DRAFT — unofficial version, compiled **11-11-2026** (**9d173f5**)

Introduction to Programming in C++

Learn the basic foundations of computer programming
Use classic C with some C++

Dr. Juan Falgueras Cano
Universidad de Málaga · 2026

Contents

Intro Computer Programming as an Art 3
0·1 But what is programming? 3
0·2 LLM or AI “assistants” 5
0·3 What Makes a Computer Programmable 5
0·4 Choosing a Language: Why C++? 6
0·5 The first exercise 8
Part 1 Computer Science and Programming 10
1·1 Computer Science and Programming 10
1·2 Functional Structure of Computers 11
1·3 Information Coding 13
1·4 Algorithms and Problem Solving 16
1·5 Programming Languages 21
1·6 General Overview of a Computer System 27
▸ Task 0: Editing and compiling our first C++ source file 30
Part 2 Introduction to a Programming Language 31
2·1 Introduction to C++ 31
2·2 Simple Data Types 34
2·3 Constants, Variables and Assignments 37
2·4 Characters and strings 48
2·5 Enumerating the values a type can hold 50
2·6 typedef: Defining new names for types 52
2·7 Random numbers 53
2·8 Operators precedence 53
2·9 Basic Input and Output 55
▸ Task 1: Basics. Sequential programs: arithmetic and standard I/O 58
2·10 Control Flow 60
2·11 Boolean Expressions 60
▸ Task 2: Booleans 64
2·12 Selection Structures 67
▸ Task 3: Control structures. Conditionals 76
2·13 Iteration Structures 79
2·14 Error and Exception Handling 84
2·15 Recommendations and Common Mistakes 86
▸ Task 4: Control structures. Single loops 89
▸ Task 5: Control structures. Double loops 93
Part 3 Procedural Abstraction 96
3·1 Top-Down Design 96
3·2 Procedures and Functions. Parameters 97
3·3 Recursion 112
▸ Task 6: Subprograms 120
▸ Task 7: Subprograms++ 124
▸ Task 8: Pre-structures 128
Part 4 Structured Data Types 130
4·1 Parameter Passing with Structured Types 131
4·2 Arrays 132
▸ Task 9: Arrays 141
▸ Task 10: Arrays++ 147
4·3 Strings of characters 152
▸ Task 11: Strings 157
4·4 Structs or records 166
▸ Task 12: Structures 171
4·5 Matrices 176
▸ Task 13: Matrices 180
▸ Task 14: Final exams problems 186
Appendix A Handling files 193
A·1 Declaring a handle to manipulate a file 194
A·2 Opening and closing the file 194
A·3 Reading and writing a text file 195
A·4 Streams as subprogram parameters 197
Appendix B Graphics, sound and WIMP with C++ 199
B·1 Getting the tools ready 199
B·2 **raylib**: drawing, animation and sound 200
B·3 **FLTK**: windows, buttons and events 204
Appendix C Writing Style 207
C·1 Great programmers comments 207
C·2 Style and Requirements 209

Intro Computer Programming as an Art

Contents of this part Book contents

0·1 But what is programming? 3
0·2 LLM or AI “assistants” 5
0·3 What Makes a Computer Programmable 5
0·4 Choosing a Language: Why C++? 6
0·5 The first exercise 8

0·1 But what is programming?

Have you ever wondered how computers operate? This is a beginner’s course designed for those learning to program for the first time.

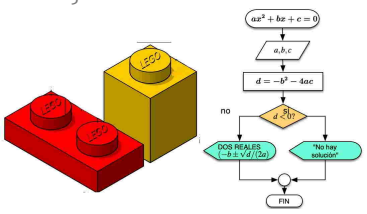
Computers execute programs, but what is programming?

> *Programming is the process of instructing a computer to perform tasks according to specifications expressed in a specific language*

Programming is fundamentally a creative endeavour. Once we have an appropriate mechanism in place, we can direct the computer to accomplish our intended objectives through multiple distinct approaches. Actions can be expressed in numerous different ways, and consequently, learning to program is analogous to learning an art form or acquiring a new language.

When we see a child using **LEGO** pieces to build a castle with those bricks and those many prefabricated pieces and figurines, we can get an idea of what programming really is. We can organise the pieces in many different ways and that just to build walls. We can be creative in the details, choosing different types of pieces, of different lengths and shapes, to build a specific thing, but we can also be creative in the organisation of the bigger parts knowing the final goal can be achieved with those modules or alternative ones. The main difference is that the final Lego construction is not meant to be understood by a computer, but by the child’s imagination.

1: Building a program is similar to assembling a LEGO model



When we artists paint a normal scene, we can do it in many different ways. Each one shows parts of the original, experience of us capturing it, which is always subjective. We also add the creativity and skill of the painter to show not only what is there, but also what the scene can convey through the paint. When we make a computer do something, even if it’s not as complex as nature, we can make it do different things in different ways. This is because we can add more layers to the actions, and we can choose from many ways to connect them, just like a child playing with Lego. So, in the end, the choice of them is an art that may reflect a subjective way of expressing our understanding of the process that the computer has to perform.

2: Las Meninas (Diego Velázquez), a masterpiece of art



James Flynn (Flynn 2014) argues that computer programming is as much an art form as a scientific discipline. Drawing on Paul Graham and Donald Knuth, he parallels coding with painting — both start from a blank space, refine toward a creative vision, and produce work for a human audience. Practices like live coding and the IOCCC contest further show how code can be expressive and aesthetic, not merely functional. Flynn concludes that programming is accessible to anyone with a computer and an idea: like a painter or composer, the programmer fills an empty page and, through iteration, arrives at something elegant and beautiful. Teaching someone to program is comparable to teaching them to paint masterpieces, albeit with a greater emphasis on semicolons and a reduced focus on berets. It could be argued that every creative discipline, including mathematics, incorporates a certain degree of artistry.

This is related to the idea that "there is no limit to how complicated everything can become, since one thing always leads to another". Thus, the complexity of the chain of actions can achieve unimaginable goals, whether they make sense or not.

For D. E. Knuth, programming is more than just a technical subject; it's a craft that combines maths skills, problem-solving creativity and human-oriented communication, creating things that can be judged by how beautiful, clear and efficient they are. This is the reason he called his 1974 essay in the journal **Communications of the ACM**: "Computer Programming as an Art".

When you're programming, you're basically choosing a final sequence of decisions and actions to reach a goal. And of course, there are dozens of ways to do it, each one with its own delightful mix of advantages, drawbacks, and "why on earth did I choose this approach?".

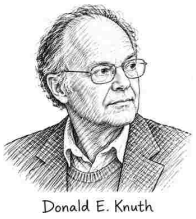
However, mathematical theorems are governed by different rules. The more complex the theorem, the fewer ways there are to prove it, and by "fewer" I mean "possibly only one, and good luck finding it". There is no computer to validate our proofs; the only hardware available is our own brain, which is notoriously flawed. The simplest theorems can be proven in many ways, but when things get complicated, the line between "deep understanding" and "an educated guess that we desperately hope is correct" becomes surprisingly thin.

You can ask an LLM to write your code for you—as long as you feed it a prompt that's detailed enough. At that point, the "creativity" mostly boils down to how cleverly you phrase your request. The model will happily fill in the blanks, churning out code stitched together from patterns it has seen before. Your job shifts from actually coding to vaguely describing what you want— basically the tech equivalent of pointing at a catalogue and saying, "*Yeah, I'll take that*," instead of grabbing tools and building the thing yourself.

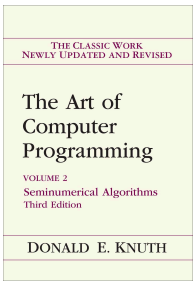
While this approach can **save time**, it does not necessarily produce better quality results. It will most likely eliminate better alternatives that would normally require more time to develop.

When programming in a programming language, we maintain focus on our objective for an extended period, which affords us the opportunity

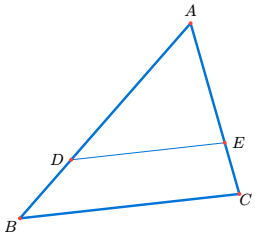
3: Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. —D.E. Knuth



4: This book (TACP vol 2, year 1969) by D.E. Knuth is and has been a reference for all subsequent books on programming.



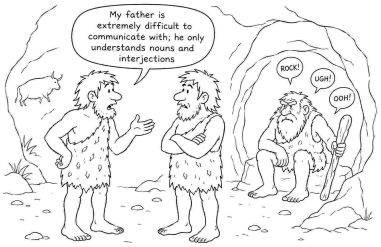
5: A mathematical theorem, such as Thales's theorem, is typically expressed in a highly specific manner and notation, quite distinct from the diverse ways that programming languages or natural language can represent it



$$\frac{AD}{AB} = \frac{AE}{AC} = \frac{DE}{BC}$$

to consider alternative approaches. Programming is typically a time-consuming activity, which explains the considerable success of large language models in assisting programmers with code development; however, reducing time is not always the optimal strategy. For a programmer, the time invested in building code is the time during which their mind adjusts and discovers—often unexpectedly—better alternatives, resulting in code that is ultimately more suited to the specific objective the programmer envisioned.

6: Choosing the correct language (syntax, grammar, etc.) is paramount to expressing ideas with precision



0·2 LLM or AI "assistants"

It is perfectly natural that these new assistants make all of us a bit uneasy: for you, because it can feel as if everything has already been done; for me, because I want you to leave this course still able to think things through on your own. We will use this course to practise exactly that skill.

How should we use them? Very carefully, and always in a way that strengthens, rather than replaces, your own thinking. In this course, our rule of thumb will be simple: during assessed work you must rely on your own understanding, and during practice you may use assistants only after you have first tried to solve the problem yourself and can explain what the tool suggested and why.

7: Some of the most popular LLM servers



What artificial intelligence can do for human thinking is worrying. On the one hand, it can reach solutions to extremely complex problems at astonishing speed; on the other hand, it risks dissolving our own thinking processes. For many people, if AI solves the problems that used to matter to them, those problems stop being meaningful, and the cognitive engagement that problem–solving once provided simply disappears.

In this course, I want you to discover that cognitive engagement with programming. If every challenge you set yourself is dissolved instantly by an assistant, you never give yourself the chance to discover that you are capable of solving it on your own. When you never have that experience, you gradually lose interest in the skill: it never becomes part of your own repertoire, it gives you no sense of power, and so you simply stop engaging with it.

Reading lists of programming exercises is the correct way to learn how to program in any language. The exercises serve a dual purpose at two levels. First, understanding the problem and what it asks is something you can accomplish relatively quickly. Second, in practice, developing the skills to solve it requires thought, though not necessarily arriving at a complete solution. However, you gain something valuable at each level.

0·3 What Makes a Computer Programmable

Computers can perform such a diverse range of tasks because they are programmable. This means that computers are not designed to perform a single, fixed function, but rather to execute any task specified by their programs.

When we turn on a light in our house, we flick a switch that connects the right wires, and magic happens. It's not much different from what we do when we program computers. Except that computers are a bit more complicated than a domestic switch. Computers also have wires, lots of

wires — so many that early computers needed whole buildings to house them.

8: Digital programming at its most fundamental level is the controlled switching of electrical signals between on and off states

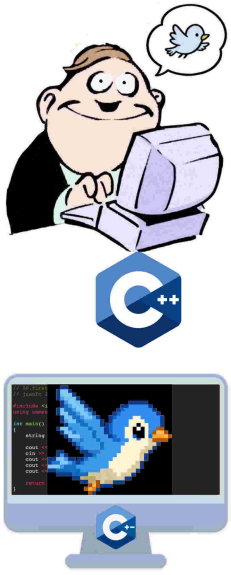


Modern computers, and even small digital smartwatches, are so complicated that they would need a whole old-style computer building full of wires to do the same thing. And even then they would be much slower, because size matters. Electrons take time to travel, and the shorter the distance, the less time it takes.

Another difference is that with computers we're not interested in switching lights on and off. Usually the first thing we want to do is display things on screens, with bright colours and lots of shiny information. So let's start by calculating something, like a sum, like `2 + 2`. Then we don't want to use our programming language to flick a switch, we want to add two numbers together and then display the result of that sum on the screen.

Conversely, the fundamental distinction of a computer from the intricate network of wires and switches discussed previously lies in its ability to propagate state changes automatically. When a connection is switched on or off, transistors follow hardwired logic to activate or deactivate other connections in response. Using the wires and switches analogy, transistors function as mechanisms that, when triggered, simultaneously connect and disconnect multiple additional components, which subsequently affect many others as a consequence. This cascading effect originates from a single initial action.

9: Programming: From Abstract Concept to Digital Implementation



This is not merely about actions, but also about states. After actions are executed, the resulting states persist, are stored in computer memory, and subsequently influence how further actions behave based on the previously recorded states.

When we write an instruction or a sentence comprising several of them, we are modifying memory states. We refer to these as variables, data structures, and so forth. Many operations produce effects on external devices and peripherals or transmit actions to other networked computers; consequently, once we execute them, we lose control over the result, as it lies outside our system.

Computing tasks can be expressed at various levels of abstraction. The highest level is human imagination, which generates concepts and scenarios that we articulate through natural language, eventually descending to programming languages. However, multiple programming languages exist to express computational actions, each with varying degrees of complexity. Some languages demand considerably more computational resources than others, depending on their abstraction level. Higher-level abstractions typically require more low-level instructions to execute.

0·4 Choosing a Language: Why C++?

The language that occupies us, C++, is a relatively low-level programming language. Each of its basic and predetermined instructions produces a relatively small number of elementary computer operations.

The following C++ program computes the sum $2 + 2$ and, more importantly, prints the result to the display:

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << 2 + 2 << endl;
  return 0;
}
```

However, a simple change such as this affects the interpretation and prints not the arithmetic sum, but rather the literal string enclosed within the quotation marks, namely: `2 +2`

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << "2 +2" << endl;
  return 0;
}
```

Several widely used programming languages feature similar syntax to this one, whilst others offer simpler, less verbose alternatives that support minimalist syntax:

```
puts 2 + 2
```

This tells the **computer** to “print the result of the sum 2 + 2 on the screen”. Relax.

Currently, there are many programming languages, some of them relatively modern, others classic, like C. Among the most popular in recent rankings are Python, C++, Java, C#, JavaScript, C, Go, Rust, TypeScript, and Swift. **20 Most Popular Programming Languages in 2026**

So it is clear that, like natural languages, talking to computers and asking them to do things is something “nationalistic”. There are no flags here, but languages define groups of people and therefore inevitably rivalry.

The reason for the existence of so many languages is that it is easy for programmers to define new languages and build translators for them. On the other hand, it is clear that some languages are better suited than others to talk about certain topics for which other languages are less expressive. If you want to build matrices, or work with text, or quickly construct complex agglomerations of information, the languages should follow different syntaxes.

C and C++ are classic programming languages that continue to be relevant today because they are efficient and give programmers precise control over what the computer does.

The simplicity of C comes from its close connection to how computers actually execute instructions. Its basic constructs correspond closely to fundamental operations such as storing values in memory and controlling the flow of execution. For this reason, C is widely used to write programs that must run very fast, such as parts of operating systems and other core software that supports more complex applications.

So the main advantages of choosing C as our learning language are its stability and simplicity. But, as you know, every good thing has its bad side. Its simplicity makes it harder to build complicated things for which

```cpp
// C++
#include <iostream>
using namespace std;

int main() {
  cout << 2 + 2 << endl;
  return 0;
}
```

```cs
// C#
using System;

class Program {
  static void Main() {
    Console.WriteLine(2 + 2);
  }
}
```

```java
// Java
public class Main {
  public static void main(String[]
  args) {
    System.out.println(2 + 2);
  }
}
```

```c
// C
#include <stdio.h>

int main() {
  printf("%d" << endl, 2 + 2);
  return 0;
}
```

```go
// Go
package main

import "fmt"
func main() {
  fmt.Println(2 + 2)
}
```

```rust
// Rust
fn main() {
  println!("{}", 2 + 2);
}
```

```python
# Python
print(2 + 2)
```

```javascript
// JavaScript
console.log(2 + 2);
```

```typescript
// TypeScript
console.log(2 + 2);
```

```swift
// Swift
print(2 + 2)
```

```ruby
# ruby
p 2 + 2
```

other languages have direct commands and ours requires a long list of delicate instructions. Stability has the disadvantage that its old syntax is difficult to adapt to modern ideas, since any changes to it would mean incompatibility with the huge pool of code already written in C.

0·5 The first exercise

The first exercise necessarily involves editing a simple program, compiling it (the process of translating our language, C++, into machine-readable code), and finally executing it to verify that it behaves exactly as intended.

To accomplish this, you must set up a programming environment on your computer. The effort required depends on your skill level and prior experience. Some of you may already have everything installed. Editing the source file is typically less demanding than the subsequent compilation and execution steps. When undertaking a new programming exercise, you will encounter three primary tasks:

1. Creating the source file. This is primarily a logistical task on your computer:
   a. Select an appropriate directory. Assuming you will be saving multiple C++ source files within it, the directory should be empty and have a descriptive name, such as `00.BasicExercises`
   b. You must initialise your programming environment. Your choice of environment depends on your computer architecture or operating system: macOS, Linux, or Windows. On macOS, I would recommend Terminal + Editor or Xcode. On Windows, **Code::Blocks** is suitable. On Linux, for those already familiar with it, numerous options exist, though Terminal + Editor remains perhaps the most instructive.
   c. Create the file `01.sum1to100.cpp` and save it (even if empty) in the directory, then write the code
2. Compile it, correcting any compiler errors as they arise. Remember to save after each correction, then recompile.
3. Execute the compiled application—your first program—which will be located alongside your source file.

After mastering these steps as a standard sequence for each exercise, attempt to modify the source program and repeat steps 2 and 3. Experimenting with modifications in this manner is the most effective way to understand what the program does.

Proceeding with the exercise, let us copy the following program (you do not need to understand it at this stage). Complete the comments at the top with the file name, your name or alias, the date, and, if you wish, what you think the program does:

```cpp
// sum1to100.cpp
// juanfc 2026-09-09
// This program sums all natural
// numbers from 1 up to 100

#include <iostream>
using namespace std;

int main() {
  int sum = 0;
  int i = 1;

```

```
  while (i <= 100) {
    sum = sum + i;  // add the current value of i to the sum
    i = i + 1;      // increment loop variable
  }

  cout << "The sum from 1 to 100 is: " << sum << endl;

  return 0;
}
```

The initial segment of every program invariably contains

```
#include <iostream>
using namespace std;

int main() {
```

whose purpose is to connect the environment in which we are developing our program: a computer terminal with keyboard and screen for input and output. It serves to include the necessary references to the operators required for interacting with these devices. We shall examine this in more detail later.

1. Modify the limit `100` to different values, recompiling and executing the program after each change. It may be more practical to reduce the limit to `10` or `5` to verify the results.
2. Attempt to sum not the natural numbers $(1 + 2 + 3 + ...)$ but their squares: $1^2 + 2^2 + 3^2 + ...$
3. Attempt to sum not all natural numbers but only the odd natural numbers: $1 + 3 + 5 + ...$

0·5.1 Checkpoint

1) What is the significance of the `//` symbol at the beginning of certain lines and following some statements in the C++ code?
2) The initial statements following program entry (after `int main()`) declare two variables, `sum` and `i`, and **assign** values to each (`0` and `1`). What would occur if we omitted this initialization, or if we assigned different values to these variables instead?
3) There is a lengthy statement concatenating multiple elements for output to the screen using `cout << ...`. Why are some parts enclosed in `"` (quotation marks) whilst others, such as `sum` or `endl`, are not? What distinguishes them?
4) The `return 0;` statement indicates to the operating system that the program executed successfully. Can you imagine how we might signal to the operating system that our program encountered problems upon exit?

References

FLYNN, James, 2014. Exploring computer coding as an art form. *The Varsity*. Online. 17 March 2014. Available from: thevarsity.ca [Accessed 11 June 2026].

Part 1 Computer Science and Programming

Contents of this part Book contents

1·1 Computer Science and Programming 10
1·2 Functional Structure of Computers 11
1·3 Information Coding 13
1·4 Algorithms and Problem Solving 16
1·5 Programming Languages 21
1·6 General Overview of a Computer System 27
▸ Task 0: Editing and compiling our first C++ source file 30

1·1 Computer Science and Programming

Computer Science is not the study of computers any more than astronomy is the study of telescopes:

> it is the science of **what**
> can be computed and **how**

10: Programming involves using computers to model problems and observe the results, much like how astronomers use telescopes to understand the universe



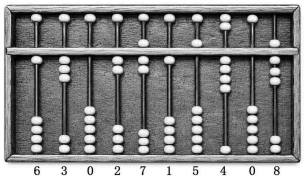
It is both a *science*, which studies what can be computed, and an *engineering* discipline, which studies how to build programs that compute it. On the other hand,

> **programming** is the craft that turns
> ideas into working artefacts,

and it will be our main tool throughout the course.

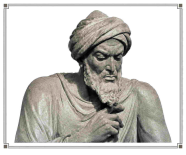
The wish to compute is much older than the machines. For thousands of years, merchants and astronomers did their arithmetic on the **abacus**, sliding beads along rods: a memory made of wood where fingers ran the procedure. The tool did not think; the person operating it followed fixed rules, step by step. That idea — *a mechanical procedure that anybody can repeat and always gives the right answer* — is the seed of everything in this course.

11: The abacus: a 4000-year-old "computer" whose processor was the person using it



In the 9th century, the Persian mathematician Muhammad ibn Musa **al-Khwarizmi** wrote down such procedures for arithmetic and equation solving. When his works were translated into Latin, his name became *algorismus* — the origin of our word **algorithm** — and the title of one of his books gave us the word *algebra*. So the central concept of computing was named after a person twelve centuries before the first computer existed.

12: Al-Khwarizmi (c. 780–850), whose Latinised name gave us the word: *algorithm*

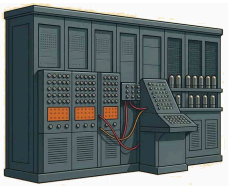


Machines slowly took over the mechanical part. Pascal (1642) and Leibniz built gear-driven calculators; in the 1830s Charles **Babbage** designed — though never finished — the *Analytical Engine*, the first machine intended to be programmable, and **Ada Lovelace** wrote what is usually considered the first program for it. She also made a remarkably modern observation:

such a machine could manipulate not only numbers but *any* symbols, music included.

The decisive steps came in the 20th century. In 1936, before any electronic computer existed, Alan **Turing** answered on paper the question of *what can be computed at all* — the theoretical foundation of Computer Science. A decade later, machines like the ENIAC (1945) filled entire rooms and were programmed by rewiring cables. The transistor, and then the microprocessor (1971), shrank those rooms onto a chip; the personal computers of the 1980s put one on every desk, and today the phone in your pocket is millions of times more powerful than ENIAC. What has never changed is what you need to make these machines useful: someone able to express a procedure precisely — a **programmer**.

13: ENIAC (1945): 30 tonnes, 18,000 vacuum tubes — programmed by rewiring cables



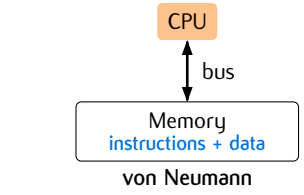
1·2 Functional Structure of Computers

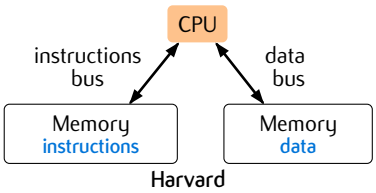
Almost every computer, from a washing machine controller to a supercomputer, follows the same architectural blueprint proposed by von Neumann: memory, processor, and input/output units connected by buses. Von Neumann's design has one defining trait: instructions and data share the same memory and the same bus. This is simple and flexible — a program can treat its own instructions as data, which is what makes compilers, loaders, and self-updating firmware possible — but it also means the processor cannot fetch an instruction and read a data value at the same time. They queue up for the same lane, one after another. This limitation is often called the *von Neumann bottleneck*.

14: John von Neumann, 1903-1957



15: The von Neumann bottleneck

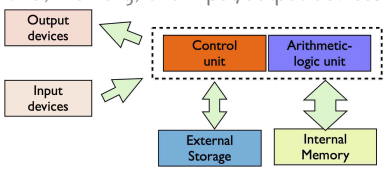




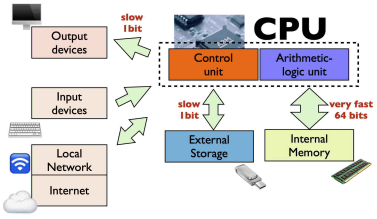
The classic alternative is the **Harvard architecture**, which keeps instructions and data in separate memories, each with its own bus. This is not just historical curiosity — it is the default choice in most microcontrollers and digital signal processors, including the 8-bit microcontrollers found in many Arduino boards, precisely because embedded systems value predictable, fast execution over the flexibility of a shared memory space.

Most modern general-purpose CPUs — those found in laptops and mobile phones — actually employ a compromise known as the **modified Harvard architecture**. Main memory is unified, as in von Neumann's model, but the processor core is flanked by two small, separate **caches**: one for instructions and one for data. The processor achieves Harvard-style parallel access for the vast majority of memory operations, which are served from these caches, while retaining the simplicity of a single address space for all remaining operations.

16: The von Neumann organisation: CPU, memory, and input/output devices



17: Kind of connections and relative speed for each of them



An important contemporary consideration is the degree of parallelism in the fetch-decode-execute cycle we will examine (Section 1·2.1). A classical von Neumann machine executes instructions sequentially on a single processor. Modern hardware, by contrast, amplifies this fundamental cycle: multicore processors execute multiple independent instruction streams concurrently, SIMD (single instruction, multiple data; you can read more about this here) units apply a single instruction to multiple data values simultaneously, and GPUs (graphics processing units) extend this to an extreme by employing thousands of simple cores optimised for precisely this kind of data-parallel computation, such as pixel rendering or neural network training. None of these architectures abandons the fetch-

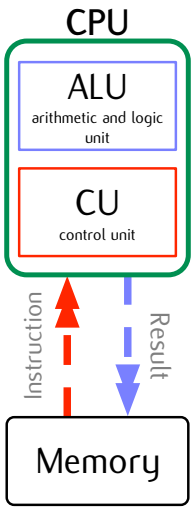
decode-execute principle—they simply execute many instances of it in parallel.

In computer science, parallelism and concurrency are two distinct concepts: a **parallel program uses multiple CPU cores**, with each core performing a task independently. **Concurrency**, by contrast, enables a program to handle multiple tasks even on a single CPU core; the core switches between tasks (that is, **threads**) without necessarily completing each one before moving to the next. A program can exhibit parallelism, concurrency, both, or neither. See Wikipedia

1·2.1 Internal Way of Working of Computers

When a computer executes a program, we say that the computer is running or executing that program. The central processing unit (CPU), as we have seen, is the component responsible for executing programs. The CPU is the most critical component in a computer because without it, the system cannot run software.

18: Main parts very simplified, of a von Neumann computer. Today CPUs are multicore: many of them working at the same time in parallel



In the earliest computers, CPUs were large devices constructed from electrical and mechanical components such as vacuum tubes and switches. Modern CPUs, known as microprocessors, are tiny integrated circuits small enough to fit in the palm of your hand. In addition to being considerably smaller than the electromechanical CPUs found in early computers, contemporary microprocessors are also significantly more powerful.

The CPU's function is to fetch instructions, decode them, and execute them to produce a result. Internally, the central processing unit comprises two principal components: the control unit (CU) and the arithmetic and logic unit (ALU). The control unit orchestrates all computer operations by determining the location of the next instruction and regulating other major computer components through control signals. The arithmetic and logic unit performs mathematical and logical operations as its name implies. The organisation of the CPU is illustrated in Figure 18. The fundamental cycle for the processor is fetching, decoding, and executing instructions.

Fetch The CPU's control unit retrieves the next instruction in the program sequence from main memory. The location of the instruction is determined by the program counter (PC), a critical pointer within the CPU. When programs start, the PC is positioned at the first instruction of the `int main() { @` ←. Numerous processor instructions enable the PC to jump from one location to another. Particularly important, as we shall see, are conditionals and loops, which cause the PC to jump to another part of the code in memory.

Decode The instruction, encoded as a numerical value, is decoded: the control unit interprets it and generates the control signals needed to carry it out.

Execute The control signals are routed to the appropriate hardware component (such as the ALU, a disk drive, or another device), which then performs the specified operation. The ALU itself can only execute basic arithmetic and logic operations (`+`, `-`, `*`, `and`, `or`, ...); if the CPU has a floating-point unit (as modern ones do), real-number arithmetic is carried out directly in hardware, otherwise it is delegated to software emulation.

Computer memory can be classified according to speed and capacity. Memory located closer to the CPU is faster but has lower capacity. The main categories are:

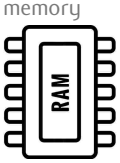
Register memory a small number of variables stored directly within the CPU, accessible at the CPU's clock speed (`A0`, `D7`).

RAM random access memory, which provides very fast access, though slower than registers, typically requiring several CPU cycles to read from or write to. It is accessed through memory addresses —a sequential enumeration of each byte in the computer's RAM (`0xAFFFFAFFFAFA` is a typical example of how to refer to a position in memory; we use hexadecimal notation).

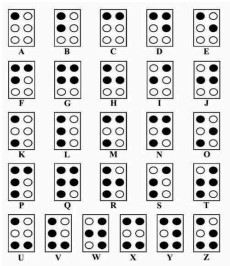
External memory connected via internal or external interfaces (e.g. SATA, USB), offering the largest capacities at the lowest cost. Content is organised in blocks called files. We do not concern ourselves with the positioning or distribution of those files on the devices; that task falls to the devices' own controllers. We need only know the name of the files (`01.myName.cpp`), the directories in which they are held (`/Users/myName/00.intro/01.myName.cpp`), and sometimes their attributes—whether they can or cannot be read by us, and so forth.

Network and Internet repositories several orders of magnitude slower than the previous options. These are accessed through URLs (uniform resource locators), which specify both the resource location and the protocol required for communication (https://www.uma.es). At the end of each URL is either a command for a server or a file directly (usually an `file.html`) to download and interpret by our web browser.

19: RAM, ROM, EPROM, and similar technologies are different forms of internal computer memory



20: Braille encoding of letters



1·3 Information Coding

Computers only manipulate two-state signals., so every piece of information — numbers, text, images, sound — must first be encoded in binary. This section examines how that encoding works and what its limits are.

- The reason computers are binary and not hexadecimal, for example, is that a transistor can **cheaply** and **reliably** distinguish just **two voltage states** (high/low). To store a hex digit natively you'd need a component that reliably held 16 distinct voltage levels — far harder to build, and much more error-prone as voltages drift with heat and noise.

The alternative is to handle information in an **analogue** way: as a physical quantity in continuous, direct correspondence with what it represents. That is what a mercury thermometer or a classic car speedometer does — and what vinyl records did for music: the groove is a physical copy of the sound wave itself. In principle nothing beats that fidelity, since digitalisation reduces reality to a finite set of numbers. In practice, though, the victory of **digital** has been total, and for good reasons: binary pulses can be stored, copied and transmitted *exactly*, millions of times, without the degradation every analogue copy suffers — errors can even be detected and corrected automatically. The jump from vinyl to the compact disc (1982) made the difference audible to everyone.

21: Vinyl records: the groove is a physical copy of the sound wave itself



Digital information has a natural minimal unit: the **bit** (*binary digit*), a single `0`-or-`1`. One bit alone says very little, so memory handles bits in groups — eight of them form a **byte** — and the meaning of a group depends entirely on the code we agree to read it with: the very same pattern of bits may be a number, a letter, or the colour of a pixel.

22: In memory, each group of bits represents a specific type of information



1·3.1 What memory can hold

The simplest memory element a computer can store is `0`/`1`. The only information it can represent is the truth value of something: `false` or `true`. These are called Boolean values, and they represent the simplest form of information—a single `true` or `false`—yet they are fundamental to programming. The next most important form of information is numbers: integers and real numbers are present in every programming language, and the hardware of every computer directly supports integer addition, multiplication, and division.

To encode integers, we use the ancient positional system; the only choice is the base—`10` in everyday life, `2` within the computer. The value of a digit depends on where it sits: the `3` in `35` is not the `3` in `53`. Thus $214 = 2 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0$, and exactly the same idea in base 2 gives, for example, $1101_2 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$. Bases 8 and 16 are also common in computing, simply because each octal or hexadecimal digit groups exactly 3 or 4 bits.

Unsigned integers. With n bits we can form 2^n different patterns, so the natural encoding covers the range 0 to $2^n - 1$: one byte holds `0`–`255`, and 32 bits reach a little over four thousand million. Note something with no equivalent in mathematics: the number of bits is *fixed in advance*. Add `1` to the largest value and the result wraps around — an **overflow**, a classic source of program errors.

$\mathbb{R} \supset \mathbb{A}_{\mathbb{R}} \supset \mathbb{Q} \supset \mathbb{Z} \supset \mathbb{N}$

$\mathbb{R}$	Real	π, e, -2π
$\mathbb{A}_{\mathbb{R}}$	Real Algebraic	$\sqrt{2}$, $-\sqrt{3}$, $\frac{1+\sqrt{5}}{2}$
$\mathbb{Q}$	Rational	$\frac{1}{2}$, $-\frac{2}{3}$
$\mathbb{Z}$	Integer	-1, -2, -3
$\mathbb{N}$	Natural	0, 1, 3

Negative integers: two's complement. Instead of storing a separate sign, computers give the leftmost bit a *negative* weight: in a byte it counts as -128 rather than $+128$. So `10000001` is $-128 + 1 = -127$, and the range of a byte becomes -128 to $+127$ (one more negative than positive). The beauty of this trick, called **two's complement**, is that the very same addition circuit works unchanged for positive and negative numbers — no special cases.

Real numbers. For reals, computers use **floating point**: binary scientific notation, storing a sign, some significant digits (the *mantissa*) and an exponent, in a standard format (IEEE 754) that in C++ you will meet as `float` and `double`. The exponent gives an enormous range, but the mantissa is finite, so almost no real number is stored exactly — not even an innocent-looking 0.1, which in binary is periodic, like $1/3$ in decimal. Every real is, in general, a *rounded approximation*; keep this in mind, because comparing two reals for exact equality is one of the classic beginner traps we will return to.

1·3.2 I/O Codes. ASCII

To exchange text with humans, computers need agreed tables that map characters to bit patterns. We look at the most common ones, from ASCII (American Standard Code for Information Interchange) to Unicode.

HI/LO	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\n 0	\0	SOH	STX	ETX	EOT	ENQ	ACK	\a	\b	\t	\n	\v	\f	\r	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x30 3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x41 4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x61 6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

The ASCII table comprises four code blocks:

Control characters (`0x00`–`0x1F`, 32 positions) control chars
: remnants from communication protocols, where individual bytes signalled functions such as the start of a data block. Notable examples include the line feed `\n` (position `0x0A`), the tab character (`\t`), and the null character (`\0`, `0x00`).

Symbols and digits (`0x20`–`0x3F`, 32 positions) standard symbols
: including the exclamation mark `!`, quotation marks, digits (`0` to `9`), and related characters.

Uppercase alphabet (`0x40`–`0x5F`, 32 positions) uppercase letters
: (`A` to `Z`) and additional symbols.

Lowercase alphabet (`0x60`–`0x7F`, 32 positions) lowercase letters
: (`a` to `z`) and additional symbols.

When we write any letter, symbol, or digit on a computer, what the system saves to memory, stores on disk, and transmits over the internet is a byte that corresponds to that character according to the ASCII table.

However, we routinely use alphabets that extend far beyond the basic 26-letter Latin alphabet. We use characters such as the Spanish ñ and emojis like 👍, which are encoded in extensions of the ASCII standard. A series of extensions to this standard began in 1992 with UTF-8, and today we can encode thousands of symbols using multiple bytes. The ASCII table forms a subset of this larger encoding scheme, which is capable of representing a vast number of symbols.

Throughout this course, we assume that ASCII encoding is used, where each character is represented by a single byte.

1·3.3 Checkpoint

1) What has made computing devices so useful and widely adopted across diverse professions and by such a broad range of people?

2) What is the most important component of a computer, and which other components are absolutely necessary for it to function?
3) What is the key difference between the von Neumann and Harvard architectures?
4) Why might a microcontroller favour a Harvard (or modified Harvard) architecture over a von Neumann one?
5) What two units comprise the CPU internally?
6) What is the difference between parallelism and concurrency?
7) Can a program be concurrent and parallel at the same time?
8) Can a program be parallel and not be concurrent?
9) What is a memory address, and what purpose does it serve?
10) Why do computers require both main memory and secondary storage?
11) Into what software category do word processing programs, spreadsheet programs, email programs, web browsers, and game programs fall?

1·4 Algorithms and Problem Solving

Before writing a single line of code, we must have a precise understanding of the problem we are solving and how to solve it. The algorithm—a finite, unambiguous procedure—serves as the bridge between problem and program. An algorithm becomes a program once it is expressed in a programming language. Prior to this stage, an algorithm remains more abstract; it can be expressed in multiple ways, and only when formalised in a programming language can it properly be termed a program. After many years devoted to teaching how to solve mathematical problems, Pólya synthesised this into a set of generic steps for solving any problem:

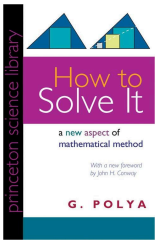
1. Understand the Problem
2. Devise a Plan
3. Carry out the Plan
4. Look Back

23: George Pólya, 1887-1985



Understand the Problem Before any code is written, we must be able to state clearly what is given (the data) and what is required (the result), together with any conditions the data must satisfy. Many failed programs stem not from mistaken logic but from an ill-defined problem: restating it in your own words, identifying inputs and outputs, and working through a small example by hand are the surest ways to expose ambiguity before it hardens into a bug.

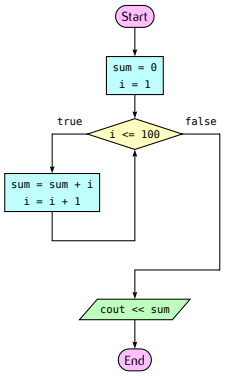
24: Pólya's Steps for Solving Any Problem



Devise a Plan Having grasped the problem, we search for a strategy connecting the data to the result. This is rarely invented from nothing: it usually means recognising the problem as similar to one already solved, breaking it down into smaller subproblems, or choosing among established techniques—iteration, recursion, divide-and-conquer—that fit its structure. A good plan can be expressed as

pseudocode or a diagram (Control Flow Diagram Figure 25), independently of any particular programming language.

25: Control Flow Diagram of the program at Section 2·1.2



Carry out the Plan Only now do we translate the plan into actual code, implementing it step by step and checking, at each stage, that the step just taken is correct before proceeding to the next. Patience here matters more than speed: a plan executed carelessly introduces errors that no amount of later debugging fully explains away.

Look Back Once a solution runs, resist the temptation to consider it finished. Examine whether the result is plausible, test it against cases you did not use whilst building it (particularly boundary cases), and ask whether the same reasoning could be reused or the solution simplified. This final step is what turns a solved exercise into transferable experience—precisely what separates a programmer who learns from one who merely finishes assignments.

1·4.1 What do we want to do with algorithms

Like a cooking recipe, an algorithm comprises an ordered sequence of precise steps that transform inputs (data) into a desired output. We define the properties that any valid algorithm must satisfy. Computing the height of a mountain from the length and angle of its shadow was one of many problems compiled by Al-Khwarizmi.

26: First book on algorithms: "Kitāb al-Jamʿwa-l-tafrīq bi-ḥisāb al-Hind" (The Compendious Book on Calculation by Completion and Balancing), Baghdad, 833 CE



Anything that can be expressed through a series of unambiguous steps and executed by some computing device can be programmed. However, we shall see that some seemingly reasonable problems—problems for which an algorithm might appear to exist—in fact have no feasible, logically correct algorithmic solution.

27: What is the algorithm for making lemonade?



Algorithm: **Lemonade**
1. In a 2 litre pitcher, combine 4 lemons' worth of juice, water, and sugar
2. Stir until the sugar dissolves
3. Chill in the refrigerator.

1·4.2 What can be done? Computability

Surprisingly, some well-defined problems admit no algorithm whatsoever, whilst others possess solutions that are prohibitively slow to be of practical use. **Computability** and **Complexity** delineate the boundaries of what programming can achieve: the former concerning *feasibility*, the latter concerning *efficiency*.

Some problems, despite being well-defined, are inherently unsolvable. The halting problem is classified as **undecidable**.

Problems that seek solutions involving self-reference—such as "this sentence is false"—are closely related. Similarly, the question of whether the set of all sets should contain itself exemplifies this issue. The halting problem specifically concerns the possibility of creating a program capable of determining whether any arbitrary program will terminate when given a particular input. Because it is self-referential in nature, such a program cannot be constructed.

1·4.3 How Difficult Can It Be? Complexity

Conversely, although many problems have known solutions, different algorithms solving the same problem do not necessarily consume equal

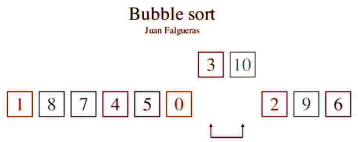
resources in terms of execution time or memory space. These two resources are interrelated: some solutions can be optimised for speed by using additional memory, and vice versa.

For instance, if you store intermediate results computed during program execution, you will require more memory space; however, you will avoid recomputing the same values repeatedly, thereby reducing execution time at the cost of increased memory consumption.

Nevertheless, the performance of different algorithms for solving identical problems can vary dramatically. Consequently, the study of computational complexity—typically focused on time efficiency—constitutes a crucial area within the theory of computation.

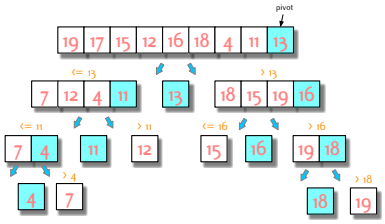
When sorting a sequence of values, you improve search performance on those values. Sorting is a fundamental operation that computers perform continuously, and implementing it efficiently is a long-studied critical concern. One approach is selection sort: identify the smallest value and place it in the first position, then advance one position and repeat with the remaining data. However, whilst conceptually straightforward, selection sort performs poorly in practice, particularly on large datasets. A significantly faster alternative is the *Quicksort* algorithm (attributed to Hoare), which partitions a large list into sublists where all elements in each sublist are smaller than those in the next, then recursively sorts these smaller lists by swapping disordered elements.

28: Bubble sort is the most widely used direct sorting algorithm



To compare algorithms fairly—independently of the particular machine, compiler, or programming mood of the day—we do not count seconds but **operations**, expressed as a function of the input size n. What matters is not the exact count but how it *grows* as n grows: this is captured by **asymptotic**, or **Big-O**, notation:

29: Quicksort, on the other hand, is one of the most efficient general-purpose sorting methods



$$O(\cdot)$$

which keeps only the dominant term and discards constant factors and lower-order details. An algorithm performing $3n^2 + 7n + 12$ operations is simply $O(n^2)$: for large n, doubling the input roughly quadruples the work, and the $+7n + 12$ becomes irrelevant.

From a complexity perspective, problems can be classified as follows:

Tractable possesses an efficient solution, typically achievable in polynomial time, $O(n^k)$.

Intractable lacks an efficient solution, exemplified by the Travelling Salesman Problem, in which one seeks the shortest path visiting each city in a given set exactly once. This problem is related to numerous others, including circuit board design, and belongs to the formally defined **NP-hard** class, which we alluded to during our discussion of decidability.

For intractable problems, the typical brute-force approach involves evaluating all possible solutions—whose number grows factorially $(n!)$ or exponentially (2^n) with n—and selecting the optimal one. There is no shortcut to the answer; instead, you must explore every possible path,

much like in a maze with no signposted exit, exhaustively comparing candidates to identify the best one.

Concretely: checking every route for the Travelling Salesman Problem with just 20 cities means examining roughly $20! \approx 2.4 \times 10^{18}$ possibilities. Even at one thousand million (10^9) checks per second, that is about 77 years of computing—for a mere 20 cities. See Figure 30

30: Exponential growth of factorials $n!$ when no efficient solution exists and it is straightforward to verify each candidate solution. The Travelling Salesman Problem exemplifies this scenario

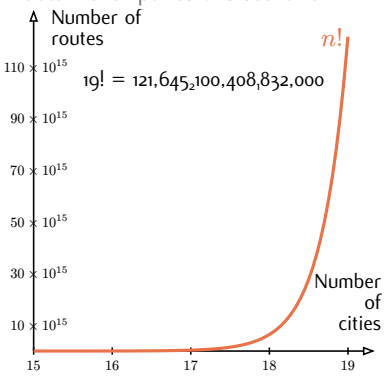
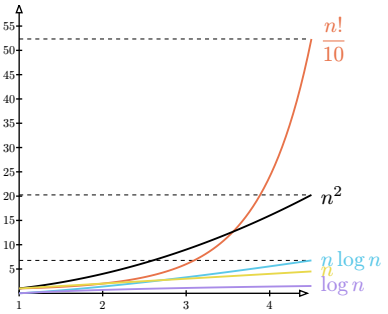


Figure 31 lines up the usual complexity classes from cheapest to most expensive: $O(\log n)$ barely grows at all—think of halving a search space at every step, as binary search does; $O(n)$ is linear, so doubling the input doubles the work; $O(n \log n)$ is what the best general-purpose sorting algorithms achieve; $O(n^2)$ is typical of comparing every pair of elements; and $O(n!)$ (with $O(2^n)$ close behind) explodes so fast that it renders even moderately sized inputs computationally hopeless. Notice how close together the first four curves stay even as n grows, whilst $n!$ shoots off the chart almost immediately: that gulf—not the gap between, say, 400 and 4,000 operations—is what separates a program that finishes instantly from one that will not finish before the sun burns out.

31: Growth of $\log n$, n, $n \log n$ (linearithmic), n^2 and $n!$ as the input size n increases



Among the tractable problems, the kind of algorithm determines its efficiency. As noted, sorting algorithms have been widely studied: in the general case, the simple ones (bubble sort, selection sort, insertion sort) run in quadratic time, $O(n^2)$, whereas the best general-purpose ones (Quicksort, merge sort, heap sort) run in linearithmic time, $O(n \log n)$—a difference that, per Figure 31, becomes dramatic well before n reaches the millions of records a real database might hold.

1·4.4 How to do it? Correctness

A program that appears to function correctly is not equivalent to one that demonstrably does so. We introduce the concept of reasoning about **Correctness** rather than relying solely on testing and conjecture. A substantial field within computer science is dedicated to formal correctness in programming, which treats algorithms and programs as mathematical entities that can be rigorously proven correct or incorrect. Whilst this approach is not necessarily commonplace in software engineering, it establishes its foundational importance. One of the key figures who studied and advocated for programming as an exact science was Dijkstra (notable references are the article (Dijkstra 1972) and the book (Dijkstra 1976)).

32: *Simplicity* is a great virtue
- but it requires hard work to achieve it
- and education to appreciate it.
- And to make matters worse: complexity sells better.



1·4.4.1 Reasoning with Loop Invariants

Formal correctness sounds abstract until you see it applied to something as ordinary as a loop. A **loop invariant** is a property that is true before the loop starts, remains true after every single iteration, and—together with the condition that finally stops the loop—lets us deduce exactly what the loop computes, without mentally tracing it iteration by iteration.

Take the algorithm for summing the first `n` natural numbers:

1. Set `sum` to `0` and `i` to `1`.
2. While `i` is less than or equal to `n`: add `i` to `sum`, then increase `i` by one.
3. Return `sum`.

The 1968 Software Crisis (a term coined at the NATO Software Engineering Conference, Garmisch, Germany, 1968) established that it was necessary to treat software development as a proper engineering discipline (with specification, design, testing, and methodology) rather than a craft, and gave rise to the structured programming movement.

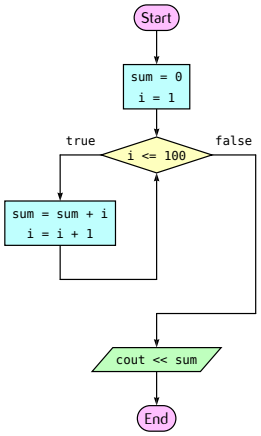
As invariant, propose: *before each iteration,* `sum` *equals* `1 + 2 + ... + (i-1)` —the sum of everything added so far. Proving a loop correct with an invariant always follows the same three-step pattern:

Initialisation Before the first iteration, `i` is `1` and `sum` is `0`. The sum "from 1 to 0" is empty, i.e. zero, so the invariant holds trivially the moment we enter the loop.

Maintenance Assume the invariant holds at the start of some iteration, so `sum` equals `1 + ... + (i-1)`. The body adds `i` to `sum`, giving `1 + ... + (i-1) + i`, which is `1 + ... + i`; it then increases `i` to `i+1`. The new value of `sum` is therefore exactly `1 + ... + ((i+1)-1)` —the invariant holds again, ready for the next iteration.

Termination The loop stops as soon as `i` exceeds `n`, i.e. when `i` equals `n+1` (since `i` only ever grows by one at a time). Substituting `i = n+1` into the invariant gives `sum = 1 + ... + n`—precisely the value the algorithm was meant to compute.

33: Control Flow Diagram of the program at Section 2·1.2



Notice what this argument achieves: it holds for **every** value of `n` at once, without running the algorithm even a single time. This is the kind of reasoning Dijkstra had in mind when insisting that a program be **proven** correct rather than merely observed to work on a handful of test cases —and it is exactly the reasoning we will revisit, applied directly to C++ code, when we study loop design later in the course.

1·4.4.2 Writing with style

When programming, the initial stage involves writing the program source code. The choice of editor is quite important at this stage. All modern editors provide *syntax colouring*, which significantly aids code writing: they highlight different syntactical tokens in distinct colours, making the program considerably easier to read.

Some novice programmers underestimate the importance of aesthetic formatting—such as spacing around tokens, code block organisation, indentation, and overall text structure—but this becomes increasingly evident as program length grows. Furthermore, it becomes crucial as program complexity increases, making it more difficult to follow the logic without proper formatting.

34: Indentation

```
int main() {
  int sum = 0;
  int i = 1;

  while (i <= 100) {
    sum = sum + i;
    i = i + 1;
  }

  cout << "The sum fro

  return 0;
}
```

A common area where novice programmers encounter difficulty is the indentation of statements and statement blocks. Programs consist of nested groups of statements, and indentation makes these groupings explicit. Without proper indentation, programs become unreadable to humans. In Figure 34, the vertical red lines indicate the two indentation levels present in that program. Typically, we indent by two spaces to advance to the next level within a block, and we consistently increase the indentation level for code blocks delimited by `{..}` curly braces.

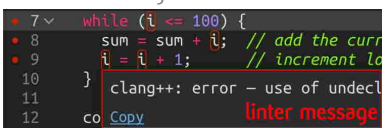
A good introductory reference on writing style is K&R indentation style.

For our course, see Section Appendix C.

1·4.4.3 Writing with the aid of linters

Linters are programs that verify code correctness at both syntactic and, to a limited extent, semantic levels. They identify potential issues such as unused or incorrectly defined variables. Importantly, they perform this analysis at the optimal time: whilst you are editing and building your source code, well before encountering runtime errors. In modern editors, linters are integrated into the programming environment for your chosen language, typically through a built-in feature or plugin.

35: Linters are particularly useful when integrated into program text editors, as they address common omissions and mistakes whilst you write code



The most effective tools minimise intrusiveness and distraction, thereby preserving your focus and maintaining development flow. In this manner, editors with linters for C++ display subtle indicators for common mistakes —such as undeclared variables and unclosed parentheses—highlighting typical coding errors rather than prescribing solutions. In Figure 35, we failed to declare the variable `int i`. The linter then highlights each occurrence with a red rectangle as we write the code, and when hovering the mouse over them, it displays an explanation of the error. This occurs whilst you are editing, even with the file unsaved and without the intervention of the compiler, yet it proves invaluable for identifying typical typos, missing variables, and many other details.

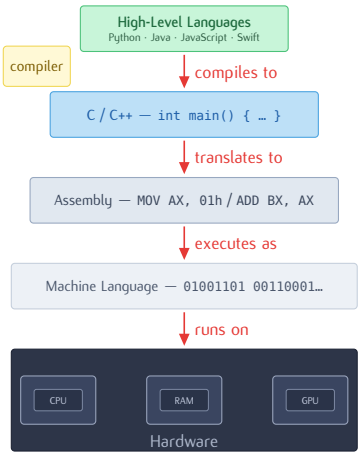
The idea behind linters is not to do the work for you, but to help you manage the many details of your programs whilst keeping your creativity unimpeded.

1·5 Programming Languages

Computers can only execute binary machine code, whereas humans rely on substantially richer forms of representation. Programming languages serve as the bridge between these two domains: they are sufficiently formal and precise for machines to execute unambiguously, yet remain readable to humans. We aim to learn the fundamental programming structures, basic data types, and essential approaches to code organisation. These concepts are well covered in one of the most classical and widely used languages: C and its augmented version C++. This particular language, its syntax, and grammar form the focus of this course.



37: Abstraction level of language, from concept to machine



Programming languages are independent of the operating system on which they are installed and are portable across different systems, thanks to the compilers and interpreters available on those operating systems. These tools convert standard syntax into the specific binary code required by each platform.

Programming languages are necessary because low-level programming —that is, programming using hardware device instructions—although it produces the fastest executable code, is cumbersome to modify and understand, and offers limited portability across platforms. Natural language, by contrast, is necessary to express the much wider variety of meanings that we humans convey to one another. This meaning is completed by visual language, body gestures, and shared context built among speakers through their communication history. People can sometimes understand each other with few or no words at all, depending entirely on contexts that emerge spontaneously, which we have learned to interpret.

