

University of Colombo School of Computing

SCS 1301 • 4C = 3L + 1P

Data Structures and Program Design in C

A Complete Reference & Study Companion

The C language • the gcc / gdb / valgrind toolchain • pointers
and memory • linear data structures • analysis of algorithms

```
struct Node {  
    int data;  
    struct Node *next; /* half the course, in one line */  
};
```

WHAT IS INSIDE

I The C Language

II Pointers, Memory and Aggregates

III The GNU Toolchain

IV Data Structures

V Analysis of Algorithms

A Worked Solutions & Cheat Sheets

Compiled from the SCS 1301 lecture series, tutorial sheets and practical
exercises, with gaps filled from Kernighan & Ritchie, Harvard CS50x, and
Ted Jensen's *A Tutorial on Pointers and Arrays in C*.

How to use this reference

This document is a single, self-contained reference for **SCS 1301 — Data Structures and Program Design in C**. It follows the order of the lecture series, but it is written to be read as a book rather than as slides: every idea that a slide states in one line is expanded here into an explanation, a diagram, a compilable program, and — where the idea is one that students reliably get wrong — a warning.

Where the material comes from

- **Lectures 1–9** supply the syllabus and the order of topics. Nothing in the lectures has been dropped.
- **Exercises I–VII** and **Lab Sheets 1–2** supply the worked problems. The dry-run questions — the ones asking “what does this print?” — are solved step by step in Appendix [A](#), because those are exactly the questions the MCQ paper is built from.
- **Gaps have been filled.** The lecture slides never cover `gdb`, and the Analysis of Algorithms slides are almost entirely images with no text. Chapter [17](#) is a full debugger tutorial and Chapter [26](#) reconstructs the algorithm analysis material in full, including every code example from those slides.
- Additional depth on pointers follows Ted Jensen’s *A Tutorial on Pointers and Arrays in C*; the pedagogical framing of memory and debugging follows Harvard’s CS50x.

How the boxes work

Five kinds of highlighted box appear throughout. They are worth learning to recognise, because they carry most of the material that is not simply “syntax”.

Key idea

The one sentence you should remember from a section. If you are revising in a hurry, read only these.

Common pitfall

A mistake that is easy to make, compiles cleanly, and then misbehaves at run time. Most C bugs live in these boxes.

Note

A detail that is true but easy to overlook — platform dependence, a standard library quirk, an edge case.

Exam focus

Something the SCS 1301 assessment structure makes likely to be asked: a definition to memorise, a distinction to be able to state, or a trace to be able to perform on paper.

Worked example

A complete problem solved from start to finish, usually taken from one of the exercise sheets.

Conventions

Code that you type into a terminal is shown on a dark background with a \$ prompt; the \$ is not part of what you type:

```
$ gcc -Wall -Wextra -g -std=c18 -o hello hello.c
$ ./hello
Hello World
```

C source is shown on a light background with line numbers. Program output, when shown separately from a terminal session, appears in a plain grey box.

Every complete program in this document has been compiled with `gcc -Wall -Wextra -std=c18` and executed; the outputs shown are the outputs actually produced. Where a result depends on the machine — pointer sizes, struct padding, uninitialised memory — this is stated explicitly rather than silently assumed.

A suggested reading path

If you are...	read...
starting the course	Chapters 1–6 in order, typing out every program.
stuck on pointers	Chapter 7, then Chapter 8, then Chapter 23. Read them with Chapter 17 (gdb) open beside you — watching a pointer change in the debugger is worth more than any diagram.
debugging something	Chapter 17, then Chapter 18.
preparing for the paper	The <i>Key idea</i> and <i>Exam focus</i> boxes, Appendix A, and Appendix E.

📖 On typing the programs out

The course forbids IDEs and requires `vim` and `gcc` on the command line. This is not gratuitous. An IDE hides the four compilation stages, hides the link step, and auto-corrects exactly the mistakes that teach you what a declaration is. Type the programs; do not paste them.

Contents

How to use this reference	iii
I The C Language	1
1 Computers, Programs and the C Toolchain	3
1.1 What a computer actually is	3
1.2 Programming languages and their generations	3
1.3 The operating system	4
1.4 Compilation versus interpretation	4
1.5 What programming is	5
1.6 The C programming language	6
1.7 The first C program	7
1.8 The four stages of compilation	8
1.9 Comments	10
1.10 Variables	11
1.11 Chapter summary	12
2 Data Types and the Representation of Values	13
2.1 The three families of types	13
2.2 The primitive types	13
2.3 How integers are represented: two's complement	16
2.4 Floating-point types	17
2.5 Character types and ASCII	17
2.6 Constants	18
2.7 Enumerations	19
2.8 typedef: naming types	20
2.9 Type conversions	21
2.10 Mathematical functions: <math.h>	23
2.11 Timing: <time.h> and clock()	24
2.12 Chapter summary	24
3 Formatted Input and Output	27
3.1 Streams	27
3.2 printf: the anatomy of a format string	27
3.3 Worked example: Exercise I, question 5	29
3.4 Escape sequences	31
3.5 puts and putchar	31
3.6 scanf: formatted input	32
3.7 Reading lines: fgets is the safe choice	35
3.8 Chapter summary	35
4 Operators and Expressions	37

