written in the code, you must edit and rebuild whenever it changes. With **user input**, a person can supply the number during a run.

Input is information going into a program. In these examples, it normally comes from your keyboard. Output is information the program shows you. An online editor may have a separate Input panel instead of letting you type directly in its output area.

The steps are: show a request, read a value, check the reading, and only then use the value. The request is often called a **prompt**.

Meet scanf and the address sign

`scanf` is a library function declared in `stdio.h`. It reads input according to a format and tries to store the result in the variables we supply.

Code fragment - use within a complete program

```c
int visitors = 0;
int read_count = scanf("%d", &visitors);
```

Part	Its job
"%d"	Ask scanf to read a decimal integer for an int.
&visitors	Tell scanf where it may store that integer.
read_count	Keep the number of values successfully assigned.

The `&` before `visitors` means "the address of this variable." An **address** identifies where a value is stored. You do not need to memorise a computer address; `&visitors` supplies it for you.

`printf` needs the value to show, so we write `visitors`. `scanf` needs somewhere to put the value, so for this ordinary integer variable we write `&visitors`. We will explain addresses and pointers slowly in Chapter 23.

One complete program with a reading check

This program introduces one small `if` check. **If** means "do this work only when the condition is true." Chapter 10 will explain all the decision forms. Here, we need a check now so we do not teach ourselves to use failed input.

visitor_input.c

```c
#include <stdio.h>

int main(void)
{
    int visitors = 0;
    printf("Enter a visitor count from 0 to 100:\n");
    int read_count = scanf("%d", &visitors);
    if (read_count != 1)
    {
        printf("Please run again and enter a whole number.\n");
        return 1;
    }
    if (visitors < 0 || visitors > 100)
    {
        printf("The count must be from 0 to 100.\n");
        return 1;
    }
    printf("Visitors recorded: %d\n", visitors);
    return 0;
}
```

Output

```
Enter a visitor count from 0 to 100:
Visitors recorded: 24
```

For this sample, type 24 and press Enter. The Output block shows only what the program writes. Your terminal may also show the 24 you typed; that keyboard echo is not a second message from `printf`.

We ask for one value, so `read_count` must be 1. `!= 1` asks whether it is anything other than 1. If reading failed, the first block prints advice and stops with `return 1;`. For this program, we use 1 to report that it could not complete the requested work successfully.

The second check rejects a negative count or a count above 100. The `||` means OR. Only after both checks have been passed do we print the recorded count.

Initialising `visitors` to zero is a clear starting habit, but it does not prove that input worked. We still check `read_count`.

Try valid and invalid input

Run the program again for each row. Each run starts fresh; this program does not yet loop back to ask again.

Type and press Enter	What should happen
24	Record 24.
0	Record 0; the lower boundary is allowed.
100	Record 100; the upper boundary is allowed.
-1	Explain the allowed range and stop.

Type and press Enter	What should happen
101	Explain the allowed range and stop.
hello	Explain that a whole number was needed and stop.

`scanf` returns `0` for this letter input because it cannot read the requested integer. If input ends before a value can be read, it can return `EOF`, a special value supplied by the header. Checking `read_count != 1` handles both cases for this one-value request.

These are first input exercises, not a complete input checker. Use small numbers that fit in int. A numeric scanf conversion outside the target type's range is not safely handled just by checking its return value. Chapter 36 will use whole-line reading and range-checked conversion for arbitrary input.

There is another limit: input such as `24abc` can successfully read `24` and leave `abc` unread. Input `24.5` can read `24` and leave `.5`. A successful `%d` conversion does not prove the whole line contained only one integer. We will solve that when we learn stronger line checks.

Read two values and check both

To read two ordinary integer values, supply two `%d` formats and two addresses. The return value must be `2` when both assignments succeed.

study_groups_input.c

```c
#include <stdio.h>

int main(void)
{
    int first_group = 0;
    int second_group = 0;
    printf("Enter two group sizes, each from 0 to 50:\n");
    if (scanf("%d %d", &first_group, &second_group) != 2)
    {
        printf("Two whole numbers were needed.\n");
        return 1;
    }
    if (first_group < 0 || first_group > 50 || second_group < 0 || second_group > 50)
    {
        printf("Each group must have 0 to 50 students.\n");
        return 1;
    }
    int total = first_group + second_group;
    printf("Students together: %d\n", total);
    return 0;
}
```

Output

```
Enter two group sizes, each from 0 to 50:
Students together: 41
```

For this run, type `18 23`. You can also type `18`, press Enter, then type `23` and press Enter. Numeric conversions skip leading spaces and newlines while looking for their next value.