Interestingly, large language models are able to follow the patterns we use and are becoming capable of communicating with us and assisting when we express concepts and problems unambiguously, so the solution is recognisable and we find it valid. The problem with large language models is that users tend to trust computer answers as if they were produced by real people who genuinely understand the unexpressed conditions of the problem, and people tend to treat large language model answers as intelligent, believing they possess the context, history, and genuine knowledge of us and what they "say".

Programming languages are trustworthy; large language model conversations and answers are not.

Everything depends on the precision of the expressions. Many programmers today believe that the future of programming languages lies with large language models—that languages such as C++, Python, and JavaScript are complicated, whereas the natural expressions we can use to query language generators are simpler, more closely aligned with our own thinking, and therefore quicker to articulate and faster to produce. This reasoning has some merit, but it comes with a significant caveat. Code generated by large language models relies on generic responses adapted to what programmers have expressed in their prompts. The more ambiguous the prompts, the less precise the generated answers become. The models must infer what the prompt omitted and patch standard solutions where the programmer lacked specificity. Ultimately, although such code is produced rapidly, it requires extensive review and is likely to fall far short of the quality of a carefully crafted program developed by people who genuinely understood what they wanted to achieve. Moreover, what people actually want changes over time, and the additional time they invest in producing the code improves its quality accordingly.

1·5.1 Language Recognition. Grammars

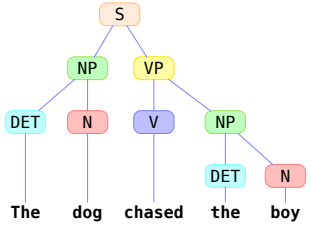
Just as natural languages are governed by grammatical rules, programming languages are defined by formal grammars that determine which sequences of text constitute valid programs. For example, "I am human" is grammatically correct, whereas "I human am" is not. A language's grammar specifies which lexical elements—individual letters and symbols—are permissible, which combinations of these elements form meaningful words in the language's vocabulary, and how these symbols may be structurally arranged to convey meaning.

Reasoning about whether a program is correct only makes sense once we can already say, unambiguously, whether it is a program at all—that is, whether it belongs to the language defined by the rules of C++. This more basic question is answered by a **grammar**: a finite set of rewriting rules that generates every valid string of the language and no others. Noam Chomsky formalised this idea in the 1950s whilst working on natural language◆, and it is now the standard tool for defining the syntax of every programming language, C++ included.

Formally, a grammar is a four-tuple:

$$G = (N, T, I, R)$$

38: Grammars define the correct sequences of text. All programming languages have a grammar



39: Noam Chomsky in 1977, now 97



◆Chomsky's original motivation was linguistic, not computational: he wanted a precise mathematical account of which sentences are grammatical in a natural language such as English. The same machinery turned out to be exactly what computer scientists needed a few years later to define programming languages, and his classification of grammars by expressive power—the *Chomsky hierarchy*—is still the one used today. For a five-minute taste of the idea straight from the source, see the short Chapter 2 of (Chomsky 1957).

Non-terminals (N) placeholder symbols standing for a syntactic category still to be expanded (written here in upper case, such as P).

Terminals (T) the actual symbols of the alphabet that may appear in a finished, valid string—letters, digits, punctuation—which cannot be expanded any further.

Initial non-terminal (I) the single non-terminal, $I \in N$, from which every derivation must start.

Rules (R) rewriting rules of the form *non-terminal* $\rightarrow$ *sequence of terminals and/or non-terminals*, each specifying one way a non-terminal may be expanded.

Each rule set is read as "P may be rewritten as…". Applying the rules repeatedly, starting from P, generates every valid string of the language.

Examples:

$$\begin{aligned} G_3: &\quad P ::= z \\ G_1: &\quad P ::= aPb \mid \varepsilon \\ G_2: &\quad P ::= aPbb \mid \varepsilon \\ G_4: &\quad P ::= aPbP \mid \varepsilon \end{aligned}$$

G_3: no recursion. There is a single rule and it contains no P, so there is nothing left to expand: $L(G_3) = \{z\}$. Recursion, not the alphabet, is what makes a language infinite.

G_1: one nested block. Expanding twice and then choosing ε: $P \rightarrow aPb \rightarrow aaPbb \rightarrow aabb$. Hence $L(G_1) = \{a^n b^n : n \geq 0\}$. It generates `ab`, `aabb`, `aaabbb`, but **not** `abab`: every a is closed by a b placed right after the inner block, so blocks can only be nested, never placed side by side.

G_2: unbalanced. Same idea, but each step adds two closing symbols: $L(G_2) = \{a^n b^{2n} : n \geq 0\}$.

G_4: matched brackets. The second P after the b lets us append another block. Reading a as `(` and b as `)`, this generates all balanced strings:

$$P \rightarrow aPbP \rightarrow abP \rightarrow abaPbP \rightarrow abab$$

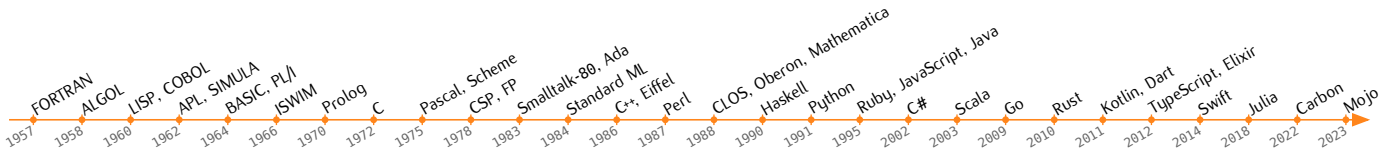
that is, `()()`. Nesting still works too: $P \rightarrow aPbP \rightarrow aaPbPbP \xrightarrow{*} aabb$, i.e. `(())`.

String	G_1	G_2	G_4	In brackets
ab	yes	no	yes	`()`
abb	no	yes	no	—
aabb	yes	no	yes	`(())`
abab	no	no	yes	`()()`
aabbab	no	no	yes	`(())()`

Only G_4 really describes balanced brackets, and it is the kind of rule that models nested **and** sequential structure in C++ (blocks inside blocks, and statements one after another).

A parser—part of the compiler, as we shall see shortly—does the reverse job: given a string, it decides whether some sequence of rule applications could have produced it, and rejects the string (a **syntax error**) if not. This is the first and cheapest kind of correctness a program can fail: before anything about its logic is even considered, it must first belong to the language the grammar defines.

1·5.2 History of programming languages



The first two decades established the two principal families that remain in use today. **FORTRAN** (1957) and **ALGOL** (1958) introduced the imperative, procedural style that **C**, **Pascal**, and their descendants inherited. **LISP** (1960), by contrast, treated programs as data and computation as function application — the foundation of every functional language from **Scheme** to **Haskell**. These represent two approaches to the same fundamental concept: both instruct the computer what to do step by step, but they diverged early. The imperative procedural dialect employs *do this, then this*, whilst the imperative functional dialect employs *this equals that*.

From the 1970s to the 1990s, programming languages evolved to address specific problems rather than embodying general philosophies. C .(1972) established itself as a portable abstraction layer close to the machine; **Smalltalk** and **C++** unified state and behaviour as objects◇; **Prolog** and **Standard ML** investigated logic and strong typing ; **Perl**, **Python** and **Ruby** fulfilled the need for rapid systems integration. Each new language represented less a paradigm shift than a pragmatic response to perceived limitations in its predecessor. Perl, which remains in active use today, brilliantly addressed the need for system administrators to automate many small, disparate tasks from the command line, initially with a procedural approach and later adding object-oriented capabilities. Python offered a higher level of abstraction, superior library handling, and significantly simplified syntax, though some argue the simplification went too far—for example, it provides only the `while` loop construct and lacks a `switch` selection structure. Ruby, which later excelled in web application development, was distinguished by its object-based design that went beyond conventional object orientation, providing a rich set of programming features.

◇ Objects encapsulate both data and behaviour, and crucially, enable the creation of descendant objects that inherit those properties and behaviour. This inheritance mechanism is fundamental to how modern operating systems support the development of complex applications: they provide classes of objects that applications can use as parents for their constituent elements, which coexist within the broader complex environment.

Strong typing means every value has a fixed, well-defined type that the compiler checks before the program ever runs, ruling out an entire class of errors — such as treating a string as a number — that would otherwise only surface at run time. Standard ML pushed this further with type inference (the Hindley–Milner algorithm): the compiler deduces the type of every expression automatically, so the programmer gains the safety of explicit typing without having to write a type annotation on every line.

Since 2000, language development has been driven primarily by two pressures: scalability and safety. **Java** and **C#** introduced managed memory to enterprise software, whilst **Go** and **Rust** addressed the need for systems programming without the frequent memory bugs that pro-

grammers introduce in C programs■. **TypeScript** retrofitted static types onto **JavaScript**□ once web applications outgrew loose scripting. The most recent entrants, **Carbon** and **Mojo**, are still establishing their position —Carbon as an experimental successor to C++, and Mojo as a Python-compatible language designed for AI workloads—a reminder that this landscape continues to evolve.

■ Memory Garbage collection is necessary but can be a headache: it frees the programmer from manually tracking every allocation, eliminating whole categories of bugs such as dangling pointers and memory leaks, but the collector's periodic sweeps introduce unpredictable pauses that can make GC-based languages like Go unsuitable for hard real-time systems. Rust sidesteps this trade-off entirely with its ownership and borrowing model, enforced at compile time, which guarantees memory safety without any run-time collector or its associated pauses.

□ JavaScript is the interactive language, the language that interacts with the user, embedded in web application pages that makes them interactive — originally designed in ten days in 1995 to add simple scripting to Netscape Navigator. Its dynamic, loosely typed nature made it easy to learn but ill-suited to the large, long-lived codebases that modern web applications became; TypeScript addresses this by adding an optional static type system, checked at compile time and then compiled away, so ordinary JavaScript is what actually runs in the browser.

1·5.3 Translators, Compilers and Interpreters

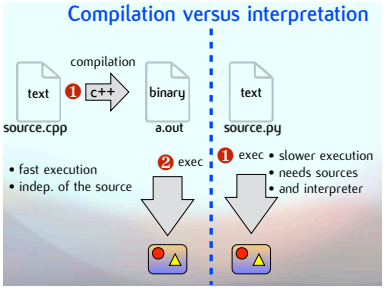
A mechanism must bridge the gap between source code and machine code. Compilers translate an entire program in advance, whereas interpreters translate and execute code dynamically—analogous to translating a book versus interpreting a live speech. Compilers and interpreters are sophisticated systems; they follow a complex sequence of steps during translation, and errors may be detected at each stage. These errors can be classified as lexical, syntactical, or semantic. Furthermore, even after a translated program (whether compiled or interpreted) has successfully passed all these stages, it may still contain logical errors that cause incorrect behaviour during execution.

Translator a general converter that translates from one language to another

Compiler converts from high-level to low-level language; it is specific to computer languages. It converts from our `source.cpp` to a machine language executable file `a.out` that can be executed repeatedly

Interpreter operates in situ by executing high-level code with no intermediate conversion; as it reads the source code, it executes it directly, without any additional steps. For example: Python, JavaScript or Ruby are interpreted languages.

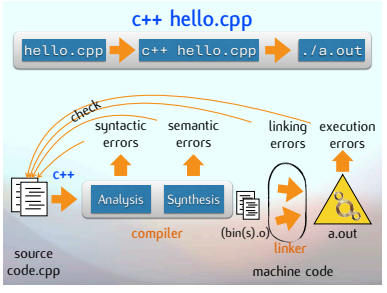
40: Compilers generate a new, independent executable file, whereas interpreters execute code as they read the original source, which they depend on



From a high-level perspective, the compiler performs the following operations on the original source file(s), various intermediate files, and standard library components (intermediate files, designated `name.o`, are typically deleted immediately by the compiler and are not visible in our sessions):

1. Parse the source file `source.cpp`
2. Generate binary code from it, producing `bin.o` (multiple source and binary `.o` files may be involved)
3. Link all `.o` files together with the required standard library components to produce an executable file `a.out`

41: Files involved in a compilation process

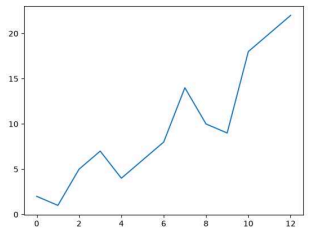


What are the advantages of interpreters? There are several significant ones. First, you obtain results and execute code immediately, which makes them particularly well-suited for testing solutions, exploring different approaches, and working with code that requires frequent modification. The second major advantage of interpreters is that they typically provide extensive libraries that offer concise solutions to common problems. For example, parsing an entire web page in Ruby uses `nokogiri` with `doc = Nokogiri::HTML(URI.open(url))`, or in Python, plotting an array with `numpy`:

```
# Python
import numpy as np
```

```
import matplotlib.pyplot as plt
plt.plot(np.array([2, 1, 5, 7, 4, 6, 8, 14, 10, 9, 18, 20, 22]))
plt.show()
```

42: Direct output from the Python `matplotlib` interpreter

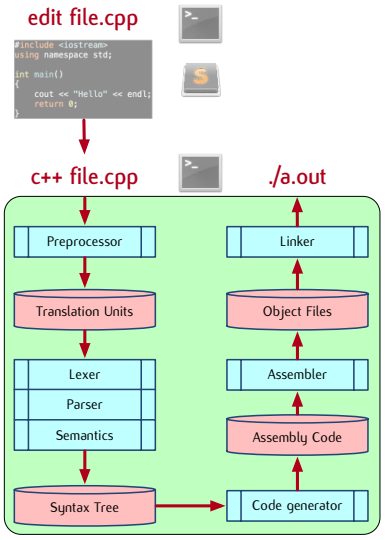


You can observe the graphics window opening on your desktop as you execute this code: Figure 42, and then modify or add any value and click to refresh the output.

1·5.3.1 Preprocessor

Before the compiler proper ever looks at your code, a separate and much simpler program called the **preprocessor** runs a purely textual pass over it. It understands nothing about C++ grammar, types, or scope—it only follows a handful of directives, each starting with `#`, and rewrites the text accordingly. Its output, not your original file, is what the rest of the pipeline actually compiles: this rewritten text is called a **translation unit** (see Figure 43).

43: Phases of compilation (and linking) from your source file `file.cpp` down to the production of the final independent, executable `a.out`



The directive you have already used in every program is `#include <iostream>`: the preprocessor locates the named header and pastes its entire contents in place of that line, verbatim, before compilation begins—so `cout` and `endl` only become known to the compiler because their declarations were physically copied into your file moments earlier. `#define` works similarly but for names of your own choosing: `#define MAX 100` replaces every later occurrence of the token `MAX` with `100`, textually, with no notion of type or scope whatsoever. A more disciplined alternative, idiomatic in C++ and the one we shall prefer throughout this course, is a properly typed `const int MAX = 100;` declaration—handled by the compiler, not the preprocessor, and therefore subject to the same type checking as everything else.

Because substitution happens blindly, before any syntax or semantic checking takes place, preprocessor mistakes surface in unusual ways: an included header that does not exist fails immediately, with no reference to what your program was actually trying to do with it, whilst a careless `#define` can silently corrupt code written far away from the definition itself. Keep this stage in mind whenever an error message points at a line you never wrote, inside a file you never opened—the preprocessor is very likely why.

1·5.3.2 Linker

In our C++ compiler, there exists a final stage in which the compiler, after compiling every source `.cpp` file that composes the program, can issue a *linking error*. These errors occur only when something is invoked but ultimately cannot be found by the compiler's linker. Although in our course we will work exclusively with programs containing a single `.cpp` source file (whereas commercial applications typically comprise dozens or hundreds of `.cpp` source files), linking errors can still occur if you reference entities that do not exist or invoke them with minor syntactical errors. For example, if you make the mistake of calling the $\sqrt{x}$ function (in C++ `sqrt(x)`) but write `sQrt()` instead, the compiler would not detect any syntax errors. This error is passed to the final phase, the linker, without any reference to the line in which it occurred in the source `myProgram.cpp` file. The linker will search for the function `sQrt()` in all the standard libraries without success and will inevitably issue an error message stating

that `sQrt()` was not found, without reference to the line in which the error occurred.

1·5.3.3 Debugging

Debugging is the process of locating bugs—errors that have made their way into your program's logic. Your program may compile and execute, but either fail noisily during execution or produce incorrect results. At this point, your only recourse is to rewrite your code, but identifying which part is failing can be difficult. When programs consist of multiple files, this problem becomes even more challenging. Fortunately, specialised programs called *debuggers* exist to help. A debugger can execute your program step by step, displaying the output alongside the source code that produces it. You can insert breakpoints at which execution halts, allowing you to inspect the current state (variable values and output so far), and they typically provide tools to examine everything your program is responsible for. However, using a debugger can be like using a sledgehammer to crack a nut. Especially for short programs like ours, it is far more practical to insert simple print statements that display variable values, thereby externalising what occurs within the program. If the program crashes, these statements also reveal when the failure occurred. For example, a simple

44: Debugging involves placing our program under scrutiny by tracing its execution line by line or stopping at breakpoints, whilst simultaneously observing changes to program variables. However, for simple programs, this can be an excessively meticulous process




```
r = a / b;
```

can crash our code if `b` is zero. To catch this happening, you can add:

```
cout << "before division. a: " << a << ", " << "b: " << b << endl;
r = a / b;
```

This will display the value of the variables `a` and `b` just before the program crashes, after which it exits without proceeding further.

1·6 General Overview of a Computer System

Hardware alone is inert: a processor, a disk, or a screen do nothing by themselves. A computer **system** only comes to life when hardware is combined with software organised in layers, each layer resting on the services provided by the one beneath it—much as a building rests on its foundations and is raised upwards floor by floor.

Operating system The permanent, lowest layer: it manages the hardware directly—CPU scheduling, memory, files, devices—and offers every other program a common platform to run on. Examples: Windows, macOS, Linux

System and development software Builds and maintains other software: compilers, linkers, debuggers, version-control systems—the translators discussed above belong here

Utilities Auxiliary tools that keep the system running smoothly but do not, themselves, produce documents for the user: file managers, antivirus software, disk-cleanup tools

Application software Produces or manipulates documents and results for the end user: word processors, web browsers

45: Computers Layers



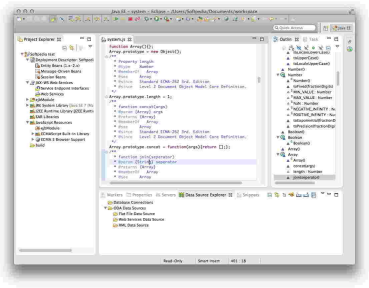
1·6.1 Integrated Development Environment

Compilers, or more broadly Integrated Development Environments (IDEs), sit in the system-and-development layer: they are the software tools responsible for building the application software we are about to write. In this course, we must first install a compiler on our personal computers to become familiar with it and work efficiently. This setup task can be challenging for an introductory course, so it is important that you complete it as soon as possible. Do not hesitate to seek assistance from instructors and classmates if needed.

An IDE integrates an editor, compiler, and terminal into a unified development environment—the programmer's equivalent of a well-equipped workshop. It is the tool you will use throughout the laboratory sessions. For our purposes, IDEs provide the three essential components required to build, compile, and test the execution of each program in this course:

1. Editor
2. Compiler
3. Terminal for execution

46: Eclipse IDE. Typically files at the left column, execution and logs at the bottom, object hierarchy at the right. Useful for building complex applications involving a bunch of files and libraries



1·6.2 Non-integrated Development Environment

In practice, you only need a UNIX/Linux command line, a terminal window, and a text editor to edit, compile, and execute the programs covered in this course. In fact, this is the preferred setup for many—myself included—as it is non-distracting and keeps your focus on what matters. It also demands significantly fewer system resources and makes every step of the compilation process transparent.

47: Icons to represent a terminal (Command Line Interpreter or CLI) window

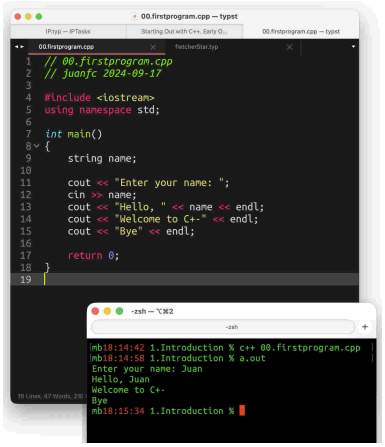


The system must have a C++ compiler. This is straightforward to set up on Linux, macOS, and Windows (with Linux virtualised).

All of this will be explained in detail during the laboratory sessions.

You can edit, compile, and execute code directly without relying on a graphical IDE. Whilst this may appear more complicated than using graphical environments, for the type of exercises we will undertake—which typically do not constitute formal projects—the additional features of graphical IDEs often introduce unnecessary complexity rather than providing genuine benefit.

48: Using only a good plain text editor or programming text editor and a terminal window (CLI) makes things considerably easier and more suitable for this course



1·6.3 Checkpoint

1) What is an algorithm?
2) Why are there so many programming languages, and why were they invented in the first place?
3) What are the fundamental components of a grammar?
4) Why are programming languages independent of the operating system and portable across different systems?
5) What is the distinction between high-level and low-level languages?
6) What does portability mean?

7) Describe the functions performed by the preprocessor, compiler, and linker.
8) What is contained in a source file, an object file, and an executable file?
9) What is an integrated development environment?
10) What is the minimum configuration required for a computer to enable programming?
11) What is a syntax error?
12) What is a semantic error?
13) What is a logic error?
14) What is a run-time error?
15) What is pseudocode?

References

CHOMSKY, Noam, 1957. *Syntactic Structures*. The Hague: Mouton.

The book that launched formal grammar as a mathematical object and, with it, the theory later used to define programming-language syntax. Written for linguists, not computer scientists, and most of it is safely skippable for a first-year programmer — but Chapter 2, "The Independence of Grammar" (about five pages), is a fast, self-contained read: it shows, with the famous sentence *Colorless green ideas sleep furiously*, that a string can be perfectly grammatical while being nonsense, and ungrammatical while using only familiar words. That single example is the whole point of a formal grammar in one sentence, and needs no background to appreciate. Read just that chapter.

DIJKSTRA, Edsger W., 1972. The Humble Programmer. *Communications of the ACM*. 1972. Vol. 15, no. 10, p. 859–866. DOI **10.1145/355604.361591**.

Dijkstra's 1972 Turing Award lecture and one of the most important texts in the history of computing. Written with remarkable clarity and conviction, it argues that programming is an intellectual discipline of the highest order — not a craft of clever tricks, but a rigorous activity demanding humility before complexity. Dijkstra traces the software crisis of the 1960s, dismisses the cult of the one-liner, and insists that programs must be composed correctly from the outset rather than debugged into correctness after the fact. The lecture closes with a call to approach every program as a Very Humble Programmer. Essential reading for first-year students: short, beautifully written, and a direct challenge to the habits of mind that produce bad code.

DIJKSTRA, Edsger W., 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall. ISBN 0-13-215871-X.

A landmark work in which Dijkstra develops a formal methodology for deriving correct programs from their specifications, introducing weakest preconditions and predicate transformers. Challenging reading, but foundational for understanding why program correctness must be reasoned about rather than merely tested. Pairs naturally with the companion article *The Humble Programmer* as a philosophical grounding for the course.

Task 0: Editing and compiling our first C++ source file

These two exercises are about getting your first programs to compile and run. You do not need to understand every line yet — focus on the edit–compile–run cycle. Always fill in the header comment with the filename, your name or alias, the date, and a brief description of what the program does.

1 **Count down.** The following program counts from 1 to 10. Modify it so that it counts down from 10 to 1.

```cpp
// 01.countTo10.cpp
// [yourShortName] [date]
#include <iostream>
using namespace std;

int main()
{
    int i = 1;
    while (i <= 10) {
        cout << i << endl;  // print the index
        i = i + 1;          // update index
    }
    return 0;
}
```

Submit it as `01.countTo10.cpp` to the `Task 0 dropbox`.

2 Copy the following program (*you do not need to understand it yet*) and optionally add a brief comment describing what you think it does.

```cpp
// 02.nameMe.cpp
// [yourShortName] [date]
// This program [does .. description]

#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    int i = 1;

    while (i <= 100) {
        sum = sum + i;  // accumulate
        i = i + 1;      // update loop variable
    }

    cout << "The result of this is: " << sum << endl;

    return 0;
}
```

Submit it as `02.nameMe.cpp` to the `Task 0 dropbox`.

o O o

Part 2 Introduction to a Programming Language

Contents of this part Book contents

2·1 Introduction to C++ 31
2·2 Simple Data Types 34
2·3 Constants, Variables and Assignments 37
2·4 Characters and strings 48
2·5 Enumerating the values a type can hold 50
2·6 typedef: Defining new names for types 52
2·7 Random numbers 53
2·8 Operators precedence 53
2·9 Basic Input and Output 55
▸ Task 1: Basics. Sequential programs: arithmetic and standard I/O . 58
2·10 Control Flow 60
2·11 Boolean Expressions 60
▸ Task 2: Booleans 64
2·12 Selection Structures 67
▸ Task 3: Control structures. Conditionals 76
2·13 Iteration Structures 79
2·14 Error and Exception Handling 84
2·15 Recommendations and Common Mistakes 86
▸ Task 4: Control structures. Single loops 89
▸ Task 5: Control structures. Double loops 93

Having studied algorithms as abstract recipes, we now need a notation to express them so that a computer can execute them. This chapter introduces C++, the language we will use throughout the course, covering its basic building blocks: data types, expressions, input and output, and the control structures that govern the order in which instructions are executed.

2·1 Introduction to C++

C++ is one of the most influential and widely used programming languages ever created, powering everything from operating systems to video games. We begin with its history, a first complete example, and the general shape every C++ program follows.

2·1.1 The Evolution of C++

C was born between 1969 and 1973 as the language of choice for the development of the operating system UNIX. Linux = free version of UNIX.

C++ was originally "C with classes" and started to be developed in 1979. C++ was created by Bjarne Stroustrup in the early 1980s as an extension of the C language, introducing higher-level abstractions whilst maintaining C's efficiency. Understanding its origins clarifies many of its design decisions and some of its idiosyncrasies.

C++ is backward compatible with C: A C++ compiler can compile C source as well. There are many other C-based languages: Objective-C, C#, etc

Version	Major changes
C++98	The first standard version
C++03	Value initialization
C++11	Lambda expressions, delegating constructors, uniform initialization syntax, `nullptr`, automatic type deduction and decltype, rvalue references, etc.
C++14	Polymorphic lambdas, digit separators, generalized lambda capture, variable templates, binary integer literals, quoted strings, etc.
C++17	Fold expressions, hexadecimal floating point literals, u8 character literals, selection statements with initializers, inline variables, etc.
C++20	Extended facilities for inspecting program entities such as variables, enumerations, classes and their members, lambdas and their captures, etc.
C++23	`std::expected` for explicit error handling, deducing this, `std::mdspan`, `std::print/std::println`, `if consteval`, multidimensional subscript operator, `std::stacktrace`, `flat_map/flat_set`, expanded ranges and views, etc.
C++26	Upcoming update

One can use any of the recent compiler versions available in most integrated development environments, and incompatibilities are rarely encountered, as the syntax used in this course is compatible across all supported versions.

2·1.2 Example of a C++ Program

The most effective approach to learning a language is to study a small yet complete program written in it. We examined a minimal example line by line, identifying the constructs that will recur in every program we write.

```cpp
// sum1to100.cpp
// juanfc 2026-09-09
// This program sums all natural
// numbers from 1 to 100

#include <iostream>
using namespace std;

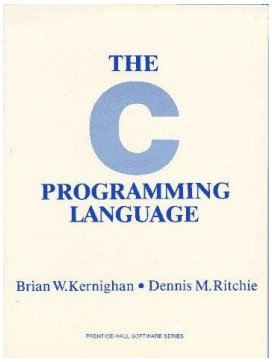
int main() {
  int sum = 0;
  int i = 1;

  while (i <= 100) {
    sum = sum + i;  // add the current value of i to the sum
    i = i + 1;      // increment loop variable
  }

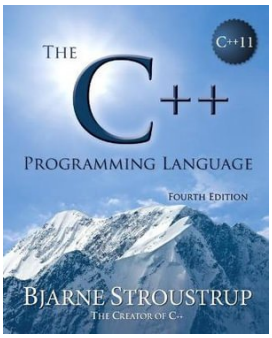
  cout << "The sum from 1 to 100 is: " << sum << endl;

  return 0;
}
```

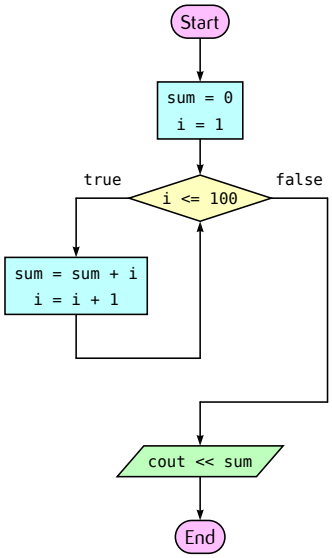
49: C official reference book 1978



50: C++ official reference book for C++11



51: Control flow for the `while` loop that sums 1 to 100 in Section 2·1.2. The `true` branch loops back to the condition; the `false` branch exits the loop and prints the result.



2·1.3 Basic Elements of C++

Like any programming language, C++ comprises a small set of fundamental components: keywords, identifiers, literals, operators, and comments. This section introduces these elements and explains the rules governing their combination.

Keywords are reserved words that the language sets aside for its own use and cannot be repurposed for any other purpose. So, for example, you can never name a variable `int while = 3;`

Control flow	`if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`, `goto`
Fundamental types	`void`, `char`, `int`, `float`, `double`, `long`, `short`, `signed`, `unsigned`
Storage & type	`const`, `static`, `extern`, `volatile`, `register`, `auto`, `typedef`
User-defined types	`struct`, `union`, `enum`
Boolean & null	`bool`, `true`, `false`, `nullptr`
Classes	`class`, `public`, `private`, `protected`, `virtual`, `friend`, `explicit`, `mutable`, `this`, `override`, `final`
Templates	`template`, `typename`
Memory	`new`, `delete`
Namespaces	`namespace`, `using`
Exceptions	`try`, `catch`, `throw`, `noexcept`
Casts	`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`, `typeid`
Operator words	`and`, `or`, `not`, `and_eq`, `or_eq`, `xor`, `xor_eq`, `not_eq`, `bitand`, `bitor`, `compl`
Char types	`wchar_t`, `char16_t`, `char32_t`
Other	`inline`, `operator`, `asm`, `decltype`, `constexpr`, `static_assert`, `thread_local`, `alignas`, `alignof`, `sizeof`

In the preceding example, we encounter the following key elements:

comments Anything written after `//` until the end of the line is completely ignored by the compiler. The preprocessor intercepts and removes these comments before the compiler processes the code.

preprocessor directive Any line beginning with `#` followed immediately by a keyword constitutes a preprocessor directive—an instruction for the preprocessor. In this case, `#include <iostream>` directs the preprocessor to insert the standard library file `iostream` from specific compiler set directories. If you need to include a standard file, use `#include "nameOfTheFile.h"` with quotation marks.

namespace declaration The statement `using namespace std;` allows the compiler to resolve names from a specified namespace without requiring the namespace prefix. For instance, this permits the use of `cout` instead of the fully qualified name `std::cout`.

the `main()` function This is the entry point of the program. It contains the initial instructions and various statements for both initialisation and execution. The first two lines within the `{.block.}` initialise variables, whilst the subsequent `while (..) {` construct implements a loop that repeatedly executes a block of statements. Finally, the result is output to standard output using the C++ syntax `cout <<`, and control returns to the operating system with exit code zero.

2·1.4 General Outline of a C++ Program

Every program we write in this course will follow the same skeleton: inclusion of libraries, declarations, and a `main()` function containing the instructions. Fixing this template now lets us concentrate later on what goes inside it. In the previous example `sum1to100.cpp` and almost the rest of our exercises, we will use the template:

```
// 13.nameOfThisProgram.cpp
// myShortName 2026-09-09
// Optionally a short description
// of the program or notes for its use

#include <iostream>
using namespace std;

int main() {
  // instructions to execute
  return 0;
}
```

2·2 Simple Data Types

Programs manipulate data, and every datum in C++ possesses a **type** that determines which values it can represent and which operations are valid for it. Simple (scalar and floating-point▲) types hold a single value and serve as the fundamental building blocks from which all other data structures are constructed.

▲A scalar type is one in which every value has a well-defined predecessor and successor. This concept applies naturally to letters and can be extended to Booleans by treating `false` as less than `true`, but it does not apply meaningfully to floating-point numbers as they are represented in computers—for instance, what is the predecessor of 3.1?

Most programming languages do not provide a comprehensive set of built-in types for encoding all possible forms of data. Programs must define more specific and complex structures to represent the information they work with, such as images, audio, or business databases. However, most programming languages, including C++, provide a selection of basic simple data types capable of encoding integers, real numbers, and characters.

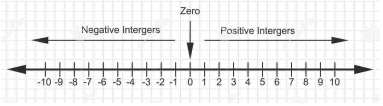
A simple type variable can only hold a single piece of information that cannot be subdivided. A physical vector, for example, can hold three simple real numbers. However, a real number itself is considered simple in that it is monolithic.

2·2.1 Predefined Simple Types

C++ provides a set of built-in types to handle characters and numbers (integers and real numbers):

Integers These represent whole numbers without a fractional part. Integers can be signed (`int`) or unsigned (`unsigned int`), and their storage capacity can vary from small (`short int`) to large (`long int`, `long long int`).

52: The integer numbers



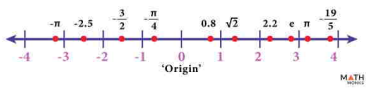
Real numbers Although many programs do not require numbers with a fractional part, real numbers are essential in many applications. However, they are more cumbersome to work with because they cannot be compared for equality reliably due to precision errors,

and they are computationally more expensive to manipulate. You can declare variables for real numbers using a smaller range (`float`) or a larger range (`double` or `long double`). Remember that

$$3.14 \times 10^{-3} \rightarrow \texttt{3.14E-3}$$

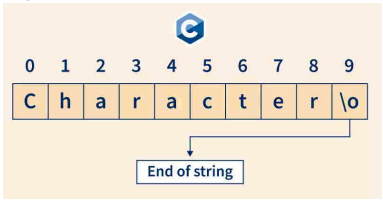
.

53: Real numbers



Characters Alphabet letters and basic writing symbols are encoded using the standard ASCII table and the more comprehensive UTF encodings, which support an extensive range of language-specific characters, mathematical symbols, and emojis. For the ASCII alphabet (the only character set our language syntax supports), you can store characters in variables of type `char`.

54: Characters, in classic C, always ending in `\0`. In C++ they are saved in the object `string`



Booleans The simplest data type, used to store the truth value of a condition, can be declared using the `bool` variable type.

```
if (a == b)
  cout << "a is equal to b";
bool isTrue = b > 5;
```

Sometimes it is convenient to write literal values with digits separated by `'` when they are long, to omit the leading `0` in `0.123`, or to omit the trailing zero after the only significant digit in `1.`, as shown below:

```
int n = 123'456;
float r = 3.141'592'653;
float p = .123;
float q = 1.;
```

However, this syntax is only supported from the C23 standard onwards.

```
float r = 2 + 3.f;
```

Since the constant `3.14` is `double` by default, if you need a literal `float` you must specify it explicitly:

```
float r = .123f;
```

In practice, C++ includes a standard library for handling complex numbers; however, we shall create them ourselves as an exercise.

```
#include <complex>
std::complex<double> z(3, 4);    // 3 + 4i
```

Conversely, C++ does not include a standard library for handling fractions. We can import them from external libraries, such as

```
#include <boost/rational.hpp>
boost::rational<int> a(1, 2), b(1, 3);
auto c = a + b;   // 5/6, automatically reduced
```

or implement them ourselves as an exercise. We will do so when we have covered structured data types.

Each type has its own range and memory footprint. Selecting the appropriate type represents the programmer's first design decision.

Encoding of:	Examples of values	Declaration Syntax
Booleans	`true` `false`	`bool`

Encoding of:	Examples of values	Declaration Syntax
Characters	a 9 ;	`char`
Natural Numbers	0 1 33	`unsigned [long] [int]`
Integer Numbers	-3 0 1000	`[long] int`
Real Numbers	$3.14 \times 10^{-3} \rightarrow$ 3.14E-3	`[long long] float/double`

You will see that the most frequently used programming type is `int`, and it is sufficiently versatile that `unsigned` is often substituted by it. Consequently, we typically use only `int` and overlook the necessity of `unsigned`△

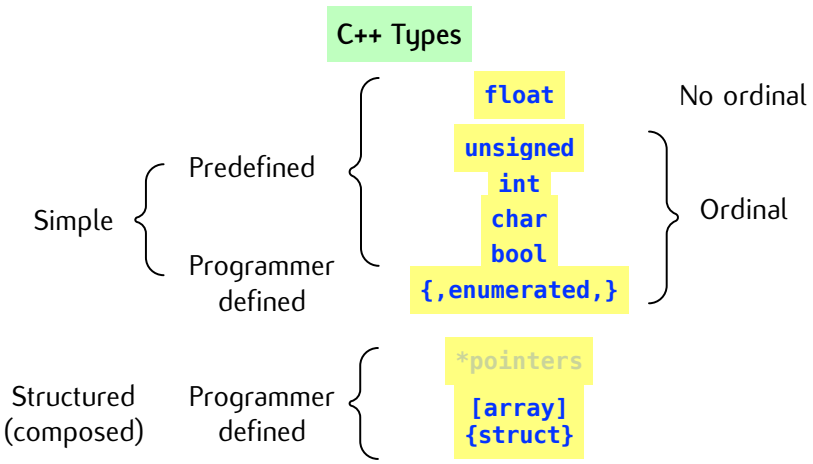
△Remember that assigning a negative number to an `unsigned` variable is quite harmful, as the result is completely incorrect.

2·2.2 Programmer-Defined Simple Types

Sometimes the predefined types do not adequately express our intent: a variable holding a day of the week is better declared as such than as a bare integer. Enumerated types let us create our own simple types, making programs clearer and safer.

Without venturing into structured multipart compositions of many simple types, we still have the possibility in C++ of creating a new simple type. We can enumerate the values a variable can hold. For example, a variable `trafficLightColor` can hold three possible values, and the best way to specify them is by their names: `GREEN`, `YELLOW` and `RED`. Our language is able to construct this list of labels (`GREEN`, `YELLOW` and `RED`) and ensure that the variable `trafficLightColor` holds one of them.

```
enum TLightColor {GREEN, YELLOW, RED};
TLightColor trafficLightColor = GREEN;
```



2·2.2.1 What a variable declaration implies

When we declare a variable, we are specifying four important things:

1) a name or identifier
2) a type, which determines how the information is encoded and interpreted
3) a size, which defines the capacity of the variable and, consequently, the amount of memory it reserves
4) an initial value with which to initialise the variable

```
int n = 10;
float r = 0.1;
```

Variables are not merely containers of information; we can operate on their contents to produce new values, typically of the same type as the operands, using operators. Each variable type has its own set of available operators.

2·2.3 Operators

Operators combine values—usually of the same type, occasionally of different types—to form expressions, which may be arithmetic, relational, or logical, depending on which operators are used. Each kind of expression draws on the operators appropriate to the types of the values involved.

Operators are the verbs of the language. We present them below, alongside their precedence rules, which determine the order in which a compound expression is evaluated.

In a programming language, we distinguish operators, which act upon values, from functions or procedures that act upon parameters. For example, `sin(x)` is the function `sin( )` acting upon the variable `x` and returning a result. Similarly, `x + y` uses the implicit binary operator `+`, which also returns a value. Operators can be unary, such as `-x`, or binary. The only ternary operator C++ provides is the **conditional operator** `?:`, which takes three operands: the condition before `?` and the two alternative values. This makes it a compact alternative to a full `if`/`else` statement when choosing between two values, though excessive nesting quickly reduces readability. We shall examine it, alongside `if`/`else`, in the section on conditionals.

The only ternary operator C++ provides is the **conditional operator** `?:`, and no other operator in the language takes three operands (the condition before `?` and the two outputs). This makes it a compact alternative to a full `if`/`else` statement when choosing between two values, though nesting it too deeply quickly hurts readability. We shall examine it, alongside `if`/`else`, in the section on conditionals.

Arity	Operators	Type
Unary	`++ --`	Unary Operator
Binary	`+ - * / %`	Arithmetic Operator
	`< <= > >= == !=`	Relational Operator
	`and or not`	Logical Operator
	`& | << >> ~ ^`	Bitwise Operator
	`= += -= *= /= %=`	Assignment Operator
Ternary	`?:`	Conditional

2·3 Constants, Variables and Assignments

Variables are named memory locations whose contents may change, whereas constants hold fixed values. The assignment instruction—placing a value into a memory location—represents the most fundamental operation in imperative programming , and merits closer examination than its straightforward appearance might suggest.

there exists another class of programming languages, less common in engineering but more prevalent in computer science research, known as *declarative* programming. In such languages, you do not specify instructions for the computer to execute; instead, you declare what you know and then query the system with questions about the expressed knowledge. These languages do not have an assignment operation

When declaring a variable, you can indicate that its value will never be modified. These are **constants**. Constants require you to specify their initial value at the point of declaration, and after this initialisation, the value cannot be changed.

```
const float PI = 3.14;
const char END = 'Q';
const int MAXSIZE = 101;
```

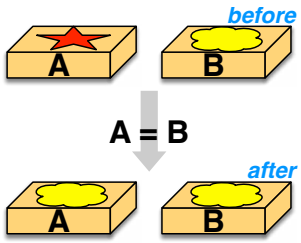
Any subsequent attempt to assign a new value to these constants will cause the compiler to emit an error and halt compilation. `PI = 3.141;`

We have already seen how to declare variables (specifying their type, identifying name, and usually an initial value), and where to declare the variables needed for our programs (close to the location where they are first used, so we understand their meaning not only from their identification name but also from their actual usage). We have also examined variable scope: the range of validity where variables can be used, and the blocks in which they are declared define the scope of that declaration.

2·3.1 The assignment operator

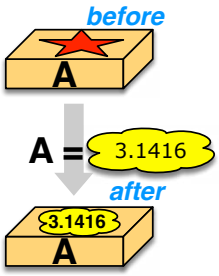
In programming languages, when you need to process information whilst preserving the original value, you typically copy the contents of one variable to another and then, if necessary, modify the copy through subsequent operations. Different languages employ varying syntax for this operation. Some use `let b = a`, interpreting the `=` sign as **copying** the value on the right into the variable on the left. Others use `b := a`. Our language, like many others, simplifies this frequently-used operation to: `b = a;` This does not indicate that the values are equivalent in a mathematical sense; rather, it signifies that after the operation executes, the contents of variable `b` shall equal the contents of variable `a`, regardless of what `b` previously contained. Following the operation, both variables hold identical contents at different locations in RAM memory.

55: Assignment of a value from a variable to another variable



You can assign many types of values to a variable, but not all of them. To assign a value to a variable, that value must be compatible with the variable's type.

56: Assignment of a value to a variable



When a constant is assigned to a variable, the literal value of the constant is copied into the variable, making the variable's contents equal to that value. Constants themselves are ephemeral: they exist only during the execution of the operation and have no persistent memory location that can be inspected or modified afterwards. Variables, by contrast, occupy a fixed, named memory location that remains accessible through their identifier throughout their lifetime—which is what permits them to be modified repeatedly as the program executes.

We can only copy things **into** variables: at the left of the `=` sign, there can only ever be a variable.

At the right of the `=` sign, however, there can be either a constant (`a = 3;`) or an expression that produces a value compatible with the variable on the left (`a = 2 + 5 * b;`).

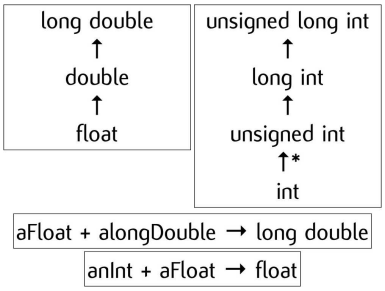
2·3.2 Type Conversions

What happens when an integer meets a real number in the same expression? C++ converts values from one type to another, sometimes silently and sometimes at our request. Understanding these conversions avoids subtle and common errors.

Often, the target on the left and the value on the right are simply incompatible, and the compiler issues an error: `aLetter = 3.1416;` makes no sense to the compiler.

In other cases, although the types differ, the compiler automatically performs a conversion of its own selection—often sensible, but not necessarily the one you intended. For example, the C++ compiler can convert an unsigned integer to a signed integer with reasonable safety, but the converse—converting a signed integer to an unsigned integer—can be problematic: `unsigned int a = -3;` stores `4294967293` in the variable `a`. Similarly, `int a = 3.9999;` stores `3` in `a`, truncating rather than rounding the fractional part.

57: Implicit type conversions C++ performs automatically. Some of them (`int` ⟶ `unsigned`) are particularly hazardous



In practical cases

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

 i = i  +  c;  // c -> int
 i = i  +  s;  // s -> int
 u = u  +  i;  // i -> unsigned int
 l = l  +  u;  // u -> long int
ul = ul +  l;  // l -> unsigned long int
 f = f  + ul;  // ul -> float
 d = d  +  f;  // f -> double
ld = ld +  d;  // d -> long double
```

2·3.2.1 Casting or explicit conversions

We can read the contents of a variable by converting it to another type, treating it as if it were being assigned to that type. In this way, we can write `int(aRealVariable)` to convert the contents of `aRealVariable` to an integer before performing further operations.

An important example is: `float r = float(1)/3;`. In this case, without explicit casting, `float r = 1/3;` would assign `0` to the variable `r`, since the division is performed on integer operands (`1` and `3`), resulting in integer division with a result of `0`.

We must pay particular attention to type combinations. Casting and implicit conversion play a crucial role in ensuring the correctness of results.

2·3.3 Integer operators

Integers are the most important data type in all languages. They are the most natural encoding in computers. Many things that are not themselves numbers still end up encoded as integers, so that we can compare, add or subtract them. Priorities, for example, are usually just `0`, `1`, `2`, ..., `10` or so.

We have the basic arithmetic operators `+`, `-` (which also works as unary negation), `*` (yes, multiplication is `*`, not `x`) and `/` (integer division):

```
int i = 0;
int j, k;

j = i + 1;
k = -j;  // unary minus: k is now -1
```

but be careful here: whenever every operand is an integer, the result is always an integer too, so

```
cout << 7 / 3 << endl; // will print 2, no fract. part
```

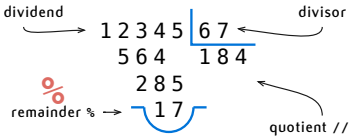
2·3.3.1 Modulus, or remainder

When dividing two integers, the fractional part is discarded (`7/3` gives `2`). To address this limitation, there is a fundamental operator — arguably even more widely used than the four basic arithmetic operations — the **modulus** or **remainder** operator `%`, which returns the remainder of integer division between two integers:

```
cout << 7 % 3 << endl; // will print 1
```

This yields the remainder when dividing 7 by 3. The modulus operator is particularly useful because it allows us to answer questions such as *is this number even?* (does `n % 2` equal `0`?) or *have we just reached column 80, counting with pos?* (does `pos % 80` equal `0`?).

58: The modulus operator is invaluable: it returns the remainder of integer division.
`12345 / 67` ⟶ `184`
whilst
`12345 % 67` ⟶ `17`



Division and modulus are tied together by an identity worth remembering — itself a small invariant, of the kind we will rely on constantly once we reach loops: for any integer `n` and any non-zero divisor `d`:

```
n == (n / d) * d + n % d
```

always holds. Whatever `/` discards, `%` gives back.

Since C++ 11, `/` truncates towards zero and `%` follows the same rule, so negative operands behave predictably:

```
cout << -7 / 2 << endl;  // -3
cout << -7 % 2 << endl;  // -1
cout <<  7 % -2 << endl; //  1
```

Before C++ 11, the rounding direction for negative operands was implementation-defined — one more reason not to trust code that relies on it without checking which standard is in force.

2·3.3.2 Increment and decrement

Apart from the new operator `%`, we have the pre/post-increment and pre/post-decrement operators, `++i`, `i++`, `--i`, `i--`. Each of the following starts afresh from `i = 1`:

```
int i = 1;
// pre-increment:
cout << ++i << endl; // i becomes 2 first, then 2 is printed
```

```
int i = 1;
// post-increment:
cout << i++ << endl; // 1 is printed first, then i becomes 2
```

```
int i = 1;
// pre-decrement:
cout << --i << endl; // i becomes 0 first, then 0 is printed
```

```
int i = 1;
// post-decrement:
cout << i-- << endl; // 1 is printed first, then i becomes 0
```

What does the following program print:

```
1 #include <iostream>                                    C++
2 using namespace std;
3
4 int main()
5 {
6     int i = 3;
7     cout << ++i << endl;
8
```

Never modify the same variable more than once inside a single expression, as in `cout << i++ + i++;`. The C++ standard does not guarantee an evaluation order here, so the result may differ between compilers — or even between optimisation levels of the same compiler.

```
9       i = 3;
10      cout << i++ << endl;
11      cout << i << endl;
12
13      i = 3;
14      cout << i++ + i << endl;
15      cout << i << endl;
16
17      return 0;
18  }
```

However, be aware of the warning that the linter and compiler will issue on line 14:

```
clang++: warning                                                    Display
warning - unsequenced modification and access to 'i' [-Wunsequenced]
```

This warning indicates that the order of operations in the sum is undefined across different compilers and may yield different results. Typically, it will print 7 (4+3), since `i` is post-incremented—that is, incremented after the entire addition is evaluated.

2·3.3.3 Compound assignment operators

And, although not exclusive to integers, the following operators can also be used with real numbers and `char`s (which behave as small integers in arithmetic) —
`+=` `-=` `*=` `/=` `%=`, etc.
they are shortcuts for an assignment combined with an operation on the same variable. Let us see an example, applying every one of them in turn:

`%=` — like `%` itself — demands integer operands: you cannot write `x %= 2.5;`. For real numbers, the equivalent is the function `fmod` from the `<cmath>` library, which we meet again in the next section.

```
int i = 20;
i += 5; // i becomes 25
i -= 3; // i becomes 22
i *= 2; // i becomes 44
i /= 4; // i becomes 11
i %= 4; // i becomes 3
```

it is not only a shortcut for writing: the compiler also tends to produce more efficient code this way.

Exercise caution with the following expression:

```
result *= a + 2;
```

`result` is multiplied by the sum of `a + 2`. Note that compound assignment operators have lower precedence than arithmetic operators, so the statement above is equivalent to

```
result = result * (a + 2);
```

And:

Combined	Equivalence
`y -= a / 2;`	`y = y - (a / 2);`
`a += b - 2;`	`a = a + (b - 2);`
`v *= 100 - p;`	`v = v * (100 - p);`
`a %= b - 1;`	`a = a % (b - 1);`
`a /= b + c;`	`a = a / (b + c);`

2·3.4 Multiple assignment

C++ allows you to assign a value (or the result of an expression) to multiple variables simultaneously. This is a frequently used convenient shortcut that simplifies code writing and is particularly appropriate in initializations.

```
cnt = sum = i = 0;
```

assigns the right-hand-side value to each variable on its left.

2·3.5 Integer overflow

Every integer type has a limited range (we tabulate them shortly), and the arithmetic operators never check whether a result still fits inside it:

```
unsigned char u = 255;
u = u + 1;
```

In C, characters share many characteristics with integers. The previous sum wraps around to `0`, this constitutes well-defined behaviour for unsigned types, yet it is mathematically nonsensical.

And

```
int n = 2147483647; // INT_MAX
n = n + 1;
```

Signed integer overflow is undefined behaviour, which in practice typically results in `-2147483648` on modern machines, is mathematically nonsensical and produces a confusing result.

Wrap-around for unsigned types is guaranteed by the standard; overflowing a signed type is not — the compiler is free to assume it never happens, which is precisely what makes it dangerous.

2·3.6 Two's Complement

The reason for this weird behaviour in integers (not so weird the behaviour in unsigned integers) is because of their encoding. An unsigned short integer (we omit the higher byte) like `3` is simply encoded as `00000011` but an integer like `-3` is encoded like: `11111101`

This is **two's complement**: the leftmost bit carries a negative weight — here, looking at a single byte, -128 instead of $+128$ — and the remaining bits are added as usual. So `11111101` is worth $-128+64+32+16+8+4+0+1=-3$.

Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1100	1101	1111
unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
int	0	1	2	3	4	5	6	7	−8	−7	−6	−5	−4	−3	−2	−1

Nobody adds eight weighted terms by hand every time, though. There is a mechanical shortcut worth memorising to understand dumps or debugging bitwise code. To encode $-n$:

1. write n in binary,
2. **flip every bit** (`0`s become `1`s and vice versa — this halfway step is called the **one's complement**),
3. then **add 1**:

```
  00000011   (3)                                                  Display
  11111100   (flip every bit: one's complement)
+ 00000001
------------
  11111101   (-3, two's complement)
```

This little recipe also explains why signed overflow looks so surprising. Take the largest positive byte, `01111111` (127 — exactly the `char` maximum in the table below), and add 1 the ordinary way:

```
  01111111   (127)                                                Display
+ 00000001
------------
  10000000
```

Read as an **unsigned** pattern, `10000000` is simply 128 — nothing odd there. But the moment the leftmost bit switches on, it stops being an ordinary digit and becomes the sign bit, worth −128: so that very same pattern, read as signed, is −128 (the `char` minimum). One more than 127 is −128 — exactly the **signed overflow** from the previous section, and exactly why the standard refuses to define it: the bits are unambiguous, but their meaning depends entirely on whether the type is signed or unsigned, and the two readings could not disagree more. The very same thing happens, one bit wider, when `int n = INT_MAX; n = n + 1;`: the sign bit flips on and `n` becomes `INT_MIN`.

2·3.7 Checkpoint

1) Describe the difference between a language keyword and a programmer-defined identifier.
2) What is the difference between a program line and a statement?
3) Explain why variables are called “variables” and what aspect of them can be changed.
4) What do variables contain when we declare their name and type?
5) Which of the following are legal C++ assignment statements?
   - 5.a) `a = 2;`
   - 5.b) `13 = x;`
   - 5.c) `2 = 2;`
   - 5.d) `x == 2;`
6) List all the variables and literals that appear below.