4.1	Assignment versus equality	37
4.2	Arithmetic operators	37
4.3	Relational and logical operators	38
4.4	Bitwise operators	39
4.5	Assignment operators	40
4.6	Operator precedence and associativity	40
4.7	Sequence points and undefined behaviour	41
4.8	Worked example: Exercise II, question 1	43
4.9	Worked example: Exercise II, question 2	44
4.10	Worked example: Exercise II, question 3 — short circuiting	45
4.11	The conditional (ternary) operator	46
4.12	The comma operator	47
4.13	Chapter summary	47
5	Control Flow	49
5.1	Statements and blocks	49
5.2	The if-else statement	50
5.3	The switch statement	51
5.4	Iteration	53
5.5	Jump statements	56
5.6	Chapter summary	57
6	Functions	59
6.1	Why functions exist	59
6.2	Anatomy of a function	59
6.3	The return type	61
6.4	Parameters and arguments	61
6.5	Declaration, definition and prototypes	62
6.6	Call by value and call by reference	62
6.7	Worked examples from Exercise IV	64
6.8	Scope, lifetime and recursion	66
6.9	Chapter summary	66
II	Pointers, Memory and Aggregates	67
7	Pointers	69
7.1	Memory as numbered cells	69
7.2	The address-of operator &	70
7.3	Declaring and using a pointer	70
7.4	The size of a pointer	71
7.5	Pointer arithmetic	72
7.6	Pointers are variables too	73
7.7	Pointers as function parameters	74
7.8	The pointer taxonomy	74
7.9	Chapter summary	80
8	Arrays and Their Relationship with Pointers	81
8.1	What an array is	81
8.2	Properties of arrays	81
8.3	Finding the length of an array	82
8.4	Array-pointer duality	82

8.5	Traversing an array	83
8.6	Passing arrays to functions	84
8.7	Multi-dimensional arrays	85
8.8	Arrays as the basis of other data structures	87
8.9	Worked examples from Exercise IV	87
8.10	Chapter summary	90
9	Strings	91
9.1	C has no string type	91
9.2	Four ways to make a string	91
9.3	The <string.h> library	93
9.4	Implementing the library functions yourself	94
9.5	Reading and printing strings	95
9.6	Arrays of strings	97
9.7	Chapter summary	97
10	Structures, Unions and Bit-fields	99
10.1	Structures	99
10.2	Memory layout, padding and alignment	101
10.3	Unions	102
10.4	Bit-fields	104
10.5	typedef with structures	105
10.6	Arrays of structures and structures as arguments	105
10.7	Chapter summary	106
11	The Preprocessor, Macros and Multi-file Programs	107
11.1	Directives	107
11.2	#include	107
11.3	Macros	107
11.4	The three macro traps	108
11.5	Multi-statement macros	109
11.6	Stringification and token pasting	110
11.7	Conditional compilation and include guards	110
11.8	Writing a C program in multiple files	111
11.9	extern: sharing across files	112
11.10A	A complete multi-file example	113
11.11	Chapter summary	114
12	Storage Classes and the Memory Layout of a Program	115
12.1	The four storage classes	115
12.2	Scope, lifetime and linkage together	117
12.3	The memory layout of a running program	117
12.4	The stack and stack frames	119
12.5	Where does each variable live?	120
12.6	Chapter summary	120
13	Dynamic Memory Allocation	121
13.1	Why static allocation is not enough	121
13.2	size_t	121
13.3	The four allocation functions	122
13.4	A complete demonstration	123
13.5	Dynamic arrays	124

13.6	Memory leaks	125
13.7	Aliasing	126
13.8	Dangling pointers	127
13.9	Correct allocation patterns	128
13.10	Chapter summary	129
14	File Operations	131
14.1	Why file handling exists	131
14.2	The FILE pointer	131
14.3	Opening and closing	131
14.4	The six file modes	132
14.5	Reading from a file	133
14.6	Writing to a file	135
14.7	Appending	135
14.8	Moving around a file	136
14.9	The read/write modes	137
14.10	Binary files	139
14.11	Error handling	140
14.12	Two utility programs	141
14.13	Chapter summary	142
15	Command-Line Arguments	143
15.1	argc and argv	143
15.2	Converting arguments	144
15.3	A realistic argument-handling skeleton	145
15.4	Exit status	146
15.5	Redirection and pipes	146
15.6	Chapter summary	146
III	The GNU Toolchain	147
16	Building Programs with gcc	149
16.1	The flags that matter	149
16.2	Reading compiler messages	151
16.3	Inspecting what was built	152
16.4	Static and dynamic linking	153
16.5	Building a multi-file project by hand	153
16.6	Chapter summary	153
17	Debugging with gdb	155
17.1	Why a debugger	155
17.2	Starting and stopping	155
17.3	Breakpoints	156
17.4	Stepping	157
17.5	Examining data	157
17.6	The call stack	158
17.7	Watchpoints	159
17.8	Worked session 1: a segmentation fault	160
17.9	Worked session 2: inspecting a linked list	161
17.10	TUI mode	162
17.11	Other useful commands	163

17.12	Command reference card	165
17.13	A debugging method	165
17.14	Chapter summary	166
18	Valgrind, Sanitizers and make	167
18.1	Why memory tools exist	167
18.2	Valgrind	167
18.3	AddressSanitizer	168
18.4	make	170
18.5	Chapter summary	171
IV	Data Structures	173
19	Abstract Data Types and the Classification of Data Structures	175
19.1	Classification	175
19.2	Abstract data types	175
19.3	The three fundamental operations	176
19.4	Choosing a structure	176
20	Recursion	179
20.1	The idea	179
20.2	Factorial	179
20.3	Fibonacci — and why naive recursion can be terrible	180
20.4	Recursion versus iteration	181
20.5	Tail recursion	181
20.6	Classical recursive problems	182
20.7	Recursion in this course	183
20.8	Chapter summary	183
21	Stacks	185
21.1	The stack ADT	185
21.2	An array implementation, in separate files	186
21.3	Application: bracket matching	188
21.4	Application: addition using stacks	189
21.5	Application: infix to postfix	190
21.6	Chapter summary	191
22	Queues	193
22.1	The queue ADT	193
22.2	The problem with a naive array queue	193
22.3	A circular array implementation	194
22.4	Priority queues	196
22.5	Dequeues	197
22.6	Stack versus queue	198
22.7	Chapter summary	198
23	Linked Lists	199
23.1	Why arrays are not enough	199
23.2	The node	199
23.3	The linked list	200
23.4	Array versus linked list	200
23.5	Singly linked list operations	200