If you enter `18 hello`, only the first number is assigned, so the return value is 1. We must not use the second value as if reading had succeeded. Our check stops before the calculation.

We also check both ranges before adding. At most 50 plus 50 is 100, so the sum fits comfortably in `int`. Checking after a potentially overflowing calculation would be too late.

Reading decimals: %f and %lf are different here

For input, the format must match the receiving variable's type exactly.

Variable type	scanf format	Example argument
int	%d	&visitors
float	%f	&distance
double	%lf	&distance
long double	%Lf	&distance
char	%c	&section

The letter in `%lf` is a lowercase L. `%Lf` uses a capital L. For `printf`, we used `%f` for both float and double because float arguments are converted to double. `scanf` has to write into the receiving variable, so its rules are different.

temperature_input.c

```c
#include <stdio.h>

int main(void)
{
    double temperature = 0.0;
    printf("Enter a temperature, such as 26.5:\n");
    if (scanf("%lf", &temperature) != 1)
    {
        printf("A numeric temperature was needed.\n");
        return 1;
    }
    printf("Temperature entered: %.1f\n", temperature);
    return 0;
}
```

Output

```
Enter a temperature, such as 26.5:
Temperature entered: 26.5
```

Type `26.5` for this example. Notice `%lf` in the reading line and `%.1f` in the printing line. This small demonstration reports a reading; it does not check whether the temperature is suitable for a particular real device or process.

Reading a character after a number

Unlike numeric formats, %c does not skip leading whitespace on its own. **Whitespace** means characters such as spaces, tabs and newlines. A newline left after typing a number can be read by the next %c.

Use a space before %c when you want the next non-whitespace character:

Code fragment - use within a complete program

```
char section = 'A';
int read_count = scanf(" %c", &section);
```

The space in `" %c"` tells `scanf` to skip whitespace before reading one character. Check `read_count == 1` before using the character. This form is not suitable if the space or newline itself is the character you want to read.

Do not solve an input problem by calling `fflush(stdin)`. That is not a portable way to discard keyboard input in standard C. Later, we will read a whole line and process it deliberately.

What about names and sentences?

A full name needs more storage than one `char`. `%s` reads a word and needs a suitable character array with a width limit; it does not read a whole sentence with spaces. We have not learned arrays yet, so we will not quietly add an unsafe name-reading example here.

The strings chapter will introduce `fgets`, which reads a line into an array with a supplied size. The input-checking chapter will combine it with numeric conversion and checks for extra characters, range errors and long lines.

For now, practise numeric input with the stated small values and always check how many assignments succeeded.

Practice: record borrowed books

Ask for the number of books borrowed today. Accept a count from 0 to 6. If reading fails or the count is outside that range, print a helpful message and stop. Otherwise print the count.

Answer: One complete answer: borrowed books

We first check whether one integer was read. Then we check the allowed range. Neither check should be skipped.

borrowed_books_input.c

```c
#include <stdio.h>

int main(void)
{
    int books = 0;
    printf("How many books, from 0 to 6?\n");
    if (scanf("%d", &books) != 1)
    {
        printf("Please enter a whole number.\n");
        return 1;
    }
    if (books < 0 || books > 6)
    {
        printf("Use a count from 0 to 6.\n");
        return 1;
    }
    printf("Books recorded: %d\n", books);
    return 0;
}
```

Output

```
How many books, from 0 to 6?
Books recorded: 3
```

Type 3 for this run. Then test 0, 6, -1, 7 and a word in separate runs. The limits of numeric scanf discussed above still apply.

Common mistakes and quick questions

Mistake	What to do instead
Missing & before an ordinary numeric receiving variable	Supply its address, such as &visitors.
Ignoring scanf's result	Check the number of assignments you expected.
Reading into double using %f	Use %lf for double input.
Adding a newline at the end of the scanf format unnecessarily	Use the shown formats; trailing whitespace can keep waiting for more input.
Using a value even after input failed	Print advice and stop before using it.