```
int main()
{
   int time;
   int space;

   time = 0;
   space = 10;
   cout << "Time is " << time << endl;
   cout << "Space is " << space << endl;
```

```
    return 0;
}
```

7) Write a multiple assignment statement that assigns `0` to the variables `counter`, `sum`, `totalPrice`, and `totalTaxes`.

8) Write statements using combined assignment operators to perform the following:

   8.a) Subtract `-4` from `price`.

   8.b) Add `-6` to `x`.

   8.c) Multiply `totalTaxes` by $1.2 \cdot 10^{-3}$.

   8.d) Divide `price` by 1.5.

   8.e) Store in `x` the remainder of `x` divided by 7.

   8.f) Add 3 times `totalTaxes` times `price` to `total`.

   8.g) Add `y / 5` to `y`.

   8.h) Divide `total` by the result of `sume` minus `10`.

   8.i) Multiply `sum` by 6 times `totalTaxes`.

9) What will the following program segment display?

```
int a, b, c;
a = b = c = 7;
b *= 2;
c -= 4;
a += 4;
a /= 3;
cout << "a: " << a << endl << "b: " << b << endl << "c: " << c << endl;
```

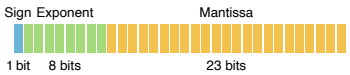
2·3.8 Float operators

The floating-point values that C++ can handle directly fall within these ranges:

	`float`	`double`	`long double`
bytes	4	8	16
exp bits	8	11	15
mant bits	23	52	112
exp bias	127	1023	16383
min	-3.40282347e+38	-1.7976931348623157e+308	-1.189731495357231765085759326628007 02e+4932
MAX	3.40282347e+38	1.7976931348623157e+308	1.189731495357231765085759326628007 02e+4932

For `double`, the internal representation of bits would be as follows:

- **`float`** 4 bytes:



- **`double`** 8 bytes:



- **`long double`** 16 bytes:



As a long double (128 bits) the number `1.0` is encoded internally as:
1 bit sign + 15 exponent + 112 fraction.
The exponent bias is 16383 = 0x3fff:

```
0 3fff 0000 0000 0000 0000 0000 0000 0000                    Display
= 1.0
```

The encoding and internal representation of floating-point values is substantially more intricate than that of integers.

In most cases, we will use the `#include <cmath>` library for floating-point operations. However, some basic operations, such as squaring or cubing a float, execute significantly faster when performed directly: `r * r` instead of `pow(r, 2)`, or `r * r * r` instead of `pow(r, 3)`. The `cmath` library provides definitions for functions such as `sqrt(x)` (the square root of `x`) and `sin(x)`, but it does not include certain transcendental mathematical functions, such as factorial, which we will implement ourselves throughout this course.

2·3.9 Formatted displaying

To make each value occupy a fixed, predictable number of columns when printed, C++ provides two useful stream manipulators:

`setw(w)` sets the width of the *next* output only, padding it with spaces (right-aligned by default) until it occupies `w` characters. Unlike most stream state, the width resets after each use, so `setw()` must be called again before every value you want padded.

`setprecision(w)` sets how many digits are shown for floating-point numbers. By default, `w` is the total number of *significant* digits displayed

/`fixed`: combined with `setprecision(w)`, `w` becomes the number of digits *after* the decimal point. `fixed` indicates that floating-point output should be printed in fixed-point, or decimal, notation.

`left` there are times when you may wish to force a value to print on the left side of its field, padded by blanks on the right. To do this you can use the `left` manipulator. It remains in effect until you use a `right` manipulator to set it back. `left`/`right` can be used with any type.

These manipulators• are defined in the header `<iomanip>`.

• Stream manipulators are ordinary functions that, when sent to a stream via `<<`, change the stream's internal formatting state rather than printing a value themselves—here, they modify the state of `cout`.

With floats, by default, `cout` shows only 6 significant digits, regardless of how many the type actually stores. If you read a value and print it back with no further thought, you may not recognise your own number:

```cpp
// iomanip.cpp                                                    C++
// juanfc 2026-09-09
//
#include <iostream>
#include <iomanip>   // for setprecision
using namespace std;

int main()
{
    double x = 3.98765999;
```

```
    cout << "'" << setw(8) << 42 << "'" << endl;    // '      42'(spaces padding)
    cout << x << endl;                              // 3.98766   (general)
    cout << setprecision(2) << x << endl;           // 4      (2 digits rounding)
    cout << setprecision(3) << x << endl;           // 3.99   (3 digits in total)
    cout << fixed << setprecision(3) << x << endl;  // 3.988  (3 decimals)

    return 0;
}
```

Another example:

```
// readFloatPrecision.cpp                                                    C++
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  float f;
  cout << "Enter a real number: ";
  cin >> f;

  cout << "Without setprecision: " << f << endl;                  // 6 significant digits
  cout << "With setprecision(9): " << setprecision(9) << f << endl; // up to float's own precision
  return 0;
}
```

Enter, say, `3.14159265` (as many digits of π as you can remember): the first line prints a rounded `3.14159`, while the second reveals what `f` actually stores — `3.14159274`, the nearest a `float` can get to π, and not a digit more, since its 23 mantissa bits only buy about 9 reliable significant digits. `setprecision`, from `<iomanip>`, is **sticky**: once set, it stays in effect for every `cout` that follows until changed again — so reset it afterwards if the rest of your output should not stay at 9 digits.

Scientific notation the mantissa is normalised to exactly one nonzero digit before the decimal point, so $1 \leq |m| < 10$, and the exponent can be any integer. Example: `47000` $\longrightarrow 4.7 \times 10^4$.

Engineering notation the exponent is restricted to multiples of 3, so the mantissa extends to $1 \leq |m| < 1000$. Example: `47000` $\longrightarrow 47 \times 10^3$. The purpose of the `×3` steps is that the exponents align with SI prefixes (10^3 = kilo, 10^6 = mega, 10^{-3} = milli, 10^{-6} = micro…), so 47×10^3 reads directly as 47 k.

Engineering notation is therefore a variant of scientific notation in which the exponent is always a multiple of three (and the mantissa lies in the range $1 \leq |m| < 1000$), so that each power of ten corresponds to an SI (International System) prefix such as kilo, mega, milli, or micro. In C++, we have `std::scientific`

2·3.9.1 Checkpoint

1) Write `cout` statements with stream manipulators that perform the following:

   1.a) Display the number `34.789` in nine spaces with two decimal places of precision.

1.b) Display the number `7.0` in a field of eight spaces with three two places of precision. The decimal point and any trailing zeros should be displayed.

1.c) Display the number `3.289e+12` in fixed-point notation.

1.d) Display the number `76` left-justified in a field of five spaces.

2) The following program segment converts an angle in degrees to radians.

```
const float PI = 3.14159;
float degrees, radians;

cout << "Enter an angle in degrees and I will convert it" << endl;
cout << "to radians for you: ";
cin >> degrees;
radians = degrees * PI / 180;

// Display the value in radians left-justified, in fixed-point
// notation, with four decimal places of precision,
// in seven spaces wide.
```

2·3.9.2 Limits for the contents of the basic types

You can find the range limits of the integer and float different sizes with this program:

```C++
#include <iostream>
#include <climits>
#include <cfloat>
using namespace std;

int main()
{
    cout << USHRT_MAX << endl;  //  65 535
    cout << SHRT_MIN << endl;   // -32 768
    cout << SHRT_MIN << endl;   //  32 767
    cout << INT_MIN << endl;    // -2 147 483 648
    cout << INT_MAX << endl;    //  2 147 483 647
    cout << ULONG_MAX << endl;  //  18 446 744 073 709 551 615
    cout << FLT_MAX << endl;    //  3.40282e+38
    cout << DBL_MAX << endl;    //  1.79769e+308
    cout << LDBL_MAX << endl;   //  1.18973e+4932
    return 0;
}
```

A not so frequent needed info is the amount of bytes each variant demands:

```C++
// sizesof.cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Sizes in bytes of:" << endl;
    cout << "char:        " << sizeof(char)        << endl;  //  1
    cout << "short int:   " << sizeof(short)       << endl;  //  2
    cout << "int:         " << sizeof(int)         << endl;  //  4
    cout << "long int:    " << sizeof(long)        << endl;  //  8
    cout << "float:       " << sizeof(float)       << endl;  //  4
    cout << "double:      " << sizeof(double)      << endl;  //  8
    cout << "long double: " << sizeof(long double) << endl;  // 16
    return 0;
}
```

And in general, the sizes and capacities for all basic types are as follows:

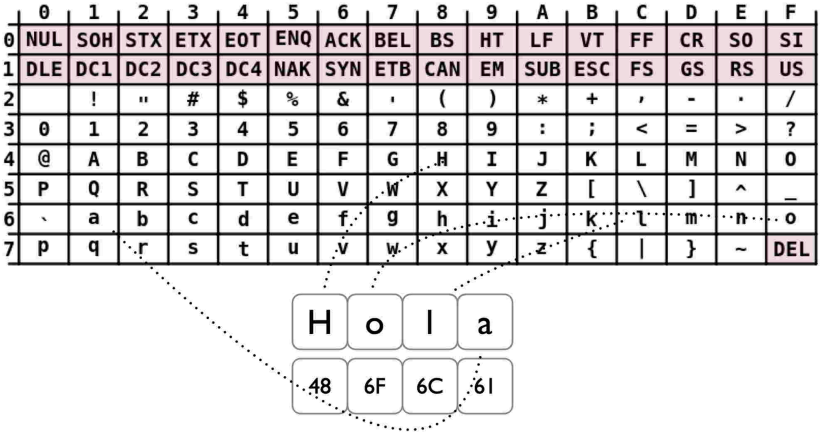
	bytes	min	max
bool	1	false	true
char	1	-128	127
unsigned char	1	0	255
short	2	-32,768	32,767
unsigned short	2	0	65,535
int	4	$-2{,}147_{1}483{,}648$	$2{,}147_{1}483{,}647$
unsigned	4	0	$4{,}294_{1}967{,}295$
long	4	$-2{,}147_{1}483{,}648$	$2{,}147_{1}483{,}647$
long long	8	$-9_{3}223{,}372_{2}036{,}854_{1}775{,}808$	$9_{3}223{,}372_{2}036{,}854_{1}775{,}807$
unsigned long long	8	0	$18_{3}446{,}744_{2}073{,}709_{1}551{,}615$
float	4	-3.40282e+38	3.40282e+38
double	8	-1.79769e+308	1.79769e+308
long double	16	-1.18973e+4932	1.18973e+4932

2·4 Characters and strings

We have learnt that there are two basic data types: numbers and the other is chars. Internally, of course, chars are encoded as we have seen in Section 2·2.1 as simple integers, but their treatment do not include summation, or division, but operations for reading/writing chars from/to the keyboard/screen. To refer to a single char, we enclose its literal char between single quotes:

```
char letter        = 'A',
     anotherLetter = '-';
```

Characters have a fundamental relationship with integers because they are internally represented as integer values. Each printable character, along with many non-printable characters, is assigned a unique numeric code. The most widely used character encoding scheme is ASCII, which stands for the American Standard Code for Information Interchange. When a character is stored in memory, the corresponding numeric code is stored rather than the character itself. When the computer is instructed to print a character value, it retrieves the integer code and maps it to the corresponding entry in the ASCII table, then renders the character on the screen. Appendix A, located at the end of this text, contains the complete ASCII character set, allowing you to identify which integer value represents each character. Note that the integer 65 represents capital A, 66 represents capital B, and so forth.




```
#include <iostream>                                                    C++
using namespace std;
```

```
int main()
{
   char aLetter;

   // We will not do this in our programs, but this is possible in C
   aLetter = 65;              // 65 is the ASCII code for the character A
   cout << aLetter << endl;  // A
   cout << char(67) << endl; // (integer forced to be a char) prints C

   return 0;
}
```

A straightforward and efficient approach to convert a lowercase or uppercase letter to its corresponding uppercase or lowercase counterpart is to add the ASCII code distance between them, since they are arranged in parallel:

```C++
#include <iostream>
using namespace std;

int main()
{
    char c;
    cout << "Enter a letter: ";
    cin >> c;

    char uppercaseC = c;
    if ('a' <= c and c <= 'z') {
       uppercaseC += ('A' - 'a');
     }

    cout << "Uppercase of " << c
         << " is " << uppercaseC << endl;

    return 0;
}
```

This is possible due to implicit conversions (`char` ⟶ `int` for the addition and subtraction operations, with the resulting `int` converted back to `char` for the assignment)◦.

◦ Some prefer to use the distance (32) as a magic number they memorise to perform this uppercase/lowercase conversion. Avoiding reliance on obscure memorised constants makes programs more readable and self-documenting.

2·4.1 Checkpoint

1) Which of the following are illegal C++ variable names, and why?
   - 1.a) `n`
   - 1.b) `1length`
   - 1.c) `sept26`
   - 1.d) `theNameOfThisVariableIsVerbose`
   - 1.e) `r&d`
   - 1.f) `_gradeIP`
   - 1.g) `_0_1_2_`
2) Are the variables `Height` and `height` considered the same?
3) Why do we prefer signed integers (`int`) to unsigned integers (`unsigned`) when representing unsigned values?

4) How would you combine the following variable declaration and assignment into a single statement?

```
int counter;
counter = 20;
```

5) How would you combine the following variable definitions into a single statement?

```
float xForce = 0;
float yForce = -10;
float zForce = 6;
```

6) What are the ASCII codes for the following characters?

6.a) `X`

6.b) `)`

6.c) `-`

6.d) `0`

7) What is wrong with the following program statement?: `char c = "X";`

8) How many bytes do each of the following literals use?

8.a) `'A'`

8.b) `"a"`

8.c) `"length"`

8.d) `'\n'`

9) What header file must you include to use string objects? Is it necessary in all compilers?

10) What value will be assigned to `x` in each of the following statements?

10.a) `x = 7 + 3;`

10.b) `x = 7 / 3;`

10.c) `x = 7 % 3;`

11) Is the following an example of integer division or floating-point division? What value will be displayed?: `cout << 16 / 3 << endl;`

2·5 Enumerating the values a type can hold

Imagine we want to work with a few colours. How would you declare the possible content of a variable of type `colour` so that when you assign a colour such as red or green, the code is easy to follow? If we declare

```
int colour;

colour = 1;
colour = 3;
```

this is not easy to understand.

When the values we handle are neither numbers nor letters and are very few in number, we can enumerate the possible values and refer to each

of them using labels or identifiers that will, from that point onwards in our program, represent a particular value suitable for assignment to that enumerated type.

```
enum TColour {BLUE, GREEN, RED, BROWN};
```

You have here 5 new names, identifiers, for your program: the 4 new labels for each of the values and the name of the new type.

Typical examples in which you can see how useful can enumerated types be:

```
enum TMonth     {JANUARY=1, FEBRUARY, MARCH, APRIL,
                 MAY, JUNE, JULY, AUGUST,
                 SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER};

enum TWeekDay   {MONDAY=1, TUESDAY, WEDNESDAY,
                 THURSDAY, FRIDAY, SATURDAY, SUNDAY};

enum TPlanets   {MERCURY, VENUS, EARTH, MARS,
                 JUPITER, SATURN, URANUS, NEPTUNE};

enum TState     {LOW=-10, HIGH=10, UNDEF};
```

and an example of their use:

C++
```
#include <iostream>
using namespace std;

enum TColour {BLUE, GREEN, RED, BROWN};

int main()
{
    TColour colorSky, colorGround;
    colorSky = BLUE;
    colorGround = BROWN;

    cout << colorSky << endl;
    cout << colorGround << endl;

    return 0;
}
```

If we do not assign values to the labels, enums are encoded internally as the sequence 0, 1, 2, ... However, we can assign any integer value to any or all of the labels. A label without an explicit value gets the previous value plus one (for example, in enum `TState {LOW=-10, HIGH=10, UNDEF}`, `UNDEF` is 11). This is useful when those integer values have semantic significance.

2·5.1 Checkpoint

1) Identify all errors in the following declaration.

```
Enum Pet = { "dog", "cat", "bird", "fish" }
```

2) Define an enumerated type to represent Spanish educational grades, ranging from `SUSPENSO` to `SOBRESALIENTE`, assigning each label the maximum value it represents so that later in a program you can compare scores against them to make decisions (for example, `mark > SUSPENSO`).

2·6 typedef: Defining new names for types

typedef

The following sentences define new types based on previously known types, or alternatively provide new names for existing types:

```
typedef int   TCounter;
typedef float TVol;
typedef TVol  TSpace;
```

From this point forward, `TCounter` is equivalent to `int`, but declaring a variable using `TCounter` significantly improves program readability:

```
TCounter cnt;
```

compared with declaring it as `int`.

The first reason to use `typedef` is to improve readability whilst simultaneously enabling parametrisation and creating more flexible programs. For example, suppose you declare a series of variables of type `TSurface` and later discover that it is better to handle these quantities (and only these) as `double` rather than `float`. You need only change the type once in the

```
typedef double TSurface;
```

declaration.

```
float l, s, v;  // are poor declarations of...
typedef float TLength;
typedef float TSurface;
typedef float TVolume;
// latter you will build variables based on these new names of types
TLength l;
TSurface s;
TVolume v;
```

In a program to handle geometrical quantities:

```
#include <iostream>
using namespace std;

typedef int TBase;
typedef TBase TArea;
typedef TBase TVolume;
typedef TBase TLength;

int main()
{
    TLength a, b, c;
    TArea s1, s2;
    TVolume v;
    // …
    return 0;
}
```

Or a situation in which you have an unwieldy type name like `unsigned short int`:

```
#include <iostream>                                                    C++
using namespace std;

// typedefs
typedef unsigned short int Tushort;

int main()
{
    Tushort width = 5;
    Tushort length;
```

```
    length = 10;
    Tushort area  = width * length;

    cout << "width:  " << width  << endl;
    cout << "length: " << length << endl;
    cout << "area:   " << area   << endl;
}
```

However, the importance of `typedef` extends beyond these two considerations. We will see later that with complex types, the use of aliases (which is all `typedef` creates) is essential.

2·7 Random numbers

Although computers cannot generate truly random numbers, it is important for both games and testing to expose our programs to random values. Random numbers, or *pseudo-random* numbers as they are properly called, are values from a mathematically generated sequence that do not follow any discernible pattern and are approximately guaranteed to be stochastically uniform; that is, their values are uniformly distributed. If we run our program and request a random value, it will always produce the same result, and the second call will yield the same value, and so on. To obtain different values each time we execute our program, we use what is called a *seed*. We seed the sequence by providing a starting point, and to ensure the sequence differs with each execution, we typically use the system time.

```C++
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
    srand(time(0)); // seeds the seq. to the system time
    for (int i = 0; i < 10; ++i) {
        cout << rand() << endl;
    }
    return 0;
}
```

`rand()` returns an unsigned integer between `0` and a maximum integer (`RAND_MAX`). To generate a random value within a specific range—for example, to simulate a dice roll—we use the formula:

```
number = (rand() % (maxVal - minVal + 1)) + minVal;
```

and for a dice roll specifically:

```
cout << rand() % 6 + 1 << endl;
```

2·8 Operators precedence

When an expression combines several operators, the compiler must decide which one to apply first. Exactly as in ordinary arithmetic `2 + 5 * 3` means `2 + 15` and not `21`, every operator carries a **precedence**: the higher its precedence, the tighter it binds to its operands, and the sooner it is applied. When two operators share the same precedence, a second rule, **associativity**, breaks the tie — for most operators, left to right.

Precedence	Operators	Assoc.
highest	`( )` grouping, function call, done before any other	
	`not` unary `-` `++` `--`	right
	`*` `/` `%`	left
	`+` `-`	left
	`<` `<=` `>` `>=`	left
	`==` `!=`	left
	`and`	left
	`or`	left
lowest	`=` `+=` `-=` `*=` `/=` `%=`	right

Two consequences of this table are worth keeping in mind. First, **multiplicative operators bind before additive ones**, so `result = 2 + 5 * b;` is read as `result = 2 + (5 * b);`: the product is formed first and only then added to 2. Second, **assignment sits at the very bottom**, so the whole right-hand side is evaluated before anything is stored — which is also why a compound assignment such as `x *= a + 2;` means `x = x * (a + 2);`, not `x = (x * a) + 2;`.

Associativity. Most binary operators are **left-associative**, so a chain groups from the left:

```
10 - 4 - 3   // means (10 - 4) - 3  ->  3,  not  10 - (4 - 3)  ->  9
```

Assignment is the notable exception: it is **right-associative**, which is precisely what makes a multiple assignment work. `a = b = c = 7;` groups as `a = (b = (c = 7));`, so the 7 travels right to left into all three variables.

Relational before logical. Comparisons bind tighter than `and`/`or`, so a range test reads naturally with no parentheses at all:

```
if (x >= 0 and x <= 100) ...   // (x >= 0) and (x <= 100)
```

The unary `not` is the trap here: being unary, it binds **very** tightly — tighter than `==` — so it grabs only the token immediately to its right:

```
not x == y     // means (not x) == y  — almost certainly NOT what you meant
not (x == y)   // this is the intended test
```

Whenever the grouping is not obvious at a glance, do not make the reader (or yourself) recall the whole table — add parentheses. They cost nothing at run time and state your intent plainly◆.

◆ One precedence trap deserves a special mention, because it involves an operator you already use constantly: `<<`. It binds tighter than the comparison operators, so `cout << x == 3;` is read as `(cout << x) == 3;` and will not do what you expect. Write `cout << (x == 3);` when you want to print the result of a comparison.

2·8.1 Checkpoint

1) In the expression `2 + 5 * b`, which operation is performed first, and why?
2) Rewrite each expression with explicit parentheses showing the grouping the compiler actually uses:
   2.a) `a - b - c`

2.b) `a = b = c`

2.c) `x < y and y < z`

2.d) `not a == b`

3) What value does `10 - 4 - 3` produce? What would it produce instead if `-` were right-associative?

4) Explain why `a = b = c = 7;` stores 7 in all three variables. Which property makes this possible?

5) Add the minimum parentheses needed so that this test means "`x` is outside the range `0` to `100`":

```
not x >= 0 and x <= 100
```

6) What does `cout << x == 3;` actually do, and how would you fix it so that it prints whether `x` equals 3?

7) Showing the order of operations step by step, what value is stored in `r`?

```
int a = 2, b = 3, c = 4;
int r = a + b * c - 6 / 3;
```

8) What are the values of the following expressions:

8.a) `9 - 1 * 5`

8.b) `6 / 2 - 4`

8.c) `9 + 9 * 2 - 6`

8.d) `5 + 19 % 3 - 1`

8.e) `8%(6+2)*22`

8.f) `44/(11-4)`

8.g) `(8+17) % 1-3`

8.h) `(3-9)*(9+9)/3`

2·9 Basic Input and Output

A program that cannot communicate with its external environment has limited practical utility. We introduce the standard streams `cin` and `cout`, which enable programs to read input from the keyboard and write output to the screen. Although numerous alternative methods exist for interacting with the computer, we focus on the most universally applicable approach: text-based terminal communication. This choice reflects both the simplicity and broad applicability across diverse and evolving application domains. Addressing graphical dialogues and complex user interfaces would distract from the fundamental principles of programming. In fact, the aforementioned interaction mechanism (`cin`/`cout`) will be used in the initial programs. In later exercises on this course, we shall typically assign values to the relevant variables directly within the program using "quick initialisation", enabling us to verify our algorithm functionality more efficiently in important cases without the repetitive overhead of manual data entry each time we execute the program.

59: Keyboard and screen are the standard input/output in our language



C++ delivers, out of the box, the code needed to read and write the basic types from and to "standard input/output":

- `char`
- `int` (and its different variants)
- `float` (and its different variants)

```
int i;
char c;
float f;

cin >> i; // will halt the program and wait for you to enter a valid int
cin >> c; // will halt the program and wait for you to enter a valid char
          // however, it will never read whitespace characters:
          //      not ' ', not newlines, not tabs
cin >> f; // will halt the program and wait for you to enter a valid real
```

Try this with:

```
// basicReading.cpp                                                    C++
#include <iostream>
using namespace std;
int main()
{
  int i;
  char c;
  float f;

  cin >> i; cout << "i: " << i << endl;
  cin >> c; cout << "c: " << c << endl;
  cin >> f; cout << "f: " << f << endl;
  return 0;
}
```

and do not hesitate to try tricking it by entering the wrong kind of thing, to see what your program does in each case.

Remember that `cin` will not read whitespace characters, which include the space character (represented as `' '`), the newline character, and the tab character. If you provide only spaces or press enter repeatedly, `cin` will continue waiting for input. The same applies to the tab key (the long horizontal key on the left side of your keyboard).

2·9.1 Reading/writing characters and strings

Remember that `cin` skips over and ignores any leading whitespace characters (spaces, tabs, or newlines) when reading data. However, once it encounters the first non-blank character and begins reading, it stops when it reaches the next whitespace character.

To handle sequences of characters, we use `string` objects. They are much more convenient than the classical approach of using arrays of characters ◇. The most popular implementations of C++ (GCC, Clang, and MSVC) do not require `#include <string>` at the beginning, as `string` is included transitively through `<iostream>`, which we always include. Therefore, unless your compiler raises an error when declaring a string, you can omit it. We only need to know that strings can be read and written using `cin` and `cout`, and that string literals are enclosed in double quotes:

◇ classically, C uses arrays of characters to handle strings, but arrays are less practical and lead to significant complexity when we want to manipulate them in various ways

```
string myName = "Juan",
       yourName = ""; // unnecessary, as strings are
                      // always guaranteed to initialise empty
```

If we need to read a string that includes spaces in it, such as `"Juan Antonio"` or similar ones, we will need to use a different approach:

```
string yourAddress;
getline(cin, yourAddress); // will include everything the user inputs
                           // until they press enter
```

When C++ encounters something it cannot parse, it does not politely tell you that you have made a mistake and ask you to try again: it silently extracts as much as it can and carries on — quietly, and usually wrongly, unless we check for it. What "as much as it can" means depends on the mistake:

- A stray letter in the middle of a number (`12a34` typed for an `int`) stops the read at the letter: `i` becomes `12`, and `a34` is left waiting in the input buffer for the **next** `cin`, which will promptly choke on it too.
- Typing a value with a decimal point where an `int` is expected (`3.14`) behaves the same way: `i` becomes `3`, and `.14` is left behind to spoil the following read.
- A value too large for the destination type (bigger than `INT_MAX`, say) is different: `cin` sets its **fail state** and clamps the variable to the type's extreme value (`INT_MAX` or `INT_MIN`). Once in a fail state, `cin` refuses **every further** extraction — your program will look as if it is silently skipping the rest of its input — until you call `cin.clear()`.

This final case represents one of the most common sources of "my program just stops working" bugs among beginners: always check `if (not cin)` immediately after reading any user input, before relying on the value you have just read.

■ We will explore how to test (`if`) the state or values of objects mid-program very shortly.

2·9.2 Checkpoint

1) How would the following number in scientific notation be represented in engineering notation with a power of ten scale factor? `6.31 × 1017`
2) What will the following code display?

```
int number;

number = 3.987:
cout << number << endl;
```

3) Why does engineering notation exist?
4) What is the difference between `cin >> s;` and `getline(cin, s);`?

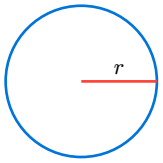
Task 1: Basics. Sequential programs: arithmetic and standard I/O

Write the following programs, naming each one as suggested. This task covers constants, arithmetic expressions, and standard input/output.

1 `01.constants.cpp` Define the following constants in C++:

1. `MAXNUMBEROFCHARS` with the value 256,
2. `PI` as `3.1416`,
3. `ENDOFLINE`, a `char` with the value `'\n'`,
4. a constant representing the distance between uppercase and lowercase characters in the ASCII table,

2 `02.circle.cpp` The following program asks the user for the radius of a circle and prints its area:



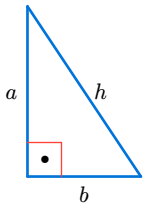

```cpp
#include <iostream>
using namespace std;

const float PI = 3.141592;

int main()
{
    float radius, area;
    cout << "Enter the circle radius: ";
    cin >> radius;
    area = PI * radius * radius;
    cout << "The area of a circle with radius " << radius
         << " is " << area << endl;
    return 0;
}
```

Modify it so that it instead asks for the radius and height of a cylinder and prints its volume ($V = S_b \times h$, where S_b is the area of the base).

3 `03.rightTriangle.cpp` Write a program that prompts the user for the two legs a and b of a right triangle and computes and prints the hypotenuse h. Include `<cmath>` for `sqrt()`. The formula is:



$$h = \sqrt{a \cdot a + b \cdot b}$$

HINT: it is generally preferable to compute `x*x` rather than call `math::pow(x, 2)` for both simplicity and efficiency.

4 `04.celsius.cpp` Write a program that asks the user for a temperature in degrees Celsius (C) and prints the equivalent in Fahrenheit (F). The conversion formula is $F = \frac{9}{5} \cdot C + 32$. › Sol

5 `05.prices.cpp` Write a program that asks the user for the price of a product and a discount percentage, and prints the final price after applying the discount. › Sol

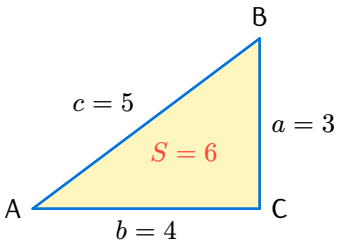
```
Price: 200
Discount (%): 15
Final price: 170
```

6 `06.seconds.cpp` Write a program that asks the user for a number of seconds elapsed since midnight and prints the equivalent time in hours, minutes and seconds. › Sol

```
Seconds: 3661   ->  1h 1m 1s                                    example
Seconds: 7322   ->  2h 2m 2s
Seconds: 60     ->  0h 1m 0s
```

7 `07.heron.cpp` Write a program that asks the user for the three sides of a triangle and prints its perimeter and area using Heron's formula:

$$S = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}$$



Include `<cmath>` for `sqrt()`. › Sol

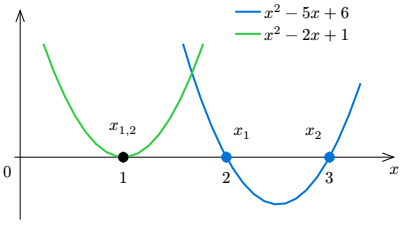
```
Side a: 3  Side b: 4  Side c: 5                                 example
Perimeter: 12  Surface: 6
```

8 `08.digits.cpp` Write a program that asks the user for a four-digit integer and prints its digits separated by spaces. Use integer division and modulo only — no strings or arrays. › Sol

```
Enter a 4-digit number: 1234  ->  1 2 3 4                       example
Enter a 4-digit number: 4001  ->  4 0 0 1
```

9 `09.quadeq.cpp` Write a program that asks the user for the coefficients a, b and c of a quadratic equation $ax^2 + bx + c = 0$ and prints its two real roots using the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



You may assume the discriminant is always non-negative. › Sol

```
a: 1  b: -5  c: 6   ->  x1: 3  x2: 2                            examples
a: 1  b: -2  c: 1   ->  x1: 1  x2: 1
```

10 `10.remainder.cpp` Write a program that asks the user for two integers and prints their quotient and remainder of the integer division, as well as the exact result as a real number. › Sol

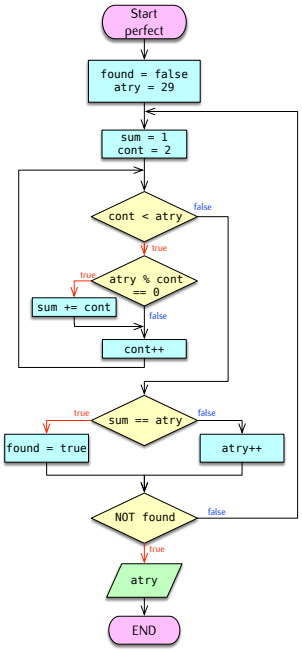
```
Dividend: 17  Divisor: 5 ->                                     example
Quotient: 3  Remainder: 2  Exact: 3.4
```

o O o

2·10 Control Flow

By default, instructions execute sequentially from top to bottom. Control flow structures—specifically selection and iteration—enable programs to follow alternative execution paths and repeat operations, which is essential for expressing algorithmic logic and creating programs that respond intelligently to different conditions.

64: Control Flow Diagram of a an algorith to find if a number is perfect



Why only these two extra ingredients—selection and iteration—on top of plain sequencing? The **Böhm–Jacopini theorem** (Böhm, Jacopini 1966) answers this precisely: any computable function can be expressed using nothing but **sequence**, **selection** and **iteration**, with no need for unrestricted jumps (`goto`). Whatever a program computes by jumping around, it can also compute using just these three constructs, at most at the cost of a few extra bookkeeping variables or some duplicated code.

This is precisely why this course forbids `goto`, `break` and `continue`: they are all controlled forms of jump, and the theorem guarantees that avoiding them costs us nothing in expressive power. Think of it as being told you can build any shape using only three types of Lego brick—individually unremarkable, but whose **composition** (one after another, nested inside one another, repeated) is enough to reach anything a chaotic pile of oddly-shaped bricks and shortcuts could.

2·11 Boolean Expressions

Every decision a program makes depends on a condition: an expression that evaluates to `true` or `false`. We examine how to construct Boolean expressions by combining relational comparisons with the logical operators `and`, `or`, and `not`. For example, `if (x > 10) {...`. The comparison `x > 10` evaluates to `true` if and only if the value stored in `x` (which must be a type compatible with relational comparisons with integers) is greater than the integer `10`. Otherwise ($x \leq 10$), the Boolean expression (`x > 10`) evaluates to `false`.

x > 5

We are accustomed to arithmetic expressions such as:

$$2 + 4 \rightarrow 6$$

When programming, we also become familiar with comparison expressions such as:

$$6 > 5 \rightarrow \texttt{true}$$

In comparisons, the result of these expressions is a Boolean value (always `true` or `false`).

In C++ we have this comparison operators:

```
a  > b
a <= b  // is a less than or equal to b
a == b  // is equal to b (be careful with double ==)
a != b  // is a different than b
// etc
```

Boolean values are not produced solely by comparisons. In general, the following all produce Boolean results (`true` or `false`):

- A Boolean constant (`true` or `false`)

- A Boolean variable
- An expression involving relational operators (`>`, `<`, etc.)
- Logical operations (`and`, `or`) between Boolean expressions. In mathematics, a **disjunction** represents the "or" relationship and is denoted by the symbol $\vee$, whilst a **conjunction** (where both sides must hold simultaneously) represents "and" and is denoted by the symbol $\wedge$.
- The negation of a Boolean expression (in mathematics, `not` is represented as $\neg$)

For example:

```
bool isEven = (a % 2 == 0);
bool aprobado = score >= 5;
bool notable = score >= 7 and score <= 8; // < == >.. always 2 operands
bool isSomething = (a == 2 or c != 10) and (not b > 5);
```

Note that chained relational operators are not permitted in this language and must be written separately:

~~`7 <= score <= 8;`~~ is `7 <= score and score <= 8;`

2·11.1 Display of Booleans

C++ can read a `bool` just as it can print one. By default, it reads and prints `0`/`1`; if we insert `boolalpha` into the stream beforehand, it reads and prints `false`/`true` Try:

```
// 01.boolReadingPrinting.cpp
// juanfc 2026-09-09
//

#include <iostream>
using namespace std;

int main()
{
  bool b = true;
  bool c = false;
  cout << b << endl;                  // prints: 1
  cout << c << endl;                  // prints: 0

  cout << boolalpha;                  // sticky: affects all later output
  cout << b << endl;                  // prints: true
  cout << c << endl;                  // prints: false

  cout << noboolalpha << b << endl;   // prints: 1 (back to the default)

  cout << "Enter true or false: ";
  cin >> boolalpha >> b;              // cin has its own flag: reads true/false
  cout << boolalpha << b << endl;     // prints: true or false

  return 0;
}
```

2·11.2 Checkpoint

Let us suppose we have:

```
int  n;
char c;
```

Build an expression that is `true` when:

1) `n` is greater than 5 (Answer: `n > 5`)

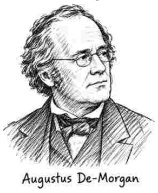
2) `n` has two digits

3) `c` is an uppercase letter
4) `c` is a letter
5) `c` is a vowel
6) `n` is odd
7) `n` ends in `0`
8) `n` ends in `00`

2·11.3 De Morgan's laws

Among the laws that enable correct manipulation of Boolean expressions, De Morgan's laws (De Morgan 1847) are particularly valuable to programmers: they state that **the negation of a conjunction is the disjunction of the negations**, and vice versa — `not (a and b)` is equivalent to `not a or not b`, and `not (a or b)` to `not a and not b`. Mastering these laws allows you to simplify conditions and avoid one of the most common sources of logical errors: incorrectly negating compound expressions. See also (Rosen 2019).

65: Augustus De-Morgan, 1806-1871



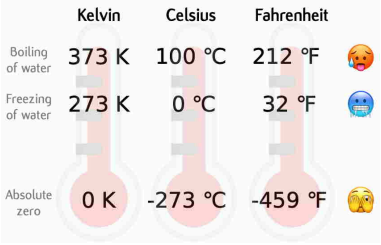
De Morgan's laws can be expressed as follows:

$$\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$$
$$\neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)$$

In C++, these laws are applied as:

```
not ( P and Q)  → not P  or not Q
not ( P  or Q)  → not P and not Q
```

66: Thermometers and negative values



For example:

```
not ( (A == B) or (A == C) ) → (A != B) and (A != C)
not ( (A == B) and (C > D) ) → (A != B) or (C <= D)
```

Logical expressions beginning with a negation are typically harder to understand. This is why De Morgan's laws are particularly useful (and will be used throughout): they allow us to rewrite the negation of a complex condition as a conjunction or disjunction of simpler, affirmative conditions. There is substantial literature on the cognitive difficulties the brain experiences when processing negative values. Some argue that Fahrenheit temperatures are less frequently negative than Celsius and are therefore easier to comprehend. This claim is highly debatable from numerous perspectives. Nevertheless, the key insight is that negations can often be handled more effectively by expressing them as positive statements of simple negations. Suppose we have the variables

```
bool rains = true;
bool carryngUmbrella = false;
```

in our program and we want to determine whether we will get wet. Typically, we get wet if it rains and we do not carry an umbrella, so the expression

```
bool illGetWet = rains and not carryngUmbrella;
```

represents this condition. The negation of this would be:

```
bool illNOTGetWet = not (rains and not carryngUmbrella);
// but better after applying De Morgan:
bool illNOTGetWet = not rains or carryngUmbrella;
```

2·11.4 Checkpoint

1) Assuming

```
x = 3;
y = 6;
z = 8;
```

indicate whether each of the following relational expressions is true or false:

2) `4 <= x + z`

3) `y == 3`

4) `x + 3 != y`

5) `z != 4`

6) `x >= 0`

7) `x <= (y * 2)`

8) Are the following relational expressions equivalent:

   8.a) `x != y` ⇔ `y >= x`

   8.b) `x >= y` ⇔ `y <= x`

   8.c) `x <= y` ⇔ `y > x`

9) Are the following expressions correct (⇒ denotes “implies”):

   9.a) `x > y and x < z` ⇒ `y < z` ?

   9.b) `x >= y and z == x` ⇒ `z == y` ?

   9.c) `x != y and x != z` ⇒ `z != y` ?

10) What will the following program segment display?

```
int a = 0, b = 2, x = 4, y = 0;
cout << boolalpha;
cout << (a == b) << " " << (a != y) << " "
     << (b <= x) << " " << (y > a) << endl;
```

Task 2: Booleans

Write the following programs, naming each one as suggested. This task focuses on Boolean values, formatted output with `cout`, and compilation basics.

1 Write a program `t0201.booleans.cpp` that prints the value of each of the following expressions, or states that the expression is invalid if it makes no sense.

```
true and true
false and true
1 == 1 and 2 == 1
1 == 1 or 2 != 1
true and 1 == 1
1 != 0 and 2 == 1
not (true and false)
not (1 == 1 and 0 != 1)
not (10 == 1 or 100 == 100)
not (1 != 10 or 3 == 4)
1 == 1 and not (1 == 1 or 1 == 0)
```

Write `cout << boolalpha;` at the top of `main()` and use the following style for each line:

```
cout << "true and true -> " << (true and true) << endl;
```

Without `boolalpha`, `cout` prints `1` or `0` instead of `true` or `false`. Use the following template: X

```cpp
// 01.booleans.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    cout << boolalpha;
    cout << "true and true -> " << (true and true) << endl;
    // ...
    cout << "nonsense      -> " << " has no sense" << endl;
    // ...
    return 0;
}
```

2 In C++, chained comparisons such as `a < b < c` do not work as in mathematics — each pair must be written separately and joined with `and`. Write a program `t0202.concatbool.cpp` that prompts the user for a number x and prints (`true`/`false`) whether $0 \leq x \leq 100$. › Sol

3 Given the declarations X

```
char c;
int x, y;
```

In C++, a Boolean expression such as `x > 5` evaluates to `true` when the value stored in `x` is greater than 5. The goal is to write simple Boolean expressions that evaluate to `true` when the condition holds and `false` otherwise.

For example: *"x is even"* → `x % 2 == 0`.

1. `x` ends in `0`,
2. `x` ends in `00`,
3. `x` has exactly 3 digits,

4. `x` belongs to $\{3, 4, 5, 6, 7\}$,
5. `x` belongs to $\{1, 2, 3, 7, 8, 9\}$,
6. `x` belongs to $\{1, 2, 3, 7, 8, 9\}$
   and simultaneously
   `y` belongs to $\{3, 6, 7, 8\}$,
7. neither `x` nor `y` is greater than `10`,
8. `c` is an uppercase letter,
9. `c` is a letter (uppercase or lowercase),
10. `c` is a lowercase vowel (`'a'`, `'e'`, `'i'`, `'o'`, `'u'`),

4 The program `t0203.calculator.cpp` below prompts the user for two `float` numbers and an operator (`+ - * /`) and prints the result: › Sol

```cpp
// 03.calculator.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    float a, b, r;
    char op;

    cout << "Enter 2 numbers: " << endl;
    cin >> a >> b;
    cout << "Enter operation (+, -, *, /): ";
    cin >> op;

    if (op == '+')
        r = a + b;
    else if (op == '-')
        r = a - b;
    else if (op == '*')
        r = a * b;
    else if (op == '/')
        r = a / b;
    else {
        cout << "Error: unknown operator " << op << endl;
        r = 0;
    }

    cout << "r: " << r << endl;
    return 0;
}
```

Extend it to support exponentiation (`^`), which C++ does not provide as an operator. Include `<cmath>` and use `pow(a, b)`. For a single value dispatched on an ordinal type, `switch` is cleaner than a chain of `if`/`else if`.

5 Write a program `t0204.demorgan1.cpp` that prompts the user for two integers `x` and `y`, then computes and prints the Boolean expression `not (x < y and y > 5)`. Also compute and print an equivalent expression with the `not` removed, applying De Morgan's laws. › Sol

6 Write a program `t0205.demorgan2.cpp` that defines `const int A = 2, B = 3, C = 2`. Starting from

```
bool r = not (found and (A==B) or (A==C));
```

rewrite the expression by removing the outermost `not` and simplifying using De Morgan's laws. Print `r` with `boolalpha`. › Sol

HINT: Assign `found = true` as a concrete value. As a mnemonic, operator precedence mirrors arithmetic: `not` (unary minus), `and` (multiplication), `or` (addition).

o O o

2·12 Selection Structures

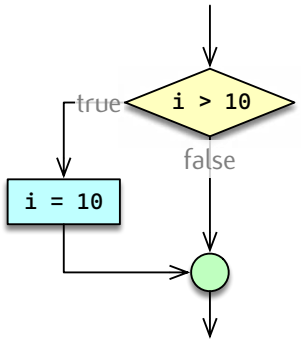
Selection structures let a program take different actions depending on a condition, just as we take an umbrella only `if` it is raining. C++ offers two of them, suited to different situations.

2·12.1 If Structure

You could think of the statements in a program as individual steps you take whilst walking along a road. To reach your destination, you must start at the beginning and take each step, one after another, until you arrive at your destination. The programs you have written so far follow a single "execution path" that the program must traverse.

The type of code shown so far is called sequence-structured because the statements are executed in sequence, one after another, from top to bottom, without branching off in another direction. However, programs often require more than one execution path (as shown in (Böhm, Jacopini 1966)) to handle situations whilst attempting to compute or accomplish something. Many algorithms require a program to execute some statements only under certain circumstances. This can be achieved with a decision structure.

67: Conditioning a part of the code



The `if` statement is the fundamental control structure for making decisions: if a condition is satisfied, execute this code. Although conceptually straightforward, nested and chained `if` statements require careful handling to maintain program readability. When the program encounters an `if` statement in its simple form, it evaluates the Boolean condition. If the condition evaluates to `true`, the *body* of the first branch (the code immediately following) is executed. If the condition evaluates to `false`, that block of code is skipped.

```C++
if (n > 10) {
    n = 10;
}
```

Example:

```
int n = 3;
if (n % 2 == 0) {
    cout << "The number " << n << " is even." << endl;
}
```

2·12.1.1 Checkpoint

1) Write an `if` statement that performs the following logic: if the value of variable `price` is greater than `100`, assign `0.2` to the variable `discountRate`.
2) What happens when the Boolean condition evaluates to `false`
3) What happens when no `{` curly braces `}` surround the next expression following the Boolean condition?

```
if (n % 2 == 0)
  cout << "The number " << n << " is even." << endl;
```

4) What happens when there is no next expression following the Boolean condition? That is, when we write

```
if (n % 2 == 0)
  ;
```

5) Write a complete program that reads an `int` age from the user and prints `"You are an adult."` only when `age` is 18 or greater. Test it with the ages 17, 18, and 40 to make sure the message appears exactly when it should.

6) Write a complete program that reads a `double` temperature and prints `"Warning: high temperature!"` only when `temperature` is above 35.0. Test it with 20.0, 35.0, and 40.0.

7) Write an `if` statement that multiplies `payRate` by `1.5` if `hours` is greater than `40`.

8) Write an `if` statement that performs the following logic: if the variable `sales` is greater than `50,000`, assign `0.25` to the `commissionRate` variable and assign `250` to the `bonus` variable.

9) What would the following code do for any value of `calls`:

```
if (calls = 20)
    rate *= 0.5;
```

10) Write the simplest form of a statement that performs the following logic: if the variable named `ticketsSold` is equal to `200`, set the Boolean flag variable `soldOut` to `true`; otherwise, set `soldOut` to `false`.

11) Write the simplest form of a statement that performs the following logic: if the variable named `ticketsSold` is equal to `200`, set the Boolean flag variable `soldOut` to `true`; otherwise, do not change `soldOut`.

12) Although the following code segments are syntactically correct, each contains a logic error. Locate the error and indicate what is wrong.

A.

```
hours = 12;
if (hours > 40);
   cout << hours << " hours qualifies for over-time." << endl;
```

B.

```
interestRate = .05;
if (interestRate = .07)
   cout << "This account is earning the maximum rate." << endl;
```

C.

```
interestRate = .05;
if (interestRate > .07)
   cout << "This account earns a 10€ bonus." << endl;
balance += 10.0;
```

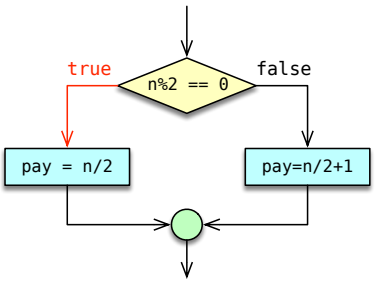
2·12.1.2 If-else Structure

When a Boolean condition evaluates to `false`, the `if` statement skips the code block and execution continues at the next statement in the program. However, this approach can be inefficient, as we know the condition is false, which often presents an opportunity to execute an alternative action. In C++, this is handled using the `else` branch of the `if` structure.

We can recognise the falsehood of the condition as an opportunity to perform a different action. Strictly speaking, this could also be achieved by evaluating a negated condition using another `if (not cond)` statement to execute the corresponding actions when `(not cond)` is `true`. The `else` branch exists precisely to take advantage of this knowledge: when we know the condition is false, we can execute alternative code without the overhead of re-evaluating the condition.

```
int n = 3; // just to see an example
if (n % 2 == 0) {
    cout << "The number " << n << " is even." << endl;
} else {
    cout << "The number " << n << " is odd." << endl;
}
```

68: If-else structure. We exploit knowledge that the condition is false to perform an alternative action




```
if (n % 2 == 0) {
    pay = n/2;
} else {
    pay = n/2 + 1;
}
```
C++

2·12.1.3 Checkpoint

1) Write an `if`/`else` statement that assigns `0.10` to `commission` unless sales is greater than or equal to `50,000.00`, in which case it assigns `0.20` to `commission`.

2) Write an `if`/`else` statement that assigns `1` to `x` if `y` is equal to `100`. Otherwise it should assign `0` to `x`.

3) Write an `if`/`else` statement that assigns `.10` to the variable `discount` if the Boolean flag variable `prepaid` is `true` and assigns `0.0` to `discount` if `prepaid` is `false`.

4) Will the `if`/`else` statement shown on the right below function exactly the same as the two separate if statements shown on the left?

```
if (x < y)
    cout << 1;
if (x > y)
    cout << 2;
```

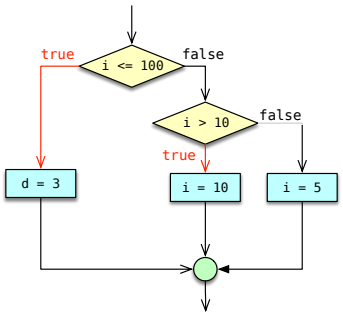
```
if (x < y)
    cout << 1;
else
    cout << 2;
```

5) Follow the instructions to complete the following program segment.

```
enum TPaint { RED, BLUE, YELLOW, GREEN, ORANGE, PURPLE };
TPaint color = GREEN;
```

Write an `if`/`else` statement that will print out "primary colour" if `color` is `RED`, `BLUE`, or `YELLOW`, and will print out `"mixed color"` otherwise. The if test should use a relational expression.

69: We can continue evaluating conditions in the else branch, knowing that the previous condition was `false`, and respond to each situation accordingly




```
if (i <= 100) {
  d = 3;
} else {
  if (i > 10) {
    i = 10;
  } else {
    i = 5;
  }
}
```
C++

2·12.2 Chaining if Structures

When a Boolean condition evaluates to `false`, you can use the `else` branch to evaluate additional conditions, knowing that the first condition was not satisfied. This allows you to chain multiple conditions together:

```
int score;
cout << "Enter your score: ";
cin >> score;
if (score >= 9) {
    cout << "Your grading is Sobresaliente" << endl;
} else if (score >= 7) {
    cout << "Your grading is Notable" << endl;
} else if (score >= 5) {
    cout << "Your grading is Aprobado" << endl;
} else {
    cout << "Your grading is Suspenso" << endl;
```

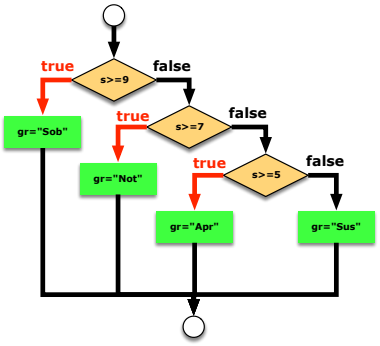
```
}
```

Note that only the final branch may omit a Boolean condition; it executes when all preceding conditions have evaluated to `false`.

The order in which you specify the conditions is critical, as each new condition is only evaluated if all preceding conditions have evaluated to `false`.

In the aforementioned example, we use if-else chaining to filter values by repeatedly testing the `score` variable. This approach is not only

70: Chaining questions



2·12.3 Nested if Statements

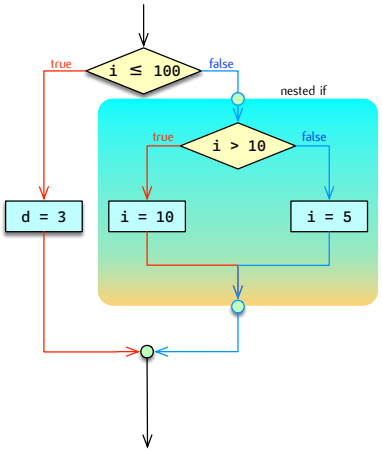
We have seen that negative answers to preceding questions can be exploited by chaining further questions. However, it is also possible to ask additional questions independently of the previous answers by nesting the question inside the corresponding branches of the if-else structure.

In Figure 71, knowing that the answer to the outer question is `false`, we can formulate an additional question to decide between two further alternative processes.

In Figure 72 we can see different logical situations in which each branch can be located.

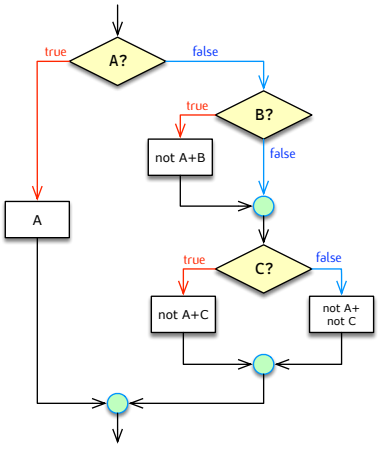
The difference between a chained-if and a nested-if is that in the latter, situations can be obtained by combining positive and negative answers from previous questions, whereas in a chained-if, each answer refines only one logical situation, creating a form of logical filtering.

71: Inside each branch, knowing the result that led to that place, we can ask further if-questions



2·12.3.1 Checkpoint

72: Situations in which each branch depends on previous answers



1) If you execute the following code, what will it display if the user enters 5? 15? 30? -1?