23.6	A complete singly linked list	202
23.7	Generic insertion and deletion	205
23.8	Stacks and queues from a singly linked list	205
23.9	Doubly linked lists	206
23.10	Circular linked lists	207
23.11	Debugging linked lists	207
23.12	Chapter summary	207
24	Trees and Graphs	209
24.1	Trees	209
24.2	Graphs	211
24.3	Chapter summary	212
25	Searching and Sorting	215
25.1	Searching	215
25.2	Sorting	216
25.3	Comparison	218
25.4	The library sort	219
25.5	Chapter summary	219
V	Analysis of Algorithms	221
26	Analysis of Algorithms	223
26.1	Why study algorithms	223
26.2	Units of measurement	223
26.3	Order of growth	224
26.4	Big-O notation	225
26.5	The other notations	226
26.6	Analysing programs	227
26.7	The ten lecture examples, analysed	227
26.8	The common complexity classes	231
26.9	Space complexity	231
26.10	Amortised analysis	232
26.11	Complexity of the structures in this course	232
26.12	Chapter summary	232
VI	Appendices	233
A	Worked Exercise Solutions	235
A.1	Exercise I — Introduction	235
A.2	Exercise II — Types, operators and expressions	235
A.3	Exercise IV — pointers, functions and arrays	239
A.4	Exercise VI — the memory simulator	240
A.5	Exercise VII — stacks and queues	242
B	Linux Command Reference	243
B.1	Setting up the environment	243
B.2	Navigation and files	244
B.3	Text processing and redirection	245
B.4	Permissions and system information	245
B.5	Lab Sheet 2 — Part B dialogue answers	246

B.6	Lab Sheet 2 — Part C dialogue answers	246
B.7	The C workflow on the command line	247
C	The vim Editor	249
C.1	Modes	249
C.2	Survival commands	249
C.3	Movement (Normal mode)	250
C.4	Editing	250
C.5	Search and replace	250
C.6	Configuration for C	250
C.7	Useful for editing C	251
C.8	Learning more	251
D	Reference Tables	253
D.1	ASCII table (printable characters)	253
D.2	printf conversion specifiers	254
D.3	Escape sequences	254
D.4	Operator precedence (highest first)	255
D.5	Standard library headers	255
D.6	The 32 C keywords (C89) plus later additions	256
D.7	Type sizes on 64-bit Linux (x86-64)	256
E	Cheat Sheets	257
E.1	The build and debug cycle	257
E.2	gdb in ten commands	257
E.3	Complexity	258
E.4	Pointers	259
E.5	Memory	259
E.6	Files	260
E.7	The twenty mistakes that cost marks	260
E.8	Definitions to be able to state	261
	Index	263

PART I

The C Language

Computers, Programs and the C Toolchain

1.1 What a computer actually is

A computer is, in the classical definition, “*a programmable, usually electronic device that can store, retrieve and process data*”. That definition is doing more work than it looks. Three words matter:

programmable

Its behaviour is not fixed in the hardware. The same silicon runs a compiler on Monday and a game on Tuesday, because the instructions it follows are themselves data stored in memory. This is the **stored-program** idea, and it is the reason software exists as a discipline at all.

store and retrieve

It has memory — a very large array of numbered cells. Almost everything in this course is ultimately about how to organise those cells.

process

It has a CPU that can perform a small set of extremely simple operations extremely quickly: add two numbers, compare two numbers, copy a value from one place to another, jump to a different instruction.

Does a computer *understand* what it does? No. It performs a fixed repertoire of primitive operations on patterns of bits. All the meaning — that this particular 32-bit pattern is a student’s mark, and that one is a memory address — exists only in the programmer’s head and in the types the programmer writes down. This is a useful thing to internalise early, because C is a language that will happily let you treat any bit pattern as any type.

Key idea

The only language a computer understands is machine code: binary patterns that the CPU decodes directly. Every programming language is a scheme for producing those patterns without writing them by hand.

1.2 Programming languages and their generations

A **programming language** is a set of constructs that a programmer combines to deliver instructions to a computer. Languages are conventionally grouped into five generations, ordered by distance from the machine.

Gen	Name	Characteristics
1GL	Machine code	Raw binary (or hexadecimal) opcodes. Machine dependent. Executed directly by the CPU with no translation.
2GL	Assembly	Human-readable mnemonics (MOV, ADD, JMP) that map one-to-one onto machine instructions. Translated by an assembler . Still machine dependent.
3GL	Procedural / high-level	English-like keywords, machine independent source. C, C++, Java, Pascal, Fortran. Translated by a compiler or an interpreter .
4GL	Domain-specific	Closer to natural problem statements, usually tied to a domain — SQL for databases, MATLAB for numerics, R for statistics. More productive than 3GLs <i>within</i> their domain.
5GL	Constraint / declarative	The programmer states the constraints and the system searches for a solution — Prolog, Mercury, and the logic programming family. Associated historically with AI research.

C sits squarely in the third generation, but it is unusually close to the second. It is sometimes called a “portable assembler” — it gives you named variables, functions and structured control flow, but it does not hide memory from you, and almost every C construct has a predictable, small translation into machine code.

1.3 The operating system

An **operating system** is software that manages the hardware and software resources of a computer and provides common services to programs. Those services are exactly the things every program needs and no program should have to reimplement:

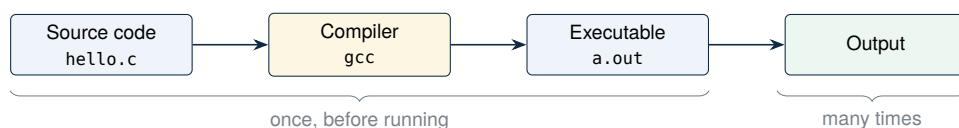
- **File management** — naming, locating, reading and writing persistent storage (Chapter 14).
- **Input/output management** — keyboards, screens, disks, networks, presented through a uniform interface.
- **Process management** — creating, scheduling and terminating running programs.
- **Memory management** — giving each process its own address space and mediating between logical and physical memory (Chapter 7).
- **Error handling and inter-process communication.**

C and UNIX grew up together, which is why C’s standard library reads like a thin layer over an operating system: `fopen`, `malloc` and `exit` are all requests to the OS wearing a C-shaped coat.