Answer: Why does scanf sometimes seem to wait forever?

It may still be waiting for a requested value. A format ending with whitespace can also wait until a later non-whitespace character arrives. Use the simple formats shown here and check whether your online editor needs input supplied in a separate panel.

Answer: Does checking read_count mean all input is safe?

No. It checks how many assignments succeeded. It does not reject every extra character or safely handle every out-of-range number. That is why we limit these first exercises and later teach whole-line parsing.

Answer: Why do we learn a little if before Chapter 10?

Input can fail, and using a value without checking would teach a bad habit. Here, if means do this block when the check is true. The next chapter explains that choice step by step and adds else and other decision patterns.

Ask clearly, match the input format to the variable, check the reading result, check the allowed range, and only then do the work.

Chapter 10: Decisions with if and else

A program should not always do the same work. It may need to reject an invalid count, show a different message for an empty room, or choose a charge. We will teach it to make these choices one question at a time.

What is a condition?

A **condition** is a value or expression used to decide whether some work should happen. For our first examples, it is a question such as `seats > 0`: are any seats available?

In C, zero means false in a condition. A nonzero value means true. A comparison such as `seats > 0` produces `1` or `0`, so it gives a clear answer for a decision.

Imagine checking the notice outside a study room. If there is space, you may enter. Otherwise, you wait. The instruction you follow depends on the answer to a question.

Do work only when a condition is true

Use `if`, put the condition in parentheses, then put the work inside braces.

seats_notice.c

```c
#include <stdio.h>

int main(void)
{
    int seats = 4;
    if (seats > 0)
    {
        printf("Seats are available.\n");
    }
    printf("Check the room notice before entering.\n");
    return 0;
}
```

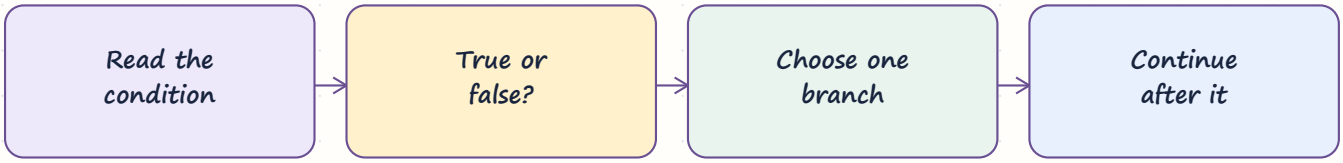
Output

```
Seats are available.
Check the room notice before entering.
```

The condition asks whether 4 is greater than zero. It is true, so C runs the printing statement inside the braces. The message after the closing brace is outside the `if` and runs afterwards either way.

Change `seats` to `0` and run again. The first message should disappear. The second should remain. This shows which statement belongs to the decision.

The general shape is `if (condition) { work }`. The words condition and work are placeholders. Use real C expressions and statements in their places.



Follow the idea from left to right. Check the chapter text for the exact C rules.

Choose one of two paths with else

`else` gives the work to do when the `if` condition is false. It does not have its own condition.

room_choice.c

```c
#include <stdio.h>

int main(void)
{
    int seats = 0;
    if (seats > 0)
    {
        printf("Choose an empty seat.\n");
    }
    else
    {
        printf("Please wait outside.\n");
    }
    printf("Keep the doorway clear.\n");
    return 0;
}
```

Output

```
Please wait outside.
Keep the doorway clear.
```

Zero is not greater than zero, so C skips the first block and runs the `else` block. Exactly one of those two blocks runs. Then execution continues after the whole decision.

Here is the path in words: check `seats > 0`; if true, show the seat message; if false, show the waiting message; then show the doorway reminder.

Keep every intended statement inside the braces

Braces group the statements belonging to a branch. **Branch** means one path the decision can take.

Code fragment - use within a complete program

```c
if (seats > 0) {
    printf("A seat is ready.\n");
    printf("Enter quietly.\n");
}
```

Both messages depend on the condition. C permits omitting braces around a single statement, but indentation alone does not group statements. We will keep braces even for one statement so later edits remain clear.