```
cout << "Enter a number: ";
cin >> number;
if (number > 0) {
  cout << "Zero    ";
  if (number > 10) {
    cout << "Ten    ";
    if (number > 20) {
      cout << "Twenty              ";
    }
  }
}
```

2) If you execute the following code, what will it display if the user enters 15 18? 15 10? 9 7?

```
cout << " Enter the number of team wins and number of team losses: ";
cin >> team Wins >> teamLosses;

if (teamWins > teamLosses)
{
  if(teamWins > 10)
     cout << "You are the champions. " << endl;
```

```
  else
     cout << "You have won more than 50% of your games. " << endl;
}
else
  cout << "Good luck in the rest of your games. ";
```

3) Which is the most general form of combining `if`s, chained-`if`s or nested-`if`s?

4) What will the following program segment display?

```
int funny = 1, serious;
if (funny != 1) {
  funny = serious = 1;
} else if (funny == 2) {
  funny = serious = 3;
} else {
  funny = serious = 5;
}
cout << funny << " " << serious << endl;
```

5) The following program is used in a book store to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
#include <iostream>                                                    C++
using namespace std;
int main()
{
  int numBooks, numCoupons;

  cout << "How many books are being purchased? ";
  cin >> numBooks;
  if (numBooks < 1)
     numCoupons = 0;
  else if (numBooks < 3)
     numCoupons = 1;
  else if (numBooks < 5)
     numCoupons = 2;
  else
     numCoupons = 3;

  cout << "The number of coupons to give is " << numCoupons << endl;

  return 0;
}
```

How many coupons will the customer get after buying 1, 2, 3, 4, 5, 10 books?

6) Write an `if`/`else if` statement that carries out the following logic. If the value of variable `quantityOnHand` is equal to `0`, display the message `"Out of stock"`. If the value is greater than `0`, but less than `10`, display the message `"Reorder"`. If the value is `10` or more, do not display anything.

7) Write an `if`/`else if` statement that performs the same actions as in the above question when the value of `quantityOnHand` is equal to `0` or is greater than `0`, but less than `10`. However, when the value is `10` or more, it should display the message `"Quantity OK"`.

2·12.3.2 Ternary decision

Although it will not be used extensively, it can serve as a shortcut to reduce code verbosity. As a point of interest, it is the only ternary operator in C, meaning it involves three parameters in its evaluation:

Normal decision	Ternary decision

`int max;`
`if (a > b)`
`  max = a;`
`else`
`  max = b;`
	`int max = a > b ? a : b;`

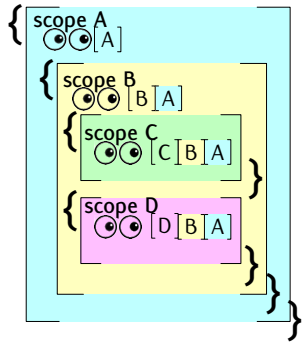
2·12.3.3 Scopes created by blocks

In C++, an opening curly brace `{` introduces a new scope, whilst `}` terminates it. This establishes a nested layer in which you can access all entities that were accessible in the enclosing scope, as well as any new variables, types, and other language elements declared within the block. When the block terminates, all entities declared inside it are destroyed and become inaccessible.

```
int a = 10;
cout << "Outside a: " << a << endl;
{
  int a = 33;
  cout << "Inside a: " << a << endl;
}
cout << "Outside, after block, a: " << a << endl;
```

What would this output?

73: Inner scopes can see enclosing ones, but not adjacent or those enclosed in them



We will later examine functions. The most important aspect of functions is that they create distinct blocks and scopes independent from the surrounding code. This isolation ensures it is safe to work with variables and definitions declared inside them, because you know they will be hidden and will not interfere with the rest of the program.

2·12.4 Switch Structure

When a decision depends on comparing a single ordinal value against several known alternatives, the `switch` structure provides greater clarity than a cascade of `if` statements. Recall that ordinal types include `bool`, `char`, `int`, and their derivatives. We also examine its distinctive fall-through behaviour.

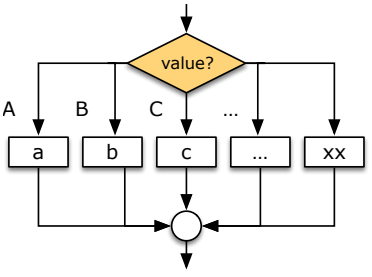
```
switch (achar) {
  case 's':
     y = sin(x);
     break;
  case 'c':
     y = cos(x);
     break;
  // ...
  default:
     cerr << "Error" << endl;
}
```

Labels can follow without a `break` statement, stacking the `case ..:`s, and this is the only way C++ provides to use multiple different keys (not many) for the same action.

```
switch (achar) {
```

```
switch (value) {
case A:
  // a
break;
case B:
  // b
break;
// ...
default:
  // xx
}
```

74: The switch control structure is straightforward in C++ and particularly convenient when a scalar value serves as the key to accessing several different actions



```
  case 's':
  case 'S':
     y = sin(x);
     break;
  case 'c':
  case 'C':
     y = cos(x);
     break;
  // ...
  default:
     cerr << "Error" << endl;
}
```

The `default` clause, if present, must be the last and is the only one that does not require a `break` statement.

A fundamental application of `switch` is the implementation of menus:

```
cout << "0: Exit" << endl;
cout << "1: Do something" << endl;
cout << "2: Do another thing" << endl;
cout << "3: etc" << endl;
int selection;
cout << "Please, user, choose an option: ";
cin >> selection;

switch (selection) {
 case 0:
   return 0; // THE END of the program
 case 1:
    // Do something
 break;
 case 2:
    // Do another thing
 break;
 case 3:
    // Do etc
 default:
    cout << "Invalid option" << endl;
 break;
}
```

If for example we are implementing a calculator:

```
float a, b, r;

cout << "Enter 2 numbers: " << endl;
cin >> a >> b;

cout << "Operation: ";
char op;
cin >> op;
switch (op) {
 case '+':
    r = a+b;
    break;
 case '-':
    r = a-b;
    break;
 case '*':
    r = a*b;
    break;
 case '/':
    r = a/b;
    break; // This one isn't necessary.
}
cout << "r: " << r << endl;
```

Typically, we will want to repeat the menu to the user:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int option;
    do {
        cout << "Enter 0 to exit" << endl;
        cout << "Enter 1 to show a message" << endl;
        cout << "Enter 2 to do something else" << endl;

        cout << "Your option?: ";
        cin >> option;
        switch (option) {
        case 1:
            cout << "The message" << endl;
        break;
        case 2:
            cout << "Another thing" << endl;
        break;
        }
    } while (option != 0);

    return 0;
}
```

Here is another example of a `switch` statement. The `toupper(grade)` function converts and returns `grade` in uppercase if it was originally lowercase▫:

▫ We will implement this as an example but it is already defined in `<cctype>` so to use it you have to remember `#include <cctype>` at the beginning of the program.

```
char grade;
cout << "Enter your control grade (A, B, C, D, E or F): ";
cin >> grade;

switch (toupper(grade)) {
case 'A':
    cout << "Excellent. "
         << "You need not take the final." << endl;
    break;

case 'B':
    cout << "Very good. ";
    grade = 'A';
    cout << "Your midterm grade is now "
         << grade << endl;
    break;

case 'C':
    cout << "Passing." << endl;
    break;

case 'D':
case 'F':
    cout << "Not good. "
         << "Go study." << endl;
    break;

default:
    cout << "That is not a possible grade." << endl;
}
```

2·12.4.1 Checkpoint

1) Explain why you cannot convert the following `if`/`else if` statement into a switch statement.

```
if (temp == 100)
   x = 0;
else if (population > 1000)
   x = 1;
```

```
else if (rate < .1)
   x = -1;
```

2) What is wrong with the following switch statement?

```
switch (temp)
{
  case temp < 0 :
    cout << "Temp is negative." << endl;
  break;
  case temp == 0:
    cout << "Temp is zero." << endl;
  break;
  case temp > 0 :
    cout << "Temp is positive." << endl;
  break;
}
```

3) What will the following program segment display?

```
int funny = 7, serious = 15;
funny = serious * 2;
switch (funny)
{ case 0 : cout << "That is funny." << endl;
           break;
  case 30: cout << "That is serious." << endl;
           break;
  case 32: cout << "That is seriously funny." << endl;
           break;
  default: cout << funny << endl;
}
```

4) Complete the following program segment by writing a `switch` statement that displays `"one"` if the user enters `1`, `"two"` if the user enters `2`, and `"three"` if the user enters `3`. If a number other than `1`, `2`, or `3` is entered, the program should display an error message.

```
cout << "Enter one of the numbers 1, 2, or 3: ";
cin >> userNum;

// Write the switch statement here.
```

5) Rewrite the following program segment using a `switch` statement instead of the `if`/`else if` statement.

```
int selection;
cout << "Which formula do you want to see?" << endl<<endl;
cout << "1. Area of a circle" << endl;
cout << "2. Area of a rectangle" << endl;
cout << "3. Area of a cylinder" << endl;
cout << "4. None of them!" << endl;
cin >> selection;
if (selection == 1)
  cout << "Pi times radius squared" << endl;
else if (selection == 2)
  cout << "Length times width" << endl;
else if (selection == 3)
  cout << "Pi times radius squared times height" << endl;
else if (selection == 4)
  cout << "Well okay then, good-bye!" << endl;
else
  cout << "Not good with numbers, eh?" << endl;
```

Task 3: Control structures. Conditionals

Solve every problem inside `main()` using conditionals. Data structures are not permitted — we will cover them later.

1 Answer these questions in a single file `t03e01.questions.cpp` labelling each one with its letter. For example (these are wrong answers!): › Sol

```
// a) P4 is executed when a == 33 and b == 0                    display
// b) a > b and b == 10
```

a) Consider the following code fragment, where P_i $(i = 1, 2, 3, 4)$ are any other code segments:

```
if      (a >  0) P1;
else if (b != 0) P2;
else if (a == b) P3;
else if (a == 0) P4;
```

For what values of `a` and `b` does P_4 run?

b) Assuming `a` and `b` are integers, write a simple Boolean expression that is `true` when neither is greater than `10`.

c) Simplify the following code fragment:

```
if (cond) {
    T;
    S;
} else {
    T;
    U;
}
```

d) Simplify the following code fragment:

```
if (cond)
    x = true;
else
    x = false;
```

e) Simplify the following code fragment using as few conditions as possible. Use the reserved words `and` `or` `not` when necessary:

```
if (age >= 65) {
    state = RETIRED;
} else {
    if (age < 18) {
        state = MINOR;
    } else {
        if ((age >= 18) and (age < 65)) {
            state = ACTIVE;
        }
    }
}
```

f) Examine the following pairs of expressions and determine whether they are equivalent. For each pair, answer with either `true` or `false`.

1: `not (a == b)` $\overset{?}{\Leftrightarrow}$ `a != b`
2: `not ((a==b) or (a==c))` $\overset{?}{\Leftrightarrow}$ `(a!=b) and (a!=c)`
3: `not ((a==b) and (c>d))` $\overset{?}{\Leftrightarrow}$ `(a!=b) or (c<=d)`
4: `not (a and b)` $\overset{?}{\Leftrightarrow}$ `a or b`
5: `a and b or c` $\overset{?}{\Leftrightarrow}$ `a and (b or c)`
6: `a and b or c` $\overset{?}{\Leftrightarrow}$ `not (a or b and c)`

2 Write a program `t03e02.prices.cpp` that calculates the price of a batch of items. The unit price decreases with quantity according to the following table (the quantity is read from the keyboard): › Sol

Quantity	Unit price
1	100
2	95
3	90
>= 4	85

The program will ask for the quantity and then print the total price. For example:

```
Quantity: 3                                                    display
Total: 270€
```

3 Build a program `t03e03.divisibleby7.cpp` that reads an integer `n` from the user and determine whether it is divisible by 7 using this algorithm:
1. double the last digit of `n` and subtract it from the remaining number;,
2. if the result is divisible by 7 (or equals 0), the original number is divisible by 7.,

For example: for `n = 203`, double the last digit ($3 \times 2 = 6$) and subtract it from the remaining part ($20 - 6 = 14$). Since 14 is divisible by 7, 203 is divisible by 7. › Sol

4 Build the program `t03e04.largest4.cpp` that asks the user for 4 integers and shows which is the largest. Do not use more types than those seen in class: simple types. › Sol

5 Write a program `t03e05.quadraticeq.cpp` to solve quadratic equations ($ax^2 + bx + c = 0$) by prompting the user for the coefficients (a, b and c) and applying the formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

taking into account: › Sol › complete

1. do not attempt to calculate the square root of a negative discriminant ($d = b^2 - 4ac$),
2. do not divide by zero (if the user enters a coefficient a equal to zero),

Inform the user when either of these two conditions occurs. Attempt to solve it (🧩) by covering all six possible situations: two distinct real roots, one repeated real root, complex conjugate roots, coefficient a equal to zero with b non-zero, all coefficients equal to zero, and only c non-zero.

6 Design a program `t03e06.timedecomposition.cpp` that asks for a number (`int`) of seconds and prints the number of days, hours, minutes and seconds that make up that amount. For example, `5000` seconds should give 1 hour, 23 minutes and 20 seconds. › Sol

7 Write a program `t03e07.dogage2.cpp` to calculate a dog's age in human-equivalent years. Ask for the dog's age in years and its weight in kg, then apply the following rule: › Sol

a) **Small** (< 10 kg): first 2 years = 12.5 human years each;, each subsequent year = 4 human years

b) **Medium** (10–25 kg): first 2 years = 10.5 human years each; each subsequent year = 4 human years

c) **Large** (> 25 kg): first 2 years = 9 human years each; each subsequent year = 5 human years.

Reject non-positive input. Print the result rounded to the nearest integer.

```
Enter dog's age in years: 15                                    display
Enter dog's weight in kg: 30
The dog's age in human years is 83
```

HINT: A more accurate (but non-linear) formula based on DNA methylation was published by Wang & Ideker, *Cell Systems*, 2020: $\text{human age} = 16\ln(\text{dog age}) + 31$. See doi:10.1016/j.cels.2020.06.006.

8 Write a program `t03e08.astro.cpp` to display the astrological sign for a given birth date. Example: › Sol

```
Enter the day of birth: 15                                      display
Enter the month of birth (e.g. March, July, etc.): May
Your Astrological sign is: Taurus
```

HINT: To look up the dates of the western zodiac signs, see here.

9 Write a program `t03e09.chinesezodiac.cpp` to display the Chinese zodiac sign for a given year of birth. Take the year–sign list from here. › Sol

```
Enter your year of birth: 2023                                  display
Your Zodiac sign: Rabbit
```

10 Write the program `t03e10.chartype.cpp` that reads a `char` from the user and prints one of the following messages: "it is a digit", "it is a lowercase vowel", "it is something else". › Sol

2·13 Iteration Structures

Whilst conditional structures enable us to build logic and flexibility into our programs, iteration structures allow us to specify actions that repeat without requiring explicit code for each occurrence.

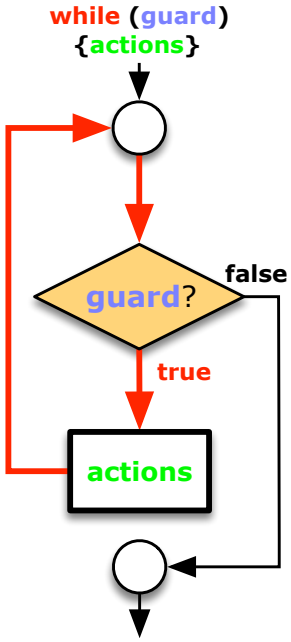
> Conditionals give our programs intelligence,
> whilst iteration gives our programs power

Computers excel at repetition: executing an operation a million times is computationally trivial. Loops are the structures that express iteration, and C++ provides two variants that differ in where and how the continuation condition is evaluated. It also provides an additional variant designed for range traversal, which is such a frequent operation in programming that it deserves its own loop construct. We are discussing `while`, `do-while`, and (range traversal) `for` loops here.

2·13.1 While Structure

The `while` loop repeats a block of code as long as a condition remains true, evaluating the condition **before** each iteration — consequently, the body may never execute. It is the most general loop construct, from which all others can be derived.

75: The while loop may never execute the action(s) in its body; the guard condition can prevent execution even once



In the following example, observe that we must obtain an initial value for `x` before entering (or deciding not to enter) the **body** of the actions the `while` loop will repeat. After each iteration, the program requests a new value of `x` from the user. The program terminates when the user provides a value of `0` or less (since it is impossible to compute $\sqrt{x}$ for non-positive values).

```C++
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float x;

    cout << "Square root of (0 or < 0 ends): ";
    cin >> x;
    while ( x > 0) {
        cout << sqrt(x) << endl;
        cin >> x;
    }

    return 0;
}
```

A practical variant of the previous program, and one that will be particularly useful given that prompting the user for values will be frequent in our exercises, is shown below:

```C++
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float x;
```

```
    cout << "Square root of (0 or < 0 ends): ";
    while (cin >> x and x > 0) {
        cout << sqrt(x) << endl;
    }

    return 0;
}
```

Note that we have omitted the separate input prompt. The input operation (`cin >>`) is performed simultaneously with the evaluation of the main loop condition (`x > 0`). This approach is possible because `cin` returns `true` when the read operation succeeds.

2·13.1.1 Checkpoint

1) What will display when the following lines of code are executed?

```
int count = 1;
while (count < 10) {
  cout << count << " ";
  count = count + 2;
}
```

2) What does the next program do:

```
#include <iostream>
using namespace std;

int main()
{
    int i = 0;
    while (i < 5) {
        cout << '*';
        ++i;
    }
    return 0;
}
```

3) Write a program that repeatedly prompts the user for real numbers until the user enters `0`. Negative numbers are permitted. Before the loop begins, declare a variable `sum` and initialise it to `0`, then add each entered value to it. When the program terminates, display the total sum of all user-entered numbers.

4) In the following program segment, which variable is the counter and which is the accumulator?

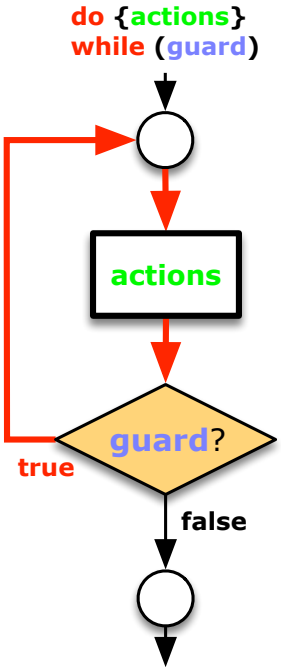
```
int number, maxNums, x = 0, t = 0;
cout << "How many numbers do you wish to enter? ";
cin >> maxNums;
while (x < maxNums) {
   cout << "Enter the next number: ";
   cin >> number;
   t += number;
   x++;
}
cout << "The sum of those numbers is " << t << endl;
```

5) Find four errors in the following code that is attempting to add up five numbers.

```
int val, count, sum;
cout << "I will add up 5 numbers. " << endl
```

```
while (count < 5) {
  cout << "Enter a number: ";
  cin >> val;
  sum = val;
}
cout << "The sum of these numbers is " << sum << endl;
```

76: The do-while loop is less frequent but remains convenient in certain cases



2·13.2 Do-While Structure

The `do-while` loop checks its condition **after** the body, guaranteeing at least one execution. It is the natural choice when something must happen before we can decide whether to repeat — for instance, validating user input.

Although `do-while` loops are less common in typical programs, they are particularly useful when an action must be performed at least once before the guard condition can be evaluated. This makes them well-suited to scenarios where the loop body needs to execute before any decision logic can be applied.

Consider a game where the user must guess a secret number known only to the program. In this scenario, prompting the user for numbers should terminate as soon as they enter the correct number. However, since the comparison between the entered number and the secret can only be performed after receiving the first input, a do-while loop is the appropriate choice:

```C++
// guessit.cpp

#include <iostream>
using namespace std;

int main() {
    int secret = 15; // not seen by the user
    int guess;
    do
        cout << "Guess our number: ";
    while (cin >> guess and guess != secret);
    cout << "You got it!" << endl;

    return 0;
}
```

Now convert it to a `while` loop program to compare which approach is simpler and more readable.

2·13.2.1 Checkpoint

1) How many times will the P actions be executed in the next iteration:

```
cnt = 0;
while (cnt < 10) {
  cnt++;
  if (cnt == 10)
    P;
}
```

2) As a function of `aValue`, what is the value of the boolean variable test after the next sentences?

```
bool test = true;
int n = aValue;
int cnt = 1;

while (cnt++ <= n)
    test = not test;
```

3) What will the following program segments display?

A.

```
int count = 3;
do {
  cout << "Hello World" << endl;
  count—;
} while (count < 1);
```

B.

```
int val = 5;
do {
    cout << val << " ";
} while (val >= 5);
```

C.

```
int count = 0, number = 0, limit = 4;
do {
    number += 2;
    count++;
} while (count < limit);
cout << number << " " << count << endl;
```

2·13.3 For-loop Structure

When the number of iterations is known in advance, the `for` loop consolidates initialisation, condition, and update into a single line. It is the idiomatic way to perform counting in C++.

A for-loop header consists of three parts: `( ① ; ② ; ③ )` followed by the body containing `actions`.

① Executed once only, before anything else occurs; always executes. The actual loop then begins

② Checked before each execution of `actions`, similar to `while` loops.

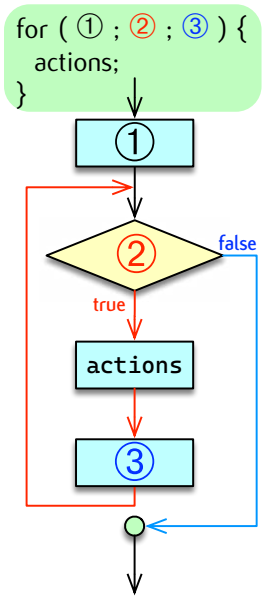
③ Executed after `actions` and before the next check of ②.

For loops are so convenient that they are frequently overused. They are used in situations where they should not be used. C++ is quite relaxed in demanding the correct use of its elements and allows you to write anything in each part, ①, ② and ③, so many programmers tend to write complex Boolean expressions in ② and miss the sense of the very word **for**: iterating along a range. The key point in the correct use of for-loops is: do I know in advance the range of the iteration, or is the number of times or the point the iteration has to cover unknown? If we do not know in advance, before the loop starts, the number of times the loop has to iterate (something that can depend on previous variables or constants), then the loop we must use is `while`.

In a typical for-loop

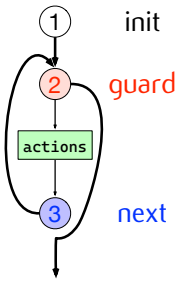
```
for (int i = 0; i < N; ++i) {
    // actions
```

77: A for-loop comprises three parts: (① ; ② ; ③) plus the body of actions




```
for ( 1; 2; 3 ) {
  actions
}
```

79: Three parts + actions



```
}
```

we know in advance the starting point and the final point and, no less important, we modify the counter **only** in the header, never in the body of actions. So, just by looking at the header of a for loop, we must be able to know the number of iterations.

Other important points with respect to this special but so important iteration structure:

semicolon Be careful that the parts (① ; ② ; ③) in the header are separated by `;`, not commas; you can use commas within any of them, as the syntax allows, for example in `for (int i = 0, j = 1; i < N and j < M; ++i, ++j)` where the `;` marks the end of a part of the header.

scope Although syntactically the header is outside the scope of the curly braces, in fact the scope of the header (① ; ② ; ③) is inside the curly braces that enclose the actions. So the declaration of `int i = 0` in the previous examples declares a variable inside the curly braces of the body of actions. This is quite useful, as you can keep external variables `i` untouched, and the actions will always refer to the "internal `i`". This makes for-loops much more independent from the surrounding context, interacting with it only through the specific actions you perform.

Remember to check that you are using this loop correctly by verifying that you can know the number of iterations from the variables and constants involved in the header.

2·13.3.1 Checkpoint

1) What is the issue with the following `for` loop:

```
for (int i = 0; i < N; ++i) {
    s += i*i;
}
cout << i << endl;
```

2) Is it guaranteed that the body of the `for` loop will execute?

3) Can we use more than one counter or index in a single `for` loop?

4) How can we perform multiple actions in the third section of a `for` loop header?

5) What does the following example do?

```
for (int i = 0; i < 80; ++i)
  cout << "*";
```

6) Compute and print the sum of all the numbers from 1 up to 1,000 using a simple for-loop.

7) A factorial is the "accumulation of products" of the natural numbers up to the number given. That is: $n! = 1 \times 2 \times 3 \times \ldots \times n$. Make a program with a for-loop to compute the factorial of 7. Be careful with the initialisation of the result: if you initialise it to 0, all the products collapse to 0. Initialise it to 1.

8) And what does the following one do?

```
int s=0;
for ( int i = 1; i <= 100; ++i ) {
  s += i*i;
}
cout << s;
```

2·13.4 Loop Design. Concept of Invariant

Writing a loop that works is easy; writing one that **provably** works is a discipline. The loop invariant — a property that is true before and after every iteration — is the key tool for designing correct loops.

We discussed loop invariants in section Section 1·4.4.1 to understand how to analyse and verify the logical correctness of code. A loop invariant is a statement about your variables that holds true immediately before the loop begins (it is **established**) and remains true at the end of every iteration (it is **maintained**). The general pattern is as follows:

```
// invariant holds here
for ( ... ; TEST CONDITION ; ... ) {
  // ...
  // invariant holds here again
}
// invariant + termination = goal
```

Two things must be checked separately: why the loop terminates, and why reaching termination, together with the invariant, gives the goal:

$$\text{invariant} + \text{termination} \Rightarrow \text{goal}$$

Example. Summing the first `N` natural numbers:

```
int sum = 0;
for (int i = 1; i <= N; ++i) {
  // invariant: sum == 1 + 2 + ... + (i - 1)
  sum += i;
  // invariant: sum == 1 + 2 + ... + i
}
// termination: i == N + 1
// invariant + termination: sum == 1 + 2 + ... + N
```

Each pass restores the invariant; once the loop terminates, invariant plus termination give exactly the goal.

2·14 Error and Exception Handling

Things go wrong: users type letters where numbers are expected, values fall outside the range a formula can accept, a division meets a zero. Robust programs anticipate these situations, and C++ provides mechanisms — from checking the state of an input stream to genuine exceptions — to detect them and react gracefully instead of crashing.

We are not building production applications here, so the failures our programs need to worry about are far more limited than in a real system. Our programs run in a sandbox: we are not reading files or fetching data from the internet yet, and we can generally expect the user to cooperate. Still, "generally" is not "always" — if you ask for a number to compute a square root, a careless or curious user may well type a negative one. When that happens and nothing in the program is watching for it, the

program does not fail gracefully: the system simply aborts it, usually with a message that means little to whoever is sitting at the keyboard.

So in situations we can foresee, it is worth protecting the program and keeping it responsive rather than letting it crash. The simplest approach is to think ahead about what can go wrong and, when it does, tell the user and perhaps let them try again. Consider:

```
try {
  💥
}
catch (..) {
  ..
}
```

```
#include <iostream>                                                    C++
using namespace std;

int main()
{
    int n, d;
    cout << "Enter numerator: " << endl;
    cin >> n;
    cout << "Enter denominator: " << endl;
    cin >> d;

    cout << n << "/" << d << " = " << n/d << endl;
    return 0;
}
```

This crashes if the user enters `0` as the denominator. The fix is a single guard:

```
#include <iostream>                                                    C++
using namespace std;

int main()
{
    int n, d;
    cout << "Enter numerator: " << endl;
    cin >> n;
    cout << "Enter denominator: " << endl;
    cin >> d;

    if (d == 0) {
        cout << "ERROR: Impossible to divide by 0" << endl;
    } else {
        cout << n << "/" << d << " = " << n/d << endl;
    }
    return 0;
}
```

We can go further and, rather than just refusing to divide, give the user another chance. This is exactly the case foreshadowed earlier in this chapter▲: reading input is itself a source of errors, since a user can type a letter where a number is expected. A `do-while` loop — checking the input **after** trying to read it — combined with `cin.clear()` and `cin.ignore()` to recover from a failed read, lets us keep asking until we get something usable:

▲See the discussion of `cin`'s fail state under Section 2·9, where we first met `if (not cin)`.

```
// readint.cpp                                                         C++

#include <iostream>
using namespace std;

int main()
{
    int n;
    bool ok;
    do {
        cout << "Enter an integer: ";
        cin >> n;
        ok = bool(cin);
```

```
        if (not ok) {
            cin.clear();              // step out of the fail state
            cin.ignore(1000, '\n');  // discard the bad input still
            waiting
            cout << "That was not a number, try again." << endl;
        }
    } while (not ok);

    cout << "Thanks, you entered: " << n << endl;
    return 0;
}
```

Notice that nothing here is new: `if`, `do-while`, and `cin` are all constructs you already know. Handling errors well is less about special syntax and more about the habit of asking, at every point data enters your program, "what if this is not what I expect?"

For genuinely unpredictable situations — where we cannot simply guard a single value with an `if`, because the failure might occur several calls deep inside code we did not write ourselves — C++ and most other languages offer a dedicated mechanism called **exceptions**: a `try` block marks a region of code as sensitive, and a separate `catch` block says what to do if something inside it fails, so that control jumps straight from the point of failure to the matching handler, skipping everything in between. That mechanism only earns its keep once our programs are broken into functions calling other functions, which is precisely where the next chapter is heading — so we leave `try`/`catch` for later and, for now, rely on the `if` and `do-while` checks above.

2·14.1 Checkpoint

1) When is it appropriate to use a `try`/`catch`/`throw` mechanism, and when would a simple check with an error message be sufficient?
2) Are all errors that the system generates visible and controllable by our programs?

2·15 Recommendations and Common Mistakes

Experience shows that beginners commonly encounter the same pitfalls: `=` instead of `==`, forgotten semicolons, and off-by-one errors in loops. This chapter concludes with a collection of best practices and frequent mistakes, so that you can learn from others' experiences rather than only from your own.

Throughout this part, we have covered the steps for solving problems, diagramming, verifying loops through their invariants, and the correct use of types and how sensitive type mixing can be. We have discussed the particularities of unsigned integers and the complexities of floating-point numbers, as well as the fundamentals of developing logic through conditionals and loops. For conditionals, the order in which conditions are evaluated is critical; for loops, the specialisation of `for` loops, which are often misused (perhaps due to their ease of writing and the quick templates that editors readily provide), is equally important.

The correct use of the appropriate loop — especially the idea of not knowing *a priori* when the iterations will finish (`while`), as opposed to

knowing the number of iterations from the information we have before entering the loop (`for`) — is crucial in this course. Bad habits arise from ignorance of these precise facts. The correct ordering of the conditions in chained or nested `if`s, or the suitability of a `switch` structure, can only be mastered through exercises. Practice is key to learning how to program; there is no substitute.

What follows in this course depends entirely on mastering conditionals and loops. What comes next has more to do with organising the code and, finally, with powerful data structures for modelling problems; but what programming really is — reasoning, being able to get from the problem to the solution, the algorithm — has already been covered.

2·15.1 Checkpoint

1) Each of the following fragments compiles without error, yet each hides one of the classic mistakes mentioned above. Find it and describe what the code **really** does.

A.

```
int n;
cin >> n;
if (n = 0) {
    cout << "n is zero" << endl;
}
```

B.

```
int i = 1;
while (i <= 10);
{
    cout << i << endl;
    ++i;
}
```

C.

```
// intended: print 1, 2, ..., 10
for (int i = 0; i <= 10; ++i) {
    cout << i << " ";
}
```

2) For each task, decide whether a `for` or a `while` loop fits better, and justify it in one line:

   2.a) print the multiplication table of a number `n`, from `n·1` to `n·10`;

   2.b) keep asking for a password until the user types the correct one;

   2.c) add up the first `100` natural numbers;

   2.d) read numbers until the user enters a negative one, then report how many were positive.

3) Given

```
unsigned int a = 3, b = 5;
cout << a - b << endl;
```

what does this print, and why? Which particularity of unsigned integers does it illustrate?

4) The following chain is meant to grade a `score` from `0` to `10`, but a `10` is reported as `"Aprobado"`. Explain why, and reorder the branches to fix it.

```
if (score >= 5) {
    cout << "Aprobado" << endl;
} else if (score >= 7) {
    cout << "Notable" << endl;
} else if (score >= 9) {
    cout << "Sobresaliente" << endl;
}
```

5) Explain why comparing two `double` values that come from different calculations with `==` is risky, even when, on paper, the two results should be identical.

We now move to two significant topics: first, organising code through subprograms; second, building genuine data structures that will provide your programs with the capability they require.

References

BÖHM, Corrado and JACOPINI, Giuseppe, 1966. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*. 1966. Vol. 9, no. 5, p. 366–371. DOI **10.1145/355592.365646**.

The theorem that gave structured programming its formal footing: any computable function can be expressed using only sequence, selection and iteration, with no need for unrestricted `goto` jumps. Dense and notation-heavy in the original, but the result itself is simple enough to state in one sentence and is the theoretical justification for this course's ban on `goto`, `break` and `continue`.

DE MORGAN, Augustus, 1847. *Formal Logic: or, The Calculus of Inference, Necessary and Probable*. London: Taylor, Walton.

The founding text of symbolic logic as a calculus, introducing what would later be formalised as De Morgan's laws — the duality rules for negation over conjunction and disjunction that every programmer encounters in boolean simplification. Of primarily historical interest, but worth citing to root the course in the intellectual tradition that connects mathematics, logic, and computation.

ROSEN, Kenneth H., 2019. *Discrete Mathematics and Its Applications*. 8. New York: McGraw-Hill Education. ISBN 978-1-259-67651-2.

The standard reference for discrete mathematics in computer science curricula. Covers logic, proof techniques, sets, relations, graphs, trees, and combinatorics with a consistently applied CS perspective. The 8th edition is well-structured for self-study and lecture support alike. Recommended background reading for students seeking to understand the mathematical foundations underlying algorithm design and program correctness.

Task 4: Control structures. Single loops

Solve every problem inside `main()` using loops and conditionals. Data structures are not permitted — we will cover them later. Problems marked 🧊 are challenging; do not hesitate to ask for help. Problems marked 🧊🧊 are for the most determined students only.

1 Write a program that finds the largest value, the smallest value, and the arithmetic mean of a series of N integers entered from the keyboard. Ask for N first, then read N values in a `for` loop. Repeated values are allowed. Round the mean to the nearest integer with: › Sol

```
int av = float(sum) / N + 0.5;
```

For example: `3 3 3 3` ⟶ largest: 3, smallest: 3, mean: 3. And: `-3 4 5 5 4 5 5 5` ⟶ smallest: −3, largest: 5, mean: 3.

2 Write a program to compute the factorial $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$ of an integer `n`. Use a `while` loop. Try `n = 6` (720) and `n = 12` (479001600). What happens for `n > 10` with `int`? › while › for

HINT: $n!$ counts the number of ways to arrange n distinct objects in a row. It is the product of all integers from 2 up to and including n.

3 Write a program that finds all multiples of 35 in the range $[1500, 2700]$ and prints them separated by commas. › while › for › steps

HINT: Iterate from `1500` to `2700` and print each number divisible by 35. Alternatively, step through the loop in increments of 35.

```
1505, 1540, 1575, 1610, ..., 2660, 2695                    display
```

4 Write a program that reads characters until a period `'.'` is entered, then prints the count of vowels, uppercase letters, and non-letter characters. Use `cin.get()` to read one character at a time — unlike `cin >>`, it does not skip spaces, tabs, or newlines: › Sol

```
char c;
c = cin.get();
while (c != '.') {
    // your code here
    c = cin.get();
}
```

An equivalent compact form uses the assignment as the loop condition:

```
char c;
while (c = cin.get() and c != '.')
    // your code here
```

Here the assignment `c = cin.get()` acts as both a statement and an expression — a common C/C++ idiom. When you want to skip whitespace, you can use `cin >> c` instead.

5 Read a non-zero integer `n` from the keyboard (validate that it is not `0`). Then read a sequence of integers terminated by `0` and determine whether `n` appears in the sequence. Stop reading as soon as `n` is found or `0` is entered, and print whether `n` was entered or not. › Sol

6 Read a sequence of characters terminated by a period `'.'` and determine whether the string `abc` appears in it. Stop reading as soon as `abc` is found or the period is reached, and print whether `abc` was entered or not. › Sol

7 Read a sequence of integers terminated by `0` and find the positions of the first and last occurrence of `12`. If `12` does not appear, print `0` for both. For example: › Sol

$$8 \quad 6 \quad 1 \quad 12 \quad -9 \quad 12 \quad 12 \quad 33 \quad 0$$

should output `4` and `7`.

HINT: Use variables `int first, last, pos, num` to track the first occurrence, the last occurrence, the current position, and the current number.

8 Read a sequence of natural numbers terminated by `0`, or stop early if the running sum of previous elements equals the current number. For example, `1 1 2 5 0`: the sum of `1+1` equals the next number `2`, so stop when `2` is read. But `1 3 1 2 0` has no such match and reads until `0`. › Sol

HINT: Use `int num, sum`. Read a number first, then loop while `num != 0` and `sum != num`; inside the loop, add `num` to `sum` and read the next number.

9 The last digit (base 10) of an integer is `n % 10`; removing it means dividing by 10. Build a program that prints the digits of an integer in reverse order. For example, `2021` prints `1202`. › Sol

10 Given a natural number n, determine whether it is a perfect square, i.e. whether there exists a natural number m such that $m^2 = n$.

HINT: Test each integer `i` starting from 1 with a `while` loop. Stop as soon as `i*i >= n` and check whether `i*i == n`. › Sol

11 Build a program that prints all divisors of a given number. For example:

```
12  ->  1 2 3 4 6 12                                    example
7   ->  1 7
```

› Sol

12 Given n, print all numbers in $[1, n]$ that are divisible by 3 or by 5 (or both). For example, for $n = 20$:

```
3 5 6 9 10 12 15 18 20                                  example
```

› Sol

13 Given a number n, determine whether it is prime. › Sol

HINT: Test each integer `i` from 2 while `i < n`. If any `i` divides `n`, stop; then check whether the loop ended because a divisor was found or because no divisor exists.

14 Given n (for example, $n = 10000$), compute the sum $1 + 2 + 3 + \cdots + n$ using a loop, then verify the result matches the closed formula $\frac{n(n+1)}{2}$. Print both values. › Sol

15 Evaluate the following series for a real number x:

$$s = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Add only the first terms up to `const int MAXPOW = 10`.

HINT: Compute each power from the previous one (multiply by x) and each factorial from the previous one (multiply by `i`), so the loop body needs neither `pow()` nor a nested factorial loop. › Sol

16 Evaluate the same series as exercise 15, but stop when the term being added falls below `const float EPSILON = 0.000001`. › Sol

17 Using the factorial program from exercise 2, compute binomial coefficients:

$$\binom{x}{y} = \frac{x!}{y!(x-y)!}$$

Ask the user for `int x, y` and minimise the number of multiplications performed. › Sol

18 Read a natural number n and compute the n-th term of the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...), defined by $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$. The sequence grows exponentially — be careful with large n. ($f_{92} = 7540113804746346429$) › Sol

19 `19.countdigits.cpp` Read an integer and print how many times each digit (0–9) appears. Ignore the sign if negative. › Sol

```
1234321  ->  1:1  2:2  3:2  4:1                    example
100      ->  0:2  1:1
-2023    ->  0:1  2:2  3:1
```

20 `20.collatz.cpp` Read a positive integer and print its Collatz sequence: if even, divide by 2; if odd, multiply by 3 and add 1. Stop at 1. Also print the number of steps. › Sol

```
6   ->  6 3 10 5 16 8 4 2 1  (8 steps)             example
27  ->  27 82 41 ...  1  (111 steps)
```

21 `21.numwords.cpp` Read characters until a full stop `'.'` and count the number of words. A word is defined as a maximal sequence of letters `a`–`z`. Use `cin.get()` — `cin >>` cannot read whitespace. › Sol

```
hello world.                  ->  2 words           example
 . one two   three   four.    ->  4 words
nospaceshere.                 ->  1 word
```

Consider tracking the state before and after each character is read. Begin in the state `inWord = false`. When you read a letter (`a–z`) **and** you were not previously in a word, increment the word count and transition to the state `inWord = true`. If the character read is not a letter, return to the state `inWord = false`. (Finite State Machines)

22 The following game is played with n objects. The machine decides who goes first; then the user and machine alternate turns, each removing 1, 2, or 3 objects. The player who takes the last object **loses**. Build a program in which the machine always wins. › alice-bob › machine

HINT: Work out which positions are **losing** for the player to move (every available move leads to a winning position for the opponent):

- $n = 1$: must take 1 $\longrightarrow$ lose.
- $n = 2, 3, 4$: can reach $n = 1$ $\longrightarrow$ win.
- $n = 5$: any move leaves a winning position for the opponent $\longrightarrow$ lose.

The pattern repeats every 4 steps: positions where `n % 4 == 1` are losing positions. The machine's strategy:

1. If `n % 4 == 1`, let the user go first.,
2. Otherwise, go first and take `(n-1) % 4` objects, leaving a losing position for the user.,
3. On each subsequent turn, if the user takes k objects, take $4 - k$, always restoring a `n % 4 == 1` position for the user.,

o O o

Task 5: Control structures. Double loops

In this task we practise nested loops (one loop, `for` or `while`, inside other). These can be challenging at first because the inner loop's bounds typically depend on the outer loop variable. For figure exercises we always use `for`-`for` loops. Do not use C++ string idioms such as multiplying a string by a number.

Figure sizes are parametrised so their size depends on a variable (`size` or whatever); give it a sample value and run the program, or prompt the user for that value if you prefer.

Figures can be solved with or without `if`-`else`; the preferred approach is to avoid complex conditionals and instead repeat simple code blocks, each one responsible for a distinct part of the figure. Expect some trial and error until the figure looks right.

Problems marked 🧊 are challenging; do not hesitate to ask for help. Problems marked 🧊🧊 are designed exclusively for the most determined students.

1 Print a filled square. Example with `side = 4`: › Sol

```
****                                                    example
****
****
****
```

Declare `int side = 4` and start with a single row of stars:

```
for (int i = 0; i < side; ++i)
    cout << '*';
cout << endl;
```

Note that `endl` comes **after** the inner loop — printing it inside would break each row into separate lines. Now wrap that block in an outer loop to repeat it `side` times:

```
for (int i = 0; i < side; ++i) {
    for (int j = 0; j < side; ++j)
        cout << '*';
    cout << endl;
}
```

Both loop bounds are the same because the figure is a square. Could you use the same index variable `i` for both loops?

```
for (int i = 0; i < side; ++i) {
    for (int i = 0; i < side; ++i)  // ← same name i?
```

Complete the program and find out.

2 Print an empty square with `side = 5`: › Sol

```
*****                                                   example
*   *
*   *
*   *
*****
```

3 Print an empty square with a double-line border, with `side = 6`: › Sol

```
******
******
**  **
**  **
******
******
```
example

4 Print a filled rhombus with `sideHalf = 4`: › Sol

```
   *
  ***
 *****
*******
 *****
  ***
   *
```
example

5 Print an empty rhombus with `sideHalf = 3`: › Sol

```
  *
 * *
*   *
 * *
  *
```
example

6 Print the following spaced triangle with `side = 6`. Dots shown below are **not** printed — they mark spaces to help you count: › Sol

```
.....*
....*.*
...*.*.*
..*.*.*.*
.*.*.*.*.*
*.*.*.*.*.*
```
example

7 Print a wine glass with half-width `widthHalf = 5`: › Sol

```
 **********
  *********
   *******
    *****
     ***
      *
      *
      *
      *
      *
   *******
*************
```
example

8 Print the following triangle with `height = 3`: › Sol

```
1
12
123
1234
```
example

9 🧊 Print the following triangle for any given height (example with `height = 6`): › Sol

```
1                                                    example
23
456
7890
12345
678901
```

10 Build a program that, given a number, computes the sum of all its proper divisors (including 1, excluding the number itself), then checks whether the number equals that sum. For example: › Sol

```
12  ->  1+2+3+4+6 = 16  ->  12 is not perfect        example
28  ->  1+2+4+7+14 = 28  ->  28 is perfect
```

HINT: A number with this property is called a perfect number.

11 🧊 Print the first N prime numbers. Ask the user for N or use $N = 100$ › Sol

12 Build a multiplication table using loops. Use `<iomanip>` and `setw(4)` to align columns. Example output (dots represent spaces — print actual spaces in your solution): › Sol

```
...1...2...3...4...5...6...7...8...9..10             example
...2...4...6...8..10..12..14..16..18..20
...3...6...9..12..15..18..21..24..27..30
...4...8..12..16..20..24..28..32..36..40
...5..10..15..20..25..30..35..40..45..50
...6..12..18..24..30..36..42..48..54..60
...7..14..21..28..35..42..49..56..63..70
...8..16..24..32..40..48..56..64..72..80
...9..18..27..36..45..54..63..72..81..90
..10..20..30..40..50..60..70..80..90.100
```

Use `const int FROM = 1` and `const int TO = 10`.

13 Build a program that prints the number of decimal digits in a given integer. For example, `1492` has `4` digits. The number `0` should print `1` — handle this edge case carefully. › Sol

14 Build a program that takes a number and a position, and prints the digit at that position (counting from the right, starting at 1). For `1492` and position `2`, print `9`; for position `0` print nothing; for position `5` print `0`. → Sol

15 🧊🧊🧊 Print a sawtooth wave with `height = 5` and `nTeeth = 3`. › Sol

```
    *       *       *                                example
   * *     * *     * *
  *   *   *   *   *   *
 *     * *     * *     *
*       *       *
```

o O o

Part 3 Procedural Abstraction

Contents of this part Book contents

3·1 Top-Down Design 96
3·2 Procedures and Functions. Parameters 97
3·3 Recursion 112
▸ Task 6: Subprograms 120
▸ Task 7: Subprograms++ 124
▸ Task 8: Pre-structures 128

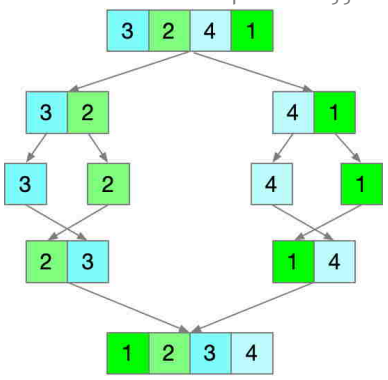
Data abstraction refers to modelling the essential information contained within data.
Procedural abstraction concerns modelling groups of actions by organising them into subprograms

One of the most influential works by Niklaus Wirth, the author of Pascal and Modula-2, is titled "Algorithms + Data Structures = Programs", published in 1976. Fundamentally, this book reflects the importance of organising not only data structures but also actions into subprograms.

3·1 Top-Down Design

When we consider a solution to a problem, we initially imagine the problem as a whole. Then, as we saw with Pólya (Section 1·4), we divide it into smaller, and typically easier to solve, parts. This technique is called divide-and-conquer. In reality, this term dates back to the Roman Empire, where it referred to the strategic advantage of keeping an adversary divided to gain power over them. However, it remains valid in other domains. When a problem exceeds our capacity to solve it in its entirety, it is better to divide it into smaller parts.

80: The divide-and-conquer strategy

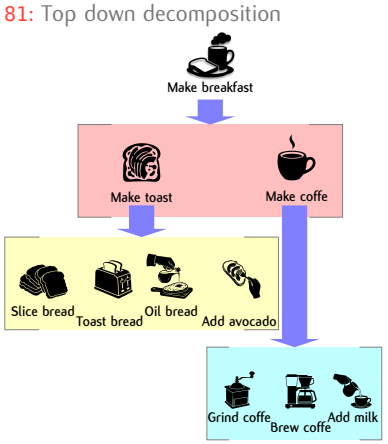


The divide-and-conquer mechanism can be understood in two ways:

top-down you start by understanding the whole and then progressively work into the details

bottom-up you do not know exactly what the complete solution will be or what it will apply to, but you begin by combining small, already solved pieces and grouping them into a solution.

81: Top down decomposition



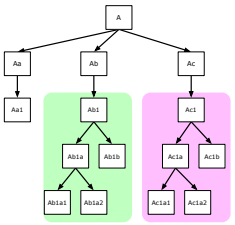
The first approach is most important for us, as engineers regularly encounter large-scale problems and must draw upon established solutions.

3·1.1 Hierarchy and coupling

When building the pieces of a top-down decomposition, we must be careful to keep the hierarchy **decoupled**. That is, the submodules — the parts we build — should be independent from one another. This is easy to check by following the calls each module makes: if, to solve its part of the problem, a module only ever needs to call its own submodules, and each of those in turn only calls submodules belonging to its own branch, we

have a clean hierarchical decomposition. If, instead, a submodule needs to call others outside its own branch to do its job, the decomposition is **coupled**.

82: Example of a clean hierarchical decomposition. Arrows represent the call dependencies that each sub-module has



Examine Figure 81 again: each small task is self-contained—it does not need to call any of its siblings to accomplish its function. This independence is precisely what maintains the hierarchy's decoupling. We will revisit this concept in more rigorous form once we begin writing our own subprograms: **cohesion** and **coupling** are two of the principal criteria we will use to evaluate whether a decomposition is sound.

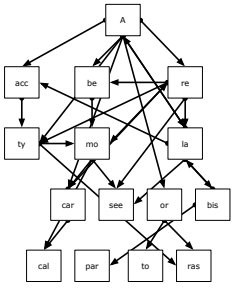
In summary, hierarchical decomposition offers three key benefits:

easier to read each component addresses a single, well-defined problem, allowing you to focus on one concern at a time. This isolation makes the code straightforward to follow.

easier to modify changes made to one component do not propagate to others. You can enhance or refine the implementation of a particular module without worrying about unintended side effects elsewhere.

easier to reuse because solutions are independent of their surrounding context and environment, you can transfer them to other problem domains with minimal or no adaptation required.

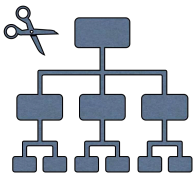
83: In a coupled decomposition, individual submodules are obscured by a mesh of interdependent calls



Accordingly, as a fundamental principle:

> Each subprogram should do one thing

84: Hierarchical decomposition makes programs easier to read, modify, and reuse



Let us now explore what C++ offers us to divide and conquer our programs.

85: Each subprogram ↔ one objective



3·1.2 Checkpoint

1) What is the principal advantage of top-down design?
2) Is structured and ordered program writing a requirement of the programming language, or is it the responsibility of the programmer?
3) Is it always necessary to write subprograms?

3·2 Procedures and Functions. Parameters

A subprogram is a collection of statements designed to perform one particular task.

Take computing a square root: the only thing the computation needs is the number to work on. So we build a subprogram to solve exactly that problem — we give it the number we want the square root of, and we expect back another number. Nothing more, nothing less.

When students begin to program, they are used to reading and printing values directly, so it is very common to see a first subprogram asking the

user for a number and printing the answer itself — for instance, printing `"The square root of x is 2.0"` from inside the subprogram. This is the exact opposite of why subprograms exist: a subprogram that only knows how to talk to the keyboard and the screen cannot be reused anywhere else. You will find plenty of solutions online (and LLM-generated code readily copies the habit) that read the input and print the result in the very body of the subprogram — resist that temptation.

If a subprogram could not receive information to work on, it would be close to useless. We give it that information through **parameters**. The idea is the same in every programming language: when you write `sqrt(x)`, `sqrt` alone names the computation, while `x` — accompanying the call in parentheses — is the parameter, the value you want that computation to work on.

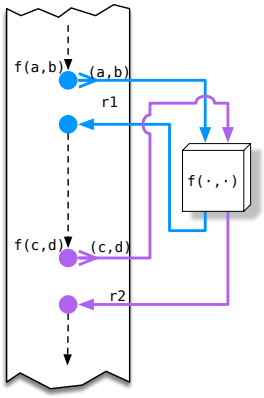
3·2.1 Example

Building a function in C++ follows this syntax:

```C++
int fact(int n)
{
    int r = 1;
    for (int i = 2; i <= n; ++i) {
        r *= i;
    }
    return r;
}
```

Four ingredients make up this definition, and you can point to each one directly in the code above: a **name** (`fact`); a **parameter list** in parentheses — here, a single integer that, inside the function, is called `n`; a **body** — the scope delimited by curly braces, where the actual work happens; and a **return type** (`int`, written before the name), paired with the `return` statement that hands the computed value, `r`, back to whoever called the function.

86: Calls to a subprogram from different points within different other subprograms



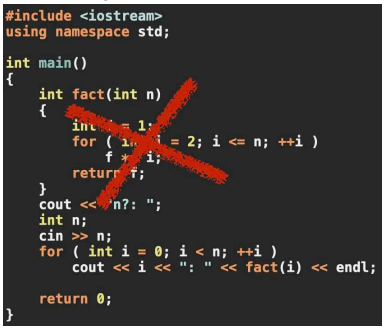
3·2.2 Definition and Declaration of Procedures and Functions. Formal Parameters

To build a subprogram we need the same four ingredients as before: a **name**, a list of **parameters**, a **body**, and a **type**. C++, however, has its own rules for where subprograms may be placed: they must all sit at the same level — nesting is not allowed, so you cannot declare a subprogram inside another one. Bear in mind that, in fact,

```
int main()
{
  // ..
}
```

is itself a kind of subprogram: it is what the operating system calls to run our program's set of actions. So, from the operating system's point of view, `main()` is a subprogram too — which is exactly why we cannot write subprograms inside `main()`; they must be written before or after it.

87: Nesting is not allowed in C++



On the other hand, in C++ anything we use — a variable, a type, a subprogram — must already have been declared before we use (or call) it. This is why, up to now, our subprograms have had to be written before `main()`:

```cpp
#include <iostream>
using namespace std;

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i ) {
        f *= i;
    }
    return f;
}

int main()
{
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i ) {
        cout << i << ": " << fact(i) << endl;
    }
    return 0;
}
```

But this technicality forces an awkward presentation of the code — the very opposite of what we have been advocating so far: top-down decomposition.

It forces us to write the details before the main idea. It is as if a book began with its appendices: to find out what the book is actually about, you would have to start reading on page 551, wading through details that mean nothing to you yet, because nobody has told you the big picture first.

3·2.2.1 Procedures vs Functions

When you call a subprogram, you typically expect to receive some information from the operation you invoked. Subprograms that return information are called **functions**, whilst subprograms that do not return anything are called **procedures**. Functions are the most common type, and sometimes we refer to all of them simply as functions; however, the syntax and usage differ between them.

88: Functions return a single value of the type they are declared to return. Procedures act on external entities such as the screen, sounds, or the internet, but do not return anything. They may, however, modify their parameters, as we shall see

```
int f()
{
    actions;
    return val;
}

void g()
{
    actions;
}
```

```
void nameOfProcedure(param, param,...) {
  // code
}
```

```
type nameOfFunction(param, param,...) {
  // code
  return something_type
}
```

Because procedures do not return anything, when called, there is nothing to assign to a variable. Calls to procedures always appear at the beginning of a line, with no `=` sign before them:

```
{
  ...
  printLines(3);
  ...
}
```

Functions, by contrast, always return a value, so in roughly 99% of cases they will either have an `=` sign or be part of an expression that uses the returned value, or perhaps serve as a parameter to another subprogram call.