1.4 Compilation versus interpretation

1.4.1 Compilation

A **compiler** reads an entire source program and translates it, ahead of time, into a target language — usually the machine code of a specific processor. The result is a standalone executable file. Translation happens once; execution happens many times.

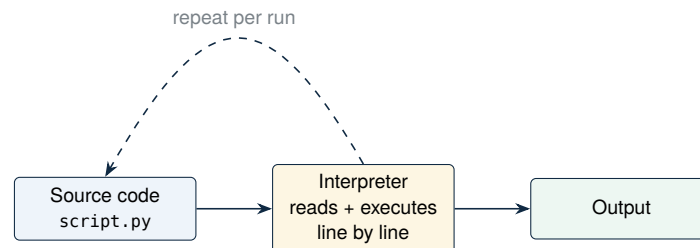


Compiled languages — C, C++, Rust, Go — are fast, because all the work of understanding the source has been done before the program starts. The cost is that the executable is tied to one

processor family and one operating system: an `a.out` built on x86-64 Linux will not run on an ARM Mac.

1.4.2 Interpretation

An **interpreter** reads the source program and executes it directly, statement by statement, without producing a separate executable. Translation and execution are interleaved and both happen every time you run the program.



Python, JavaScript, Perl and Ruby are interpreted. They are slower — the interpreter re-does the parsing work on every run and cannot see the whole program at once to optimise it — but they give the programmer facilities that compiled languages cannot easily offer: inspecting and modifying the program while it runs, immediate feedback with no build step, and no need to know the target machine.

1.4.3 Hybrid implementation

Java is neither. Its compiler (`javac`) translates source to **bytecode** — machine code for a machine that does not exist, the Java Virtual Machine. At run time the JVM interprets that bytecode, and typically compiles hot paths to real machine code as it goes (just-in-time compilation). C# and modern Python implementations work the same way. This category is called **hybrid implementation**, and it buys portability (bytecode runs anywhere a JVM runs) at some cost in start-up time.

✓ Exam focus

Be ready to state the distinction crisply: a compiler translates the whole program ahead of time and produces an executable; an interpreter translates and executes statement by statement at run time; a hybrid compiles to an intermediate bytecode which is then interpreted or JIT-compiled. C is compiled. Java is hybrid. Python is interpreted.

1.5 What programming is

Programming is the act of writing a sequence of instructions, in a programming language, that a computer can follow to accomplish a task. The sequence is called a **program**; the activity is also called coding.

It is worth saying plainly that this is as much a craft as a science. Two programs that produce identical output can differ enormously in how easy they are to read, extend and debug — and the whole point of studying data structures is that they can differ enormously in how fast they run, too.

1.6 The C programming language

C was created by **Dennis Ritchie** at Bell Labs in **1972**. It evolved from two earlier languages, **BCPL** (Basic Combined Programming Language, Martin Richards, 1966) and **B** (Ken Thompson, 1969). Its rise was inseparable from UNIX: when UNIX was rewritten in C in 1973, it became the first operating system written in a high-level language, and therefore the first that could be ported to new hardware by writing a new compiler instead of rewriting the OS.

1.6.1 Why learn C

- **It is everywhere it matters.** Operating system kernels (Linux, Windows, macOS/XNU), embedded and real-time systems, device drivers, databases, language runtimes (CPython, the JVM), and network stacks are written in C or its derivative C++.
- **It is efficient.** C imposes almost no run-time machinery of its own: no garbage collector, no bounds checking, no hidden allocations. What you write is close to what runs.
- **It teaches the machine.** You cannot write C without confronting memory addresses, allocation, lifetime and layout. Every one of those concepts is present in higher-level languages too — C just refuses to hide them.
- **It has a steep learning curve.** This is a genuine cost, and it is mostly concentrated in one place: pointers. Chapters 7 and 8 exist to flatten that curve.

1.6.2 C standards

C was standardised in the United States by ANSI and then internationally by ISO in **1989/1990** (“ANSI C” or C89/C90). Subsequent revisions:

Standard	Year	Notable additions
C89 / C90	1989/90	Function prototypes, <code>void *</code> , the standard library.
C99	1999	// comments, <code>long long</code> , <code>bool</code> via <code><stdbool.h></code> , variable-length arrays, declarations anywhere in a block, <code>inline</code> , <code>restrict</code> .
C11	2011	Multi-threading (<code><threads.h></code>), atomics, anonymous structs and unions, <code>_Static_assert</code> , bounds-checked library functions.
C17 / C18	2018	Bug fixes and clarifications only — no new features. The current document is ISO/IEC 9899:2018 .
C23	2024	<code>true/false/uintptr</code> as keywords, <code>typeof</code> , binary literals, <code>%b</code> .

This course targets **C18**. Select it explicitly:

```
$ gcc -std=c18 -Wall -Wextra -o prog prog.c
```

Note

If you omit `-std`, gcc uses its own default (currently `-std=gnu17`), which enables GNU extensions that are *not* standard C. Code that compiles under `gnu17` may fail elsewhere. Being explicit costs nothing and prevents surprises.

1.7 The first C program

```

1  hello.c
2  #include <stdio.h>
3
4  /* main begins the execution of the program */
5  int main(void)
6  {
7      printf("Hello World!\n");
8      return 0;
9  }

```

Seven lines, and every one of them is worth a paragraph.

#include <stdio.h>

A **preprocessor directive**. Before the compiler proper sees your file, the preprocessor textually replaces this line with the entire contents of the standard input/output header, which contains the *declaration* of `printf`. Without it the compiler does not know what `printf` is.