Do not write a semicolon immediately after `if (seats > 0)`. That would make an empty statement the controlled work. A following block could then run without being controlled by the condition you intended.

More than two choices with else if

Suppose a reading club uses three messages: no sessions attended, some sessions attended, or all four sessions attended. We can test the cases in order.

attendance_message.c

```c
#include <stdio.h>

int main(void)
{
    int sessions = 3;
    if (sessions == 0)
    {
        printf("Start with the next reading session.\n");
    }
    else if (sessions < 4)
    {
        printf("Keep going; some sessions remain.\n");
    }
    else
    {
        printf("All four sessions completed.\n");
    }
    return 0;
}
```

Output

```
Keep going; some sessions remain.
```

For this example, `sessions` is known to be from 0 to 4. First, `sessions == 0` is false. Next, `sessions < 4` is true, so that block runs. The final `else` is skipped. Once a branch is chosen, later alternatives in the chain are not tested.

If a value comes from input, validate its allowed range before using a chain that assumes it is valid. A negative session count would otherwise reach the wrong message here.

When testing thresholds, order matters. For grade-like categories, test the highest threshold first if lower checks would also be true. Trace one sample value through each condition before running the program.

Separate if statements are different

An `else if` chain chooses one branch. Two separate `if` statements ask two independent questions, so both blocks may run.

separate_checks.c

```c
#include <stdio.h>

int main(void)
{
    int minutes = 45;
    if (minutes >= 20)
    {
        printf("You have time for a short revision.\n");
    }
    if (minutes >= 40)
    {
        printf("You also have time for a practice problem.\n");
    }
    return 0;
}
```

Output

```
You have time for a short revision.
You also have time for a practice problem.
```

Forty-five satisfies both conditions. If we changed the second `if` to `else if`, its block would be skipped after the first true branch. Choose the form that matches the actual problem.

A decision inside a decision

*A **nested if** is an `if` inside another decision. It helps when a second question matters only after the first answer.*

lab_booking.c

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    bool registered = true;
    bool has_slot = false;
    if (registered)
    {
        if (has_slot)
        {
            printf("Your lab visit is ready.\n");
        }
        else
        {
            printf("Please choose a lab time.\n");
        }
    }
    else
    {
        printf("Complete registration first.\n");
    }
    return 0;
}
```

Output

```
Please choose a lab time.
```

The student is registered, so we enter the outer block. Only there do we ask about the slot. Since `has_slot` is false, the inner `else` prints the time-selection message.

Use braces and indentation to show which `else` belongs to which `if`. Without braces, C associates an `else` with the nearest unmatched `if`, which may not be the one you intended.

Combine tests and check a range

For a mark to be valid from 0 to 100, both checks must be true: `mark >= 0 && mark <= 100`. The `&&` means AND.

To reject a mark outside that range, use `mark < 0 || mark > 100`. The `||` means OR: being too small or too large is enough to reject it.

Do not write `0 <= mark <= 100`. C does not interpret a chained comparison like a mathematics sentence. Write each comparison completely and join them with a logical operator.

It can help to say the rule aloud first: "the count is below zero OR above the limit." Then translate each part into code. Parentheses can make longer combinations easier to follow.

Check before dividing

We can use short-circuit evaluation to ask about a divisor before doing a division. The right side of `&&` runs only when the left side is true.

enough_per_group.c

```c
#include <stdio.h>

int main(void)
{
    int sheets = 20;
    int groups = 0;
    if (groups > 0 && sheets / groups >= 3)
    {
        printf("Each group can receive at least three sheets.\n");
    }
    else
    {
        printf("Check the group count and available sheets.\n");
    }
    return 0;
}
```

Output

```
Check the group count and available sheets.
```

`groups > 0` is false, so `sheets / groups` is not evaluated. No division by zero takes place. Reversing the order would attempt the division too early. These fixed, small counts also avoid a number-range problem.

Another clear option is to reject an invalid divisor in a separate `if` and stop before reaching the calculation. Choose the form you can explain reliably.

Choose a value with the conditional operator

For a simple choice between two values, C provides `condition ? value_if_true : value_if_false`. This is the **conditional operator**, also called the ternary operator because it has three operands.

Code fragment - use within a complete program