```
{
  ...
  int r = fact(n);
  float term = x / fact(++n);
  printLines(fact(2));
  ...
}
```

Can you imagine any use of a procedure without parameters? They are not forbidden at all, and the only practical purpose they could serve is to modify something outside our program. For example, clearing the screen:

```
  ...
  clrscr();
  ...
```

3·2.2.2 Checkpoint

1) Is it syntactically permissible to write:

```
int x = f(g());
```

2) If `p()` is a procedure and `f()` is a function, is the following syntax permissible?

   2.a) `int x = f()`

   2.b) `p(f());`

   2.c) `f(p());`

3) When would a parameterless procedure be of practical interest?

3·2.2.3 Interfaces. Prototypes

In some languages, despite offering numerous mechanisms for organising algorithms, subprograms, methods and so forth, they do not provide straightforward ways to write subprograms in the desired order. The order in which code must be written is determined by the compiler's internal workings.

89: A prototype informs the compiler that a subprogram with that interface exists and permits subsequent source lines to use it

```
int f(int n);
```

C++ has provided a solution to this problem from the outset. Function-argument checking — what we now call a **prototype** — was one of the first features Stroustrup added to "C with Classes", and he made it compulsory as early as 1982, years before the language was even renamed C++ (1983) or reached its first standard, C++98△. A prototype allows us to order our subprograms as we wish: the only requirement is to declare in advance that a subprogram with a given name and parameters will exist — its actual definition, the body, can be deferred until later:

90: Prototypes do not need to show parameter names, except where a name aids readability, e.g. for defaulted parameters

```
int f(int);
```

△ B. Stroustrup, *A History of C++: 1979–1991*.

```
#include <iostream>                                          C++
using namespace std;

int main()
{
    int fact(int n);   // prototype: fact() exists,
                       // its definition comes later

    cout << "n?: ";
    int n;
    cin >> n;
```

```
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return 0;
}

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}
```

This way, we start reading the program the natural way: we meet `main()` — the outline of what the program does — right away, and only go looking for the details of `fact` afterwards, if we care to.

> **Prototypes** or **interfaces**
> relax the order of placement of subprograms by making them easier to read naturally from the abstract idea we want to solve down to the implementation details,
> from top to bottom.

3·2.2.4 Top-Down Program Organisation

A C++ source file should follow this same top-down order throughout, progressing from the most general elements to the most specific:

1. Includes, ordered from general to specific headers.
2. Constants (`const`).
3. Type definitions (`typedef`).
4. `int main()`, accompanied by the prototypes of any subprograms it requires.
5. Subprogram bodies, ordered from the most abstract to the most detailed.

We have seen the general practice of including headers at the top of the `.cpp`, which makes sense, as it documents the program's external dependencies. It is worth noting here that the inclusion of headers implies definitions, and when we include many headers, some definitions may inadvertently clash with others from other headers, so a healthy rule of inclusion, in general, is to include the most abstract headers first, and our own program's headers last.

When we write `#include <cmath>` we are asking the preprocessor to look for the file `cmath` in the standard directories our C++ settings have in its configuration. In a complex program made of more than one `.cpp` file, it is necessary to have common definitions for all the files, and those definitions (constants, types, prototypes) are held in headers `.h` like `geometry.h` that our files `.cpp` will include: `#include "geometry.h"` without any `<>`, directly telling the preprocessor to locate the file at the path we write, relative to our own program.

So for the inclusion of headers a general to particular order makes sense.

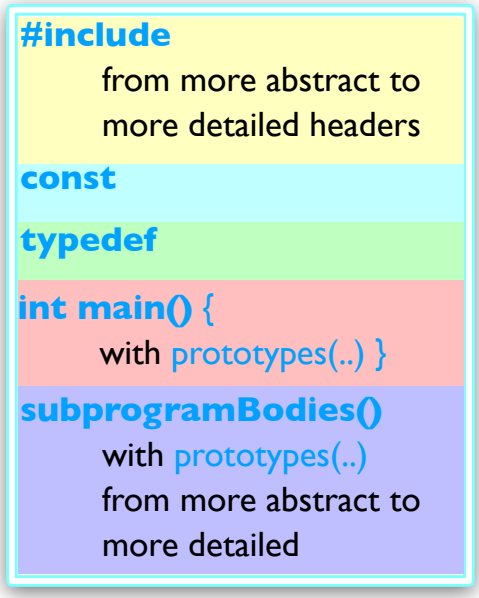
Constants, on the other hand, are the safest elements of our language, but many others can depend on them, so the location of constants should be

just after the inclusion of headers, again ordered from general to specific (the constant π, for instance, would hardly depend on any other).

Types, on the other hand, are typically dependent on constants (arrays especially) and should be right after constants, as they define the general organisation of our data, are not harmful, demand neither memory nor time from our final program, and are necessary for the subprograms that follow.

The `main()` is the index, the guide to what our program is meant to do, so it must be the first "function" declared, and at the top of it, before any variable declarations, come the prototypes of the subprograms it will need to do its job — as a declaration of dependency, no more than really necessary.

Then, finally, after `main()` come the bodies of the subprograms, from the most abstract to the most detailed.



3·2.2.5 Checkpoint

Design the interfaces for the following subprograms to compute:

1) the area of a circle
2) the maximum of three numbers
3) the ASCII code of a letter
4) the letter at a given position in the alphabet
5) the number of days between two given dates (interface only)
6) whether a number is even or odd
7) whether a given date falls in a leap year

3·2.3 Calling Procedures and Functions. Actual Parameters

We now have subprograms with parameters. Four questions come up around them, over and over, so it is worth settling the vocabulary once and for all:

formal parameter the type and name we give a parameter in the subprogram's own declaration. This is very much like the counter of a `for` loop: `i` is declared in the loop's header, in parentheses, and yet it behaves as if it belonged inside the body — it is usable throughout it, and disappears once the loop ends (Figure 73). A formal parameter works the same way: it is written in the subprogram's parentheses, right next to its name, but it is scoped to the body exactly as if it had been declared inside it.

actual parameter the value, or expression, we supply at each individual call to the subprogram. The only rule is that it must be assignable to its corresponding formal parameter — the same compatibility rule that governs any ordinary assignment. Actual parameters are matched to formal parameters one by one, in the order they are written, not by name.

returned value when a subprogram returns a value (procedures, as we shall soon see, return nothing at all), that value must be compatible in assignment with the subprogram's own type, and each `return` statement hands back exactly one value. C++ does allow several `return` statements scattered through a function, but for now we will keep to the convention of writing a single one, at the end of the body, once the value to hand back has been fully computed — a lone exit point is far easier to read and to reason about.

default values the formal parameters at the tail of the parameter list may be given a default value. This saves you from repeating, at every call, whatever value that parameter typically takes: if a call omits it, the default written in the declaration is used instead. C++ only allows this for a trailing suffix of the list — once one parameter has a default, every parameter after it must have one too, or the compiler would no longer be able to tell which values in a shorter call belong to which parameters.

The fragment below does not define `combinatorial`, `printComplex`, `printlns`, or `exseries` — it only declares their prototypes, to show, side by side, several ways of calling them:

```cpp
#include <iostream>
using namespace std;

int main()
{
    int combinatorial(int n, int p);
    void printComplex(float real, float im);
    void printlns(int n = 1);
    float exseries(float x, float precis);

    int a = 5;
    float p_im = -1.5, x = 3.0, y = -2.0, r;

    cout << combinatorial(a, 2);
    printComplex(2 + 3.0, p_im);
```

```
    printComplex(x, y);
    printlns();
    printlns(3);
    r = exseries(0.5, 0.00001);

    return 0;
}
```

Notice `printComplex(2 + 3.0, p_im)`: the expression `2 + 3.0` is a `double`, yet `real` expects a `float` — it is silently converted, exactly as the assignment-compatibility rule for actual parameters promised above. Notice also `printlns()` and `printlns(3)`: the same subprogram called with and without its (defaulted) parameter.

In prototypes, it is not mandatory to give the names of the formal parameters (except for parameters that have a default value, which needs a name to attach the default to); you can write only what matters for the call itself: the type.

```C++
int main()
{
    int combinatorial(int, int);
    void printComplex(float, float);
    void printlns(int n = 1);
    float exseries(float, float);
    // ..
}
```

3·2.4 Passing Parameters by Value and by Reference

We have learnt that a subprogram can return a value — but only one value, one single piece of information.

However, it is quite common for a program to need a subprogram that returns two, or more, values, and that is simply not possible with a function as we have defined it so far.

Imagine we need a function to compute the solutions of a second-degree equation. There can be two solutions, so we might be tempted to write something like:

```C++
float, float quadEq(float a, float b, float c)
{
  return x1, x2;
}
```

This does not compile: a function has exactly one return type and returns exactly one value — never a comma-separated list of either.

To work around this limitation, C++ offers another mechanism.

Remember that actual parameters are, so far, always *copied* into formal parameters — that is the default mechanism in most programming languages. But there is another way to pass a value to a formal parameter: passing it **by reference**.

So far, when values are passed by copy, what the formal parameter — and hence the subprogram — receives is a plain copy: exactly as when one variable is assigned to another, the copy keeps no further connection to the original. You can change your copy inside the subprogram, but the

original actual parameter is safe; nothing the subprogram does to its own copy can reach it.

But C++ offers a different mechanism for passing parameters, passing the parameter **by reference**.

Original C could only pass parameters by copy or, for pointer-like access to the original, by explicitly passing a pointer to it. C++ adds a third, more controlled way: passing by reference.

Passing a parameter by reference uses this syntax:

```C++
#include <iostream>
using namespace std;

int main()
{
    void swap(int& a, int& b);

    int var1 = 1;
    int var2 = 2;
    swap(var1, var2);
    cout << var1 << ", " << var2 << endl;
    return 0;
}

void swap(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}
```

In the example above, the parameters are passed by reference. The procedure `swap` receives references to the actual variables `var1` and `var2` when it is called on line 10. A reference, once modified, modifies the very variable it refers to — so, inside `swap`, line 18 immediately changes `var1` (through `a`), and line 19 immediately changes `var2` (through `b`). Any other call, from `main()` or from any other scope, would send the corresponding references to `swap()`, and the corresponding variables would be affected in exactly the same way.

In fact, the declaration of references in C++ is not restricted to parameters, but in practice parameters are pretty much the only place you will actually use them.

Remember: to receive an actual parameter by reference, declare the matching formal parameter as

```
type& name
```

The `&` may sit next to the type (`type& name`), next to the variable name (`type &name`), or on its own with spaces on both sides (`type & name`) — C++ does not care about the spacing, but the `&` itself must be there.

One classic pitfall: in a multi-variable declaration such as `int& a, b;`, the `&` only binds to `a` — `b` is declared as a plain `int`, not a reference. When in doubt, declare one reference per line.

Because a reference parameter gives the subprogram direct access to the caller's own variable, only an actual variable — something with an address, an *lvalue* — can be passed to it. A literal or the result of an expression, such as ~~`swap(1, 2)`~~ or ~~`swap(a + 1, b)`~~, has nowhere for the reference to point to, so C++ refuses to compile it.

The swapping example above could not be written in C++ in any other way — short of resorting to a far clumsier way of wrapping the information, or C-style pointers.

Passing by reference is not only useful for returning multiple values or allowing a subprogram to modify the caller's variables; it also avoids the cost of copying, which becomes significant once variables are no longer single numbers but rather larger structures such as arrays and records, which we will cover in the next chapter. However, we will not address this type of subtle performance optimisation in this course.

Although C++ allows parameters to be passed by reference in both functions and procedures, many authors — and even entire programming languages — forbid passing by reference in functions altogether, reserving it for procedures: in that view, a function should only ever communicate through the single value it returns, never by silently modifying its parameters.

Remember:

exact type matching passing by reference requires that the actual and formal parameters be of exactly the same type.

side effects actual variables passed by reference are modified implicitly, which makes the program less safe. It is the programmer's responsibility to make it evident that changes or initial values are set implicitly in the subprogram call.

As a final example, consider the second-degree equation subprogram we discussed earlier. For this, we pass a, b and c from the equation $ax^2 + bx + c$ to the procedure and expect it to populate the two by-reference parameters typically placed at the end of the parameter list:

```
void quadEq(float a, float b, float c, float& x1, float& x2)
{
    ...
}
```

What if you wanted to determine the nature of the solution? There are several possibilities that we could enumerate as follows:

```
enum TYPEEQ {ANYX, ABSURD, LINEAR, DOUBLE, IMAG, TWODIFSOLS};
```

How could we modify the procedure (or function) to also receive this information?

In summary, passing parameters by reference entails the following implications:

- 👎 Susceptibility to side-effect issues
- 👎 Actual parameters must be variables of identical type, which precludes the use of constants or expressions as actual parameters, thereby limiting their applicability
- 👍 Improved performance, particularly for large data structures, as no copying is required
- 👍 Ability to return two or more values

Therefore,

Only use `&` when you need to return more than one value

3·2.5 Checkpoint

1) How many return values can a function have?
2) Apart from the requirement to obtain two or more results, when would parameters passed by reference be useful?
3) Can a function receive and modify a parameter passed by reference? Why is this approach appropriate or inappropriate?
4) What makes sending parameters by reference less convenient?
5) Is it necessary for the prototype's formal parameters to have names?
6) What is a default value? Where should parameters with default values be positioned in the parameter list?

3·2.6 Modularisation Criteria

When to use subprograms?

So far we have written many programs without ever needing to split part of them off into a subprogram. But as programs grow, the number of variables and small tasks `main()` has to juggle turns a monolithic `main()` into something hard to follow. On the other hand, many programs need to perform the same operation, or extract the same kind of information from different values, over and over again. Repeating the same calculation again and again is not only tedious — it is also hard to maintain.

Three criteria, in particular, help decide when a piece of code deserves to become a subprogram of its own:

different tasks when part of a program is devoted to something clearly different from the goal of the surrounding code, split it off into a subprogram. To do so, you need to know exactly what information that separate goal depends on — and this is itself a good test: if you can name the handful of parameters a piece of code depends on, and nothing else, that piece of code is naturally independent from the rest, and a good candidate for a subprogram.

repetition the previous criterion is often, by itself, reason enough to produce clear modules. But sometimes a program is not complex enough to have obviously separable parts, and yet still needs to repeat the same operation several times over. Whenever you can pin down exactly what information an operation needs and what value it produces, that repetition alone is reason enough to turn it into a subprogram — even where the rest of the program has no other natural seams.

readability even small parts of a program — ones that could be solved in a line or two of code — deserve to become an independent

subprogram if they are important and play a well-defined role. Take the "is this an uppercase letter" query, `'A' <= c and c < 'Z'`: it could be expressed far more clearly as:

```
bool isUpper(char c)
{
  return 'A' <= c and c < 'Z';
}
```

at the bottom of our `.cpp` file. This function could easily be modified later — to handle accented letters, say — without touching the rest of the algorithm that uses it.

The next program pushes this even further: every detail lives at the bottom, and reading it from the top down tells the whole story.

```
// lcm.cpp
// juanfc 2026-09-09
// least common multiple

#include <iostream>
using namespace std;


int main()
{
    int lcm(int a, int b);

    cout << lcm(4, 6) << endl;
    cout << lcm(21, 6) << endl;
    return 0;
}

int lcm(int a, int b)
{
    int gcd(int a, int b);

    return (a*b)/gcd(a,b);
}

int gcd(int a, int b)
{
    int smallest(int a, int b);

    int m = smallest(a, b);
    while (a % m != 0 or b % m != 0)
        --m;
    return m;
}

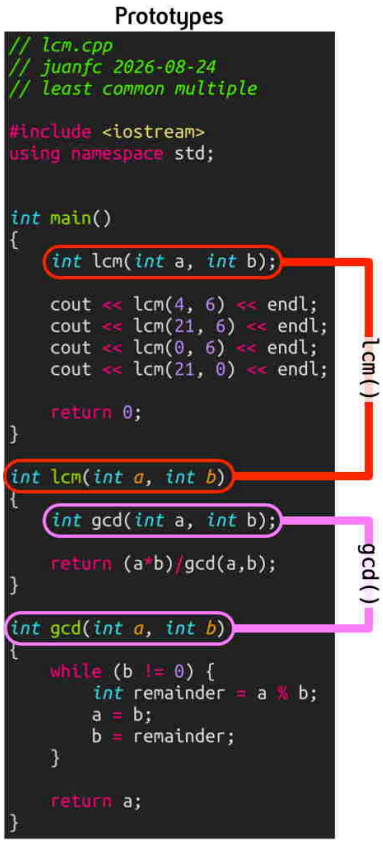
int smallest(int a, int b)
{
    return (a < b) ? a : b;
}
```

3·2.7 Local and Global Variables. Side Effects

Every variable we have declared so far — inside `main()`, inside a subprogram's body, or as one of its formal parameters — is a **local variable**: it exists only within the scope of the block that declares it, and disappears the moment that block ends, exactly as we saw earlier in this chapter (Figure 73).

We cannot go on any longer without forbidding you from using what are called **global variables**. We have not needed one so far, so it should be clear that we can live without them — but they pose a luring temptation all

91: From the index representing `main()` down to the details, each subprogram is independent and can be placed in any region of our `.cpp` file. The internal declaration of prototypes at the top of each subprogram documents and enables their freedom to be arranged in the order of our choice.



the same: just declare the variable outside the scope of every subprogram, and there it is, reachable from anywhere.

Global variables do have a real advantage: they save you from passing the same parameters around everywhere. In a program built around one key piece of information that almost everything works on, it can feel odd not to simply declare it globally, once, before `main()`, as one glorious piece of shared information.

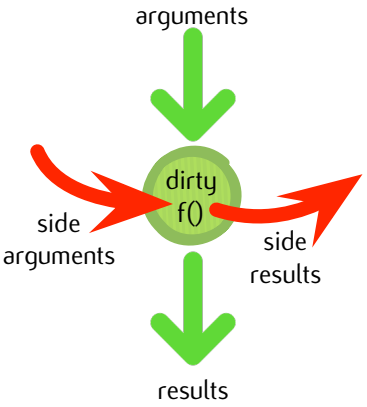
The bad news is that they are entirely forbidden in this course.

Older languages, such as early Fortran or BASIC, relied heavily on global variables — those were different times. Modern languages make it far easier, and far safer, to pass information around explicitly instead.

Global variables are very hard to control. They live in the global scope, so every scope, no matter how deeply nested, can read them — and, far more dangerously, can write to them too. Imagine a program full of subprograms calling one another, each one silently tweaking the same global variable. You call a subprogram to do one specific thing and, after the call returns, that important variable — never part of the parameter list, never part of the subprogram's declared interface — has changed anyway. Wrong.

This is exactly what is called a **side effect**: a change a subprogram makes to the state of the program beyond the value it returns, invisible unless you go and read the subprogram's entire body. A subprogram that only reads and writes its own parameters and local variables has no side effects; one that reaches out and modifies a global variable always does — and that is precisely what makes global variables so treacherous to debug.

92: The interface or prototype of subprograms should define their requirements and the scope of their operation. If they require external or global variables, they become coupled to the environment and cannot be relocated. If they operate on external variables, this is even more problematic, as it produces side effects.





3·2.8 Checkpoint

1) Why are parameters passed by reference not considered a side-effect issue?
2) Why do many poorly structured programs tend to use global variables?

3·2.9 Preconditions and Postconditions. Handling Exceptional Situations

As we saw in Section 2·14, handling exceptions in full is not a central concern of an introductory programming course. Even so, two ideas from that title are worth pinning down properly now that we have subprograms:

precondition what a subprogram expects to already be true — about its parameters, typically — before it starts doing its job. Checking that a precondition holds is, by convention, the **caller's** responsibility; a subprogram only promises to behave correctly when its preconditions are satisfied.

postcondition what a subprogram guarantees to be true once it returns — about its result, and about anything else it is allowed to change — provided its precondition was met. This is the promise the subprogram makes back to whoever called it.

Take `gcd(a, b)` from earlier in this chapter (Figure 91): it has an implicit precondition — `a` and `b` must both be different from `0` — and a postcondition: the value it returns is the greatest common divisor of `a` and `b`. Nothing in the code said so explicitly there; the compiler would not have stopped you from calling `gcd(0, 0)`, but the **contract** would have been broken all the same.

Whether a given situation is treated as an **exception** — rather than just another branch of ordinary logic — depends on three things:

how the operating system and hardware react when we ask them to perform an action, different systems can react differently to the very same request — dividing by `0` being a classic example.

what we, the programmers, decide counts as exceptional versus a situation the program should simply absorb as part of its normal flow. It is exactly this level of tolerance that turns a situation into an exception, and it is exactly what a precondition is for: stating, for each subprogram — and for the program as a whole — what it does and does not expect to receive.

the kind of application we are building it makes a real difference whether this is a simple algorithm exercise, a commercial program meant to run untouched for years, or firmware embedded in a chip, where reacting correctly to every unexpected situation can be critical.

All this said, remember that in Section 2·14 we already met two of the three commands C++ uses to handle exceptions — `try` and `catch` — and promised to come back to the third once subprograms existed. That third command is `throw`: it is how a subprogram reports that one of its preconditions has been broken, handing a value — often just a descriptive message — up to whichever `catch` is prepared to deal with it.

Exceptions make much more sense once subprograms exist, and rather little without them: when a subprogram calls another, which in turn calls another, an exception thrown deep inside can be caught several scopes higher up, without every intermediate subprogram having to check for it and forward it by hand, and without the whole program having to crash.

A simple example, reusing `lcm`, `gcd`, and `smallest` from earlier in this chapter (Figure 91), now guarded against the division by zero that an unchecked `lcm(0, n)` would otherwise cause:

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

int main()
{
    int lcm(int a, int b);

    try {
        cout << lcm(4, 6) << endl;
        cout << lcm(21, 6) << endl;
        cout << lcm(0, 6) << endl;   // breaks the precondition
    } catch (const domain_error& e) {
        cout << "ERROR: " << e.what() << endl;
    }
    return 0;
```

```
}

int lcm(int a, int b)
{
    int gcd(int a, int b);

    return (a*b)/gcd(a,b);
}

int gcd(int a, int b)
{
    if (a == 0 or b == 0)
        throw domain_error("gcd requires a and b to be non-zero");

    int smallest(int a, int b);

    int m = smallest(a, b);
    while (a % m != 0 or b % m != 0)
        --m;
    return m;
}

int smallest(int a, int b)
{
    return (a < b) ? a : b;
}
```

Notice too that `gcd` checks its precondition explicitly, with a plain `if`, before it computes anything at all. The call to `lcm(0, 6)` on line 12 breaks that precondition two calls deep — inside `gcd`, which throws on line 29 — and yet `main()` catches it directly, on line 13, without `lcm` needing to know anything about it. Lines 10 and 11 already ran and printed their results; the rest of the `try` block, everything after line 12, is simply skipped — exactly the jump described earlier in this chapter.

This matters: an actual division by zero — writing `a % 0` and letting C++ evaluate it — is not something a `try`/`catch` can rescue you from. It is undefined behaviour, and in practice it crashes the program outright, exactly like the unguarded `n/d` back in Section 2·14. Exceptions only help with mistakes we catch **before** they happen and report **deliberately**, with `throw`.

Note that when we say

```
throw domain_error("gcd requires a and b to be non-zero");
```

we are not **returning** a value: this time `gcd` gives `main()` no `int` back. Instead it abandons its own work — and `lcm`'s, still waiting on it — and the thrown object travels up the call stack until it meets a `catch` whose type matches. That match is the whole point: our `catch (const domain_error&)` receives this because a `domain_error` is exactly what was thrown, and, being a standard exception, it carries the `.what()` message we printed. A bare `string` would not have matched it at all — which is why we throw the standard types from `<stdexcept>` rather than improvising with a raw value.

3·2.10 Inline subprograms in C++

When calling a subprogram, it is necessary to generate and insert preparatory code before the call and resumption code after it to restore the current scope. Just as `const type&` is a C++ optimisation that avoids the internal overhead incurred by passing objects, the compiler also provides an optimisation available from version 2.0 known as `inline`. You can prepend any subprogram declaration with the `inline` keyword, and the compiler will then optimise the generated code by eliminating the context switch overhead. Rather than performing an actual function call, the compiler instead substitutes the function body (typically small) directly at each call site. For example, every invocation of `inline int fact(int n);` is replaced by the body code itself.

In this course, we are not particularly focused on compiler-specific optimisations, and for this reason you will not encounter `inline` optimisation in practice, just as you will not see `const type&` in any parameter passing. However, it is worth understanding why these optimisations exist and who might benefit from their use.

3·2.11 Checkpoint

1) At what level in the subprogram hierarchy is it appropriate to use a `try`/`catch`/`throw` mechanism?
2) Is `throw` equivalent to the `return` statement?

3·3 Recursion

A subprogram is not restricted to calling other subprograms: it can call itself. When it does, we call it **recursive**. It sounds almost like cheating — solve a problem by calling the very subprogram that is trying to solve it — but it is one of the oldest and most elegant ideas in programming, and C++ needs no special syntax for it at all: a function may appear inside its own body exactly as it would call any other function.

> To understand recursion, you must first understand recursion.

That joke contains the whole warning this section exists to give: recursion, done carelessly, goes nowhere but in circles. Done properly, it turns problems that are naturally defined in terms of smaller versions of themselves into code that is almost a direct transcription of their definition.

```C++
int main()
{
  void a();

  a();  // we do not want `main` to be recursive

  return 0;
}

void a()
{
  if (cond) {
    // BASE CASE, no a()
  } else {
    a();
  }
}
```

3·3.1 Concept of Recursion

A recursive subprogram always needs two ingredients:

base case at least one situation, checked first, where the subprogram can answer directly, without calling itself again. This is what eventually stops the recursion.

recursive case the situation where the subprogram calls itself, but on a **smaller** version of the same problem — one that is guaranteed to be closer to a base case than the one it started with.

Miss the base case, or fail to make the problem smaller on every recursive call, and the subprogram calls itself forever — or rather, until it runs out of **stack**: every call, recursive or not, needs its own private storage for its parameters and local variables, stacked on top of the calls that are waiting for it to finish. A recursion with no way out eventually stacks so many of these that the program crashes with a **stack overflow**, C++'s version of running out of "call slots".

We already have a perfect example on hand: `fact`, from earlier in this chapter. Its own mathematical definition is already recursive — $n! = 1$ when $n \leq 1$, and $n! = n \times (n-1)!$ otherwise — so translating it into C++ is close to copying the formula directly:

```C++
int fact(int n)
{
    if (n <= 1)
        return 1;              // base case
    return n * fact(n - 1);    // recursive case: a smaller problem
}
```

Compare this with the iterative version we wrote earlier, with its explicit `for` loop and running total `r`: nothing here counts or accumulates by hand. `fact(5)` simply says "5 times whatever `fact(4)` turns out to be", and trusts the rest to sort itself out, one smaller call at a time, all the way down to `fact(1)`, which does not need to trust anything — it just answers.

3·3.2 Examples

Take the running sum we designed around a loop invariant earlier in this chapter: `sum == 1 + 2 + ... + N`. Read as a recursive definition instead of a loop, the same idea says: the sum of the first `n` naturals is `0` when `n` is `0`, and `n` plus the sum of the first `n - 1` naturals otherwise.

```C++
int sumUpTo(int n)
{
    if (n == 0)
        return 0;              // base case
    return n + sumUpTo(n - 1); // recursive case
}
```

It helps, the first few times, to unwind a small call by hand and watch nothing get computed until the base case is finally reached:

```Display
sumUpTo(4)
      = 4 + sumUpTo(3)
      = 4 + (3 + sumUpTo(2))
      = 4 + (3 + (2 + sumUpTo(1)))
      = 4 + (3 + (2 + (1 + sumUpTo(0))))
      = 4 + (3 + (2 + (1 + 0)))
      = 10
```

Each line waits, unresolved, for the call nested inside it to answer first — exactly the stack of pending calls described above, four deep at its worst point here.

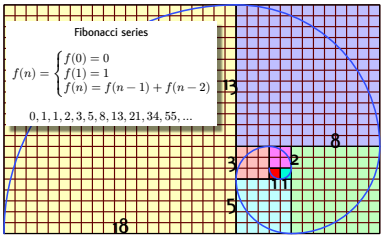
`fact` and `sumUpTo` both make a single recursive call each time, so their calls form a simple chain, as long as `n`. Not every recursion is so tame. The Fibonacci sequence — each number the sum of the two before it —

is recursive by definition, and the direct translation makes **two** recursive calls per call:

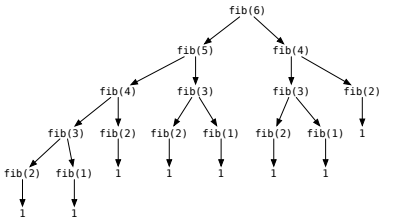
```cpp
int fib(int n)
{
    if (n <= 1)
        return n;                    // base case
    return fib(n - 1) + fib(n - 2);  // two smaller problems
}
```

Instead of a chain, the calls now form a branching tree: `fib(5)` calls `fib(4)` and `fib(3)`, `fib(4)` calls `fib(3)` and `fib(2)`, and so on — with `fib(3)` computed twice along the way, `fib(2)` several times more, and the repetition getting rapidly worse as `n` grows. This is the same lesson from Section 1·4, back when we first compared algorithms by how their cost **grows**: a recursive definition copied faithfully into code is not automatically an efficient program, even when it is a perfectly correct one.

93: The Fibonacci sequence is an important topic of study in mathematics



94: Observe that in a recursive solution to the Fibonacci sequence, although the simplest to write and understand, the **repetitive calls** to the same function grow exponentially with the depth of recursion

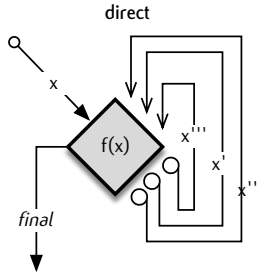


3·3.3 Types of recursion

Subprograms can call themselves back in more than one way. It is worth naming the differences, since the vocabulary is common enough that you will meet it again.

Direct recursion the subprogram calls itself, exactly as `fact`, `sumUpTo`, and `fib` did above.

95: Direct recursion: the subprogram calls itself

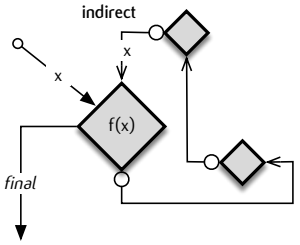


Indirect recursion a subprogram calls a second one, which — perhaps after a chain of further calls — ends up calling the first one again. The simplest case is just two subprograms calling each other:

```cpp
void a()
{
  b();
}

void b()
{
  a();
}
```

96: Indirect recursion: A function calls another function, which in turn calls the first function



As drawn here there is no base case in sight, so `a` and `b` would call each other forever. In a real program, exactly as with direct recursion, at least one of the two must eventually take a path that skips the call to the other — otherwise the pair recurses without end, just as surely as a single subprogram missing its base case would.

Two more distinctions matter: where, inside the subprogram, the recursive call sits, and how many recursive calls it makes.

Tail recursion the recursive call is the very last statement the subprogram executes — nothing is left to do once it returns.

```cpp
#include <iostream>
using namespace std;

int main()
{
  void countDown(int n);
```

```
  countDown(5);

  return 0;
}

void countDown(int n)
{
  if (n != 0) {
    cout << n << " ";
    countDown(n - 1);   // the last thing this call does
  }
}
```

Tail recursion is the easiest one to convert into simple iteration.

Head recursion the recursive call is the first statement the subprogram executes — everything else happens only once it returns.

```
void countUp(int n)                                                C++
{
  if (n != 0) {
    countUp(n - 1);      // the first thing this call does
    cout << n << " ";
  }
}
```

The two look almost identical, yet `countDown(3)` prints `3 2 1`, while `countUp(3)` prints `1 2 3`. In `countDown`, each call prints **before** handing off to the next, so the numbers appear in the order the calls are made. In `countUp`, each call waits for its recursive call to finish completely before it prints anything, so the numbers appear in the reverse order — the smallest call, the one that unwinds first, prints first.

- Tail recursion has one more advantage: because nothing happens after the recursive call, some compilers can turn it into an ordinary loop behind the scenes, avoiding the stack-growth cost described in Section 3·3.4 below.

Linear recursion the subprogram makes exactly one recursive call each time it runs, as `fact`, `sumUpTo`, `countDown`, and `countUp` all do — their calls form a simple chain.

Tree recursion the subprogram makes more than one recursive call each time it runs, as `fib` does above. Its calls form a branching tree rather than a chain, and the cost can grow explosively: with two recursive calls at almost every step, the number of calls made roughly doubles with each increase in `n`, giving the exponential time complexity we already noted for `fib`.

Recursion can solve some genuinely delicate problems with remarkably little code. Many data structures you will meet later in your studies — trees, graphs — are naturally traversed with recursive algorithms, one call handling each branch.

The Towers of Hanoi puzzle is often held up as the best example of that elegance. An iterative solution does exist, but its code looks nothing like the puzzle; the recursive solution, by contrast, reads almost like the puzzle's own rules.

Imagine a procedure that moves a whole stack of `n` disks from one rod to another, using the third rod as temporary storage along the way: `moveGroupUsing(n, base, inter, dest)` moves `n` disks from rod `base` to rod `dest`, parking disks on rod `inter` whenever it needs somewhere to put them down. The recursive insight is that moving `n` disks reduces to three smaller moves: move the top `n - 1` disks out of the way, from `base`

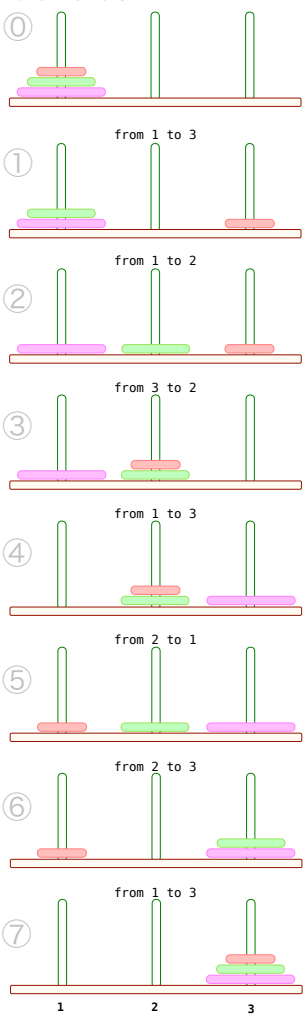
onto `inter` (using `dest` as the temporary spot for that smaller move); move the one remaining, largest disk directly from `base` to `dest`; then move those same `n - 1` disks from `inter` onto `dest` (now using `base` as the temporary spot). Each "smaller move" is the very same problem, one disk shorter — which is exactly what makes it recursive, and exactly what the base case (`n == 0`: nothing to move) eventually stops.

```cpp
// hanoi.cpp
// juanfc 2026-09-09
#include <iostream>
using namespace std;

int main()
{
    void moveGroupUsing(int n, int base, int inter, int dest);
    int noDiscs;
    do {
        cout << "No. of discs (0 to end): ";
        cin >> noDiscs;
        moveGroupUsing(noDiscs, 1, 2, 3);
    } while (noDiscs > 0);
    return 0;
}

void moveGroupUsing(int n, int base, int inter, int dest)
{
    if (n > 0) {
        moveGroupUsing(n - 1, base, dest, inter);
        cout << "from " << base << " to " << dest << endl;
        moveGroupUsing(n - 1, inter, base, dest);
    }
}
```

97: Hanoi Towers



For two discs, `moveGroupUsing(2, 1, 2, 3)` prints:

```
from 1 to 3
from 1 to 2
from 3 to 2
from 1 to 3
from 2 to 1
from 2 to 3
from 1 to 3
```

exactly the well-known three-move solution: the small disc out of the way, the big disc across, the small disc back on top.

3·3.4 Recursion versus Iteration

Anything a loop can compute, a recursive subprogram can compute too, and vice versa — the two are, in that sense, equally powerful. What differs is cost and clarity, and both cut in opposite directions.

Iteration reuses the same handful of variables on every pass, so its memory use stays constant regardless of how many times the loop runs; a `for` or `while` looping a million times costs no more storage than one looping ten. Recursion, by contrast, opens a new stack frame on every call and only closes it once that call returns, so its memory use grows with how **deep** the recursion goes — `fact(20)` opens twenty frames before the first one can close. Push that depth far enough — a large `n`, or a missing base case — and the program crashes with a stack overflow long before an equivalent loop would have shown any strain. Recursive calls also carry the ordinary overhead of any function call, paid once per element processed; a loop pays no such toll.

Against that cost, recursion offers clarity, precisely when a problem's own definition is recursive: `fact` and `sumUpTo` above read almost exactly like the mathematical definitions they implement, with no bookkeeping variable required to carry a running total across iterations. It even explains a detail from earlier in the course: when we first mentioned Quicksort (Figure 28), we said it "recursively sorts" the sublists it creates — now you know exactly what that meant. That algorithm, and others like it, split a problem into smaller pieces of the same problem — precisely the shape recursion is built for — which is why you will keep running into it as soon as the structures we work with, starting in the next chapter, stop being single numbers.

For the simple, numeric problems we have met so far, though, iteration is usually both clearer and cheaper: no risk of stack overflow, no per-call overhead, and the loop-invariant discipline from earlier in this chapter to reason about correctness directly. As a rule of thumb for this course: default to iteration, and reach for recursion when a problem is naturally defined in terms of smaller versions of itself, so that the recursive solution is dramatically clearer than any loop you could write for it.

This is exactly the **tail recursion** we distinguished from head recursion above — the case where the recursive call is the very last thing the subprogram does. It is a useful thing to know exists; we will not rely on it in this course.

3·3.5 General algorithm for converting recursion into iteration

This section is optional and goes beyond the syllabus. It is here for the curious reader who wants to see **why** recursion is, in a precise sense, more powerful than iteration.

Any recursive function can be rewritten as an iterative one. The reason is simple: recursion is nothing more than the automatic use of a hidden data structure — the **call stack**. Every time a function calls itself, the machine quietly saves the current parameters, the local variables, and the exact point to resume once the call returns. Iteration does none of this for you. So the trick is to **make that stack explicit** and manage it by hand.

The universal recipe. Replace every recursive call with the bookkeeping the machine was doing on your behalf:

1. Keep an explicit `stack` of **frames**. Each frame stores the saved parameters, the saved local variables, and a **return label** — a marker telling you which point of the code to continue from once that call finishes.
2. Wrap the whole body in a single loop that runs `while` the stack is not empty.
3. Turn each recursive call into: push a new frame (with the values you will still need afterwards and the return label), reset the parameters to the arguments of the call, and jump back to the start.
4. Turn each `return` into: store the result, pop the top frame, restore its saved values, and jump to its return label.

That is all. The transformation is completely mechanical and **absolutely general** — it works for every recursive function, no matter how tangled — and it is, frankly, ugly: you are hand-writing the exact machinery a compiler generates for you. Schematically:

```text
push initial frame
while stack is not empty:
    read the top frame, jump to its return label
  L_start:
    ... work ...
    // for a recursive call f(args):
    save (locals, L_after) on the stack
    parameters <- args
    goto L_start
```

```
  L_after:
    ... use the returned value ...
    // for "return E":
    result <- E
    pop the frame
    goto (that frame's return label)
```

Notice the explicit **return labels** and jumps: this is precisely the low-level control flow that the elegant recursive version hides — which is exactly why, in real code, we never write like this. (Knuth 1974) works through the readable transformations in full detail.

The easy special case: tail recursion. When the recursive call is the **last** thing the function does — nothing is left to compute after it returns — there is no saved work to stack up. The frame is useless, and the whole thing collapses into a plain loop with no stack at all:

```C++
// tail-recursive
int gcd(int a, int b)
{
    return b == 0 ? a : gcd(b, a % b);
}
// iterative — no stack needed
int gcd(int a, int b)
{
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

This is the case compilers optimise automatically (**tail-call optimisation**), and the one you can always rewrite by hand without any auxiliary structure.

How far does recursion really go beyond iteration? Here is the deep point. For tail recursion we needed no stack; for the general recipe we needed one. Could we always be cleverer and get rid of it, keeping just a fixed number of ordinary variables and a loop? In general, the answer is **no**. Viewed as abstract program **schemas**, recursion is strictly more powerful than iteration: there are recursive definitions that cannot be simulated by any loop using only a bounded amount of extra memory — the auxiliary stack is genuinely unavoidable ((Paterson, Hewitt 1970)). So the stack in our recipe is not laziness on our part; it is essential. That unbounded, last-in-first-out memory is precisely the extra power recursion buys you over a plain `for` or `while`.

In practice, once you apply the mechanical recipe to a **specific** function, you can often simplify the result into an idiomatic loop. (Bird 1977) catalogues these stack-based techniques systematically, and (Barron 1968) is the classic short monograph on recursion and its removal. But the ugly general form is your guarantee: the conversion is **always** possible.

3·3.6 Checkpoint

1) Every recursive subprogram needs two ingredients. Name them, and say what each one is responsible for.

2) What happens if a recursion has no base case, or if the recursive call is not made on a **smaller** problem? What is the name of the crash it eventually causes?

3) `countDown` and `countUp` differ by a single line. What does each print for `n = 4`, and why does one of them come out in reverse order?

4) Classify each of the following as **direct** or **indirect** recursion, and as **linear** or **tree** recursion: `fact`, `fib`, and a pair of procedures `a` and `b` in which `a` calls `b` and `b` calls `a`.

5) The recursive `fib` is perfectly correct, yet very slow. In terms of the shape its calls take, explain why its running time grows so fast.

6) Compared with an equivalent loop, what extra cost does recursion pay in memory, and what failure does very deep recursion risk?

7) Which kind of recursion can be turned into a plain loop with no extra memory at all, and why? And why, in general, can the auxiliary stack **not** always be removed?

8) Write a recursive `int power(int base, int exp)` (assume `exp >= 0`) that returns `base` raised to the power `exp`. Point out its base case and its recursive case.

References

BARRON, David W., 1968. *Recursive Techniques in Programming*. New York: American Elsevier. Computer Monographs.

The classic short monograph on recursion and its removal.

BIRD, Richard S., 1977. Notes on Recursion Elimination. *Communications of the ACM*. 1977. Vol. 20, no. 6, p. 434–439. DOI **10.1145/359605.359630**.

A systematic catalogue of stack-based methods for removing recursion, compared by running cost.

KNUTH, Donald E., 1974. Structured Programming with go to Statements. *ACM Computing Surveys*. 1974. Vol. 6, no. 4, p. 261–301. DOI **10.1145/356635.356640**.

The definitive account of turning recursion into iteration by hand, working through the readable transformations step by step. The provocative title notwithstanding, it argues for good program structure, not for `goto`.

PATERSON, Michael S. and HEWITT, Carl E., 1970. Comparative Schematology. In: *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. New York: ACM. 1970. p. 119–127.

Proves, at the level of abstract program schemas, that recursion is strictly more powerful than iteration: in general the auxiliary stack cannot be eliminated.

Task 6: Subprograms

Make sure you have completed all exercises in Task 5 before starting this one. From this point forward, most problems should be solved using subprograms. The typical structure is: `main()` calls one subprogram, which in turn calls others. Most subprograms should perform computations or modify parameters passed by reference — **they should not read with `cin` or print with `cout`** unless specifically designed to do so.

Problems marked 🧊 are challenging; do not hesitate to ask for help. Problems marked 🧊🧊 are intended only for the most determined students.

1 Suppose the following variables are declared at the top of `main()`:

```C++
int a, b, c;
bool theEnd;
```

and that the following subprogram prototypes are available:

```C++
bool one  (int  x, int y);
void two  (int& x, int y);
int  three(int  x);
```

Which of the following calls from inside `main()` are legal? For those that are not, explain why.

1. `if ( one(a, b) ) {`,
2. `two(a, b+3)`,
3. `theEnd = one(c, 5)`,
4. `theEnd = two(c, 5)`,
5. `two(a, three(a))`,
6. `two(three(b), c)`,
7. `if (three(a)) {`,
8. `b = three(two(a, 5))`,
9. `two(4, c)`,

Write a file `t06e01.answers.txt` with each of the previous calls followed by your comments.

2 Build a function that returns the number of digits needed to represent `n` in a given `base`: › Sol

```
int numDigits(int n, int base=10);
```

From `main()`, call it with `n = 2021` in base `10` and base 2. It should return `4` and `11` respectively ($2021 = 11111100101_2$). `numDigits(0)` should return `1`.

3 Build a function that returns the p-th digit of `n` in base `base`: › Sol

```
int digit(int n, int p, int base);
```

Digits are counted from the right, starting at `0`. For example, `digit(25064, 1, 10)` returns `6` and `digit(25064, 10, 10)` returns `0`.

4 Build a function that computes the n-th integer power of a real number:

```
float powx(float x, int n);
```

Compute it by repeated multiplication (`r *= x`, starting from `1`). The exponent `n` may be negative, zero, or positive. From `main()`, ask the user for `x` and `n` and print the result. › Sol

5 Build a procedure that solves the quadratic equation $ax^2 + bx + c = 0$ and returns its two roots through reference parameters:

```
void quadEq(float a, float b, float c, float& x1, float& x2);
```

The roots are $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. From `main()`, ask for a, b, c and print `x1` and `x2`. Assume the discriminant $d = b^2 - 4ac$ is positive and $a \neq 0$.

For a more complete solution that handles all cases, add an integer output parameter `type` encoding the kind of solution found: › Sol

1. two distinct real roots,
2. real and imaginary parts of a conjugate complex pair,
3. a repeated real root,
4. a single root (because $a = 0$),
5. no solution ($a = b = 0$, $c \neq 0$),
6. any x is a solution ($a = b = c = 0$),

Build:

```
void quadEq(float a, float b, float c, float& x1, float& x2, int& type);
```

and test it with:

```
// (type 1) 2 roots: 1 and -2
quadEq( 1,  1, -2, x1, x2, type);
// (type 3) 1 double root
quadEq( 1, -2,  1, x1, x2, type);
// (type 4) 1 root: 0.5 only
quadEq( 0, -2,  1, x1, x2, type);
// (type 5) impossible
quadEq( 0,  0,  1, x1, x2, type);
// (type 2) complex: +- i
quadEq( 1,  0,  1, x1, x2, type);
// (type 6) any x
quadEq( 0,  0,  0, x1, x2, type);
```

6 A year is a **leap year** if it is divisible by 4 but not by 100, or if it is divisible by 400. For example, 2016 and 2000 are leap years; 1800, 1900, 2100, and 2300 are not. Build a function that receives a year and returns whether it is a leap year. From `main()`, ask the user for a year and print `"leap year"` or `"not a leap year"`. › Sol

7 Build a function `double harmonic(int n)` that returns the sum of the first `n` terms of the harmonic series: › Sol

$$\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots.$$

From `main()`, ask for `n` and print the result. Verify that the sum grows without bound as `n` increases.

8 Build a function `double alternate(int n)` that returns the sum of the alternating harmonic series: › Sol

$$\sum_{i=1}^{n} (-1)^{i+1} \frac{1}{i} = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots.$$

From `main()`, call it with a given `n` and print the result. Verify that it converges to $\ln 2 = 0.6931471805\ldots$

9 Build a function `float gravity(float m1, float m2, float d)` that returns the gravitational force between two masses m_1 and m_2 separated by distance d: › Sol

$$F_g = -G \frac{m_1 m_2}{d^2}$$

where $G = 6.67428 \times 10^{-11}$ N $\frac{\text{m}^2}{\text{kg}^2}$ (Cavendish constant).

Use it to compute:

1. The gravitational acceleration at the surface of the Earth ($m_1 = 5.974 \times 10^{24}$ kg, $d = 6,378,140$ m, $m_2 = 1$ kg),
2. The acceleration at geostationary orbit (**35,786 km** above the surface),
3. The surface gravity of asteroid Bennu (radius 245 m, mass 7.329×10^{10} kg)

10 Using the leap-year function from exercise 6, build a function that receives a complete date (year, month, day) and returns whether it is a valid calendar date. From `main()`, ask the user for year, month, and day, and print `"legal date"` or `"not legal date"`. › Sol

11 Build a function `bool isPrime(int n)` that returns `true` if `n` is prime. Start by trying all divisors (`i`) from 2 up to `n`; then optimise by stopping `i` at $\sqrt{n}$ and skipping even divisors. › Sol

HINT: Instead of computing $\sqrt{n}$, which requires a floating-point series computation from `<math>` library, just check `d*d < n and n%d != 0` at each iteration.

12 Using `bool isPrime(long long n)` from previous exercises, implement a function `unsigned long long nPrime(int n)` that returns the n-th prime number. Test it with: › Sol

```cpp
int main()
{
    unsigned long long nPrime(int);

    for (int i = 1; i < 100; i += 10)
        cout << i << ": " << nPrime(i) << endl;
    return 0;
}
```

Expected output:

```
 1:   2                                                          Display
11:  31
21:  73
31: 127
41: 179
51: 233
61: 283
71: 353
81: 419
91: 467
```

Then test it by computing the sum of the first `1000` prime numbers. Finally, determine and tell whether that sum is prime.

o O o

Task 7: Subprograms++

Make sure you have completed all exercises in Task 6 before starting this one. From this point forward, most problems should be solved using subprograms. The typical structure is: `main()` calls one subprogram, which in turn calls others. Most subprograms should perform computations or modify parameters passed by reference — **they should not read with `cin` or print with `cout`** unless specifically designed to do so.

Problems marked 🧊 are challenging; do not hesitate to ask for help. Problems marked 🧊🧊 are intended only for the most determined students.

1 Write a simple function without parameters that returns the length of the longest sequence entered via keyboard in which the characters `0` and `1` remain in ascending order. For example:

```
10100001102 -> 6 corresponding to 000011                    example
00112 -> 4 no disorder
112 -> 2 no disorder
002 -> 2 no disorder
2 -> 0 nothing
```

The procedure (for example: `int lss(void);`) will return this length, representing the longest ascending sequence, until a `2` is read, and `main()` will print this number. › Sol

2 🧊 Using `digitAtp()` from previous Tasks, build a procedure `void printSumSymDigits(int n)` that prints the sums of symmetrically placed digit pairs. For example, `2354869` gives: › Sol

```
2354869: 11 9 13 4                                          example
6582: 8 13
121: 2 2
0: 0
```

where `11 = 2+9`, `9 = 3+6`, `13 = 5+8`, `4` (middle digit, no pair). You will also need a helper `int noDigits(int n)`.

HINT: Subprograms should not read or print (no `cin`/`cout`) unless specifically designed to do so. Here it is convenient to reuse `digitAtp()` and `noDigits()`. A possible skeleton:

```cpp
int main()                                                  C++
{
    void printSumSymDigits(int);

    printSumSymDigits(2354869);

    return 0;
}

void printSumSymDigits(int n)
{
    int digitAtp(int, int);
    int noDigits(int);