The comment

Ignored entirely — in fact removed by the preprocessor before compilation. It has no effect on the generated code.

int main(void)

The **entry point**. Execution of every C program begins here. It returns `int` because the program returns a status code to the operating system. `void` in the parameter list means “takes no arguments” — see the note below.

{ ... }

A **block**. It groups the statements that make up the function body.

printf("Hello World!\n");

A call to the standard library function that writes formatted output to the screen. The `f` stands for *formatted*. `\n` is an escape sequence meaning “newline”.

return 0;

Hands control back to the OS with exit status `0`, which by convention means success. Any non-zero value signals an error.

✗ **int main() vs int main(void)**

In C — unlike C++ — an empty parameter list `int main()` does *not* mean “no parameters”. It means “unspecified parameters”, a legacy form that disables argument checking. Always write `int main(void)` when the program takes no command-line arguments, and `int main(int argc, char *argv[])` when it does (Chapter 15).

1.7.1 Building and running it

```

$ gcc -std=c18 hello.c          # produces a.out
$ ./a.out
Hello World!

$ gcc -std=c18 -o hello hello.c # choose the output name
$ ./hello
Hello World!

```

The `./` is not decoration. It tells the shell to look for `hello` in the current directory; without it, the shell searches only the directories listed in `$PATH`, which for good security reasons does not include `..`.

1.8 The four stages of compilation

“Compiling” is really four programs run in sequence. `gcc` is a *driver* that invokes them for you. Understanding the four stages is the single most useful piece of infrastructure knowledge in this course, because every build error you will ever see comes from a specific one of them.

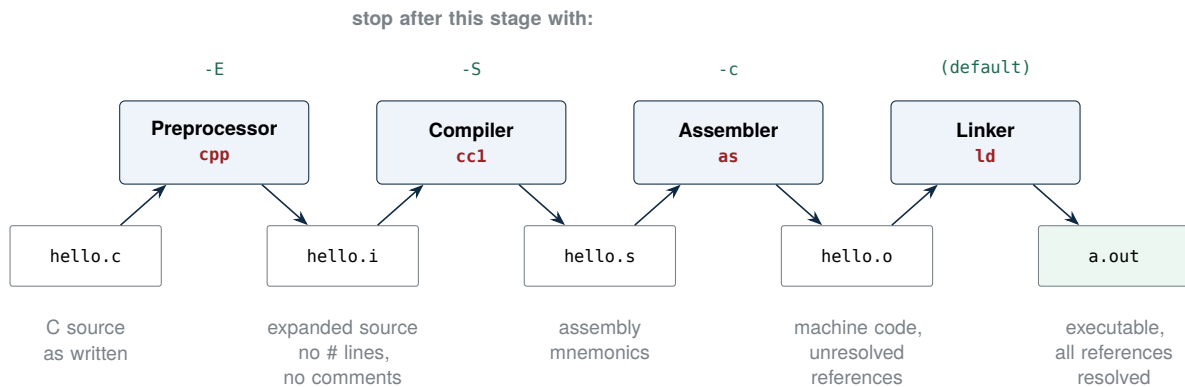


Figure 1.1. The four stages of a C build, the artefact each produces, and the gcc flag that halts the process there.

1.8.1 Stage 1 — the preprocessor

The preprocessor is a text substitution engine. It knows nothing about C grammar. It does exactly three things:

1. Replaces every `#include` with the contents of the named file (recursively).
2. Expands every `#define` macro.
3. Removes comments, and evaluates conditional compilation (`#ifdef`, `#ifndef`, `#if`).

To see its output:

```
$ gcc -E hello.c > hello.i
$ wc -l hello.i
741 hello.i
$ tail -8 hello.i
```

```
# 2 "hello.c" 2

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

Seven lines of source became 741. All but the last handful came from `stdio.h` and the headers it includes in turn. Notice what happened to your program:

- The `#include` line is gone, replaced by hundreds of declarations.

- The comment is gone — **the compiler never sees comments at all**. This answers Exercise I question 3(d): comments are stripped at the preprocessing stage, so they cannot possibly affect the generated code.
- Your actual code is completely unchanged. The preprocessor did not compile anything.
- Lines like `# 2 "hello.c" 2` are **line markers**. They tell the compiler which original file and line the following text came from, so that error messages point at your file and not at line 638 of the expanded soup.

1.8.2 Stage 2 — the compiler

The compiler proper takes the preprocessed source and produces **assembly language** for the target CPU. This is where all the real work happens: lexical analysis, parsing, type checking, optimisation, and code generation. This is also where the vast majority of your error messages come from.

```
$ gcc -S hello.c          # produces hello.s
$ cat hello.s
```

```
.file "hello.c"
.text
.section .rodata
.LC0:
.string "Hello World!"
.text
.globl main
.type main, @function
main:
.LFB0:
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    ret
```

Read it once, slowly; it repays the effort.

- `.LC0` labels the string constant, placed in the read-only data section `.rodata`. This is why string literals must not be modified (Chapter 9).
- `pushq %rbp / movq %rsp, %rbp` is the **function prologue**: it sets up this function's **stack frame** (Chapter 12).
- `leaq .LC0(%rip), %rax` loads the *address* of the string, not the string. `movq %rax, %rdi` puts it in `%rdi`, the register used for the first argument on this platform. Arguments are passed in registers.
- `movl $0, %eax` is return 0 — the return value goes in `%eax`.

Exercise I, question 3(e): where did printf go?

The assembly calls `puts`, not `printf`. The optimiser noticed that the format string contains no conversion specifiers and ends in a newline, and replaced the call with the cheaper, equivalent `puts("Hello World!")` — `puts` appends the newline itself. This substitution is legal because the observable behaviour is identical. Compile with `-fno-builtin` and the call to

`printf` reappears.

1.8.3 Stage 3 — the assembler

The assembler translates the assembly mnemonics into actual machine instructions and produces an **object file**: binary machine code plus a symbol table.