```
int copies = 3;
int charge = (copies <= 5) ? 2 : 1;
printf("Charge per copy: %d\n", charge);
```

Here, `copies <= 5` is true, so the expression supplies 2. That becomes the starting value of `charge`. Only the chosen value expression is evaluated.

Use this for a short value choice, not for a long sequence of tasks. An ordinary `if/else` is easier for our first multi-step problems. Avoid nesting several `?:` choices while you are learning.

Solve a problem: choose a locker size

A campus desk lends two locker sizes. One or two bags need a small locker. Three or four bags need a large locker. Other counts are not accepted. Ask for a bag count, check it and choose a message.

Plan: check that reading one integer worked; reject counts outside 1 to 4; use a small-locker message for 1 or 2; otherwise use a large-locker message.

locker_choice.c

```
#include <stdio.h>

int main(void)
{
    int bags = 0;
    printf("Enter a bag count from 1 to 4:\n");
    if (scanf("%d", &bags) != 1)
    {
        printf("Please enter a whole number.\n");
        return 1;
    }
    if (bags < 1 || bags > 4)
    {
        printf("This desk accepts 1 to 4 bags.\n");
        return 1;
    }
    if (bags <= 2)
    {
        printf("Choose a small locker.\n");
    }
    else
    {
        printf("Choose a large locker.\n");
    }
    return 0;
}
```

Output

```
Enter a bag count from 1 to 4:
Choose a large locker.
```

Type `3` for the sample. The reading succeeds, `3` is within the accepted range, and `3 <= 2` is false. The last `else` chooses the large locker.

As in Chapter 9, use small representable integers for this first scanf exercise. It is not a whole-line parser. Chapter 36 will address oversized input and extra text.

Test the edges of a decision

A **boundary** is a value where a rule starts, stops or changes. Testing only `3` would miss several possible mistakes in the locker program.

Input	Expected message
0	This desk accepts 1 to 4 bags.
1	Choose a small locker.
2	Choose a small locker.
3	Choose a large locker.
4	Choose a large locker.
5	This desk accepts 1 to 4 bags.
a word	Please enter a whole number.

Run each case separately. Check both the text and whether an invalid case stops before any locker message. This is a useful testing habit for every decision you write.

Practice with explained answers

A room uses three notices based on the number of waiting students: `0` means no queue, `1` to `5` means a short queue, and `6` or more means a busy queue. For now, use a known nonnegative count written in the code. Write one chain that prints exactly one notice.

Answer: One complete answer: queue notice

First test the special zero case. Next test up to five. A remaining valid nonnegative count must be at least six.

queue_notice.c

```c
#include <stdio.h>

int main(void)
{
    int waiting = 5;
    if (waiting == 0)
    {
        printf("No queue.\n");
    }
    else if (waiting <= 5)
    {
        printf("Short queue.\n");
    }
    else
    {
        printf("Busy queue.\n");
    }
    return 0;
}
```

Output

```
Short queue.
```

Try 0, 1, 5 and 6. If you later read this count from a user, add the reading and nonnegative-range checks before this chain.

Common mistakes and quick questions

Mistake	What to check
Writing if (seats = 0)	= assigns; use == when comparing for equality.
Putting ; straight after the if condition	It creates an empty controlled statement.
Trusting indentation without braces	Braces decide which statements form the block.
Using separate if statements when only one result is wanted	Consider an if/else if/else chain.
Validating after a dangerous calculation	Check first, then calculate.

Answer: Can I use if without else?

Yes. Use it when some work is needed only for a true condition and no alternative work is needed. Code after the decision continues normally unless a statement such as return exits the function.

Answer: Is else compulsory after else if?

No, but without a final else, it is possible that none of the blocks runs. Add a final else when your problem needs a result for all remaining cases.

Answer: Are negative numbers false in C?

No. Zero is false; negative nonzero numbers are true too. Use a comparison such as count > 0 when you specifically mean a positive number, rather than just a nonzero number.

Answer: When will we learn switch and loops?

Chapter 11 will choose among named menu options with switch. Chapters 12 and 13 will repeat work with loops. A decision chooses a path; a loop makes work happen again.

State the rule in words, turn it into a clear condition, keep the work inside braces, and test values on both sides of every boundary.