    // cout << ...
}

int digitAtp(int n, int pos) { /* ... */ return ...; }
int noDigits(int n)          { /* ... */ return ...; }
```

3 Build a function `bool isPalindromic(int n)` that returns `true` if `n` reads the same forwards and backwards (e.g. `898`, `1719171`, and `1` are palindromes; `13` and `123` are not). Reuse `digitAtp()` and `noDigits()` from the previous exercises. › Sol

HINT: Compare digit `i` with digit `l-i`, iterating `i` from `0` to `l/2` while the symmetrical digits match, where `l` is the number of digits. Use a `while` loop: you must stop as soon as a mismatch is found, and a `for` loop would require either a `break` in the body or a variable guard to do so. Wrong if used in this course.

4 Build a function `bool isPalindromic(int n)` that returns `true` if `n` reads the same forwards and backwards (e.g. `898`, `1719171`, and `1` are palindromes; `13` and `123` are not). Build `digitAtp()` using the function `powerint()` › Sol

5 Build a **procedure** `void primeDecomposition(int n)` that prints the prime factorisation of `n`. For example: $1200 = 2^4 \times 3^1 \times 5^2$. › Sol

HINT: You do not need to call `nPrime()`. Instead, try dividing `n` by each integer from 2 upwards, counting how many times each divides it with a double loop, and reducing `n` each time until it falls below 2. We print results as we find them because we have no data structure yet to store them.

6 Goldbach's conjecture states that every even number greater than 2 is the sum of two primes. Build a procedure `void testGoldbach(int from, int to)` that verifies this for all even numbers in `[from, to]`. For the even numbers just above 2: › Sol

example
```
 4 = 2 + 2
 6 = 3 + 3
 8 = 3 + 5
10 = 3 + 7
12 = 5 + 7
```

If all numbers in the range satisfy the conjecture, print: `"From 20 to 2000 they follow Goldbach's"`. If one fails (say `3232`), stop and print `"3232 DOES NOT FOLLOW Goldbach's"`.

7 The greatest common divisor (GCD) of two natural numbers is the largest natural number that divides both. For example, $\gcd(8, 12) = 4$. Build `int gcd(int a, int b)` in three steps:

1. **Brute force.** Build `int gcdBruteForce(int a, int b)`: start from the smaller of `a` and `b` and test every integer downward until you find one that divides both. › Sol
2. **Euclid's algorithm** uses the properties:

$$\begin{cases} \gcd(a, b) = \gcd(b, a - b) \text{ if } a > b \\ \gcd(a, b) = \gcd(b, a) \\ \gcd(a, 0) = a \end{cases}$$

› Sol For example: $(12, 18) \to (18, 12) \to (6, 12) \to (12, 6) \to (6, 6)$, so $\gcd(18, 12) = 6$. A subtraction-based implementation:

```cpp
int gcd(int a, int b)
{
    do {
        if (a < b)
            swap(a, b);
        a -= b;
    } while (b != 0);
    return a;
}
```

Now rewrite it replacing subtractions with the `%` operator for a faster version.

Build a procedure `void simplify(int num, int den, int& nnum, int& nden)` that reduces the fraction `num/den` to lowest terms using `gcd()`, returning the result in `nnum` and `nden`. Test from `main()`: › Sol

```cpp
int main()
{
    void testIt(int, int);

    testIt(12,   18);  // 12/18:   2/3
    testIt(54,   24);  // 54/24:   9/4
    testIt(12,   30);  // 12/30:   2/5
    testIt(1260, 600); // 1260/600: 21/10
    return 0;
}
```

8 Build `int lcmBruteForce(int a, int b)` for the least common multiple using brute force: starting from the larger of `a` and `b`, search upward for the first number that is a multiple of both. › Sol

9 Build `int lcm(int a, int b)` using the identity (see here): › Sol

$$\operatorname{lcm}(a, b) = a \cdot b / \gcd(a, b)$$

10 One method to approximate $\sqrt{x}$ uses the identity:

$$\sqrt{n} = \sqrt{p + q} \approx \sqrt{p} + \frac{q}{2\sqrt{p}}$$

The idea is to find the largest perfect square $p \le n$, then apply the formula. For example, if $n = 968$: since $\sqrt{961} = 31$, we get $\sqrt{968} \approx 31 + \frac{7}{2 \cdot 31} = 31.112$. (The exact value is $31.11270\ldots$)

Build `int prevSq(int n)` that returns the integer p such that $p^2 \le n < (p+1)^2$. For example, `prevSq(968)` returns `31`. › Sol

11 In a non-executable file `14.rec2iter.cpp`, design the iterative equivalent of the following recursive procedure: › Sol

```cpp
bool f(int n);
int  g(int n);

void rec(int n)
{
    if (not f(n)) {
```

```
        // any statements that do not modify n
        rec(g(n));
    }
}
```

12 Design a recursive procedure that reads a sequence of characters of arbitrary length from the keyboard, terminated by a full stop (`'.'`), and prints the sequence in reverse order. The full stop character is not printed; it serves only as a terminator.

The procedure should have no parameters. `main()` will test the procedure. For example, if you enter `hola` from the keyboard, the output should be `aloh`. › Sol

13 The greatest common divisor (GCD) of two natural numbers p and q is the largest integer that divides both. Euclid's algorithm keeps subtracting the smaller value from the larger until both are equal. For example, starting from $p = 18$, $q = 12$: › Sol

```
18, 12  (p > q  =>  p -= q  =>  p = 6)                    Display
 6, 12  (q > p  =>  q -= p  =>  q = 6)
 6,  6  (p == q  =>  gcd = 6)
```

Implement a **recursive** version:

```
int gcd(int p, int q);
```

14 Copy, compile, and run the following program, which repeatedly asks the user for two integers, performs integer division, and handles division by zero with a `try`/`catch` block: X

```
// t07e14.trycatch.cpp                                    C++
#include <iostream>
#include <stdexcept>
using namespace std;

int main()
{
    int dividend, divisor;
    cout << "Enter dividend and divisor (0 0 to quit): ";
    cin >> dividend >> divisor;
    while (dividend != 0 or divisor != 0) {
        try {
            if (divisor == 0)
                throw invalid_argument("Divisor cannot be zero");
            cout << dividend << " / " << divisor
                 << " = " << dividend / divisor
                 << " remainder " << dividend % divisor << endl;
        } catch (invalid_argument& e) {
            cout << "Error: " << e.what()
                 << ". Please do not enter zero as the divisor." << endl;
        }
        cout << "Enter dividend and divisor (0 0 to quit): ";
        cin >> dividend >> divisor;
    }
    return 0;
}
```

Task 8: Pre-structures

Make sure you have completed all exercises in Task 7 before starting this one. Since we have not yet introduced data structures, none of the solutions below should use arrays or any other structured type. You will use them freely from Task 9 onwards.

1 Build the program `t08e01.readnum.cpp`. We want to read a number from the user, but only accept it if it falls within a given range. If the number is out of range, keep prompting until a valid one is entered. Build the function: › Sol

```
int readNum(int from, int to);
```

that returns an integer entered by the user, rejecting any value outside `[from, to]`. Call it from `main()` and print the result.

2 Build the program `t08e02.difextremes.cpp` *(no subprograms needed)*. Read integers from the user until `0` is entered; `0` is not part of the sequence. Print the difference between the maximum and minimum values. Assume at least one number is entered before `0`. › Sol

```
1 2 3 4 0    -> 3                                    example
-2 2 0       -> 4
2 2 0        -> 0
-2 0         -> 0
```

3 Build a function `bool lowUpBalanced()` that reads characters from the user (spaces are skipped automatically with `cin >>`). Input ends when a period `'.'` is entered. The function returns `true` if the number of upper-case letters equals the number of lowercase letters, and `false` otherwise. Use the following `main()`: › Sol

```
int main()                                                C++
{
    bool lowUpBalanced();

    if (lowUpBalanced())
        cout << "Equal number of lower and upper chars" << endl;
    else
        cout << "No equal number of lower and upper chars" << endl;
    return 0;
}
```

4 Two numbers are amicable when the sum of the proper divisors of each is equal to the other. Build the function `areAmicable()` that receives two numbers and returns whether they are amicable. From `main()` call a procedure `testAmicables()` that receives two numbers and prints the result. Also print all pairs of amicable numbers less than `3000`. › Sol

```
int main()                                                C++
{
    void amicLessThan(int n);
    void printRes(int a, int b);

    printRes(284, 220);
    printRes(121, 212);

    amicLessThan(3000);
    return 0;
}
```

```
void amicLessThan(int n)
{
    bool amicable(int a, int b);
    int addDivisors(int n);
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j < i; ++j)
            if (amicable(i, j))
                cout << i << " " << j << endl;
}

void printRes(int a, int b)
{
    bool amicable(int a, int b);

    cout << "The numbers " << a << " and " << b <<
     (amicable(a, b)? " ARE ": " are NOT ") << "amicable" << endl;
}
```

5 Build the program `t08e05.difgreatests.cpp`. Read integers from the user until `0` is entered; `0` is not part of the sequence. Print the difference between the largest and the second-largest values. Assume at least two numbers are entered before `0`. › Sol

```
-3 4 -22 0          -> 7                                    example
1 2 0               -> 1
2 2 0               -> 0
-3 5 -3 8 6 9 0     -> 1
```

6 Build the program `t08e06.summing.cpp` *(no subprograms needed)*. Read integers from the user one at a time, accumulating their sum. Stop as soon as the running sum exceeds `max = 10`. Do **not** include the number that pushes the sum over the limit. Print the final sum. › Sol

o O o

Part 4 Structured Data Types

Contents of this part Book contents

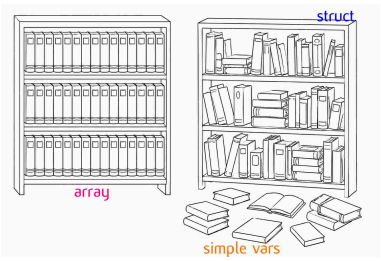
4·1 Parameter Passing with Structured Types 131
4·2 Arrays 132
▸ Task 9: Arrays 141
▸ Task 10: Arrays++ 147
4·3 Strings of characters 152
▸ Task 11: Strings 157
4·4 Structs or records 166
▸ Task 12: Structures 171
4·5 Matrices 176
▸ Task 13: Matrices 180
▸ Task 14: Final exams problems 186

So far we have only seen types of variable capable of holding a single piece of information — that is, whose content cannot be decomposed into pieces with a separate meaning of their own. Such types are also called *monolithic*.

Simple variables are enough to solve many algorithms, but computers exist for more than analysing the behaviour of a handful of values. The tasks that matter most usually involve collections of data: statistics over lists of complex records, or the processing of natural data — sound, weather readings, images — which often means handling huge numbers of individual data points.

Even modest programs that work with similarly-treated data, a few dozen values say, become cumbersome with single variables: you would need to repeat the same action, line after line, once for every single variable. That is exactly why data structures — compositions of simple variables, collections of data referred to by a single collective name — matter so much. And the most important of them all is the array.

98: Simple variables are independent and must be accessed individually, whereas structured data types can be manipulated as a whole through a single identifier



4·0.1 Templates

The kinds of structures we will use in this course — arrays, character strings, records, and arrays of arrays, or matrices — are all going to be provided by the C++ Standard Template Library (STL), rather than the classic C-style approach older courses might have used instead. Relying on STL means relying on a genuine C++ compiler rather than a plain C compiler, and it means using an aspect of the language that is more advanced still: instances of **templates**

Templates in C++ is a mechanism that enables you to write generic code —code that works with many different types without rewriting the same function for each one.

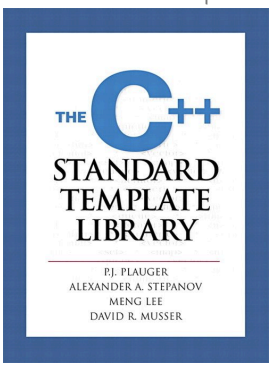
Function overloading lets you reuse one name, but you still have to write every body by hand, even though they are identical except for the type:

```
void mySwap(int& a, int& b)       { int t = a; a = b; b = t; }
void mySwap(float& a, float& b)   { float t = a; a = b; b = t; }
void mySwap(char& a, char& b)     { char t = a; a = b; b = t; }
```

```
void mySwap(string& a, string& b) { string t = a; a = b; b = t; }
```

and so on for every type you might need to swap. With templates, a single definition is enough:

99: STL: C++ Standard Template Library




```
template <typename T>
void mySwap(T& a, T& b) { T t = a; a = b; b = t; }
```

The compiler generates the version for each type you actually use.

After the declaration of the previous template, you can directly use

```
int a = 1, b = 2;
mySwap(a, b);         // T = int

string s = "x", t = "y";
mySwap(s, t);         // T = string
```

the compiler deduces the type and builds the necessary code for each call as our program makes, no fewer no more than those.

So we are going to rely on STL, which means relying on templates and classes too. But do not worry: we will only ever need to **use** two of them, the `string` class and the `array` class (itself built with templates) — and using them, rather than writing your own, keeps the syntax refreshingly simple, as we will see as soon as we look at `array` and `string` themselves in the sections that follow.

4·1 Parameter Passing with Structured Types

One important aspect we covered shortly after introducing simple types, in Section 2·2, was passing parameters — values and references — to subprograms (Section 3·2). That is exactly the first question we want to address here: does anything change once the parameters are structured rather than simple?

The main advantage of using templates for arrays and strings (strings are also templates; for normal characters, we do not need to use the template constructor, as it is already provided) is that they become **copyable**. This matters considerably to us, since we already know that passing parameters by copy is our default, safe method of passing them. Moreover, parameter passing is not the only context that benefits: returning an STL-based structure from a subprogram works the same way — you never need to think about the fact that it is a structure at all. It gets copied exactly as if it were a simple data type.

In classic C, arrays are declared in a completely different way — `int a[3];` — and they are not classes: they are not copyable, so passing or returning them would require an entirely different mechanism. Once passed to a subprogram, they **decay** into a bare pointer, losing all memory of their own size in the process. `array`, being a proper class, suffers from neither of these problems.

All STL-based types are classes, and classes — like the C `struct` they descend from — are copyable by default. This copyability ultimately derives from this property, and it is why all of them, `string` and `array` included, are copyable.

In summary, passing structured parameters will not change any of our habits: as long as we stick to STL-based structures, everything we learnt in Section 3·2 still applies exactly as it did for simple types.

Some programmers, keen to optimise whenever a structure might prove to be large — something we have no way of knowing and do not need to worry about in this course — pass structured data types as `const type& structuredParam` instead. This is not incorrect, and plenty of C++ literature actively encourages it; however, it complicates interfaces and does little to help you learn how to program in general. It draws you into the more intricate details of the C++ language, some of which change from version to version and do not carry over to other languages at all. Therefore, in our examples, we will not use this `const&` style of passing parameters.

4·2 Arrays

> Arrays are the most important data structures in computing because of their ability to handle large volumes of data efficiently and straightforwardly

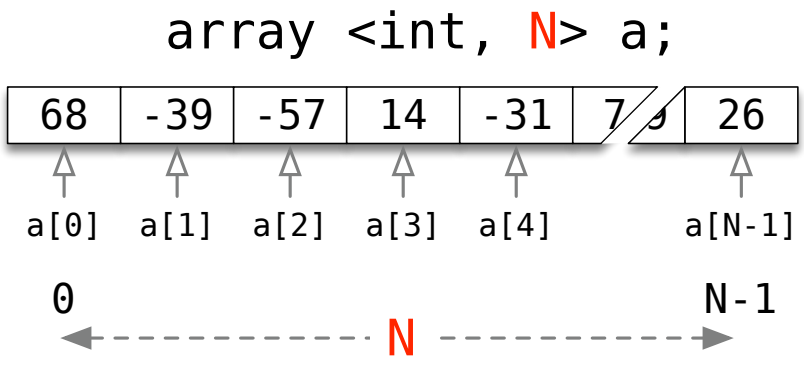
Arrays are the basis of character strings and are the single most important data structure in programming languages. In fact, we can think of a computer's RAM as one huge array of bytes — and, from the hardware's point of view, that is literally true. From now on we will define arrays and learn how to declare and handle them; afterwards we will extend what we have learned here to character strings (Section 2·9.1), since strings are nothing more than specialised arrays of characters. Finally, once structures have been introduced, we will come back to arrays to build matrices.

4·2.1 Definition. The Array Data Type

In general

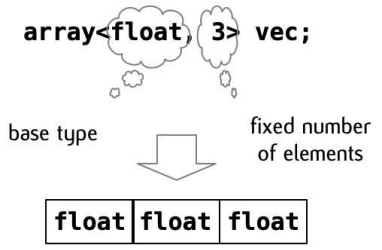
> An array can store and work with multiple values
> of the same data type,
> indexed by an integer index starting at `0`

> An array in C++ is the repetition of the same base type `N` times,
> where `N` is a constant known in advance:
> `array<baseType, N> arrayName;`



The fact that `N` is a constant directly implies that the array is a **static** structure: it cannot grow or shrink. The number of elements is something the programmer must decide at the moment the program is written. This is the harshest restriction of arrays — one that other STL types such as `vector` do not share — but it is precisely this fixed, known size that makes an array's syntax so convenient when it comes to building multi-dimensional arrays (matrices), something very important in programming.

100: The simple one-line declaration of an array involves creating many variables at the same time



Do not forget to `#include <array>`, the header where the `array` template is defined. If you forget it, the errors the compiler issues are baffling and hard to relate to the real problem — be careful with this.

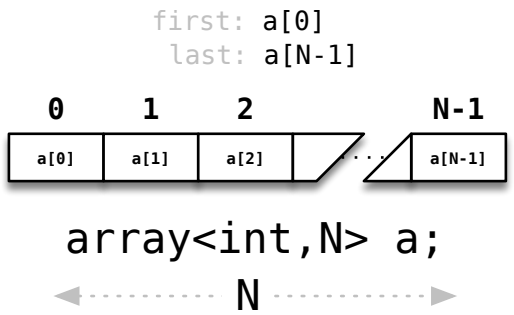
In C++, arrays are laid out in memory as a sequence of `baseType` variables, one right after the other, and each of them is referred to simply by its position — a position that starts at `0`. So, strictly speaking, it is not really an "index" that locates each variable, but the displacement (shift) from the origin of the array in memory.

The memory consumption of an array depends on its data type and the number `N` of elements it contains.

4·2.2 Access of array elements

Although an array has a single name, its `N` elements can be accessed and manipulated as individual variables. This is possible because each element is assigned a numerical index, known as a subscript. The subscript identifies a specific element within the array. The first element has subscript `0`, the second element has subscript `1`, and so on, with the last element having subscript `N-1`.

So, to access an element of an array we need both the array's name and a simple integer: an index (shift) that starts at `0` and ends at the number of elements minus `1` (`N-1`).



Each position is an **independent** variable:

```C++
#include <array>
int main()
{
  array<int, 10> a;
  a[0] = 33;
  cout << a[9] << endl; // will print garbage
  return 0;
}
```

To access **all** the elements — to traverse the array — `for` is the loop of choice, since we already know in advance exactly how many iterations we need (`N`). But `for` is only the right tool when we intend to go through

the whole array without stopping partway; if instead we might need to stop early — for instance, as soon as we find what we are looking for — `while` remains the correct choice, following the same `for`/`while` rule we already established.

```cpp
#include <array>
int main()
{
  array<int, 10> a;
  for (int i = 0; i < 10; ++i) {
      a[i] = i*i;
  }
  for (int i = 0; i < 10; ++i) {
      cout << a[i] << ", ";
  }
  cout << endl;
  return 0;
}
```

A word of warning about `[]`. The subscript operator `[]` does **not** check whether the index is in range. Since RAM really is one huge array, an out-of-range access such as `a[10]` on an `array<int, 10>` simply reads or writes whatever bytes happen to lie next to your array. This is **undefined behaviour**: the program may crash, print nonsense, or — worst of all — appear to work today and fail tomorrow. Keeping every index within `0 .. N-1` is entirely the programmer's responsibility.

Because of this, `std::array` also offers a **bounds-checked** access, `.at(i)`, which verifies the index and throws a `std::out_of_range` exception instead of silently corrupting memory. It is slightly slower than `[]`, but far safer while you are learning or debugging:

```
array<int, 10> a;
a[10]    = 5;   // undefined behaviour: no check at all
a.at(10) = 5;   // throws std::out_of_range
```

4·2.3 Array Operations

Arrays have their own companion operations. Array parameters require the formal and actual parameters to have exactly the same type: same element type and same size. You do not need to declare a named type for this, because every occurrence of `array<float, N>` denotes the same type. Still, defining an alias makes the code easier to read and to maintain, and lets us change the size in one place. For this reason, we will always declare arrays as new named types with `typedef` before using them:

```
const int N = 3;
const int NPER = 55;
typedef array<float, N> TVector;
typedef array<int, NPER> TAges;
```

But be careful: if you do not give any initial value to a local array, its contents are indeterminate (garbage), as with simple variables.

To send a whole array as a literal, we can build a temporary of the exact array type:

```
printVec(TVec{0,1,2,3,4});   // N must be 5 here
```

The number of values cannot exceed `N`; fewer values are allowed, and the rest are set to `0`.

Array sizes are compile-time constants. Best practice is to declare them as `const`ants, as in the examples above. We can also ask the array about its size with the member function `.size()`. Unlike a plain C array, `std::array` is a class, but it stores no extra data: the size is part of its type.

```C++
const int NPER = 55;
typedef  array<int, NPER> TAges;
int main()
{
  TAges ages;
  cout << ages.size() << endl; // -> 55
}
```

Using `N` or `v.size()` in the loop condition is equally correct — but note that `.size()` actually documents the code **less** than a well-chosen constant name such as `N` or `MAX_NUMBER` does, since the name of the constant tells the reader what the number means.

Comparison of arrays: You can not only assign one array to another (of the same type), you can also compare them:

```C++
int main()
{
  TVec a, b;
  a = b;
  bool eq = (a == b);
  bool ne = (a != b);
  bool gt = (a > b);
  ...
}
```

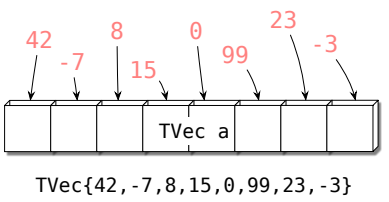
To understand the `>`, `<`, `<=` and `>=` comparisons, think of the way words are ordered in a dictionary (lexicographic order): the two arrays are compared cell by cell, starting at index `0`; as soon as one cell is found to differ (greater or less) from the corresponding cell in the other array, the comparison stops there and that difference decides the result.

4·2.3.1 Quick initialisation

Assigning initial values to simple variables at declaration has proven to be highly practical. Not only does it prevent accidental omission of critical initial values for accumulators, for instance, but it also enhances code clarity by coupling the assignment with the declaration, making the process more convenient and semantically meaningful.

But what about arrays? Usually an array holds many more elements than a simple initialisation can comfortably afford, so the right approach depends on the problem much more than it did for simple variables. On the other hand, it is precisely with arrays that quick initialisation is of the most use. Picture yourself debugging a program: on every run you would otherwise have to type in a long list of values by hand — most likely the very same values as the previous run. You could copy-paste them, but is that always possible or convenient? It quickly becomes tedious and is of no help at all when you are hunting for a bug.

101: Quick initialisation of arrays



How does C++ do quick initialisation with arrays? What do you think the next program will display?

```C++
#include <iostream>
```

```
#include <array>
using namespace std;

const int N = 5;
typedef array<int,N> TVec;

int main()
{
    void printVec(TVec v);

    TVec garb;
    printVec(garb); // random numbers

    TVec v =       {1,2,3};
    printVec(v); // 1 2 3 0 0

    printVec(TVec{0,1,2,3,4});
    //            0 1 2 3 4

    return 0;
}

void printVec(TVec v)
{
  for (int i = 0; i < N; ++i)
      cout << v[i] << " ";
  cout << endl;
}
```

The answer is that, an array being an *aggregate*, if you supply fewer values than the array can hold, the standard says every remaining element is copy-initialised from an empty list `{}` — that is, value-initialised. For `int`, value-initialisation means zero-initialisation, so those elements become `0`. In short: the values you write are assigned in order, and the rest are set to `0`.

But, be careful, if you do not give any initial value, the contents of the array is, as usually with simple variables, garbage, random memory left there by other programs.

So to initialise a long vector with all zeros, it is enough:

```
TVec v = {};  // all the elements to 0
TVec w = {0}; // if you want to be more explicit
```

The "empty list" `{}` is filled with the **zero** of the base type: `0` for an array of `int`, `0.0` for `float`, and for `char` the null character. For example, `typedef array<char, N> TChars; TChars cs = {};` fills every element with '`0`', the character `NUL` (code `0`), **not** the space `' '` (code 32). For Booleans, an array of Booleans, it is `false`.

We often need to send literal values to subprograms. With simple types this is as easy as writing the literal itself (`someFunction(3)`), but to send a whole array as a literal we must tell the compiler the exact array type of that literal — that is, we must cast it:

```
printVec(TVec{0,1,2,3,4,5,6,7,8,9});
```

will send the literal array `{0,1,2,3,4,5,6,7,8,9}` and before that, for the compiler to accept it as an array of exactly the same type the formal parameter in `printVec(TVec)` is expecting, we have to "convert" it to `TVec`. We can use this to make quick tests of our subprograms; for example, to test a `TVec bubbleSort(TVec);` function we only need a few lines in `main()`:

```C++
int main()
{
  void printVec(TVec);
  TVec bubbleSort(TVec);

  printVec(bubbleSort(TVec{9, 5, 4, 3, 2, 7}));
  printVec(bubbleSort(TVec{2, 4, 5, 1, 3, 4}));
  return 0;
}
// ..
```

As a summary, quick initialisation is a very useful tool we will use extensively. You can set the initial values of a (not too long) array once and for all, using the following simple syntax:

```
1 typedef array<float,3> TVector;
2
3 TVector v1 = {3, 5, -2};    // quick init
4 TVector v2 = {{1,2}};       // first 2 initialised *
5 TVector v3 = {0}; // only the first is 0, the rest default to 0 too
```

In line 4 we used `{{double, curly, braces}}` that are the most general way of initialising an array: the outer braces initialise the `std::array` object itself, and the inner braces initialise the raw array it wraps internally.

`{single braces}` also work: the compiler infers the missing inner braces (*brace elision*). For arrays of arrays or `struct`s, it is clearer to write all the braces.

Imagine testing a program with `10` or `20` element arrays: each time you run it to test it, you would have to type/paste in every single number, probably the same values as the last time. Quick initialisation is recommended for these cases, and it is a great time-saver while debugging.

```C++
#include <iostream>
#include <array>
using namespace std;

const int N = 3;
typedef array<int,N> TVec;

int main()
{
  int sum(TVec);

  TVec a = {1, 2, 3};

  cout << sum(a) << endl;
  return 0;
}

int sum(TVec v)
{
  int s = 0;
  for (int i = 0; i < N; ++i)
    s += v[i];
  return s;
}
```

4·2.4 A trap: sizes are unsigned

The values returned by `size()` and `length()` are not `int`: they are *unsigned* numbers, which cannot be negative. This is a historical inheritance from C, and it causes a nasty surprise whenever an `unsigned` and an `int` meet in the same expression: the `int` is silently converted to `unsigned`, and a negative value turns into a huge positive one.

```
string s = "ab";                                                    C++
cout << s.length() - 3 << endl;   // prints 18446744073709551615, not
-1

int i = -1;
if (i < s.length())               // false!  -1 becomes a huge number
    cout << "-1 < 2" << endl;
```

Since C++20 there is also a free function `ssize(s)` that returns a signed value, but we will keep using `int(s.length())`, which works in every version of the language.

Our rule: convert the size to `int` as soon as you get it, so that `unsigned` values never take part in your arithmetic or comparisons. Either write the conversion explicitly:

```
for (int i = 0; i < int(s.length()); ++i)
```

or store the size in an `int` once and use that variable:

```
int n = s.length();                                                 C++
```

With this rule, `int(s.length()) - 3` is `-1` and `i < int(s.length())` is `true`, as you would expect. Some compilers can warn about these situations, but good habits should not depend on warnings.

4·2.5 Examples with Arrays

The best way to learn to use arrays is by building things with them. Given a type such as

```
const int N = 3;
typedef array<int,N> TVec;
```

a typical "toolbox" of subprograms to master would be:

```
void initArr(TVec& a, int value = 0);
void printArr(TVec a);
void readArr(TVec& a);
TVec readArr();
int  sumArr(TVec a);
bool isIn(TVec a, int x);
int  indexOf(TVec a, int x);
int  count(TVec a, int x);
bool anyReps(TVec a);
int  findPosFirstMax(TVec a);
```

A few of these signatures are worth a comment:

- `initArr` and the first form of `readArr` **modify** the array, so they take it by reference (`TVec&`); `printArr`, `sumArr`, `isIn`, `indexOf`, `count`, `anyReps` and `findPosFirstMax` only **read** the array, so it is passed by value.
- `readArr` is shown in two different styles: one fills a reference parameter and returns nothing, the other builds a brand-new array and returns it (`TVec readArr()`). Both are correct and idiomatic; pick one style and stay consistent within the same program.
- `indexOf` and `findPosFirstMax` return a position as an `int`; by convention, a value such as `-1` is used to signal "not found".

As a worked example, here is `initArr`, which fills every cell with a given value, and `printArr`, which prints the elements in order, separated by spaces:

```C++
void initArr(TVec& a, int value = 0)
{
  for (int i = 0; i < N; ++i)
    a[i] = value;
}

void printArr(TVec a)
{
  for (int i = 0; i < N; ++i)
    cout << a[i] << " ";
  cout << endl;
}
```

A second example is worth studying closely, because it illustrates the `for`/`while` rule. `isIn` tells us whether a value `x` appears in the array. Since we can stop the moment we find it, the number of iterations is **not** known in advance — so the loop of choice is `while`, not `for`:

```C++
bool isIn(TVec a, int x)
{
  int i = 0;
  bool found = false;
  while (i < N and not found) {
    found = (a[i] == x);
    ++i;
  }
  return found;
}
```

Notice the disciplined style: a single `return`, no `break` to jump out of the loop, and the word operators `and`/`not`. Compare this with `sumArr`, which must **always** visit every cell in order to add it up: there the count of iterations is fixed at `N`, so `for` is the right choice. Choosing between `for` and `while` is not a matter of taste — it is dictated by whether the number of repetitions is known before the loop begins.

Try implementing the rest — `readArr`, `sumArr`, `isIn`, `indexOf`, `count`, `anyReps` and `findPosFirstMax` — on your own before moving on to the exercises.

Input/output of arrays: As you may have noticed, we have not discussed `cin >> a` (or `cout << a`) because these operations do not exist. Since arrays are programmer-defined types, `cin`/`cout` has no built-in mechanism to handle them. It is therefore the programmer's responsibility to implement custom routines for reading and writing arrays.

4·2.6 Checkpoint

1) What would be the contents of the array `cs` after `typedef array<char, 3> TChars; TChars cs = {'a'};`?
2) Given `typedef array<int, 5> TVec; TVec v = {7, 8};`, what does `v` contain? And what about a `TVec w;` declared with no initialiser at all?
3) For an `array<int, 10> a;`, what is the valid range of indices? At the level of memory, what happens when you write to `a[10]`?
4) What is the difference between `a[i]` and `a.at(i)`? When would you prefer each one?

5) Why must an array type be given a name with `typedef` before it can be passed to a subprogram? What goes wrong if two structurally identical arrays are declared without sharing a `typedef`?

6) `.size()` returns the number of elements of an array. Why might a named constant such as `N` document a loop condition better than `a.size()`, even though both are correct?

7) Explain the difference between the initialisers `{{3, 5, -2}}` and `{3, 5, -2}`. Why do the double braces exist?

8) How would you initialise an `array<float, 100>` so that every element is `0.0`, writing as little as possible?

9) Two arrays of the same type are compared with `a < b`. Describe, in your own words, the rule used to decide the result. Which everyday ordering does it resemble?

10) When traversing an array, when should you reach for `for` and when for `while`? Give a one-line example of a task that needs each.

11) Why do `cin >> a` and `cout << a` not work for an array `a`? Whose responsibility is it, then, to read and print arrays?

12) Of the toolbox functions, which take their array by reference (`TVec&`) and which by value (`TVec`)? What is the criterion that decides this?

13) Write the body of `int sumArr(TVec a)` and of `bool anyReps(TVec a)` (true if some value appears more than once). For each one, decide whether `for` or `while` is appropriate and justify your choice.

Task 9: Arrays

Throughout these exercises, assume the following setup:

```cpp
#include <array>
const int N = 3;
typedef array<int, N> TVec;
```

although the specific types and sizes may vary per exercise. We use *position* as a synonym for *index*: the first position is `0` and the last is `N-1` (equivalently, `a.size()-1`).

To avoid retyping test data on every run, we initialise arrays with brace-initialised literals directly in `main()` and pass them to subprograms as arguments. Exercise 1 is the exception: it requires reading from and writing to the terminal.

1 Write a `main()` program that declares the following subprograms, tests each one with appropriate `TVec` arguments (using brace-initialised literals where possible), and prints any returned values: › Sol

```cpp
void initArr0(TVec& a);       // set every element to 0
void printArr(TVec a);        // print all elements separated by spaces
void readArr(TVec& a);        // read N values from the keyboard (procedure)
TVec readArr();               // read N values from the keyboard (function)
int  sumArr(TVec a);          // return the sum of all elements
bool isIn(TVec a, int x);     // return true if x appears in a
int  indexOf(TVec a, int x);  // return the index of the first occurrence of x
int  count(TVec a, int x);    // count how many times x appears
bool anyReps(TVec a);         // return true if any element is repeated
int  findPosFirstMax(TVec a); // return the index of the (first) maximum
```

2 Using `TVec` from exercise 1, build a procedure `void printRev(TVec a)` that prints the array in reverse order. For example: › Sol

```cpp
int main()
{
    void printRev(TVec a);

    printRev(TVec{{1, 20, 3, 40, 5}}); // 5 40 3 20 1

    cout << endl; // necessary to flush data on the screen
    return 0;
}
```

You can also test it with `printRev(readArr())`: read an array and immediately pass it to `printRev()`.

3 Define a new type `THeights` as an array of `N = 5` floats. Declare a variable `alumHeights`, initialise it with five values (e.g. `1.83 1.78 1.55 1.90 1.60`), and build the function `float mean(THeights a)` that returns their average. Check that the example gives `1.732`. › Sol

4 Using `TVec` with `float` elements, build the function `float scalarProd(TVec a, TVec b)` that returns the scalar (dot) product of two vectors. The scalar product is $\sum_{i=0}^{N-1} a_i \cdot b_i$. › Sol

```cpp
int main()
{
```

```
    void printArr(TVec a);
    float scalarProd(TVec a, TVec b);

    TVec a = {{1, 2, 3}},
         b = {{3, 1, 1}};
    printArr(a); cout << endl; // 1 2 3
    printArr(b); cout << endl; // 3 1 1
    cout << "Scalar product: "
         << scalarProd(a, b)   // 8
         << endl;
    return 0;
}
```

5 Build a function `int numberOfNeg(TVec a)` that returns the number of negative elements in `a`. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    int numberOfNeg(TVec a);

    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    cout << "Negatives: " << numberOfNeg(a) << endl; // 1

    a = TVec{{0, 1, 0}};
    printArr(a); cout << endl;
    cout << "Negatives: " << numberOfNeg(a) << endl; // 0

    return 0;
}
```

6 Build a function `float minVec(TVec a)` that returns the smallest value in `a`. › Sol

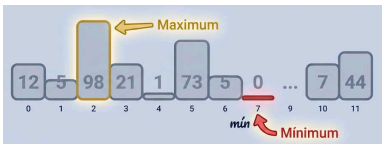
```
int main()                                                    C++
{
    void printArr(TVec a);
    float minVec(TVec a);

    TVec a = {{1, -2, 10}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -2

    a = TVec{{-10, -1, -20}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -20

    a = TVec{{4, -10, -2}};
    printArr(a); cout << endl;
    cout << "Min: " << minVec(a) << endl; // -10

    return 0;
}
```



7 Build a procedure `void minMaxVec(TVec a, int& mn, int& mx)` that returns both the minimum and maximum of `a` through reference parameters. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    void minMaxVec(TVec a, int& mn, int& mx);

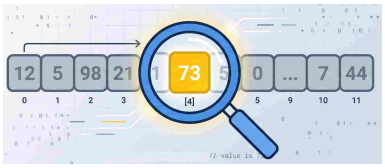
    int mn, mx;
    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    minMaxVec(a, mn, mx);
```

```
    cout << "Min: " << mn << endl; // -22
    cout << "Max: " << mx << endl; // 1

    a = TVec{{0, 1, 0}};
    printArr(a); cout << endl;
    minMaxVec(a, mn, mx);
    cout << "Min: " << mn << endl; // 0
    cout << "Max: " << mx << endl; // 1

    return 0;
}
```

8 **(Find.)** Build a function `int findArr(TVec a, int x)` that returns the **first position** at which `x` appears in `a`, or -1 if it is not found. › Sol




```
int main()                                                    C++
{
    void printArr(TVec a);
    int findArr(TVec a, int x);

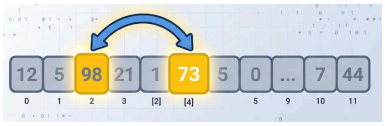
    TVec a = {{1, -22, 0}};
    printArr(a); cout << endl;
    cout << "find 1: "  << findArr(a,  1) << endl; // 0

    a = TVec{{0, 2, 10}};
    printArr(a); cout << endl;
    cout << "find 10: " << findArr(a, 10) << endl; // 2

    return 0;
}
```

9 Using `void swap(int& a, int& b)` — which exchanges the **contents** of `a` and `b` — build a procedure `void invertArr(TVec& a)` that reverses the elements of `a` in place.

HINT: You only need to swap the first half of the array with the second half; be careful not to swap elements twice. › Sol




```
int main()                                                    C++
{
    void printArr(TVec a);
    void invertArr(TVec& a);

    TVec a = {{1, 2, 3, 4, 5}};
    printArr(a); cout << endl;
    invertArr(a);
    cout << "after invert:" << endl;
    printArr(a); cout << endl; // 5 4 3 2 1

    return 0;
}
```

10 Build a function `int find12(TVec a)` that returns the first index at which the pair `1, 2` appears consecutively in `a`, or -1 if it does not appear. For example, in `{-2, 3, 1, 2, 0, 1, 2}` it returns `2`.

HINT: Be careful not to read past the last element of the array. › Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    int find12(TVec a);

    TVec a = {{3, 1, 2, 3, 4}};
    printArr(a); cout << endl;
    cout << "1,2 at: " << find12(a) << endl; // 1
```

```
    a = TVec{{3, 2, 2, 2, 1}};
    printArr(a); cout << endl;
    cout << "1,2 at: " << find12(a) << endl; // -1

    return 0;
}
```

11 Represent a 3-D point as `typedef array<float, 3> TPos`. Given two points `r1` and `r2`, build a function `float distance(TPos r1, TPos r2)` that returns the Euclidean distance between them. You will need `sqrt()` from `<cmath>`. › Sol

```
int main()                                                    C++
{
    float distance(TPos r1, TPos r2);

    TPos a = {{0, 0, 1.5}},
         b = {{0, 1.3, 0}};
    cout << "Distance: " << distance(a, b) << endl; // 1.98494

    return 0;
}
```

12 Using a `TVec` of `N` floats, build:

```
void stats(TVec a, float& mean, float& mn, float& mx, float& stdev);
```

that computes the mean, minimum, maximum, and corrected sample standard deviation of `a`:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} (a_i - \overline{a})^2}{N-1}}$$

where $\overline{a}$ is the mean. That formula requires two passes over the array (one for the mean, one for the squared differences). Try to implement it that way first; then try the algebraically equivalent single-pass formula:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} a_i^2 - \left(\sum_{i=0}^{N-1} a_i\right)^2 / N}{N-1}}$$

› Sol

```
int main()                                                    C++
{
    void printArr(TVec a);
    void stats(TVec a, float& mean, float& mn,
               float& mx, float& stdev);

    float mean, mn, mx, stdev;
    TVec a = {{1, 2, 3, 4, 5}};
    printArr(a); cout << endl;
    stats(a, mean, mn, mx, stdev);
    cout << "mean:    " << mean  << endl; // 3
    cout << "min:     " << mn    << endl; // 1
    cout << "max:     " << mx    << endl; // 5
    cout << "std dev: " << stdev << endl; // 1.58114

    return 0;
}
```

13 Using the procedure from exercise 12, compute the mean, minimum, maximum, and σ of the annual mean temperatures in Málaga from 1906 to 2025 (data from weatherandclimate.eu, column "año"). Set `N = 120` and use the data below. Then call `fitLine()` (provided) to fit a straight line $y = ax + b$ to the data, where $x = 1$ is the year 1906, and report the slope, intercept, and RMS error. › Sol

```C++
int main()
{
    void stats(TVec a, float& mean, float& mn,
               float& mx, float& stdev);
    void fitLine(TVec y, float& a, float& b, float& rms);
    void printArr(TVec a);

    TVec means = {{
        16.5, 17.1, 17.7, 17.0, 17.1, 17.4, 17.6, 17.3, 17.5, 17.8,
        17.4, 17.1, 17.3, 17.3, 18.1, 17.5, 17.6, 16.9, 17.7, 17.2,
        18.0, 18.2, 17.6, 18.0, 17.8, 18.2, 17.2, 17.7, 17.2, 17.5,
        17.3, 18.0, 17.4, 17.4, 17.7, 17.3, 17.3, 17.5, 17.2, 18.1,
        17.5, 18.2, 18.1, 18.6, 18.4, 17.7, 18.3, 17.9, 17.5, 18.7,
        16.8, 17.5, 18.0, 18.0, 17.8, 17.9, 18.2, 17.9, 18.0, 18.0,
        18.2, 17.7, 17.8, 17.3, 17.6, 17.1, 17.0, 17.2, 17.4, 17.4,
        17.3, 17.7, 17.8, 17.8, 17.8, 18.0, 17.8, 18.3, 17.6, 18.1,
        18.0, 18.5, 18.3, 18.6, 18.6, 17.8, 17.9, 17.6, 18.5, 19.2,
        18.0, 18.9, 18.5, 18.6, 18.3, 19.3, 19.2, 19.0, 18.7, 18.6,
        19.0, 19.0, 19.0, 19.2, 18.9, 19.3, 19.0, 19.1, 19.9, 19.7,
        19.6, 19.3, 19.0, 19.6, 19.5, 19.3, 20.0, 20.1, 20.0, 19.8
    }};
    printArr(means); cout << endl;

    float mean, mn, mx, stdev;
    stats(means, mean, mn, mx, stdev);
    cout << "mean:    " << mean  << endl; // 18.0833
    cout << "min:     " << mn    << endl; // 16.5
    cout << "max:     " << mx    << endl; // 20.1
    cout << "std dev: " << stdev << endl; // 0.799303 (changes with
    method)

    float a, b, rms;
    fitLine(means, a, b, rms);
    cout << "a = " << a << " degC/year" << endl; // 0.0179966
    cout << "b = " << b << " degC"      << endl; // 16.9945
    cout << "RMS = " << rms << " degC"  << endl; // 0.495051

    return 0;
}
```

Use the following helper to fit the line:

```C++
void fitLine(TVec y, float& a, float& b, float& rms)
{
    float sx = 0, sy = 0, sxy = 0, sx2 = 0;
    for (int i = 0; i < N; ++i) {
        float x = i + 1;
        sx  += x;
        sy  += y[i];
        sxy += x * y[i];
        sx2 += x * x;
    }
    float meanx = sx / N, meany = sy / N;
    a = (sxy - sx * meany) / (sx2 - sx * meanx);
    b = meany - a * meanx;

    float sumsq = 0;
    for (int i = 0; i < N; ++i) {
        float x = i + 1;
        float diff = y[i] - (a * x + b);
        sumsq += diff * diff;
    }
```

```
    rms = sqrt(sumsq / N);
}
```

The slope tells you how many degrees per year Málaga is warming.

14 Build a function `int sum2Biggest(TVec a)` that returns the sum of the two largest elements of `a`. › Sol

```
int main()                                                    C++
{
    int sum2Biggest(TVec a);

    cout << sum2Biggest(TVec{{1, 2, 3, 4, 5}}) << endl; //  9
    cout << sum2Biggest(TVec{{1, 5, 3, 4, 5}}) << endl; // 10

    return 0;
}
```

o O o

Task 10: Arrays++

Throughout these exercises, assume the following setup:

```cpp
#include <array>
const int N = 5;  // for example
typedef array<int, N> TVec;
```

although the specific types and sizes may vary per exercise. We use *position* as a synonym for *index*: the first position is `0` and the last is `N-1` (equivalently, `a.size()-1`). For many problems it is useful to have `void printArr(TVec a)` already implemented. Testing from `main()` is easiest by passing brace-initialised arrays directly as arguments.

1 Build a subprogram that receives a `TVec` and a value `x`, and returns the **last** position at which `x` appears. If the value is not found, return -1: › Sol

```cpp
int rfind(TVec a, int x);
```

For example,
`rfind( TVec{ 3, 0, -10, 80, 8}, 3)` should return `0`
`rfind( TVec{-1, 3,  3,  3, 3}, 2)` should return `-1`

2 Suppose `TVec` may contain repeated elements. Define the **distance** between two equal elements as the difference between their positions. For example, in `3 4 3 5` the two `3`s are at distance 2. Build a subprogram that finds the **largest distance between repetitions** of its second parameter. Return -1 if the value is absent; return 0 if it appears only once: › Sol

```cpp
int maxDistBetwReps(TVec a, int x);
int main()
{
    int maxDistBetwReps(TVec a, int x);

    cout << maxDistBetwReps(TVec{{10,10,4,3,5,6}}, 10) << endl; //  1
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,3}},  3) << endl; //  4
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,6}},  1) << endl; //  0
    cout << maxDistBetwReps(TVec{{ 1, 3,4,3,5,6}}, 33) << endl; // -1

    return 0;
}
```

3 Build a subprogram that returns the first position at which a repeated element appears. If no element is repeated, return -1: › Sol

```cpp
int firstRep(TVec a);
```

For example:
- `firstRep( TVec{1, 2, 3, 2} )` ⟶ `1`
- `firstRep( TVec{1, 2, 3, 4} )` ⟶ `-1`
- `firstRep( TVec{1, 1, 2, 3} )` ⟶ `0`

4 Build a subprogram that finds the **first peak** of an array. A peak is an element strictly greater than both its neighbours ($a_{i-1} < a_i > a_{i+1}$). The first and last elements are peaks if they are strictly greater than their single neighbour. Return the position of the peak, or -1 if none exists: › Sol

```cpp
int firstPeak(TVec a);
```

- `firstPeak(TVec{{3,2,9,7}})` ⟶ `0`
- `firstPeak(TVec{{3,3,3,3}})` ⟶ `-1`
- `firstPeak(TVec{{3,3,3,4}})` ⟶ `3`

- `firstPeak(TVec{{3,3,5,4}})` ⟶ `2`

5 🧊 Determine whether two arrays are **circularly equal**: one is a rotation of the other. Assume neither array contains duplicate elements. › Sol

```
bool circularlyEq(TVec a, TVec b);
```

For example,
- `{{1,2,3,4,5}}` and `{{5,1,2,3,4}}` are circularly equal, whereas
- `{{1,2,3,4,5}}` and `{{1,2,3,5,4}}` are not.

🧊🧊 A more challenging variant would involve arrays that contain repeated elements: `eqCircWithReps()`

6 🧊 Change `N` to 16 and implement a subprogram that returns the length of the longest run of consecutive equal elements: `int lcs(TVec a);` › Sol

A *run* is a maximal subsequence of identical elements.

- `lcs( TVec{3,9,9,0,9,9,9,9,0, 9, 3, 9, 9, 9, 9, 1} )` ⟶ `4`
- `lcs( TVec{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16} )` ⟶ `1`

Also implement: `int lcs(string s)` (using exactly the same algorithm).

- `lcs("aabbcccaaabdddddd")` ⟶ `5`

If no element is repeated, return `1`.

7 🧊 Build a subprogram that finds the **trend** of an array: the element that appears strictly more often than any other. If two or more elements tie for the highest frequency, there is no trend:

```
void trend(TVec a, int& tr, bool& ok);
```

Use `ok` to signal whether a trend exists. › Sol

```
2 5 5 2 5 2 5 5 trend: 5                                    example
1 1 1 1 1 1 1 1 trend: 1
2 1 2 2 1 1 2 1 has NOT a trend
```

8 Suppose we are given the floating-point coordinates (x_i, y_i) of a cloud of points, and we want to find the best-fit line $y = ax + b$ — the line closest to all points in the least-squares sense. The slope and intercept are given by: › Sol

$$a = \frac{n\sum_{i=0}^{n-1} x_i y_i - \sum_{i=0}^{n-1} x_i \sum_{i=0}^{n-1} y_i}{n\sum_{i=0}^{n-1} x_i^2 - \left(\sum_{i=0}^{n-1} x_i\right)^2}$$

$$b = \frac{\sum_{i=0}^{n-1} y_i - a\sum_{i=0}^{n-1} x_i}{n}$$

Since b depends on a, a must be computed first. The function returns two results, a and b, so the interface uses reference parameters:

```
void linearInterpolation(TVec x, TVec y, float& a, float& b);
```

Here `TVec` is an array of `float`. Call it from `main()` with:

```
linearInterpolation({{0,1,3,5,7}}, {{1,0,2,4,6}}, a, b);
```

and verify that $a \approx 0.804878$ and $b \approx 0.024390$.

9 **Purge.** In computing, *purging* means removing duplicates from a list. The old VAX/VMS operating system offered a `PURGE` command that deleted older versions of files (which were numbered with a suffix such as `name.ext;122`). Build both a procedure and a function that purge a `TVec` by keeping only the **first** occurrence of each element and removing any later duplicates: › Sol

```
void purge(TVec& v);
TVec purge(TVec v);
```

The procedural version must work in-place, without copying the array or using extra storage. It may help to first implement:

```
void deletePos(TVec& v, int pos);
```

Because the resulting array is shorter than the original, fill the unused trailing positions with `NUL=0`. (This works as long as the array never legitimately contains zeros.)

10 **Throwing Darts.** Write a function that accepts an array of throw radii (`N = 3` integers) and returns a total score according to these rules: › Sol

- `0` points — radius `>` `10`
- `5` points — `5` `<=` radius `<=` `10`
- `10` points — radius `<` `5`
- BONUS: if every radius is `<` `5`, add `100` extra points

```
scoreThrows(TVec{{ 1,  5, 11}}) // returns  15          example
scoreThrows(TVec{{15, 20, 30}}) // returns   0
scoreThrows(TVec{{ 1,  2,  3}}) // returns 130
```

11 Write a function that accepts an array and an integer `n`, and returns the number of index pairs (i, j) with $i < j$ such that $|a_i - a_j| = n$: › Sol

```
int pairsDiff(TVec a, int n);
```

```
pairsDiff(TVec{{1,1,5,6,9}}, 4);
    // 3: pairs (0,2), (1,2), (2,4)
pairsDiff(TVec{{1,1,3,3,9}}, 2);
    // 4: pairs (0,2), (0,3), (1,2), (1,3)
```

12 **Closest sum.** Given an unsorted array of integers and a target value, find the pair of elements whose sum is closest to the target. Return the two values through reference parameters: › Sol

```
void closestSum(TVec ns, int goal, int& a, int& b);
```

(This is a classic problem from *Programming Pearls* by Jon Bentley.) Also write a helper `void test(TVec a, int goal)` that prints:

```
goal:  4;  array:  9 -3 -10 -4 -5   ->  9 + -5          Display
goal:  0;  array:  1 -1  -2 -4  0   ->  1 + -1
goal:  6;  array:  2  2 -10 10  5   ->  2 +  5
```

when called from `main()` with:

```
test(TVec{{ 9, -3, -10, -4, -5}}, 4);                   C++
test(TVec{{ 1, -1,  -2, -4,  0}}, 0);
test(TVec{{ 2,  2, -10, 10,  5}}, 6);
```

13 **Midpoint sum.** Given an array of integers, return the index of the element for which the sum of all elements to its left equals the sum of all elements to its right. Return -1 if no such element exists: › Sol

```
int midpointSum(TVec a);
```

HINT: The element at the returned index is not included in either sum. The first and last indices cannot be midpoints.

```
midpointSum(TVec{{ 4, 1, 7, 9,  3, 9}}) // 3  (4+1+7=12, 3+9=12)   C++
midpointSum(TVec{{ 4, 4, 4, 4,  4, 4}}) // -1
midpointSum(TVec{{16, 4, 4, 4,  4, 4}}) // 1
```

14 **Manhattan distance.** The Manhattan distance between two grid points is the total number of horizontal and vertical steps needed to travel from one to the other (named after the grid-like street layout of Manhattan, New York). Write: › Sol

```
int manhattanDistance(TCoord a, TCoord b);
```

where `typedef array<int,2> TCoord` holds (x, y) coordinates.

```
manhattanDistance([1,1], [1,1]) // 0                    example
manhattanDistance([5,4], [3,2]) // 4
manhattanDistance([1,1], [0,3]) // 3
```

15 **Parity outlier.** An array contains integers that are all odd or all even, except for exactly one outlier. Return the **position** of that outlier. If more than one element has a different parity from the rest, return -1: › Sol

```
int parityOutlier(TVec a);
```

```
parityOutlier({{  0, 100,  4, 11}}) // [3] -> 11        example
parityOutlier({{160,  19, 11, 13}}) // [0] -> 160
parityOutlier({{  1,   1,  2,  1}}) // [2] -> 2
parityOutlier({{  1,   1,  2,  2}}) // -1  (no single outlier)
```

16 Fill an array of `100` floats with random values using: X

```
void randFloatArr(TVec& a, float min, float max)        C++
{
    for (int i = 0; i < int(a.size()); ++i)
        a[i] = (float(rand()) / RAND_MAX) * (max - min) + min;
}
```

Then print the array and its average.

17 Using `randFloatArr()` from previous exercise, fill a random array, print it, sort it with the procedure below, and print the sorted result: X

```C++
void bubbleSort(TVec& a)
{
    for (int top = int(a.size()) - 1; top >= 1; top--)
        for (int i = 0; i < top; ++i)
            if (a[i] > a[i + 1])
                swap(a[i], a[i + 1]);
}
```

o O o

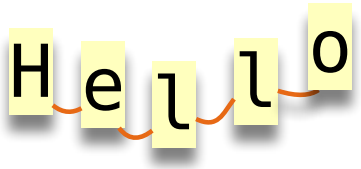
4·3 Strings of characters

Character strings enable computers to process text and communicate with users

In written or spoken form, text remains the only reliable medium for conveying abstract concepts. Consequently, it continues to be the primary foundation of information systems, despite substantial efforts to incorporate alternative media such as graphical interfaces, virtual reality systems, and animated synthesis. This fundamental role is further reinforced by the proliferation of knowledge databases—legal, commercial, and otherwise—deployed on the Internet, particularly through web services and, more recently, by the advent of Large Language Models (LLMs), whose natural working medium is text. Text remains far more efficient than images or other media for these purposes. Images, unless labelled and converted to strings of labels, are unsuitable for reasoning or logical precision. Mathematics relies on textual representations of symbols and expressions; natural language is inherently text-based; and programming, at any level—from high-level expressions to machine code—is fundamentally text.