```
$ gcc -c hello.s           # or: gcc -c hello.c (runs stages 1-3)
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, not stripped
$ nm hello.o
0000000000000000 T main
                  U puts
```

`nm` lists the symbols. `T main` means “`main` is defined here, in the text section”. `U puts` means “`puts` is undefined — somebody else must supply it”. The object file is machine code with holes in it.

1.8.4 Stage 4 — the linker

The linker fills the holes. It takes one or more object files, plus the object files it needs from the standard library, resolves every undefined symbol against a definition somewhere, assigns final addresses, and writes an executable.

```
$ gcc hello.o              # produces a.out
$ ./a.out
Hello World!
```

✗ Compiler error or linker error?

Learn to tell them apart from the message alone — it tells you which stage failed and therefore what kind of mistake you made.

error: 'x' undeclared

Compiler. You used a name the compiler has never seen. Missing declaration, typo, or missing `#include`.

warning: implicit declaration of function 'foo'

Compiler. You called a function with no prototype in scope. Almost always a missing header.

undefined reference to 'sqrt'

Linker. The compiler was happy (it saw the prototype in `math.h`) but no object file supplies the code. Add `-lm`.

multiple definition of 'count'

Linker. Two object files both define the same global. Usually a variable defined in a header (Chapter 11).

✓ Exam focus

Memorise the four stages, their tools, their flags and their outputs: `cpp` → `-E` → `.i`; `cc1` → `-S` → `.s`; `as` → `-c` → `.o`; `ld` → (none) → `a.out`.

1.9 Comments

C has two comment forms.

```
/* A block comment.
   It may span multiple lines.
   It is the only form in strict C89. */

// A line comment: everything to the end of the line. Added in C99.
```

Block comments **do not nest**. Writing `/* outer /* inner */ */` terminates at the first `*/`, leaving a stray `*/` that is a syntax error. To disable a region of code that already contains block comments, use `#if 0 ...#endif` instead (Chapter 11).

Comments should explain *why*, not *what*. A comment that restates the syntax is noise:

```
i++;           /* increment i      <- useless */
i++;           /* skip the sentinel <- useful  */
```

1.10 Variables

A **variable** is a named container for a value that may change. In C every variable must be declared with a type before use.

Declaring a variable

```
<data_type> <variable_name>;
<data_type> <variable_name> = <initial_value>;
```

Rules for identifiers:

- May contain letters, digits and underscores.
- Must begin with a letter or an underscore — *not* a digit.
- Are case sensitive: `count`, `Count` and `COUNT` are three different variables.
- May not contain spaces or punctuation such as `!`, `#`, `%`, `$`, `-`.
- May not be a reserved keyword (`int`, `if`, `return`, ...).

Exercise 1, question 1 — which of these are valid?

Candidate	Verdict	Reason
ABC\$13	✗	\$ is not a legal identifier character in standard C.
ACB76	✓	Letters then digits; starts with a letter.
abc123_5	✓	Letters, digits, underscore, starts with a letter.
98test	✗	Starts with a digit.
test item	✗	Contains a space — this is two tokens.
fullCredit	✓	Perfectly legal (camelCase).
abc_ABC	✓	Legal; note it differs from <code>abc_abc</code> .
\$cd	✗	Starts with \$.
int	✗	Reserved keyword.
if-else	✗	- is the subtraction operator, not an identifier character.

Note: gcc does accept \$ in identifiers as a GNU extension, so `ABC$13` may compile on your machine. Under `-std=c18 -pedantic` it is rejected, which is the answer expected here.

✗ Names beginning with an underscore

Identifiers starting with an underscore followed by a capital letter (`_Foo`), and any identifier containing a double underscore (`my__var`), are *reserved for the implementation* at every scope. They are legal to write but you must not use them — they may collide with internals of the compiler or standard library. Start your names with a letter.

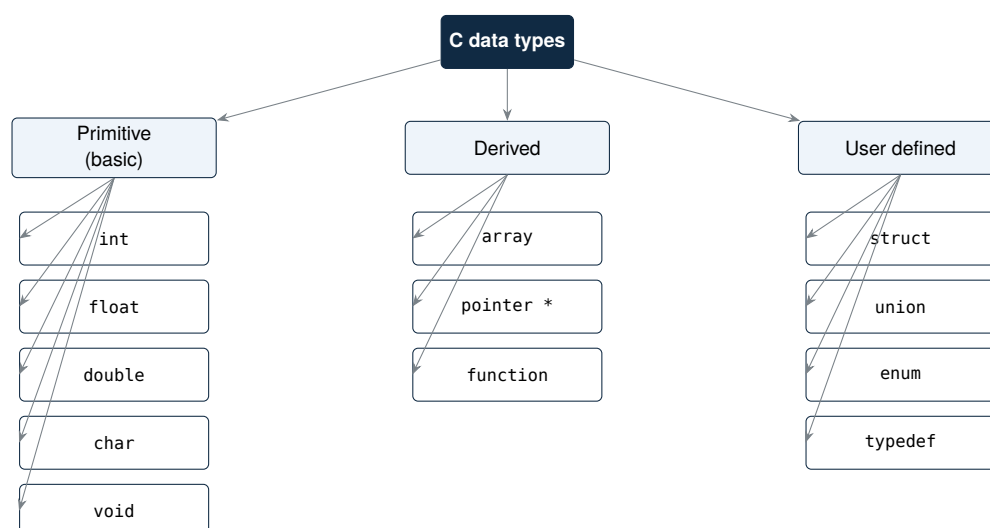
1.11 Chapter summary

- Computers execute machine code; every language is a scheme for producing it. Languages are classed 1GL–5GL by distance from the machine.
- Compilers translate ahead of time to a standalone executable (C); interpreters translate and execute simultaneously (Python); hybrids compile to bytecode then interpret it (Java).
- C was written by Dennis Ritchie in 1972, evolved from BCPL and B, and is standardised as ISO/IEC 9899:2018 (C18).
- A build has four stages — preprocess, compile, assemble, link — stopped by `-E`, `-S`, `-c` and nothing respectively.
- The preprocessor removes comments and expands includes and macros; the compiler produces assembly; the assembler produces object code with unresolved symbols; the linker resolves them.
- Identifiers: letters, digits, underscores; must not start with a digit; case sensitive; not a keyword.

Data Types and the Representation of Values

Every value in a C program is a pattern of bits in memory. A **type** is the answer to two questions about that pattern: *how many bytes is it?* and *how should those bytes be interpreted?* C will let you get either answer wrong, so it is worth being precise.

2.1 The three families of types



Note

`void` is not really a data type — it is the absence of one. Depending on context it means *no value* (a function returns nothing), *no type* (a `void *` points at untyped memory) or *no parameters* (`int main(void)`).

2.2 The primitive types

There are four basic types — `int`, `float`, `double`, `char` — modified by the qualifiers `short`, `long`, `signed` and `unsigned`.

Type	Typical size	Min guaranteed	Range (typical, 64-bit Linux)
char	1 byte	1 byte	−128 to 127 (signedness is implementation defined)
signed char	1 byte	1 byte	−128 to 127
unsigned char	1 byte	1 byte	0 to 255
short	2 bytes	2 bytes	−32,768 to 32,767
unsigned short	2 bytes	2 bytes	0 to 65,535
int	4 bytes	2 bytes	−2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	2 bytes	0 to 4,294,967,295
long	8 bytes	4 bytes	$\approx \pm 9.22 \times 10^{18}$
long long	8 bytes	8 bytes	$\approx \pm 9.22 \times 10^{18}$
float	4 bytes	—	$\approx \pm 3.4 \times 10^{38}$, 6–7 significant digits
double	8 bytes	—	$\approx \pm 1.8 \times 10^{308}$, 15–16 significant digits
long double	16 bytes	—	80-bit extended precision on x86

✗ The standard guarantees ranges, not sizes

C does *not* say `int` is four bytes. It says `sizeof(char) = 1` and `char ≤ short ≤ int ≤ long ≤ long long`, with minimum ranges as above. On a 16-bit embedded target `int` is two bytes. Code that assumes otherwise is not portable. When you need an exact width, use `<stdint.h>`: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`.

2.2.1 Finding the sizes yourself

`sizeof` is not a function — it is a compile-time *operator*. It yields the size in bytes of a type or an expression, as a `size_t`.

sizes.c --- Exercise I, question 2

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("%-20s %zu bytes\n", "char",      sizeof(char));
6      printf("%-20s %zu bytes\n", "short",     sizeof(short));
7      printf("%-20s %zu bytes\n", "int",       sizeof(int));
8      printf("%-20s %zu bytes\n", "long",      sizeof(long));
9      printf("%-20s %zu bytes\n", "long long", sizeof(long long));
10     printf("%-20s %zu bytes\n", "float",     sizeof(float));
11     printf("%-20s %zu bytes\n", "double",    sizeof(double));
12     printf("%-20s %zu bytes\n", "long double", sizeof(long double));
13     printf("%-20s %zu bytes\n", "void *",    sizeof(void *));
14     return 0;
15 }
```

```

char          1 bytes
short         2 bytes
int           4 bytes
long          8 bytes
long long     8 bytes
float         4 bytes
double        8 bytes
long double   16 bytes
void *        8 bytes
```