Text in C++ is supported by encoded `char`s and `string`s of chars.

105: Strings of characters are fundamental to information processing and are present in every aspect of programming



Recall (Section 2·4) that a single character is referred to using single quotes:

```
char c = 'X';
```

However, to represent a word, a sentence, a paragraph, or any longer piece of text, we need more than one character — we need an array of characters. Although something like `array<char, 256> aVarString;` is possible, character strings are so fundamental to programming that C++ gives them their own dedicated type, `string`, instead of leaving us to manage a raw `array<char, N>` by hand. Compared with a plain array of `char`, a `string`:

- Has no fixed size limit — a `string` grows and shrinks dynamically, unlike an `array`, whose size `N` is fixed forever once declared.
- Comes with specialised input/output operations, with variants such as `cin >> s` and `getline(cin, s)` that read either a single word or a whole line at a time.
- Offers a whole family of specialised operations (searching, replacing, inserting, comparing…) optimised for working with text.

4·3.1 Definition. The string Data Type

Declaring a string of characters could not be simpler:

```
string name;
string address = "C/ Larios 6, Malaga";
name = "Juan";
```

106: String declaration is straightforward, with the initial value being empty. To read them, we use `cin` and `getline()`

```
string s;
string name = "Juan";
getline(cin, s);
if (name.length() > s.size()) {
```

Strictly speaking, `string` lives in the `<string>` header, and you should `#include` it explicitly. In practice, if you have already included `<iostream>`, `string` will often work anyway, because `<iostream>` happens to drag `<string>` in indirectly on most implementations — but this is not guaranteed by the standard.

Do not forget that string literals are always written between double quotes, and that it is perfectly fine to have an empty string, `s = "";`. In fact, you do not even need to worry about initialising a string to empty explicitly: by definition, a freshly declared `string` already starts out empty, so `string name;` alone already guarantees that `name` is `""`.

4·3.2 Operations on Strings of Characters

Unlike an `array`, whose size `N` is fixed forever, a `string` **can** grow and shrink — that is precisely its main advantage over a plain array of `char`. What you cannot do is grow or shrink it **through indexing**: `s[i]` only lets you read or overwrite a character that already exists within the current length of the string; it can never be used to add a character past the end, nor to remove one. To change the length of a string you must use one of its own operations — such as `+=`, `.append()`, `.insert()`, `.erase()` — or reassign it completely. For example:

```
string s = "cloud";
s += 's'; // now s contains "clouds" -- its length grew by one
```

You can freely overwrite an existing position, `s[0] = 'C'`, but you cannot use indexing to add a character beyond the current end of the string: ~~`s[5] = 's'`~~ is undefined behaviour if `s` only has 5 characters (positions `0`..`4`). The same applies in reverse: indexing alone can never shrink a string. Growing and shrinking are only possible through the string's own operations, such as the ones shown above.

The most important method (in classes, the internal subprograms are called **methods**) for strings is `.length()` — or, equivalently, `.size()`, since `string`, like the other STL container types, also supports this common interface — which tells us the single most important global property of a string: its number of characters. So:

```
int main()                                                    C++
{
  string s; // s.length() -> 0
}
```

So, to build a new string out of an existing one, you add characters one at a time and reassign the result to a (possibly new) variable:

```
string toLowercase(string s)                                  C++
{
  string r;
  for (int i = 0; i < int(s.size()); ++i)
    r += tolower(s[i]);
    // r.append(1,tolower(s[i])); // an equivalent alternative
  return r;
}
```

Note that `s.size()` returns an unsigned type; that is why we wrap it in `int(...)` in the loop above, as explained in "A trap: sizes are unsigned".

Input/output operations on strings come in two flavours:

```
string s;
cin >> s;              // reads only the next word (stops at the first
blank or newline)
cout << s << endl;

getline(cin, s);       // reads a whole line, until \n
getline(cin, s, '\t'); // reads until a tab character
```

There is a classic pitfall when the two are mixed. `cin >> s` stops as soon as it meets the first whitespace character — including the newline produced by pressing Enter — but it leaves that newline sitting unread in the input buffer. If a `getline` call comes right after, it will consume the rest of that **same** line — whatever is left after the word, which may even be empty — instead of waiting for a fresh line of input:

```cpp
#include <iostream>
using namespace std;

int main()
{
  string s;
  cin >> s;           // does not read "spaces"
  cout << s << endl;
  getline(cin, s);    // reads only what is left of the
                      // SAME line, not a new one!
  cout << s << endl;
  return 0;
}
```

For instance, if the user types `hello world` and presses Enter, `cin >> s` reads `"hello"`, and the `getline` right after reads `" world"` (note the leading space) — the rest of that same line — not the next line the user might type afterwards. Keep this in mind whenever you mix `>>` and `getline` on the same stream.

Interesting observation:

A `string` can change its length, but not through indexing: `s[i]` lets you modify a character that already exists, never add or remove one. To change the length you must use the string's own operations (`+=`, `.append()`, `.insert()`, `.erase()`...) or reassign it completely. Of these, `+=` is especially cheap: a `string` keeps some spare room (its *capacity*) beyond the characters it currently holds, so most appends simply write into that room at constant cost. Only when the capacity is exhausted does the string allocate a larger buffer and copy its contents. Hence, to add characters efficiently, use `+=` (or `.append()`), not `s = s + c`.

We measured both ways of appending one character `200,000` times:

```
s += c    ->   0 ms
s = s + c -> 875 ms
```

With `+=`, the capacity grows in jumps (15⟶30⟶60⟶120⟶240... in GCC compiler's library), roughly doubling each time it runs out, not one character at a time. This *amortised growth* makes `+=` $O(1)$ amortised per append. The exact growth factor depends on the implementation.

By contrast, building a string of n characters with `s = s + c` in a loop is $O(n^2)$: each iteration builds a brand-new temporary string of size $k+1$, copying all k existing characters, and then moves it into `s`. There is no spare room to reuse. The total work is:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

That is why `200,000` appends took 875 ms instead of about `0` ms. Doubling n makes the first approach take about twice as long, and the second about four times as long.

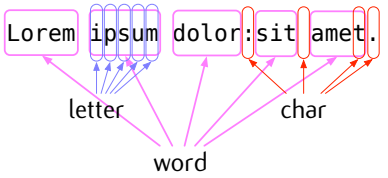
4·3.3 Examples with Strings of Characters

The string library offers many more useful operations. Several of them make for great practice exercises, and we will indeed use them that way. Here is a short tour through some of the most common ones, together with what each line actually prints:

```cpp
// ... (assume #include <iostream>,
// #include <string>, using namespace std;, and this inside main())
string s = "Juan";
string s2(s), s3 = s2;                          // () is like =
cout << s << endl;                              // Juan
cout << s + " Falgueras" << endl;               // Juan Falgueras
        s += " (end)";
cout << s << endl;                              // Juan (end)
cout << "s.empty():     "    << s.empty() << endl;  // s.empty():    0
cout << "s.length():    "   << s.length() << endl;  // s.length():   10
cout << "s.max_size(): " << s.max_size() << endl;  // a huge,
                                                   // implementation-defined
                                                   // number
cout << s.append(" bye") << endl;               // Juan (end) bye
cout << s.find("end", 2) << endl;               // 6
cout << "At 6: " << s.at(6) << endl;            // At 6: e
cout << "At 6: " << s[6] << endl;               // At 6: e
cout << s.replace(6, 3, "hello") << endl;       // Juan (hello) bye
cout << s.insert(10, " and bye") << endl;       // Juan (hell and byeo) bye
cout << s.erase(10, 4) << endl;                 // Juan (hell byeo) bye
cout << s.substr(6, 9) << endl;                 // hell byeo
```

Notice that `.append()`, `.replace()`, `.insert()` and `.erase()` all mutate `s` in place **and** return a reference to `s` itself — which is why chaining them directly inside `cout <<` works; `.find()` returns the position of a match (or a special "not found" value); `.at()` and `operator[]` return a reference to a single character; and `.substr()` builds and returns a brand-new string, leaving `s` untouched. Take the time to trace the position and length used in each call by hand against the printed result of the line before it — it is one of the best ways to get comfortable with zero-based indexing.

107: Any ASCII is a `char`; a letter is any uppercase or lowercase alphabetic symbol `'a'`-`'z'` a word is a sequence of letters. So, `chars` include punctuation marks such as commas and full stops, as well as whitespace characters: spaces, tabs, and newlines



4·3.4 Checkpoint

1) What is the output of the next fragment of code:

```cpp
string name = "Juan";
name += name;
cout << name << endl;
```

2) What is the content of a freshly created string: `string s;` — and what does `s.length()` return at that point?

3) Given `string s = "programming";`, what value does `s.length()` return, and which index positions are valid in `s[i]`?

4) Is the following a correct way to append a character to a string? Explain why or why not, and show the right way to do it.

```cpp
string s = "cat";
s[3] = 's';
```

5) When the index `i` is beyond the last character of `s`, how does `s.at(i)` behave differently from `s[i]`?

6) The user types `hello world` and presses Enter. What do `first` and `rest` contain afterwards, and why?

```
string first, rest;
cin >> first;
getline(cin, rest);
```

Task 11: Strings

Recent compilers do not require the inclusion of the `<string>` header because they include it in `iostream`, for the economy of syntax, we wont include it, but if you have any problems with your compiler and strings not being recognize, just `#include <string>`.

In these problems, we will make extensive use of the definition of **word**: any contiguous sequence of *lowercase* letters `['a'..'z']` (no digits, no spaces, no punctuation marks).

1 **Finding substrings.** Implement a subprogram that mimics the `string::find` method with the following interface:

```
int find(string s, string pat);
```

This function should return the index of the first occurrence of `pat` within `s`. If the pattern is not found, return -1.

HINT: Construct a single `while` loop that searches for the j-th character of the pattern (namely `pat[j]`) by incrementing `i`. Keep j at `0` unless `s[i+j]` equals `pat[j]`, in which case increment `j` whilst holding `i` fixed. When `j` reaches the length of the pattern, the pattern has been found; otherwise, reset `j` and increment `i`. › Sol

2 **Indexed substrings.** Build a subprogram to emulate `string::substr` with the interface below, but using indexes that can be negative (counting backwards from the end):

```
string substrneg(string s, int from, int end=-1);
```

If `from` or `end` are negative, count from the last character, where -1 is the last. If no third argument is supplied, it defaults to -1, meaning all characters from `from` to the end. The characters at both `from` and `end` are included. › Sol

```
tryThis("abcdefgh", 2);       //  "cdefgh"                    C++
tryThis("abcdefgh", -2);      //  "gh"
tryThis("abcdefgh", -1);      //  "h"
tryThis("abcdefgh", -3, -2); //  "fg"
tryThis("abcdefgh", 4, 4);    //  "e"
tryThis("abcdefgh", 4, -2);   //  "efg"
tryThis("abcdefgh", 4, 3);    //  ""
tryThis("abcdefgh", 40, 50); //  ""
```

3 **Palindromes.** Build a function

```
bool isPalindromic(string s);
```

that tells whether the string is a palindrome (reads the same forwards and backwards). **Ignore** case and everything that is not a letter (`'a'..'z'`, `'A'..'Z'`) when scanning. A good collection of examples is at palindromelist.net. › Sol

```
int main()                                                    C++
{
    void test(string s);

    test("");
    test("a");
    test("ab");
    test("aaab");
    test("A Santa dog lived as a devil God at NASA.");
    return 0;
```

```
}

void test(string s)
{
    bool isPalindromic(string s);

    if (isPalindromic(s))
        cout << s + ": IS palindromic" << endl;
    else
        cout << s + ": is NOT palindromic" << endl;
}
```

4 **To int.** Build a function:

```
int str2int(string s);
```

that analyses the string `s` containing a decimal number and returns its value as an integer. From `main()`, try with `"123"`, `"2021"`, `"01"` and print their integer values: `cout << str2int("123") << endl;` › Sol

HINT: Start with `result = 0` and scan the string, adding each character as the integer value it represents. For the char `'3'` we want the integer `3`, so compute `s[i] - '0'` (C++ allows subtracting chars directly, using their ASCII codes). Also note that each new digit shifts the previously accumulated value by 10 ×: `res = 10 * res + s[i] - '0';`

5 Extend `str2int` from the previous exercise to accept a second parameter for the base (e.g.
2 or 16). For bases greater than 10, the digits above 9 are represented by the letters A, B, C..., so `'A'` stands for 10, `'B'` for 11 and so on. For a letter, the value of the digit is `tolower(s[i]) - 'a' + 10`; for a decimal digit it is still `s[i] - '0'`. Each new digit then updates the result as `res = base * res + digit`. › Sol

Try with:

```C++
int main()
{
    int str2int(string s, int base=10);

    cout << str2int("121231")       << endl;  // 121231
    cout << str2int("1111",   2)    << endl;  // 15
    cout << str2int("77",     8)    << endl;  // 63
    cout << str2int("A",     16)    << endl;  // 10
    cout << str2int("FFFF",  16)    << endl;  // 65535
    return 0;
}
```

6 Write a function `int firstInt(string s)` that receives a string such as `"You owe me 10, doesn't you?"` and returns the first integer it contains. For that example the result is `10`. › Sol

Try with:

```C++
int main()
{
    int firstInt(string s);

    cout << firstInt("You owe me 10, doesn't you?") << endl; // 10
    cout << firstInt("No ten, but -12")             << endl; // 12
    return 0;
}
```

7 **Comparing.** Build a function:

```
int strcmp(string s1, string s2);
```

that compares `s1` and `s2`, returning

$$\begin{cases} -1 & \text{if } s_1 < s_2 \\ 0 & \text{if } s_1 = s_2 \\ +1 & \text{if } s_1 > s_2 \end{cases}$$

Note that *airplane* is less than *earth*, and *ex* is less than *example*. › Sol
Try with:

```
int main()                                                          C++
{
    int strcmp(string a, string b);

    cout << strcmp("",          "")          << endl; //  0
    cout << strcmp("a",         "b")         << endl; // -1
    cout << strcmp("b",         "a")         << endl; //  1
    cout << strcmp("a",         "a")         << endl; //  0
    cout << strcmp("avion",     "avioneta") << endl; // -1
    cout << strcmp("ave",       "avioneta") << endl; // -1
    cout << strcmp("avioneta", "ave")       << endl; //  1

    return 0;
}
```

You will need to compare both strings char by char simultaneously, checking that you stay within bounds:

```
while (i < int(a.length()) and i < int(b.length())
                      and a[i] == b[i])
    ++i;
```

8 **Counting words.** Build a function:

```
int noOfWords(string s);
```

that returns the number of words in a string. › Sol

Try with:

```
int main()                                                          C++
{
    void tryWith(string s);

    tryWith(" i   love   programming");        // 3
    tryWith("  some; (like) a2b");             // 4
    tryWith(" live on time, - emit no evil "); // 6
    return 0;
}
```

9 **Nth word.** Implement a function:

```
string wordn(string s, int n);
```

that returns the nth word contained within the input string. › Sol

```
int main()                                                          C++
{
    string wordn(string, int);

    cout << wordn("-i.   to programming", 1) << endl;    // i
```

```
    cout << wordn("i. -- (to) programming", 2) << endl;  // to
    cout << wordn("--- i. to programming  ", 4) << endl; //

    return 0;
}
```

10 Build a function

```
string capitalit(string s);
```

that returns the input string with each word capitalised. › Sol

```
int main()                                                    C++
{
    string capitalit(string s);

    cout << capitalit(" F.   of,   programming") << endl;
    // " F.   Of,   Programming"
    cout << capitalit("INDEPENDENTLY the case The Original Chars HaD")
    << endl;
    // "Independently The Case The Original Chars Had"
    return 0;
}
```

11 Write a procedure `coincidences()` that takes two input parameters: strings `str1` and `str2` containing only lowercase letters. The procedure should return two output values as parameters: the count of letters in `str2` that match letters at the same positions in `str1`, and the count of letters in `str2` that appear in `str1` but at different positions.

If a character appears multiple times in one string but only once in the other, it counts as 1. › Sol

For example:

```
int main()                                                    C++
{
    void printOutput(string pat, string s);

    printOutput("ARXMBA", "RZXMKJH"); // -> 2, 1
    printOutput("JAKZEF", "AOA");     // -> 0, 1
    printOutput("JAKZAF", "AA");      // -> 1, 1
    printOutput("EA", "AA");          // -> 1, 0
    return 0;
}
```

12 **Scrabbling.** Build a function:

```
bool scrabble(string s1, string s2);
```

that determines whether the two strings are permutations of each other —that is, they have the same length and each character appears the same number of times. For example, `"friendship"` and `"rifendihps"` are permutations of each other. A useful approach for checking whether two strings are permutations of each other is to count the frequency of each letter. Write:

```
TCharFreq freqs(string s);
```

that returns the frequency of each ASCII character in `s`, where `typedef array<int,256> TCharFreq;`. To test it, print each character in `s` followed by its frequency. For example, for `"Charles' suit"`:

```
   1 ' 1 C 1 a 1 e 1 h 1 i 1 l 1 r 1 s 2 t 1 u 1          Display
```

› Sol

13 An **isogram** is a word with no repeating letters, consecutive or non-consecutive. Implement a function that determines whether a string containing only letters is an isogram. Assume the empty string is an isogram. Ignore letter case. › Sol

```
isIsogram("Dermatoglyphics") == true                    Examples
isIsogram("aba")             == false
isIsogram("moOse")           == false  // ignore letter case
```

HINT: Use the solution of exercise 11 after downcasing the letters.

14 **Break camelCase.** Build a function that breaks camel-cased strings by inserting an underscore between the lowercased words. Assume every uppercase letter starts a new word. › Sol

```
int main()                                                                C++
{
    string breakCamelCase(string s);

    cout << breakCamelCase("camelCasing") << endl; // camel_casing
    cout << breakCamelCase("__A__aA_A_")  << endl; // ___a__a_a__a_
    cout << breakCamelCase("camelC")      << endl; // camel_c
    cout << breakCamelCase("camElC")      << endl; // cam_el_c
    cout << breakCamelCase("APPLE")       << endl; // _a_p_p_l_e
    return 0;
}
```

15 You have a restricted arithmetic expression in a string such as `"2+6-9-4"`, containing only integers, the operators `+` and `-`, and spaces. The following function evaluates it:

```
int express2int(string s)                                                 C++
{
    int r = 0;
    for (int i = 0, sign = 1; i < int(s.size()); ++i) {
        if (s[i] == '+')
            sign = 1;
        else if (s[i] == '-')
            sign = -1;
        else if (s[i] >= '0' and s[i] <= '9')
            r += sign * (s[i] - '0');
    }
    return r;
}
```

Extend the algorithm to support numbers with more than one digit and spaces.

HINT: Build each number as a temporary `int` until you reach a new sign; then use the temporary, reset it, and interpret the new sign. › Sol

Try with:

```
int main()                                                                C++
{
    void test1(string s);
    void todo(string s);

    test1("2+6-4");          // 2+6-4: 4
    test1("6-4");            // 6-4: 2
    test1("0-1+1");          // 0-1+1: 0
```

```
    todo("2+634-4");          // 2+634-4: 632
    todo("  634   -  4 ");    //   634   -  4 : 630
    todo(" 123 ");            // 123 : 123
    todo(" -123 - 1");        // -123 - 1: -124
    return 0;
}
```

16 🧊 **Build a string from an int.** Build

```
string int2str(int n);
```

that converts an integer to its string representation.

HINT: It is easy to get the last digit: compute the remainder modulo the base (default `10`), then add `'0'` to get the corresponding char. Divide by `10` and repeat until the quotient is `0`. The result is reversed, so reverse it before returning. › Sol

17 Build

```
string diff(string s, string t);
```

that subtracts the characters in `t` from those in `s`. For example: `diff("telephone", "phonograph")` is `"telee"`, and `diff("holograph", "dog")` is `"hloraph"`.

HINT: Use the `freqs` function from exercise 11. › Sol

18 🧊 There are two types of wildcard pattern matching: globbing (used in Unix file matching) and regular expressions. Globbing uses two metacharacters: `'*'` and `'?'`. Build:

```
int glob(string s, string pat);
```

interpreting `'?'` in `pat` as *any single character*, so `"exer01.cpp"` matches the pattern `"exer01.cpp"`.

HINT: Similar to exercise 7 (`strcmp`), but in the comparison check `pat[j] == '?' or s[i+j] == pat[j]`. › Sol

🧊🧊🧊 The complete solution (or nearly complete) includes `*` as a possible part of the pattern, meaning *zero or more* characters at that position in the string. For example in `glob("ejer01.cpp", "ejer*.cpp")` the `*` matches `01` › Sol

19 A simple way to extract fragments of a string separated by a given character is to call a function repeatedly until it returns an empty string. Build:

```
string nextPiece(string s, char sep='\0');
```

The second parameter is the separator. On the first call, supply the separator to initialise a `static char theSep` and reset the static running index to `0`. Subsequent calls without a separator resume from where the previous call left off. Static variables in functions preserve their values between calls. › Sol

```
nextPiece("*abc *cde**fg*", '*') // -> ""                    Examples
nextPiece("*abc *cde**fg*")      // -> "abc "
nextPiece("*abc *cde**fg*")      // -> "cde"
nextPiece("*abc *cde**fg*")      // -> ""
nextPiece("*abc *cde**fg*")      // -> "fg"
```

20 **Valid braces.** Write a function

```
bool validBraces(string s);
```

that takes a string with braces and determines whether their order is valid. Input strings may contain `(`, `)`, `[`, `]`, `{`, `}`. A string is valid if each closing brace closes its corresponding opening brace with no unmatched opener in between. For example, `"is (a ){valid}[strINg]"` and `"([{}])"` are valid, while `"not(}valid"` and `"[({})](]"` are not.

```
validBraces( "(){}[]" ) => true                              Examples
validBraces( "(}" )     => false
validBraces( "[(])" )   => false
validBraces( "([{}])" ) => true
```

HINT: Use a string as a stack. Each time you find an opening brace from `"([{"`, push it onto the end of the string. Each time you find a closing brace, it must correspond to the last opening brace pushed. › Sol

21 **EAN validation.** An International Article Number (EAN) is a 13-digit barcode: 12 digits of data followed by a single check digit. The check digit is computed from the 12-digit string as follows: multiply each digit at an odd position (1st, 3rd, ...) by 1 and each digit at an even position (2nd, 4th, ...) by 3, sum the results, then: `checksum = 10 - (sum % 10)` (or `0` if `sum % 10 == 0`).

For example, for `"400330101839"`:

```
4x1+0x3+0x1+3x3+3x1+0x3+1x1+0x3+1x1+8x3+3x1+9x3              Display
= 4+0+0+9+3+0+1+0+1+24+3+27 = 72
10 - (72 % 10) = 8  =>  EAN: 4003301018398
```

Build a function that validates a given EAN code, returning `true` if valid. Assume the input is always exactly 13 digits. › Sol

```
isValidEAN("4003301018398") => true                          Display
isValidEAN("4003301018392") => false
```

22 **RGB to hex conversion.** The function

```
string rgb2hex(TRGB a);
```

receives RGB decimal values and returns the corresponding hexadecimal string. Valid decimal values are $0–255$; any argument outside that range should be clamped to the nearest valid value.

HINT: Write a helper that converts an integer to its two-digit hexadecimal string, padding with a leading zero if necessary. › Sol

```
rgb2hex(TRGB{255, 255, 255}) // "FFFFFF"                    Examples
rgb2hex(TRGB{255, 255, 300}) // "FFFFFF"
rgb2hex(TRGB{  0,   0,   0}) // "000000"
rgb2hex(TRGB{148,   0, 211}) // "9400D3"
```

23 The **Hamming distance** of two equal-length strings is the number of positions at which they differ — the number of character substitutions required to transform one into the other. Write:

```
int hamming(string a, string b);
```

The strings contain only `0`s and `1`s. No need to validate the input. › Sol

```
hamming("100101", "101001") == 2                            Examples
hamming(  "1010",   "0101") == 4
```

24 **Patterns.** Write

```
bool wordPattern(string pattern, string str);
```

that checks whether `str` follows the same sequence as `pattern`. Each letter in the pattern stands for a distinct word. › Sol

```
wordPattern("abab", "truck car truck car");       // true       Examples
wordPattern("aaaa", "dog dog dog dog");           // true
wordPattern("abab", "apple banana banana apple"); // false
wordPattern("aaaa", "cat cat dog cat");           // false
```

25 **Look-and-say.** The look-and-say sequence (explained by Conway here) begins:

```
1                                                           example
11
21
1211
111221
312211
13112221
1113213211
```

Each element is built by *reading* the previous one: count each run of identical digits and say "count digit".

Build

```
string lookandsay(string seed);
```

Starting from `seed = "1"`, iterate and print the first `20` elements.

First build a helper `int countReps(string s, int pos)` that, starting at `pos`, counts how many times `s[pos]` repeats consecutively. Convert that count to a char (always `'1'`, `'2'`, or `'3'`) and concatenate it with the digit itself to form the next sequence. › Sol

26 **Roman numbers.** Build a function:

```
int rom2int(string s);
```

that parses the string `s` containing a Roman numeral and returns its integer value. Accept both uppercase and lowercase letters.

```
rom2int("iIi") // 3                                                    Display
rom2int("iv")  // 4
rom2int("liv") // 54
```

HINT: Reading the string backwards, you only need to subtract a digit's value when it is smaller than the previous one. › Sol

27 Build a function:

```
TNat sumNat(TNat a, TNat b);
```

where `typedef string TNat;` serves as a container of digits for arbitrarily large natural numbers.

```
int main()                                                             C++
{
    test("1", "1");
    test("122345", "98765");
    test("11112222333334444555566666777778888999",
         "11112222333334444555566666777778888999");
    return 0;
}
```

HINT: Reverse both numbers before adding, use a `carry` variable, then reverse the result before returning.

Using the ideas developed so far, build:

```
TNat multBy(TNat a, TNat b);
```

Check your solution with:

```
25389450228724393822272644606314242417158103448 43                    Display
* 98464993182266617687544411784683307548280902514 =
24999720436728450547751653056583602336163680792674621614491391110567144360637821957437760563530 2
```

Can you use your previous functions to build a short solution for:

```
TNat pow(TNat b, TNat exp);
```

› Sol

Check with:

```
25389450 ^ 9 =                                                         Display
438409178051912501457984911781933184948537741593458789062500000000 0
```

o O o

4·4 Structs or records

Packaging different types of variables under a single name allows a program to handle complex information hierarchically and enables it to model real-world data effectively

108: A `struct` is a container of arbitrary data, with fields accessed by their `.names` and without any implicit ordering



A `struct` or record is an aggregate of possibly different data types grouped under a single name. They complement what the array offers, in the sense that they can hold different types in the same collection; but they also have a limitation arrays do not have: their scalability and the way you access them are not like the array's — you declare each field one by one, rather than giving a single total count as with arrays, and accessing the elements, called *fields*, is done through their own names, so easy `for` loops with an index are no longer possible.

For a small number of components there is not a big difference between the two, as the "advantages" and "disadvantages" vanish and the only difference left is the way you access the cells/fields. With structures there is no obvious order for the fields in their use, and you call each field by its own name.

On the other hand, the syntax used to access elements goes well beyond structs; you can find it in places as remote as internet resource locators, which — although in reverse order — express `detailOfProtocol.machine.place`, as in www.uma.es where you access the Spanish domain, inside it the University of Málaga, and inside that, the web service (`www`). We will use the same syntactic style with records: from the container to the contents, using dots to mean "belongs to".

4·4.1 Definition. The struct Data Type

To define a `struct` we first need to decide the names and types of its fields. The novelty (and perhaps an inconsistency in C++) is that, instead of `typedef`, we use a new reserved word to create the type: `struct`:

```
struct TName {
    type1 field1;
    type2 field2;
    type3 field3;
    …
};
```

It is possible (in an old-fashioned style) to put the name of the new type after the fields, next to the closing curly brace — notice that with records we have inverted the usual order, giving the name of the new type first and its composition afterwards. There is a good reason for this: as mature programmers know well, a long list of details makes it harder to find the important part, in this case the name of the type. On the other hand — not for how we will use them here, but for an important mechanism C++ offers to chain blocks into flexible-length structures such as linked lists — placing the struct's name before its fields is far more convenient, since the fields can then refer back to it recursively through pointers.

An example:

```
1 #include <iostream>
```

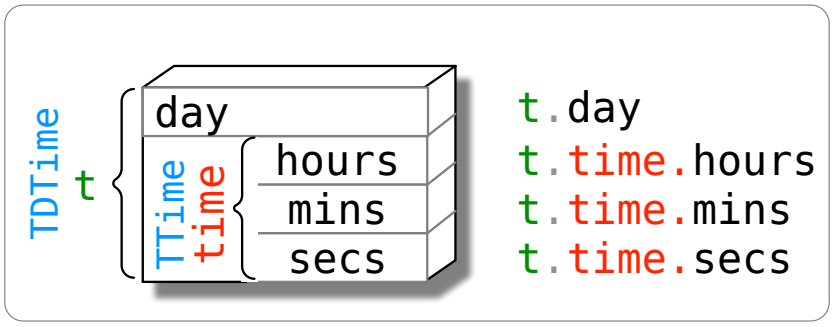
```
2  using namespace std;
3
4  // types
5  typedef string TMonth;
6
7  struct TDate {
8      int day;
9      TMonth month;
10     int year;
11 };
12
13 const TDate TODAY = {9,"Sept", 2026};
14
15 int main()
16 {
17     TDate birthD, currD;
18
19     birthD = TDate{1, "Jan", 2026};
20
21     cout << birthD.day << endl; // -> 1
22
23     return 0;
24 }
```

Initialisation of structs is straightforward: the only requirement is the order of the fields (line 13).

To build a value of a specific struct type on the fly, we prefix the braced list of values with the struct's name — a functional-style cast (line 19).

Accessing fields is easy: you only need to use the dot, the generic key: `.` (line 21)

Structs can be passed directly as value or reference parameters and returned in the same way. They behave exactly as expected of basic variables. In the following example:



The definitions are:

```
struct TTime {
    int hours;
    int mins;
    int secs;
};
struct TDTime {
    int day;
    TTime time;
};
```

The dot notation `.` following the name of the variable provides access to its members. It is like a key to enter into the box of the structure which it follows:

```
TDTime t;
t.day = 13;
t.time.hours = 2;
```

```
t.time.mins = 0;
t.time.secs = 20;
```

The contents of this variable would be: `{13,{2,0,20}}`. We could also quick-initialise it with:

```
TDTime t = {13,{2,0,20}};
```

Alternatively, if you are not initialising at creation time but need to assign a value later, you can construct a literal value by casting the contents:

```
TDTime t;
// ...
t = TDTime{13,{2,0,20}};
```

You can also assign default initial values to a struct's fields — something plain C structs could not do — which is quite useful:

```
struct TList {
    TVec v;
    int n = 0;  // all TList new vars will
                //  start with this value here
};
```

To access its elements:

```
TList l;
l.v[0] = 33;
++l.n; // our list has now one value
```

In fact, C++ classes are nothing more than extensions of struct definitions, to which C++ added encapsulation, generic programming, and behaviour through methods — whose syntax is already familiar to us: `a.length()`, just as when we access a field, except that these are methods, which can be private to the class.

4·4.2 Operations on Records

As arbitrary data types, the only operation you can perform on `struct`s is comparison, and only for equality or inequality.

As an example of use

```
struct TTime {                                                C++
    int hours, mins, secs;
};


int main()
{
    int toSecs(TTime t);

    TTime atime = {23, 30, 12};
    cout << "Secs: " << toSecs(atime) << endl;
    return 0;
}

int toSecs(TTime t)
{
    return t.hours * 3600 +
           t.mins * 60 +
           t.secs;
}
```

You could also have declared `array<int, 3> TTime;` and referred to the elements by `atime[0]`, etc. But the use of named fields makes the program much more readable and self-documenting.

With structs you can copy their contents as a whole to variables of the same type, but you cannot compare them, as you could with arrays. The reason is not hard to guess. You cannot compare `TRec a, b;` `a == b` or `a != b` or `a > b`, etc.

4·4.3 Examples with Records

In normal situations, complex data will require complex structures to represent it. You will typically find nested structures:

```C++
struct TDate {
    int day;
    TMonth month;
    int year;
};
struct TEmployee {
    int code;
    float salary;
    TDate joiningDate;
};
int main()
{
  TEmployee empl = {101, 2000, {19, "Aug", 1960}};
  empl.joiningDate.year++;
  printMonth(empl.joiningDate.month);
  //
}
```

Observe the chain of dots (lines 14, 15).

4·4.4 Lists or Sequences of Elements

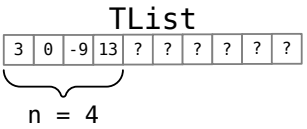
For us, an Open Array, or List, will be a container of an indefinite number of elements. We saw that arrays' most important limitation is that they are static — they cannot shrink or grow — so they are bad containers for variable-length amounts of information. Especially in scripting languages (JavaScript, Python, Ruby, etc.), arrays are inherently flexible in length (as well as in the type of their components), so this detailed construction would not be needed; but an easy way to work around C++'s fixed container size is to use an array together with an occupancy index. Sometimes this is not even necessary, since you can fill the array with an impossible (sentinel) value and treat cells holding it as empty — but that is not a generally valid mechanism, and it has disadvantages. The following approach is universally valid:

```C++
const int N = 100;
typedef array<int, N> TVector;

struct TOpenArr {
  TVector ar;
  int noElems = 0;
};
```

We have a container (unavoidably with a maximum capacity) but a flexible amount of data inside it. We can insert, delete, or append elements to this list; it can be treated as a simple store, but also as an ordered array of elements.

109: A list is a container. You can exploit its ordering properties, but it can also be used simply as storage with variable capacity




```
struct TList {
  int n = 0;
  TVec ar;
};
```

4·4.4.1 Checkpoint

1) Define a `struct TPoint` with `float x` and `float y`, and write the prototype of a function `distance(TPoint a, TPoint b)` returning the distance between two points.
2) Given `struct TBook { string title; string author; int year; float price; };`, write the declaration and initialisation of a constant `TBook` for a book of your choice.
3) Why can two `TBook` variables only be compared field by field “by hand”, and not with `==`?
4) Design a `struct TStudent` that nests a `TDate` (birth date) and a `TOpenArr` (from this section) holding the student’s exam marks.
5) What would go wrong if you tried to initialise a `struct`’s fields by giving a total count, the way you do with `array<int, N>`?

Task 12: Structures

1 Please respond to these short questions with short answers. You can build a `.cpp` file full of comments with the answers to each of them. X

1. Tell 2 differences between structures and arrays,
2. What is the advantage, if any, in declaring a `TComplex` as a `struct` instead of an `array`? `TComplex` is a structured variable containing the 2 real numbers that define a mathematical complex number,
3. What is the most important drawback of classical arrays/matrices (i.e. `int a[3];`, `float m[5][4];`, or non-STL arrays/matrices) that `struct` has never suffered from?,
4. What do you think was the syntactic inspiration for the Internet dot notation `https://www.uma.es/`? Is the order of the components the same as the order we use to access the fields in our `struct`s?,
5. Why do you think that, in general, structures cannot be compared with each other?

2 Given the next constants, types, and variables:

```C++
const int LLEN    = 50000;
const int SUBLLEN =    20;

typedef array<string,LLEN>    TList;
typedef array<string,SUBLLEN> TSublist;

enum TItemType {HARDW, SOFTW};
// Alternatively, we could declare:
//    typedef int TItemType;
//    const int HARDW = 0;
//    const int SOFTW = 1;

struct TRecList {
    TList list;
    TSublist sublist;
};

struct TCatalRec {
    string itemName;
    TItemType itemType;
    TRecList subcatal;
};

typedef array<TCatalRec,SUBLLEN> TCatal;

TCatal catal;
TCatalRec aCatal;
TRecList aList;
int count;
string aName;
```

Which of the following phrases are correct? (assume the variables already have adequate values): X

1. ```C++
   if (TCatalRec.itemType == HARDW)
       count++;
   ```
2. `catal[1].subcatal.list[2] = aCatal;`,
3. `catal[5].TRecList = aList;`,
4. `aCatal.itemName[2] = aList.list[2];`,
5. `catal[1].list.list[1][2] = aName[2];`,
6. `catal[19].list[1] = aName;`,
7. ```C++
   if (catal[19].itemType == SOFTW) {
   ```

```
    catal[10].TRecList.sublist[3][9] = 'A';
}
```

,

8. `aCatal = catal[5];`,
9. `aCatal.subcatal = aList;`,

3 We want to work with complex numbers

$$[\text{real-part}] + i\,[\text{imaginary-part}]$$

so we define the structure:

```C++
struct TComp {
    float re, im;
};
```

Write the types and subprograms necessary for the following `main()`:

```C++
int main()
{
    TComp w = {2, 1};
    TComp z = {4, 3};

    printComp(sumComp(z, w));
    printComp(subComp(z, w));
    printComp(mulComp(z, w));
    printComp(divComp(z, w));

    return 0;
}
```

If you do not remember the complex number operations, look them up on Wikipedia. › Sol

4 Define the most appropriate constants and types for `TPatient` with:

- An open identifier code (with numbers and letters).
- An array of integers with the `744` (24×31) blood-pressure values corresponding to every hour in a month.
- An array with a maximum of `1 000` real elements with the levels of a particular gene expression.
- The total number of gene expressions in the previous array (always smaller than `MAXGEN = 1000`).

Then, using this type: › Sol

1. Build a function that receives a `TPatient` and returns the maximum blood pressure,
2. Build a function that receives a `TPatient` and returns, through two by-ref parameters, the day and hour of the first maximum blood pressure. Assume day `0` hour `0` corresponds to the first hour of the first day,
3. Build a function that receives a `TPatient` and a blood-pressure value and returns the number of times the patient had an equal or higher blood pressure.

5 Define the most appropriate constants and types for `TStudent` with:

- ID (string).
- An array of integers with the identifiers of every subject the student has enrolled.

- An array of reals with the marks corresponding to each of the previous subjects.
- The number of subjects currently enrolled, always smaller than `MAXNOSBJ = 50`.

Then:

1. Build a function to get the minimum grade across all enrolled subjects,
2. Build a function to check (true/false) whether a `TStudent` passes a threshold `vmark` in all marks,

Assume subjects below the number enrolled all have a valid mark, and that the student enrolls subjects from `0` to the last sequentially without omission. › Sol

6 An **open array** is a structure containing an array (with a static maximum size `N`) and an integer telling how many elements are actually stored (`0` to `N`). › Sol

```C++
const int N = 100;
typedef array<int, N> TVector;
struct TOpenArr {
    int noelems;
    TVector ar;
};
int main()
{
    void printOArr(TOpenArr);
    TOpenArr sumOArr(TOpenArr, TOpenArr);

    TOpenArr a = {3, {{1, 2, 3}}};
    TOpenArr b = {3, {{3, 2, 1}}};
    printOArr(sumOArr(a, b));
    return 0;
}
TOpenArr sumOArr(TOpenArr a, TOpenArr b)
{ ... }
void printOArr(TOpenArr a)
{
    for (int i = 0; i < a.noelems; ++i)
        cout << a.ar[i] << " ";
    cout << endl;
}
```

Fill in the function `sumOArr` that sums 2 open arrays.

7 There are two ways to represent polynomials of one variable $p(x) = a_0 + a_1 x + a_{n-1}x^{n-1} + \ldots + a_n x^n$:

1. A simple array of floats `array<float, N> TPoly` where each value `a[i]` is the coefficient a_i in $a_i x^i$. The index corresponds to the degree of the monomial: `a[0]` is the constant term and `a[N-1]` is the coefficient of x^{N-1}, so `TPoly` can hold polynomials of degree up to $N-1$. For example, the polynomial $x^5 - 3.25x^3 + 2.25x$ would be represented with `N = 6` as: `{{0, 2.25, 0, -3.25, 0, 1}}`. All elements in the array contain meaningful values. Some or all of them may be zero,
2. An **open array** of pairs of numbers —

`struct TMono {float coef; int deg;}` — each storing the degree and coefficient of one monomial, sorted so that the first pair has the highest degree. The previous example becomes◆:

◆ In this case an open array is not strictly *necessary* — a zero coefficient signals no further monomials. It is used here for practice with open arrays.

deg	5	3	1	—
coef	1	−3.25	2.25	0

For each representation construct the data structure and the following subprograms:

1. Print a `TPoly`,
2. Add a `TMono` to a `TPoly`, modifying the latter by ref:
   ```
   void addMono(TPoly& p, TMono m);
   ```
   or alternatively:
   ```
   void addMono(TPoly& p, int degree, float coef);
   ```
3. Add a `TPoly` onto a `TPoly`, modifying the latter by ref:
   ```
   void addPoly(TPoly& a, TPoly b);
   ```

This and next problems have the solution later.

8 Using the `TPoly` type from the previous exercise, build a function that receives a polynomial and a point x and returns the value of the polynomial at that point:

```
float evalPoly(TPoly p, float x);
```

9 The **Horner method** is an efficient way to evaluate polynomials at a point x, avoiding repeated power computations. On Wikipedia you will find a solution in C. Give two better solutions, one for each of the representations developed in the previous exercise.

10 Build a function that returns the **derivative** of a `TPoly`:

```
TPoly derivative(TPoly p);
```

As you can read on Wikipedia, the derivative of

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

is

$$n \cdot a_n x^{n-1} + (n-1) \cdot a_{n-1} x^{n-2} + \ldots + 2 \cdot a_2 x + a_1$$

11 Build a function that returns the **sum** of two `TPoly` values:

```
TPoly sumPoly(TPoly a, TPoly b);
```

Summing polynomials is simply summing their corresponding coefficients.

12 Multiplying a `TPoly` by a real number means multiplying each coefficient by the parameter: › Sol

```
TPoly multBy(TPoly p, float b);
```

Multiplying a `TPoly` by a monomial means multiplying each coefficient by `c` and adding the monomial degree `deg` to each degree:

```C++
TPoly multBy(TPoly p, int deg, float c); // or
TPoly multBy(TPoly p, TMono m);
```

Once you have those, multiplying a `TPoly` by another `TPoly` is summing the polynomials obtained by multiplying every monomial of one of them with the other polynomial:

```
TPoly multBy(TPoly p, TPoly q);
```

The simplest representation is form **(a)** from the previous exercise: the direct array of coefficients. After solving the helper functions, form **(b)** based on structs is more flexible.

13 Chemical compounds consist of many identical molecules. A molecule is a sequence (an array) of elements, each of which is an atom (e.g. `Fe`, `H`, `Ni`) repeated a certain number of times. Each element (`TElement`) stores a string for the atom symbol together with its frequency (an integer in the range 1–9). A molecule (`TMolecule`) is an open array of `TElement` values.

Write a function

```
TMolecule string2Molec(string s);
```

that parses the input string and returns a `TMolecule`.

Write also a procedure to print them:

```
void printMol(TMolecule);
```

in the classic way, so that
`printMol(TMolecule{{"Na", 1}, {"O", 1}, {"H", 1}})`
prints `NaOH`.

Test it with these examples: › Sol

```C++
string2Molec("NaOH")
    // {{"Na", 1}, {"O", 1}, {"H", 1}}
string2Molec("Rb2SeO4")
    // {{"Rb", 2}, {"Se", 1}, {"O", 4}}
```

An atom symbol always starts with an uppercase letter; any additional letters are lowercase. To simplify, assume the frequency is 1 when no digit follows, and otherwise always between 2 and 9.

o O o

4·5 Matrices

In Mathematics and Physics we know there are matrices, frequently two-dimensional, with which you can express geometry, interlink relationships, etc. There are also three-dimensional, and even higher-dimensional, matrices. In Geography, Electronics, Game Theory, and countless other areas, arranging data this way is often the only way to model the information.

$$\begin{pmatrix} 10 & 2 & -3 & -7 & -2 & 4 & 5 & 5 & -3 & -1 \\ 4 & 10 & -2 & 6 & -1 & -7 & 6 & 2 & 9 & -8 \\ -5 & 1 & 3 & 9 & 3 & -2 & -2 & 8 & 9 & 4 \\ -7 & 2 & -3 & 7 & -1 & 4 & -4 & 8 & 9 & -3 \\ 9 & -8 & 3 & 0 & -9 & -7 & 1 & 4 & -4 & 8 \\ 9 & -8 & 9 & 1 & 2 & -9 & 8 & -8 & -4 & -4 \\ 3 & -6 & 4 & -2 & -1 & 1 & -6 & -6 & -8 & -3 \\ -3 & 8 & -5 & -6 & 9 & -7 & 2 & -7 & -2 & 1 \\ -3 & -9 & -4 & -5 & 0 & 5 & 5 & 0 & -9 & -7 \\ 2 & 8 & 10 & 9 & 0 & -1 & 9 & -1 & 5 & -4 \end{pmatrix}$$

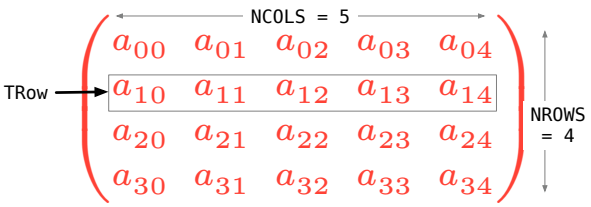
We can easily build matrices in C++ — using a syntax that is perhaps simpler than in other languages — as arrays of arrays:

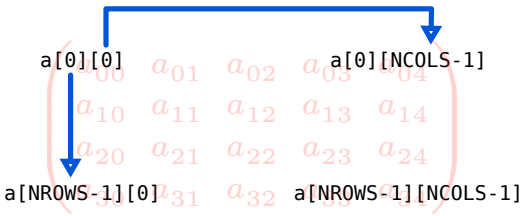
```C++
1 const int NROWS = 4;
2 const int NCOLS = 5;
3 typedef array< float,NCOLS> TRow;
4 typedef array<TRow,NROWS> TMatrix;
```

So in line 3 we have a normal array of floats composing a single horizontal line of the matrix, and in line 4 we repeat those lines `NROWS` times.

When the middle type `TRow` is of no use in our program we can skip its declaration and directly compose the matrix at once:

```
typedef array< array<float, 5>, 4> TMatrix;
```





When we access the elements of a matrix, we have to be careful with the order of the indexes: the first always refers to the row (starting at `0` for the first row), and the second, to the column:

```
const int NROWS = 4;
const int NCOLS = 5;
typedef array<array<float,NCOLS>,NROWS> TMatrix;

int main()
{
    TMatrix a;
    a[0][1] = 32.3;
```

Some languages, but not ours, allow bundling all the indexes of a multi-dimensional array into a single one, ~~[0, 1]~~.

Remember that access to a single row is also possible: `a[2]` gives you the third row. Whether or not you declared it as its own type, `typedef array< float,NCOLS> TRow;`, you can send that row as a parameter or assign it as a whole to another row — having a named `TRow` type simply makes those declarations easier to write; the row itself behaves exactly the same either way.

Traversing a matrix is straightforward if the walk is complete, with `for`s. For example, to initialise a matrix to `0`:

```
TMatrix m;

for (int r = 0; r < NROWS; ++r)
    for (int c = 0; c < NCOLS; ++c)
```

```
        m[r][c] = 0;
```

Identity matrix:

```
const int N = 3;
typedef array<array<float,N>, N> TSqMat;

TSqMat identityMatrix()
{
  TSqMat m;
  for (int r = 0; r < N; ++r)
    for (int c = 0; c < N; ++c)
      if (r == c)
        m[r][c] = 1;
      else
        m[r][c] = 0;
  return m;
}
```

For the exercises the first tool you will need is

```
void printMat(TMatrix m)                                                C++
{
  for (int r = 0; r < NROWS; ++r) {
      for (int c = 0; c < NCOLS; ++c)
          cout << m[r][c] << " ";
      cout << endl;
  }
}
```

Initialising through `cin >> m[r][c]` can be cumbersome, even for small matrices, particularly when debugging your code by running it repeatedly. We therefore strongly recommend using the quick initialisation method shown here instead:

```
const int NROWS = 4;                                                    C++
const int NCOLS = 5;
typedef
 array<array<float,NCOLS>,NROWS> TMatrix;

int main()
{
    TMatrix a = {{
        { 1,  2,  3,  4,  5},
        { 6,  7,  8,  9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20},
    }};
    //
}
```

Note that although the arrays of numbers do not require double braces, the array of rows, being composite rather than simple numbers, requires double braces. You can always use double braces for both:

```
    TMatrix a = {{                                                      C++
        {{ 1,  2,  3,  4,  5}},
        {{ 6,  7,  8,  9, 10}},
        {{11, 12, 13, 14, 15}},
        {{16, 17, 18, 19, 20}},
    }};
```

Traversing a matrix is less straightforward if, for some reason, we need to stop partway through:

```
bool isIn(TMatrix m, int x)                                             C++
{
  bool found = false;
```

```
  int r = 0;
  while (r < NROWS and not found) {
    int c = 0;
    while (c < NCOLS and not found) {
      found = m[r][c] == x;
      ++c;
    }
    ++r;
  }
  return found;
}
```

Do not do this, tempting as it may be:

```
bool isIn(TMatrix m, int x)                                    C++
{
  for (int r = 0; r < NROWS; ++r)
    for (int c = 0; c < NCOLS; ++c)
        if (m[r][c] == x) return true;
  return false;
}
```

Multidimensional Arrays: You can always extend matrices to more dimensions by nesting one more `array` inside the `typedef`:

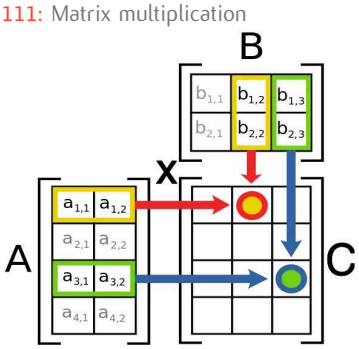
```
const int NROWS = 4;
const int NCOLS = 5;
const int NDEPTH = 3;
typedef array<array<array<float, NDEPTH>, NCOLS>, NROWS> TCube;

TCube cube;
cube[0][1][2] = 32.3;
```

Everything we have seen for matrices — declaration, access order (outermost index first), traversal with nested `for`s — extends the same way: one more index, one more nested loop.

We leave the exercise of matrix multiplication for the student: $A \times B = C$, where, as you can see, a precondition is that the number of columns of the first matrix (A) must match the number of rows of the second (B). To simplify this, without losing the generality of the algorithm, do it with square matrices (`TSqMat`):

111: Matrix multiplication




```
const int N = 3;
typedef array<array<float,N>, N> TSqMat;

TSqMat mult(TSqMat a, TSqMat b);
```

4·5.1 Checkpoint

1) Write the prototype of a function that returns the sum of all the elements of a `TMatrix`.
2) Write the prototype of a function that returns the sum of a single row, given its index.
3) Why does `printMat` need `NROWS` and `NCOLS` to be visible, instead of receiving them as parameters?
4) What would `cube[1][0][2]` mean for the `TCube` type defined above?

5) You need a `bool isIn(TMatrix m, int x)` that stops as soon as it finds `x`. Which loop must you use — `for` or `while` — and why is the crossed-out `for` version above not acceptable?

6) With `NROWS = 4` and `NCOLS = 5`, which element does `a[3][4]` refer to, and what is the valid range of each of the two indexes?

7) In the quick initialisation `TMatrix a = {{ ... }};`, why does the outer list of rows need double braces while each inner row of numbers may use single braces?

8) Write the loop or loops needed to compute the sum of the main-diagonal elements (`m[i][i]`) of a `TSqMat m`. How many loops are really necessary?