🔧 Print sizeof with %zu

sizeof yields a `size_t`, which is an *unsigned* type of implementation-defined width. `%d` is wrong and will provoke a warning under `-Wall`; `%zu` is the correct specifier. See Section 13.2.

2.2.2 Limits: <limits.h> and <float.h>

Rather than hard-coding ranges, ask the implementation.

limits.c --- Exercise I, question 4

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <float.h>
4
5  int main(void)
6  {
7      printf("CHAR_BIT   = %d bits per byte\n", CHAR_BIT);
8      printf("CHAR_MIN   = %d   CHAR_MAX   = %d\n", CHAR_MIN, CHAR_MAX);
9      printf("SCHAR_MIN  = %d   SCHAR_MAX  = %d\n", SCHAR_MIN, SCHAR_MAX);
10     printf("UCHAR_MAX   = %u\n", UCHAR_MAX);
11     printf("SHRT_MIN   = %d   SHRT_MAX   = %d\n", SHRT_MIN, SHRT_MAX);
12     printf("INT_MIN    = %d\nINT_MAX    = %d\n", INT_MIN, INT_MAX);
13     printf("UINT_MAX   = %u\n", UINT_MAX);
14     printf("LONG_MIN   = %ld\nLONG_MAX   = %ld\n", LONG_MIN, LONG_MAX);
15     printf("ULONG_MAX  = %lu\n", ULONG_MAX);
16
17     printf("\nFLT_MIN    = %e   FLT_MAX    = %e\n", FLT_MIN, FLT_MAX);
18     printf("FLT_DIG    = %d significant decimal digits\n", FLT_DIG);
19     printf("FLT_EPSILON= %e\n", FLT_EPSILON);
20     printf("DBL_MIN    = %e   DBL_MAX    = %e\n", DBL_MIN, DBL_MAX);
21     printf("DBL_DIG    = %d significant decimal digits\n", DBL_DIG);
22     printf("DBL_EPSILON= %e\n", DBL_EPSILON);
23     return 0;
24 }
```

```

CHAR_BIT   = 8 bits per byte
CHAR_MIN   = -128   CHAR_MAX   = 127
SCHAR_MIN  = -128   SCHAR_MAX  = 127
UCHAR_MAX   = 255
SHRT_MIN   = -32768  SHRT_MAX   = 32767
INT_MIN    = -2147483648
INT_MAX    = 2147483647
UINT_MAX   = 4294967295
LONG_MIN   = -9223372036854775808
LONG_MAX   = 9223372036854775807
ULONG_MAX  = 18446744073709551615