Task 13: Matrices

$$M_{\text{NROW}=4 \times \text{NCOL}=5} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

```
typedef array<array<int, NCOL>, NROW> TMat;

typedef array<array<int, N>, N>     TSqMat;
```

1 Build a procedure

```
void printMat(TMat m)
```

that prints the matrix `m` on the screen. › Sol

2 Build a procedure

```
void fillRandMat(TMat& m)
```

that fills the matrix `m` with random numbers. Call `srand(time(0))` once to seed the random sequence, then use `rand()`, which returns a pseudo-random number in in $[0, \texttt{RAND_MAX}]$ Use `printMat` from previous exercise afterwards. › Sol

3 Build a function that returns an identity square matrix: › Sol

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

```
TSqMat identSqMat();
```

4 Build a function that determines whether the `TSqMat` it receives is **diagonal** (all elements are `0` except those on the main diagonal): › Sol

$$\begin{pmatrix} 33 & 0 & \cdots & 0 \\ 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 33 \end{pmatrix}$$

```
bool isDiagonal(TSqMat m);
```

5 Build a function that returns the **sum** of its two `TMat` parameters: › Sol

```
TMat sumMats(TMat a, TMat b);
```

6 Build a function that **transposes** the `TSqMat` it receives: element $[i, j]$ swaps with element $[j, i]$.

```
TSqMat transpose(TSqMat a);
```

HINT: Be careful not to swap each pair of elements twice. Traverse only the elements above the main diagonal (j from $i + 1$ to N). › Sol

7 Build a function that determines whether the `TSqMat` it receives is **symmetric** (a matrix is symmetric if it equals its own transpose): › Sol

```
bool isSymmetric(TSqMat m);
```

8 Build a procedure that swaps two rows of a `TMat`:

```
void swapRows(TMat& m, int r1, int r2);
```

Try declaring `TMat` as an array of `TRow` and swapping the corresponding row objects directly. › Sol

9 Build a procedure that finds and returns the coordinates (x, y) of the **first local maximum** cell — one that is strictly greater than all of its existing neighbours (8 neighbours in the interior, 5 on an edge, 3 at a corner). Scan from top-left to bottom-right and return the first one found. If no such cell exists, return -1 in both coordinates.

```
void firstMax(TMat m, int& x, int& y);
```

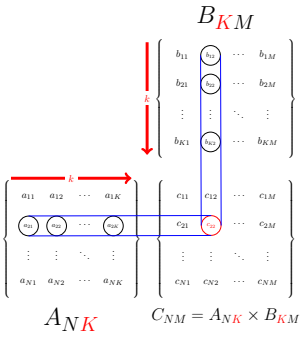
Use the following helper, which returns `true` when the index pair (i, j) is outside the matrix (so that out-of-bounds neighbours are treated as smaller than any interior value): › Sol

```
bool gt(TMat m, int v, int i, int j)                                C++
{
    return (i >= 0 and i < NROWS and j >= 0 and j < NCOLS) ?
                v > m[i][j] : true;
}
```

10 Multiply two matrices and return the resulting product:

```
TMatNM multMat(TMatNK a, TMatKM b);
```

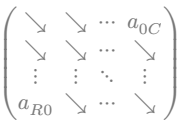
Recall that cell (i, j) of the result is the dot product of row i of the first matrix by column j of the second. › Sol



Because we are not using square matrices here, three separate type aliases are needed:

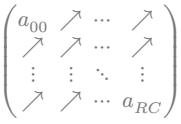
- `TMatNM` $(N \times M)$,
- `TMatNK` $(N \times K)$, and
- `TMatKM` $(K \times M)$.

11 A **diagonal** of a matrix consists of the elements reached by moving diagonally: both indices increase together for downward diagonals (↘), while the row index decreases as the column index increases for upward diagonals (↗). Every such diagonal has a sum, and diagonals vary in length.

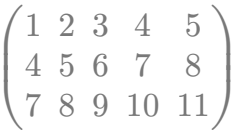


Find the greatest diagonal sum for a matrix

```
typedef array<array<int, C>, R> TMat
```



For example, given:




```
int main()
{
    void test(TMat);
    test(TMat{{{{1, 2, 3,  4,  5}},
                {{4, 5, 6,  7,  8}},
                {{7, 8, 9, 10, 11}}}});
```

prints the matrix and 21 as its `maxDiag`. The diagonal sums are: › Sol

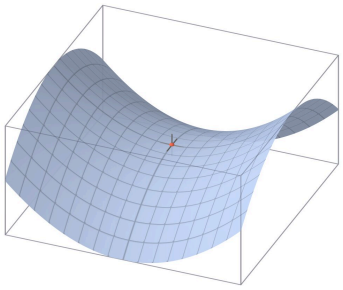
```
Downward:  7,   4+8,  1+5+9,  2+6+10,   3+7+11,  4+8,    5                Display
           7,    12,     15,      18,       21,   12,    5

Upward:    1,   2+4,  3+5+7,   4+6+8,    5+7+9, 8+10,   11
           1,     6,     15,      18,       21,   18,   11

Maximum sum: 21
```

12 A `TMat` has a **saddle point** at a cell that is a local maximum with respect to its row neighbours and a local minimum with respect to its column neighbours, or vice versa.

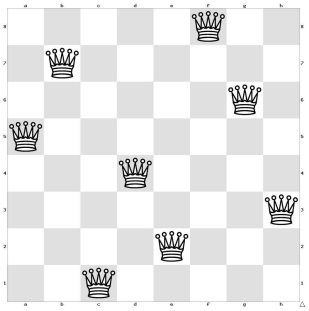
A horizontal local maximum (or minimum) is a cell strictly greater (or smaller) than its left and right neighbours; at a border, only the single existing neighbour counts. The same definition applies vertically.



A `TMat` can have more than one saddle point. Build a function that traverses the whole `TMat` and returns another `TMat` of the same size filled with:

- `+1` where there is a row-maximum / column-minimum saddle point,
- `-1` where there is a column-maximum / row-minimum saddle point,
- `0` everywhere else › Sol

13



A chessboard is a `TSqMat` with $N = 8$. Each player has 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns — conveniently represented with a C++ `enum` or integer constants. This exercise, however, focuses on a different chess-based problem: the eight queens puzzle. The goal is to place eight queens on an 8×8 board so that no queen threatens any other. Build a function that determines whether a given placement is threat-free:

```
bool noThreaten(TSqMat m);
```

In the `TSqMat` board, `true` means a queen occupies that cell. For each cell containing `true`, verify that all cells reachable from it (same row, same column, both diagonals) are free of queens. › Sol

14 Magic Squares

A magic square is a `TSqMat` of $N \times N$ **distinct** integers (each used exactly once) in which the sums of every row, every column, and both main diagonals are equal. This common value is the **magic constant** $\frac{n(n^2+1)}{2}$. (A more comprehensive formula is available here.)

Magic squares exist for all N except $N = 2$. The oldest known example, the unique 3×3 magic square dating from 650 BC, is:

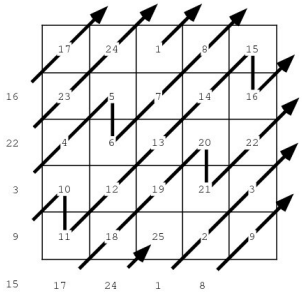
$$\begin{pmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{pmatrix}$$

Adding 19 to every element yields another valid magic square:

$$\begin{pmatrix} 23 & 28 & 21 \\ 22 & 24 & 26 \\ 27 & 20 & 25 \end{pmatrix}$$

H. Coxeter's algorithm constructs magic squares for any odd N:

- Start by placing 1 in the middle cell of the top row.
- Move diagonally up and to the right, placing consecutive numbers in empty cells.
- When crossing a boundary, wrap around to the opposite side (as if the matrix were tiled infinitely).
- If the destination cell is already occupied, move one cell down instead and continue.



Implement `void coxeter(TSqMat& magic);` Test with odd values of N and display the results. › Sol

15 Chess knight tour

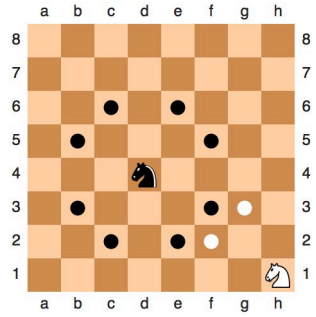
A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once.

This exercise does not ask you to **find** such a tour; instead, you must **verify** whether a given tour is correct. › Sol

The tour is encoded in a `TSqMat` of integers (assume $N = 8$): the cell numbered 1 is the starting square, the knight must then land on the cell numbered 2 via a legal jump, and so on. A complete, valid tour covers all $8 \times 8 = 64$ squares. Build:

```
bool okTour(TSqMat m);
```

A knight can jump to at most 8 positions: two squares horizontally and one vertically, or two squares vertically and one horizontally, in any combination of directions.



An example of a valid tour:

```
TSqMat m = {{                                                    C++
    {{ 1, 64, 61, 42, 37, 40, 55, 52}},
    {{60, 43,  2, 63, 56, 53, 36, 39}},
    {{ 3, 62, 59, 20, 41, 38, 51, 54}},
    {{44, 19,  4, 57, 50, 21, 32, 35}},
    {{ 5, 58, 45, 18, 31, 34, 49, 22}},
    {{14, 17,  6, 25, 46, 11, 30, 33}},
    {{ 7, 26, 15, 12,  9, 28, 23, 48}},
    {{16, 13,  8, 27, 24, 47, 10, 29}}
}};
```

$$\begin{pmatrix} 1 & 64 & 61 & 42 & 37 & 40 & 55 & 52 \\ 60 & 43 & 2 & 63 & 56 & 53 & 36 & 39 \\ 3 & 62 & 59 & 20 & 41 & 38 & 51 & 54 \\ 44 & 19 & 4 & 57 & 50 & 21 & 32 & 35 \\ 5 & 58 & 45 & 18 & 31 & 34 & 49 & 22 \\ 14 & 17 & 6 & 25 & 46 & 11 & 30 & 33 \\ 7 & 26 & 15 & 12 & 9 & 28 & 23 & 48 \\ 16 & 13 & 8 & 27 & 24 & 47 & 10 & 29 \end{pmatrix}$$

16 Project — Game of Life

Conway's Game of Life is a cellular automaton in which a matrix of cells evolves through discrete time steps (generations) based solely on the states of neighbouring cells.

The universe is an infinite two-dimensional grid; each cell is either `ALIVE` or `DEAD` and interacts with its 8 neighbours (horizontally, vertically, and diagonally adjacent). At each generation the following transitions occur simultaneously:

1. A living cell with **fewer than 2** living neighbours dies (underpopulation).
2. A living cell with **2 or 3** living neighbours survives.
3. A living cell with **more than 3** living neighbours dies (overpopulation).
4. A dead cell with **exactly 3** living neighbours becomes alive (reproduction).

Summarised as a table:

neighbours	state	→	new state
< 2	ALIVE	→	DEAD
$= 2, 3$	ALIVE	→	ALIVE
$= 3$	DEAD	→	ALIVE
> 3	ALIVE	→	DEAD

In a simple C++ program, the grid can be a `TMat` no larger than a typical terminal window (around `40` rows × `100` columns). Initialise it with `rand()` as in exercise 2, and redraw it each generation after clearing the screen by sending:

```
const string CLRSCR = "\x1B[H\x1B[J";
```

(On Windows you may need `system("cls")` instead.)

The core subprogram is:

```
void nextGeneration(TMat& m)
```

To avoid having previous states interfere with the next generation, **modify the array in place** using temporary state values for each cell during the evaluation pass. A second pass then writes the definitive values. Cell states can be represented with:

```
typedef int TState;
const int DEAD     = 0;
const int ALIVE    = 1;
const int TODIE    = 2;
const int TOBEBORN = 3;
```

Boundary cells can be treated as surrounded by dead cells, or you can **wrap** the indices so that opposite edges are neighbours (as if the grid were folded into a torus). › Sol

After displaying a generation, the program can wait for a keypress or pause automatically with `usleep(100000)` (100 ms) before drawing the next one. A helper to fill the grid randomly:

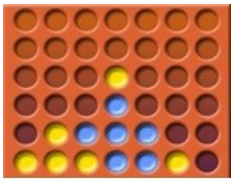
```
void arrAleat(TMat& a)
{
    for (int i = 0; i < NROWS; ++i)
        for (int j = 0; j < NCOLS; j++)
            a[i][j] = (TState)(rand() % 2);
}
```

17 *(From exam)* Design a program, organised into subprograms, that reads from the keyboard a natural number `M` (which must be greater than `0` and at most a known constant, e.g. `MAX = 10`). The program then reads an indefinite sequence of **distinct** integers ending in `0` and computes the average of the `M` largest numbers in the sequence. If the sequence contains fewer than `M` numbers, compute the average of those that were entered.
› Sol

```
Number of highest numbers averaged (between 1 and 10): 4  Display
7 8 1 2 3 4 5 6 0  ->  (5 6 7 8)  6.5

Number of highest numbers averaged (between 1 and 10):  5
1 2 3 0  ->  (1 2 3)  2
```

18 *(From exam)* **Four in a row**



The game is played on a vertical board with 6 rows and 7 columns. A token dropped into a column falls to the lowest empty cell in that column. Two players alternate turns, each using tokens of a different colour (represented here as `1` and `2`; `0` means an empty cell). The first player to form a line of four tokens of the same colour — horizontally, vertically, or diagonally — wins.
Design the data structure for the board and implement the following two subprograms. To test them, write a `main()` that initialises an empty board, makes several calls, and prints the results (including the full board contents). › Sol

a. `insertToken` — stacks a token into a column:

```
void insertToken(TBoard& tab, int token, int col,
                 bool& ok, int& row);
```

If the column is full, return `false` in `ok`; otherwise return `true` and the `row` in which the token was placed. The token value and column number are always valid.

b. `fourInArow` — checks efficiently whether the most recently added token creates a winning line:

```
bool fourInArow(TBoard tab, int row, int col);
```

Returns `true` if `tab` contains four equal tokens in a line (horizontal, vertical, or diagonal) one of which is at position `(row, col)`. Returns `false` if no token is present at that position. The row and column values are always valid.

19 The **distance between two occurrences of a word** in a text is the number of words that appear between them. Design a program that reads words from the keyboard until the word `end` is entered, then prints the maximum distance between each pair of repeated words. Words that appear only once are not printed. › Sol

```
Enter text (end to finish):                                    Display
por favor traeme el abrigo que tengo en el armario
que tengo que ir por el descampado a por el pan end

Maximum distances between repeated words:
  por   : 13
  el    :  6
  que   :  4
  tengo :  4
```

Notes:

- The text contains an indefinite number of words.
- It ends with the sentinel word `end`.
- Each word consists solely of lowercase letters and has an indefinite but bounded length.
- Words are separated by spaces.
- At most `MAX_DIFF_WORDS` (a known constant) distinct words will appear in the text.

o O o

Task 14: Final exams problems

After reviewing the contents of the first topic, please answer the following questions. You may record your answers in a file named `02.theoryQuestions.txt` by listing the question number followed by your response. Be sure to include 1) the name of the file 2) your name/alias and the date at the beginning of the file. Remember to upload the file to the corresponding task Dropbox on Campus Virtual.

1 A program reads a positive `pattern` integer (re-prompting if ≤ 0), then a sequence of integers ending in `0`, and prints all positive integers in the sequence whose digits sum matches that of `pattern`. Negatives are discarded; no repeats in output. › Sol

- Build `sumDigits()` to compute the digit sum of a positive integer.
- Read the integers and print those whose digit sum matches `pattern`.

To be considered:
- Modularisation and correct loop choice matter; efficiency is evaluated.
- The sequence is unlimited, terminated by `0` (not included).
- Integers are separated by spaces or newlines.
- The maximum number of different matching integers is `NMAX_DIFF` (e.g. `50`).

Display
```
Enter the pattern: -3
Enter the pattern: 0
Enter the pattern: 153
Enter integers (0 to end):
1242 564 -6576 9 1242 1523 7689713 3 -531
Result: 1242 9 45
```

2 From `main()`, call `test()` receiving an $R \times C$ integer matrix (`R=4`, `C=3`). It displays the matrix with `display()`, then calls `firstCent()` to find and return the indexes of the first cell equal to the sum of its neighbours (3, 5 or 8 neighbours, depending on position). Returns $(-1, -1)$ if none found. Efficiency is evaluated. › Sol

$$M = \begin{pmatrix} 3 & 1 & 2 \\ 2 & 16 & 2 \\ 1 & 3 & 2 \\ 5 & 1 & 6 \end{pmatrix}$$

- $M_{0,0} : 3 \neq \sum\{1, 2, 16\} = 19$.
- $M_{1,1} : 16 = \sum\{3, 1, 2, 2, 2, 3, 1, 2\} \longrightarrow$ returns $(1, 1)$.

To be considered:
- Modularisation is evaluated. You may add subprograms.
- Values may be negative or zero.
- Quick-initialise and send the matrices as parameters; do not read them from keyboard.

```
Matrix:                                                          Display
3 1 2
2 16 2
1 3 2
5 1 6
First cell satisfying the condition: (1, 1)

Matrix:
2 1 7
2 5 8
2 2 1
7 7 2
First cell satisfying the condition: (-1, -1)

Matrix:
2 1 7
2 5 8
2 2 1
7 6 9
First cell satisfying the condition: (3, 2)
```

3 A mountain profile is stored as an array of `N=10` integer heights. For example:

```
                    *                                            Display
                *   *
            *   *   *
*           *   * * *
*           * * * * *
* *       * * * * * *
* * *     * * * * * *
* * * * * * * * * *
5 3 2 1 3 6 4 7 5 8
```

Has peaks at indexes: 0 (height 5), 5 (height 6), 7 (height 7) and 9 (height 8). A position is a peak when both neighbours are shorter; the extremes are peaks when their only neighbour is shorter. The distances between consecutive peaks are: $5 - 0 = 5$, $7 - 5 = 2$ and $9 - 7 = 2$.

Build `peaksDists()` receiving an array of heights and returning a zero-terminated array with the distances between consecutive peaks. For the example above: 3 distances {`5, 2, 2, 0`}, printed as:
`Distance between a peak and the next: 5 2 2` › Sol

4 The **distance between two words** in a text is the number of words between them: between `a` and `d` in `"a b c d"` the distance is 3. Read a sequence of words ending in `end` and print the maximum distance between repeated words. Non-repeated words do not appear in the output. flux › Sol

```
Type in the words (end to finish):                               Display
can you can a can as a canner can can a can end

Max distance between repeated words:
a: 4
can: 4
```

Observations:
- All words are lowercase; only spaces between them.
- The sequence length is unlimited.
- Total different words limited to `MAXPAL` (known constant).

5 Read a natural number `m` with `0 < m <= MAXDIFNUMS` (re-prompting until valid), then a sequence of natural numbers ending in `0`. Display the `m` greatest numbers each with the **positions in which they appeared**. If fewer than `m` numbers were entered, display all of them. › Sol

```
Type in m: 5                                                    Display
Type in a sequence of natural numbers, 0 ends it:
2 3 4 7 28 4 5 1 1 1 9 7 4 4 28 2 1 3 3 6 0
The 5 greatest ones and their positions are:
   7:  4 12
  28:  5 15
   9: 11
   5:  7
   6: 20
```

Starting data structures and prototypes:

```C++
const int MAXDIFNUMS = 50;
const int MAXREPETIT = 10;

typedef array<int,MAXREPETIT> TPoss;
struct TNum {
  int num;
  TPoss pos;
};
typedef array<TNum,MAXDIFNUMS> TSeq;


int main()
{
  void addNumbers(TSeq& seq, int m);
  void displayNumbers(TSeq seq);
  void init(TSeq& seq);
  TSeq seq;
  init(seq);
  int m = 5;
  addNumbers(seq, m);
  displayNumbers(seq);
  return 0;
}
```

Suggested subprograms:

```
int lenSeq(TSeq seq);
int posn(TSeq seq, int n);
int posMin(TSeq seq);
void initPos(TPoss& poss);
void writeFirstTime(TSeq& seq, int i, int n, int pos);
void addPos(TSeq& seq, int i, int pos);
```

6 A **word-search puzzle** is an $N \times N$ char matrix. Build a subprogram that receives the matrix and a string, and returns whether the string was found, and if so, its starting position and direction (`NORTH`, `SOUTH`, `EAST`, `WEST`). The word appears at most once. Also build `main()` to read the matrix (8 strings of 8 chars, `N=8`), read the search string, call the subprogram, and print the result or `"it is not there"`. › Sol

```
8 strings with 8 chars each:                                    Display
  wynapjgu
  fplmodrd
  ipiankee
  riurkxrd
  cjogomid
  ikgoxpiq
  lcdrghef
  sgnpycft
Enter a word to find: program
The word "program" is in the position (7,3) North
```

HINT: To populate the matrix, pass a string parameter containing all matrix characters without spaces or line breaks, then assign the characters to the matrix cells sequentially. Afterward, you can prompt for a word to search for.

7 Define `TList` (containing up to `MAX` integers). Implement `sieve()` that takes a `TList list1` and an integer `n` as parameters, returning a `TList list2` containing only those numbers from `list1` that appear **exactly** `n` times, without duplicates. › Sol

```
list1 is: 1 3 4 3 1 3 0 -6 4                                    Display
n is: 2
Then list2 will be: 1 4
```

8 Work with **incomplete matrices** `TMat` of maximum size `MAX×MAX`: › Sol

```
const int MAX = 10;
struct TMat {
    int nrows, ncols;
    array< array<int,MAX>, MAX> data;
};
```

Build procedures to read and print `TMat`. Then, build a subprogram to **multiply** two matrices into a third. Ask the user for dimensions before reading elements. Assume the matrices are multipliable (columns of first = rows of second); do not check this precondition.

To multiply two matrices, we must multiply each row of the first matrix with each column of the second. These are **scalar** multiplications, so we must multiply each corresponding element and **sum** these products. That is: given $\boldsymbol{A} = a_{ij}$ and $\boldsymbol{B} = b_{ij}$, the product $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ has elements:

$$C_{ij} = \sum_{k=0}^{M} A_{ik} \cdot B_{kj}$$

To find the element C_{ij}, we iterate over an index k that ranges over the common dimension M, which equals both the number of columns in the first matrix and the number of rows in the second matrix.

9 The **centre** c of a vector v of N natural values is the index

$$1 \leq c \leq N-2$$

satisfying: › Sol

$$\sum_{i=0}^{c-1}(c-i)\cdot v_i = \sum_{j=c+1}^{N-1}(j-c)\cdot v_j$$

Build `centreOfVector()` receiving `v` and returning two values by reference: whether a centre exists, and if so, its index.

Display
```
vec is: 6 2 3 0 1
Then centre is 1   // 1*6 = 6  IS EQUAL TO  1*3 + 2*0 + 3*1 = 6

vec is: 1 2 1 1 0
No centre
```

HINT: You can try calling a procedure `test(TVec v)` twice from `main()`, once with each of these `v`ectors, and display the results there.

10 Read a `suffix` string and a sequence of words, some of which terminate with that suffix. Print all words ending with that suffix, without repetitions.

Display
```
Suffix?: ing
Type in a sequence of words (end to finish):
i decided going hiking instead of going riding bike end

Words ending in "ing":
going hiking riding
```

Observations: › Sol

- All words are lowercase; only spaces between them.
- The number of words, the sequence length is unlimited.
- Total different words limited to `MAXPAL` (known constant).

11 🧊 Design:

```
int noOcurrs(TSeqNum seq, TPerm aPermutation);
```

`TSeqNum` is an array of `SEQ_LEN=11` integers; `TPerm` is an array of `PERM_LEN=4` integers. `noOcurrs()` returns how many times `aPermutation` occurs as a permutation of a same-size portion of `seq`, sliding `aPermutation` along `seq`.

To check whether a portion of `seq` is a permutation of `aPermutation`, build:

```
int countFrom(TSeqNum a, int from, int x);
```

returning how many times `x` appears in `a` starting at `from` for `PERM_LEN` positions. In the example, `noOcurrs()` returns 2. › Sol

12 🧊 **Run-length encoding** compresses strings with long runs of repeated chars. `rle()` replaces runs of equal chars by the char once, followed by `/` and the count — but only when the run exceeds `THRESHOLD`. Shorter runs are copied unchanged.

With `THRESHOLD=4`, the next compression occurs:

abcaaaaaaadbbbbbbdddvvvvxxaaaaaaaaccc
abca/7db/6ddd v/4xxa/8ccc

Build:

```
string rle(string s);
```

As a first simplification, assume run lengths are always less than `10`. (For longer runs, an extra delimiter char would be needed.) › Sol

13 Build `sliding()` (complete its interface) that receives two strings and two by-reference integers. In the first by-reference integer it returns the shortest displacement of the second string over the first that yields the greatest number of character matches, and through the last by-reference integer, returns that number of matches.

For `acbaabch` and `aabghc`: displacement 3, matches 4. › Sol

```
acbaabch  // displ: 0; coinc: 2                                    Display
aabghc
acbaabch  // displ: 1; coinc: 1
 aabghc
acbaabch  // displ: 2; coinc: 1
  aabghc
acbaabch  // displ: 3; coinc: 4
   aabghc
...
acbaabch  // displ: 7; coinc: 0
       aabghc
```

14 Build:

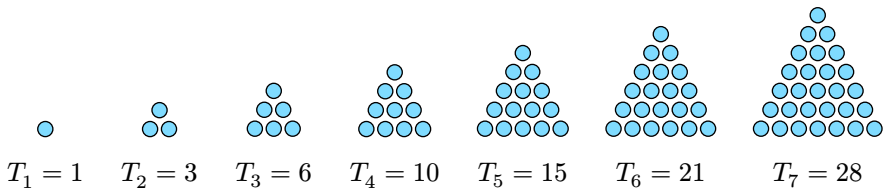
```
void clockwise(TSqMat& m);
```

where `TSqMat` is a square matrix of size `N`. `clockwise()` rotates the matrix `m` 90° clockwise **in-situ** without using auxiliary variables. › Sol

HINT: decompose the operation into two steps, with the first being **transpose**.
For `N=5`:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix} \xrightarrow{\texttt{clockwise()}} \begin{pmatrix} 21 & 16 & 11 & 6 & 1 \\ 22 & 17 & 12 & 7 & 2 \\ 23 & 18 & 13 & 8 & 3 \\ 24 & 19 & 14 & 9 & 4 \\ 25 & 20 & 15 & 10 & 5 \end{pmatrix}$$

15 A number is said to be the kth *triangular number* when it equals the sum of the first k natural numbers: $T_k = 1 + 2 + ... + k$.



In **Bulgarian solitaire**, a deck of N cards is divided into piles with n_i cards each, so $\sum n_i = N$. At each step, one card is removed from every pile; the removed cards form a new pile. Empty piles are deleted.

If $N = T_k$ then the game always reaches the stable configuration $\{1, 2, ..., k\}$ in at most $k^2 - k$ moves.

For $N = 15$ with initial piles $\{5, 7, 3\}$:

```
5 7 3                                                    Display
4 6 2 3
3 5 1 2 4
2 4 1 3 5
1 3 2 4 5
2 1 3 4 5
1 2 3 4 5
```

Design a program using a struct to hold the number of piles and their sizes. For the example above the initial state would be `{3, {5, 7, 3}}`, supporting up to `MAX=20` piles. Implement procedures to perform one step, print the current state, and check whether the stable state has been reached. The program prints each configuration until stability, as shown above. › Sol

o O o

Appendix A Handling files

Contents of this part Book contents

A·1 Declaring a handle to manipulate a file 194
A·2 Opening and closing the file 194
A·3 Reading and writing a text file 195
A·4 Streams as subprogram parameters 197



Files are essential for any program whose data must outlive a single run —that is, for almost every useful program.

A file is a repository of information stored on an external device separate from main memory. Files are persistent: they remain intact even after the system is shut down.

There are two main types of files: binary and text files. Text files are designed for human reading and can contain human-readable markup, such as `<bold>This is bold</bold>`. Binary files, by contrast, depend on specific programs that understand their structure. Over time, most binary formats—with the exception of well-established standards like `.pdf` or `.jpg`—risk becoming obsolete unless you maintain the software capable of reading and manipulating them, which is often impractical. Consequently, text files remain the only truly universal format with genuine prospects for long-term preservation.

Until the internet era, file extensions were the main way to determine a file's format. This remains valid, but transmitting files as byte streams—for example, linked from `.html`—made it necessary to augment them with metadata (the MIME type) to inform the browser what kind of file it is downloading, enabling it to handle the content appropriately.

Files are identified within each directory solely by their name, so each name must be unique within that directory.

File extensions serve to indicate the format of the content and/or which program can handle them.

- `.cpp` C++ text source file
- `.h`, `.hpp` C++ text header file
- `.html` Hypertext Markup Language file
- `.pdf` Binary document with formatted text, images, etc.
- `.jpg`, `.gif`, etc. Binary images
- `.zip` Compressed archive of one or more file(s)
- `.txt` Text file

Working with files in all programming languages requires the following four steps:

1. Declare the file.
2. Open the file.
   a. If the file does not yet exist and you attempt to open it for reading, an error will occur.

   b. If the file does not yet exist and you open it for writing, it will be created empty.
   c. If the file already exists and you open it for writing, its previous contents will be deleted and it will become empty.
   d. If you open it in *append* mode, the file is created if it does not exist; if it does, its contents are kept and everything you write goes after them.
   e. You can also open it for both reading and writing. Then the file must already exist and its contents are kept, but you can move the cursor around and overwrite specific parts; this is mostly done with binary files.
3. Read from and/or write to the file.
4. Close the file.

A·1 Declaring a handle to manipulate a file

In C++ there are three main types of file streams (other constructs such as streams for communication channels are treated similarly, but are not covered here).

- `ifstream`: *input* file stream, to read existing files.
- `ofstream`: *output* file stream, to create files and write to them.
- `fstream`: file stream for both reading and writing.

A·2 Opening and closing the file

The simplest scenario is opening an existing file for reading as text:

```C++
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in;

  in.open("customer.txt");
  if (not in) {
    cerr << "Error opening customer.txt" << endl;
  } else {
    // ... read from in ...
    in.close();
  }
  return 0;
}
```

Observe we need the header `<fstream>`, which contains the declarations necessary to handle files in C++.

You can declare and open the file in one step, passing its name to the constructor:

- `ifstream in    (filename);`
- `ofstream out   (filename);`
- `fstream  inout (filename, ios::in | ios::out);`

Both `open` and these constructors accept an optional second parameter, the *open mode*. `ifstream` always adds `ios::in`, `ofstream` always adds `ios::out`, and `fstream` defaults to `ios::in | ios::out`. Several flags can be combined with `|`:

- `ios::in` Input mode. Information is read from the file.

- `ios::out` Output mode. Information is written to the file. On its own, it empties an existing file (it implies `ios::trunc`).
- `ios::app` Append mode. If the file already exists, its contents are preserved and all output is written at the end of the file.
- `ios::ate` (*at end*) The cursor is placed at the end of the file right after opening; unlike `app`, it can later be moved back.
- `ios::trunc` If the file exists, its contents are deleted.
- `ios::binary` Binary mode. Bytes are transferred exactly as they are, without translating line endings (which matters on Windows).
- `ios::noreplace` (C++23, with `out` only) If the file already exists, opening fails, so you cannot overwrite it by accident.

Files are susceptible to the common issue of not existing or being unreadable when expected. Therefore, before proceeding with read operations, the first and most critical check we must perform is whether the file opened successfully.

```
in.open("customer.txt");
if (not in)
if (in.fail())
if (not in.is_open())
```

These are three ways to verify that the file opened correctly.

A file name without a directory, such as `"customer.txt"`, is relative to the program's *working directory*: the directory from which the program is run, which is not necessarily where the source code or the executable are. IDEs often run programs from a different folder, and this is the most frequent cause of "file not found" errors. If in doubt, run the program from a terminal in the directory where the file is, or give the full path.

When an application terminates, the system closes all the files that the application opened, and a stream is also closed automatically when its variable goes out of scope. However, it is good practice to close files explicitly as soon as you no longer need them: output is kept in a memory *buffer* and is only guaranteed to reach the disk when the file is closed (or flushed), and closing also releases the file for other programs. You may also want to close a file and reopen it later, perhaps in another mode, within the same program. The `.close()` method is used for this purpose:

```
in.close();
out.close();
```

A·3 Reading and writing a text file

We have explored various methods for opening a file, but how do we work with its contents?

Files are not like arrays; you cannot access their contents by index. When you open a file, a position marker called the *cursor* indicates where the next read or write will happen. Depending on the mode in which you open the file, it starts at the beginning (when opened for reading or writing) or at the end (when opened for appending).

After each read or write operation, the cursor advances accordingly.

Reading and writing use exactly the same operators and functions as `cin` and `cout`: `<<`, `>>`, `getline`, etc. Here is a simple program that writes two

names, one per line, closes the file as a demonstration, and then reopens it in append mode to add two more lines.

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  fstream f;
  f.open("demofile.txt", ios::out);             // created, or emptied
  if (not f) {
    cerr << "Cannot create demofile.txt" << endl;
  } else {
    f << "Juan" << endl;
    f << "Miguel" << endl;
    f.close();

    f.open("demofile.txt", ios::out | ios::app); // kept, write at the
    end
    f << "Paquito" << endl;
    f << "Vicenta" << endl;
    f.close();
  }
  return 0;
}
```

To write a single character use `out.put(c)`; its reading counterpart is `in.get(c)`, which reads one character—spaces and newlines included, unlike `>>`. In both cases the cursor advances one position.

When reading a file, you need to determine when to stop. Files are read sequentially, and you know you have reached the end because a read operation *fails*. Every read—`in >> x`, `getline(in, s)`, `in.get(c)`—returns the stream itself, which evaluates to `true` if the read succeeded and `false` otherwise. Hence the idiom: use the read itself as the loop condition, as in `while (in >> x)` or:

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream in("textfilewithlines.txt");
  if (not in) {
    cerr << "Cannot open textfilewithlines.txt" << endl;
  } else {
    string s;
    while (getline(in, s)) {
      cout << s << endl;           // display it
    }
    in.close();
  }
  return 0;
}
```

A very common mistake is to write `while (not in.eof())`. The function `eof()` only becomes `true` *after* a read has tried to go past the end, so the loop body runs once more with a failed read, typically processing the last item twice or processing garbage.

If you have a file `people.txt` with one person per line, the name and the age separated by `;`:

```
Juan Luis; 21                                                    Display
Pedro; 18
Juanito Maravillas; 14
```

the following program loads it into an array:

```
#include <iostream>                                              C++
#include <fstream>
#include <string>
#include <array>
using namespace std;

const int MAX = 200;
struct TPers {
  string name;
  int age;
};
typedef array<TPers,MAX> TPeople;

int main() {
  TPeople persons;
  int cnt = 0;                    // number of persons read

  ifstream in("people.txt");
  if (not in) {
    cerr << "Cannot open people.txt" << endl;
  } else {
    while (getline(in, persons[cnt].name, ';')) {
      in >> persons[cnt].age;     // reads the age
      in.ignore(100, '\n');       // discards the rest of the line
      cout << persons[cnt].name << " (" << persons[cnt].age << ")" <<
      endl;
      ++cnt;
    }
    in.close();
  }
  // persons[0] .. persons[cnt-1] hold the data of the file
  return 0;
}
```

Two details merit consideration. First, we assume that the array capacity (`MAX`) is sufficient to accommodate the number of lines in the file. Second, `in.ignore(100, '\n')` is necessary because the `>>` operator stops immediately before the newline character; without it, that `'\n'` would become the first character of the next name.

A·4 Streams as subprogram parameters

Stream variables must always be passed to functions by reference: they cannot be copied, and every read or write changes them (their cursor and their error state).

```
bool openFileIn(ifstream& file, string name) {
  file.open(name);
  return file.is_open();
}
```

Better still, declare the parameter as `ostream&` (or `istream&` for reading). `cout` and any `ofstream` are both `ostream`s, so the same function works for the screen and for a file:

```
void printPerson(ostream& out, TPers p) {
  out << p.name << "; " << p.age << endl;
}
```

then

```
  printPerson(cout, persons[0]);    // on the screen
  printPerson(outFile, persons[0]); // into a file
```

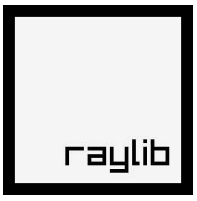
Note that `printPerson` writes in the same format that the previous program reads, so a file saved with it can be loaded again later.

Appendix B Graphics, sound and WIMP with C++

Contents of this part Book contents

B·1 Getting the tools ready 199
B·2 **raylib**: drawing, animation and sound 200
B·3 **FLTK**: windows, buttons and events 204

119: **raylib** for graphics, animation and sound



120: FLTK to build WIMP programs



121: A typical Window, Icon, Menu and Pointe application



So far every program in this book has talked to the world through `cout` and `cin`: plain text in a terminal. This appendix shows how to open a real window, draw in it, animate, make sound, and react to the mouse. None of this is part of standard C++ — the language itself knows nothing about windows — so we lean on two small, free, cross-platform libraries.

We use **raylib** for graphics, animation and sound, and **FLTK** for traditional windowed programs with buttons and menus (what is known as a *WIMP* interface: Windows, Icons, Menus, Pointer). Both are open source, both run on Windows, macOS and GNU/Linux, and both are lightweight enough to install in a few minutes.

raylib gives you a blank canvas and a fast loop — perfect for games, drawings and animations, but it has no buttons or menus of its own. **FLTK** gives you the buttons and menus, but it is not meant for drawing games.

B·1 Getting the tools ready

You only have to do this once. The instructions differ per operating system, so read the one section that matches your computer.

A practical tip before you start: on macOS and GNU/Linux the simplest and most reliable way to build these programs is the **Terminal**, using a single command you will reuse every time. On Windows, an IDE such as **Code::Blocks** is the smoother path, at least for **raylib**. Do not fight your system; use the tool that is easiest there.

B·1.1 On macOS

First, if you have not yet installed it, install Apple's command-line compiler (this also gives you `c++`):

```
xcode-select --install
```

Then install the libraries with **Homebrew** (`brew`):

```
brew install raylib
brew install fltk
```

That is all. Skip ahead to *How to compile and run*.

B·1.2 On GNU/Linux (Debian / Ubuntu)

Install the compiler and build tools, then the libraries:

```
sudo apt install build-essential
```

```
sudo apt install libraylib-dev
sudo apt install libfltk1.4-dev      # older systems: libfltk1.3-dev
```

If your distribution does not carry `libraylib-dev`, follow **raylib**'s *"Working on GNU Linux"* wiki page to build it from source once.

B·1.3 On Windows (Code::Blocks)

raylib. The easy route is **raylib**'s own Windows installer, which bundles a ready-to-use compiler (MinGW-w64) together with the library. After installing it to `C:\raylib`:

1. In **Code::Blocks**, open *Settings* → *Compiler* → *Toolchain executables* and point the compiler directory to `C:\raylib\MinGW`. Do *not* press "auto-detect".
2. Open the raylib project template that comes with the installer, or, in a new project, add these libraries in the linker settings: `raylib`, `opengl32`, `gdi32`, `winmm`.

FLTK. **FLTK** on Windows needs a little more setup than on macOS or Linux. The smoothest beginner path is to install **MSYS2**, add the **FLTK** package from its shell (`pacman -S mingw-w64-ucrt-x86_64-fltk`), and then compile from that shell with exactly the same `fltk-config` command shown below for macOS and Linux. Your instructor may also provide a preconfigured setup. If **FLTK** gives you trouble on Windows, do the **FLTK** exercises on a lab Mac or Linux machine.

B·1.4 How to compile and run

On macOS and GNU/Linux you compile from the Terminal. The command has an unusual looking piece, `$(...)`, so let us take it apart.

For a **raylib** program:

```
c++ myprogram.cpp    $(pkg-config --libs --cflags raylib)
./a.out
```

For an **FLTK** program:

```
c++ myprogram.cpp    $(fltk-config --cxxflags --ldflags)
./a.out
```

The `$(...)` part runs a little helper program (`pkg-config` for **raylib**, `fltk-config` for **FLTK**) that prints out all the technical flags the library needs: where its header files live and which libraries to link against. The shell pastes that output straight into your compile line. The point is that **you never have to memorise those flags** — the helper knows them. `-o myprogram` just names the output file; `./myprogram` runs it.

On Windows with **Code::Blocks** you do not type this: you press *Build and run* (the green gear-and-arrow), and the project settings you configured above supply the same information.

B·2 raylib: drawing, animation and sound

Every **raylib** program has the same skeleton: open a window, then loop until the user closes it. On each turn of the loop you clear the screen and draw the current picture. Because you redraw everything about sixty times per second, moving a shape a little each turn produces animation. This is called a *game loop*, and *you* are in charge of it.

B·2.1 A window with a message

Our first program opens a window, draws a circle (just the outline — a *circumference*), and writes a word in its centre.

```cpp
// raylibHello.cpp
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibHello.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"
#include <string> // we are not including iostream, so..

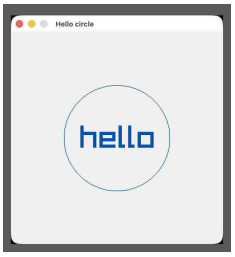
int main()
{
    InitWindow(400, 400, "Hello circle");
    SetTargetFPS(60);

    std::string message = "hello";
    int fontSize = 30;

    while (not WindowShouldClose()) {
        BeginDrawing();
            ClearBackground(RAYWHITE);
            DrawCircleLines(200, 200, 100.0f, DARKBLUE);
            DrawText(message.c_str(),
                     200 - MeasureText(message.c_str(), fontSize) / 2,
                     200 - fontSize / 2,
                     fontSize, DARKBLUE);
        EndDrawing();
    }

    CloseWindow();
    return 0;
}
```

122: The graphic output of a simple text and circle from **raylib**



Three things to notice:

1. `while (not WindowShouldClose())` is the game loop. It ends only when the user closes the window or presses `Esc`.
2. **raylib** draws text from its **top-left corner**, not its centre. To centre a message of width w inside a space of width T, the left margin is $(T - w)/2$. That is why we subtract half the text width (given by `MeasureText`) and half the font size. The very same idea will centre a whole window later on.
3. **raylib** is a C++ library with a C interface, so `DrawText` expects a `const char *`, not a `std::string`. We hand it one with `message.c_str()`. This little bridge appears whenever you use a C library from C++.

B·2.2 An animation: a bouncing ball

To animate, we keep the ball's position and velocity in a `struct`, and on each turn of the loop we move it and bounce it off the walls. Grouping the five numbers into one `Ball` keeps the program tidy: the ball travels from function to function as a single thing.

```cpp
// raylibBall.cpp
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibBall.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"
```

```
struct Ball {
    float x;
    float y;
    float velocityX;
    float velocityY;
    float radius;
};

// Prototypes
void updateBall(Ball &ball, int width, int height);
void drawBall(Ball ball);

int main()
{
    const int screenWidth = 600;
    const int screenHeight = 400;

    InitWindow(screenWidth, screenHeight, "Bouncing ball");
    SetTargetFPS(60);

    Ball ball = {screenWidth / 2.0f, screenHeight /
                                2.0f, 4.0f, 3.0f, 20.0f};

    while (not WindowShouldClose()) {
        updateBall(ball, screenWidth, screenHeight);

        BeginDrawing();
            ClearBackground(RAYWHITE);
            drawBall(ball);
        EndDrawing();
    }

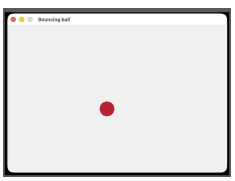
    CloseWindow();
    return 0;
}

void updateBall(Ball &ball, int width, int height)
{
    ball.x = ball.x + ball.velocityX;
    ball.y = ball.y + ball.velocityY;

    // Bounce off the walls by reversing the velocity component
    if (ball.x - ball.radius < 0 or ball.x + ball.radius > width) {
        ball.velocityX = -ball.velocityX;
    }
    if (ball.y - ball.radius < 0 or ball.y + ball.radius > height) {
        ball.velocityY = -ball.velocityY;
    }
}

void drawBall(Ball ball)
{
    DrawCircle(int(ball.x), int(ball.y), ball.radius, MAROON);
}
```

123: A simple animation of a ball bouncing against walls



The physics is the one from secondary school: when the ball touches a wall, the velocity component perpendicular to that wall changes sign.

This example also shows the two ways of passing a parameter side by side:

1. `updateBall(Ball &ball, ...)` receives the ball **by reference** (`&`) because it must **change** it — and those changes must survive back in `main`.
2. `drawBall(Ball ball)` receives it **by value** (a copy) because it only **reads** it to draw. A common C++ refinement is `const Ball &ball`

for read-only access: no copy, and the compiler forbids accidental changes.

B·2.3 Making sound: a little scale

A digital sound is just a list of numbers, the *samples*. If we produce `44100` samples per second following $\sin(2\pi f t)$, the sound card turns them into a tone of frequency f. Changing f changes the note. Here we build each note of the scale do-re-mi-fa-sol and play it.

```cpp
// raylibSound.cpp
// juanfc 2026-09-09
//
// brew install raylib
// c++ raylibSound.cpp $(pkg-config --libs --cflags raylib)

#include "raylib.h"
#include <array>
#include <cmath>
#include <thread>
#include <chrono>

// Prototypes
Sound makeNote(float frequency);

int main()
{
    InitAudioDevice();

    // do      re       mi       fa       sol   (A4 = 440 Hz)
    std::array<float, 5> scale = {261.63f, 293.66f, 329.63f, 349.23f,
    392.00f};

    for (float frequency : scale) {
        Sound note = makeNote(frequency);
        PlaySound(note);
        while (IsSoundPlaying(note)) {
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
        UnloadSound(note);
    }

    CloseAudioDevice();
    return 0;
}

Sound makeNote(float frequency)
{
    const int sampleRate = 44100;
    const int durationMs = 400;                          // 0.4 seconds
    const int frameCount = sampleRate * durationMs / 1000;

    // One sine-wave sample per frame, as 16-bit signed values
    std::array<short, frameCount> samples;
    for (int i = 0; i < frameCount; i = i + 1) {
        float t = float(i) / float(sampleRate);
        samples[i] = short(8000.0f * std::sin(2.0f * PI * frequency *
        t));
    }

    Wave wave;
    wave.frameCount = frameCount;
    wave.sampleRate = sampleRate;
    wave.sampleSize = 16;
    wave.channels   = 1;
    wave.data       = samples.data();

    Sound note = LoadSoundFromWave(wave);   // raylib copies the
    samples in
    return note;
```

```
}
```

Two ideas worth pausing on.

The choice between `for` and `while` is not decorative here. We loop over the five notes with a `for`, because the number of turns is **known in advance**. We wait for a note to finish with a `while`, because we **cannot know** how many turns that will take — it depends on the sound. This is exactly the rule from the chapter on loops.

The wait uses the standard library, `std::this_thread::sleep_for`, and **not raylib**'s own `WaitTime`. The reason is a subtle trap: **raylib**'s timing functions (`GetTime`, and `WaitTime` underneath) are only started up by `InitWindow`. This program opens no window, so **raylib**'s clock never starts, and `WaitTime` would wait forever — you would hear a single note and the program would hang. The standard `sleep_for` does not depend on **raylib**'s clock, so it works whether or not there is a window. The lesson: know **what each call sets up**. `InitAudioDevice` gives you sound, but not the clock.

1) Why does the note loop use `for` while the "wait until finished" loop uses `while`? Could you swap them?
2) Change `durationMs` to make longer notes. Which single line has to change?
3) Add two more notes (la, si) to complete the octave. What frequencies do you need, and does any other line change?

B·3 FLTK: windows, buttons and events

FLTK programs look different from **raylib** ones. You do **not** write a loop. Instead you build the window, tell **FLTK** which of your functions to call when things happen, and then hand control to **FLTK**. This is **event-driven** programming.

B·3.1 A window with a message

```cpp
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main()
{
    Fl_Window *window = new Fl_Window(340, 180, "Hello");
    Fl_Box *box = new Fl_Box(20, 40, 300, 100, "Hello, world!");

    box->box(FL_UP_BOX);
    box->labelsize(28);
    box->labelfont(FL_BOLD);

    window->end();
    window->show();

    return Fl::run();
}
```

Three points that surprise newcomers:

1. The `new` here is **not** the careless raw `new` we avoid elsewhere. **FLTK** owns its widgets: each child (the box) attaches itself to the current window, and the window's destructor deletes its children. Creating a widget on the stack instead would make the window try to delete a stack object — a crash. In **FLTK**, `new` is the contract.
2. `window->end()` means "I have finished adding children to this window". Everything created between the window's constructor and `end()` is attached to it automatically.
3. `Fl::run()` is **FLTK**'s event loop. It is what the game loop was in **raylib**, but here the **library** runs it, not you. It waits for clicks and keystrokes and returns only when the window closes.

B·3.2 A button that reacts

To make something happen on a click we register a **callback**: a function **FLTK** will call for us when the button is pressed. We keep everything the callback needs in a `struct`, so we still use no global variables. We also centre the window on the screen.

```cpp
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
#include <FL/Fl_Button.H>
#include <cstdio>

struct Counter {
    Fl_Box *display;
    int clicks;
};

// Prototypes
void onClick(Fl_Widget *widget, void *data);

int main()
{
    Fl_Window *window = new Fl_Window(340, 180, "Click me");
    Fl_Box *box       = new Fl_Box(20, 30, 300, 60, "No clicks yet");
    Fl_Button *button = new Fl_Button(120, 110, 100, 40, "Click");

    box->box(FL_UP_BOX);
    box->labelsize(20);

    Counter counter = {box, 0};
    button->callback(onClick, &counter);

    window->end();
    window->position((Fl::w() - window->w()) / 2,
                     (Fl::h() - window->h()) / 2);
    window->show();

    return Fl::run();
}

void onClick(Fl_Widget *widget, void *data)
{
    Counter *counter = (Counter *)data;
    counter->clicks = counter->clicks + 1;

    char text[64];
    snprintf(text, sizeof(text), "Clicks: %d", counter->clicks);
    counter->display->copy_label(text);
}
```

The central idea is the **callback**, and it is best understood by contrast with **raylib**. In **raylib you** kept asking, on every turn of the loop, "has the user done anything?" — that is *polling*. In **FLTK** you leave your phone number

and **FLTK** calls **you**: you register `onClick`, and **FLTK** invokes it exactly when the click happens. You no longer control the loop. This hand-over is called *inversion of control*: "don't call us, we'll call you".

Some details that connect back to earlier chapters:

1. The `void *data` is C's way of passing "your own stuff" to a function that does not know its type. We put everything the callback needs in a `struct`, pass its address, and recover it inside with the cast `(Counter *)data`. Note that `counter` lives in `main` for the whole run of `Fl::run()`, so its address stays valid — no dangling pointer.
2. We call `copy_label`, not `label`, on purpose. `label` would store only a **pointer** to `text`, which is local and dies when `onClick` returns — you would be pointing at dead memory. `copy_label` asks **FLTK** to keep its **own copy**. Always ask: who owns this memory?
3. Centring the window reuses the margin formula from the circle example: `Fl::w()` and `Fl::h()` are the screen size, `window->w()` and `window->h()` the window size, and $(T - w)/2$ is the margin. It must come **before** `show()`, because the window takes its position when it is shown.

1) In `onClick`, what would go wrong if we used `label` instead of `copy_label`?

2) The parameter `widget` is unused here. What does **FLTK** pass in it, and when would you need it?

3) Rewrite the centring so the window sits in the **top-right** corner instead of the centre.

Appendix C Writing Style

Contents of this part Book contents

C·1 Great programmers comments 207
C·2 Style and Requirements 209

C·1 Great programmers comments

Before the rules, the spirit. Here is how several of the field's masters describe *clean code*.

These definitions were gathered by Robert C. Martin in *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008. (Martin 2008)

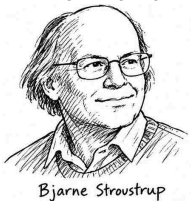
> Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.
>
> — Grady Booch

124: Author of "Object-Oriented Analysis and Design with Applications"



> I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.
>
> — Bjarne Stroustrup

125: The creator of C++ and author of *The C++ Programming Language*



> Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.
>
> — Dave Thomas

126: Founder of OTI and architect of the Eclipse strategy



> I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares.
>
> — Michael Feathers

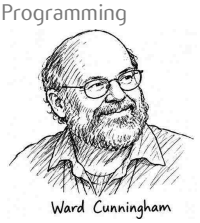
127: Author of *Working Effectively with Legacy Code*



You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.

— Ward Cunningham

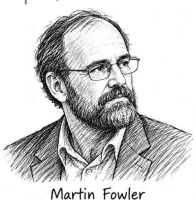
128: Inventor of Wiki and co-inventor of eXtreme Programming



Any fool can write code that a computer can understand. Good programmers write code that humans can understand

— Martin Fowler

129: Developer, 1963-



References

MARTIN, Robert C., 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. ISBN 9780132350884.

C·2 Style and Requirements

The table below summarises the key points covered throughout the course; ◆ indicates prerequisites:

	Style Standards and Requirements (◆ Required)
1	◆ Never declare global variables outside of subprograms
2	◆ Do not interrupt any loops with `return` or `break` statements
3	◆ Only the selection and iteration structures covered in class notes, slides, and demonstrations are permitted (`if`, `switch`, `while`, `do-`while"), `for`, and so on). Other constructs, such as ~~`goto`~~ and ~~`continue`~~, are strictly prohibited
4	◆ The boolean condition in the second section of a `for` loop should depend only on the iteration variable, and the iteration variable must be updated exclusively in the third section of the loop declaration
5	◆ Do not use C-style array declarations such as `int a[…];`. Always use STL `string` and `array<..>`). All `array` declarations must be `typedef`'ed
6	Pay careful attention to the indentation and formatting of your code, preferring the Kernighan/Ritchie style

7	Program structure outline:
	`#include // (from general to specific headers)`
	`const`
	`typedef`
	`int main()  // with prototypes(..) at the beginning of their body`
	`subprogramBodies() // from more abstract to more detailed { with prototypes(..) }`

8	Use `for` loops only when you know before the loop starts how many iterations will occur; apply them to ranges or sequences with well-defined bounds. Do not include conditions within the loop that could change during iteration
9	Use `while` loops for general-purpose iteration, particularly when loop execution depends on the loop body's side effects
10	Use descriptive names for all identifiers, especially for variables with large scope or significant importance throughout the program (`finalLength`, `volume`, `avgPrice`, etc.)
11	Reserve short standard names exclusively for short-lived variables such as loop indices, counters, and similar constructs (`i`, `s`, `n`, etc.)
12	Declare variables in their respective scopes and place them close to their first use, especially for short-lived variables. Avoid grouping variable declarations at the beginning of subprograms
13	Names of new types created with `struct`, `enum`, or `typedef` should begin with the letter T (for example, `TList`, `TStudentIP`)
14	Declare necessary subprogram prototypes at the beginning of each subprogram, before declaring local variables
15	Only subprograms **designed for user interaction** should perform input/output operations via `cin`/`cout`. Subprograms that process parameters and return results, or subprograms not intended for user interaction, should not perform input/output operations
16	Use a single `return` statement only at the end of functions. Multiple `return` statements are permitted only within a clear final `if`/`switch` structure
17	Use the words `and` `or` `not` to combine booleans instead of the less readable characters `&&` `||` `!` (Remember that you can always use `<` `>` `<=` `>` `!=` `==` in comparisons)

o O o