FLT_MIN    = 1.175494e-38   FLT_MAX    = 3.402823e+38
FLT_DIG    = 6 significant decimal digits
FLT_EPSILON= 1.192093e-07
DBL_MIN    = 2.225074e-308   DBL_MAX    = 1.797693e+308
DBL_DIG    = 15 significant decimal digits
DBL_EPSILON= 2.220446e-16
```

Note the asymmetry: `INT_MIN` is `-2147483648` but `INT_MAX` is `2147483647`. There is one more negative value than positive. The next section explains why.

2.3 How integers are represented: two's complement

An n -bit unsigned integer is just binary: the bits are weighted $2^{n-1}, \dots, 2^1, 2^0$. Signed integers use **two's complement**, in which the top bit carries a *negative* weight.



Two's complement has three properties that explain almost everything about integer behaviour in C:

1. **Zero has one representation.** (Sign-and-magnitude and one's complement both have $+0$ and -0 ; two's complement does not.)
2. **Addition and subtraction need no special sign handling.** The same adder circuit works for signed and unsigned values — this is why the representation won.
3. **The range is asymmetric.** With n bits the range is -2^{n-1} to $2^{n-1} - 1$. Hence `INT_MIN` has no positive counterpart, and `-INT_MIN` overflows.

To negate a two's-complement number: invert all bits, then add one.

$$5 = 0000101 \xrightarrow{\text{invert}} 1111010 \xrightarrow{+1} 1111011 = -5$$

✗ Signed overflow is undefined behaviour

Unsigned arithmetic *wraps*: it is defined to be modulo 2^n , so `UINT_MAX + 1 == 0` is guaranteed. Signed overflow is **undefined behaviour** — the compiler is entitled to assume it never happens and to optimise on that basis. A loop like `for (int i = 1; i > 0; i++)` may therefore be compiled into an infinite loop rather than terminating on overflow. Never rely on signed wraparound.

2.3.1 Signed/unsigned comparison — a classic trap

```

trap.c
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = -1;
5     unsigned b = 1;
6     if (a < b) printf("as expected\n");
7     else printf("surprise: -1 is not less than 1\n");
8     return 0;
9 }

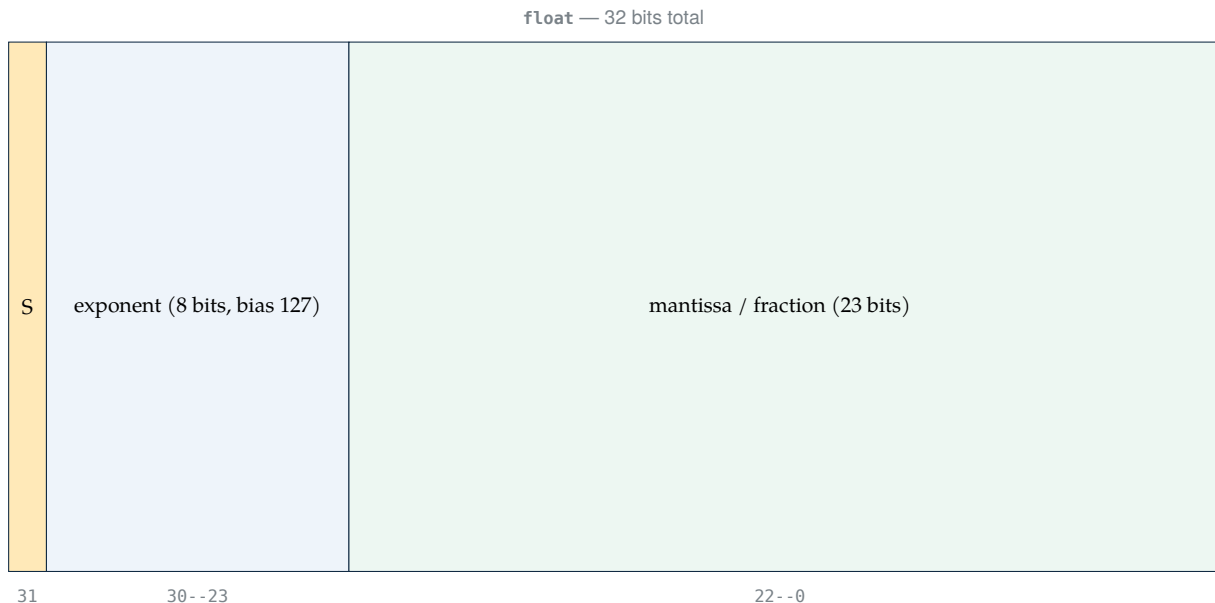
```

```
surprise: -1 is not less than 1
```

When a signed and an unsigned operand of the same rank meet, the *signed* one is converted to unsigned. `-1` becomes `4294967295`, which is not less than `1`. Compile with `-Wall -Wextra` and `gcc` warns: comparison of integer expressions of different signedness. Heed it.

2.4 Floating-point types

float and double store numbers in IEEE 754 format: a sign bit, a biased exponent, and a fractional mantissa.



The value is $(-1)^S \times 1.\text{mantissa} \times 2^{(\text{exponent}-127)}$. Two consequences matter for this course:

- **Precision is finite and relative.** A float carries about 7 significant decimal digits, a double about 16. Beyond that, digits are noise. This is why Exercise I asks you to print 125.456734516547332453425 as a double and observe what survives.
- **Most decimals are not representable.** 0.1 in binary is a recurring fraction, exactly as 1/3 is in decimal. It is stored as the nearest representable value, which is not 0.1.

✖ Never compare floats with ==

```
double x = 0.1 + 0.2;
if (x == 0.3) printf("equal\n");      /* does NOT print */
printf("%.20f\n", x);                 /* 0.30000000000000004441 */
```

Compare with a tolerance instead:

```
#include <math.h>
if (fabs(x - 0.3) < 1e-9) printf("close enough\n");
```

📖 The f suffix

An unsuffixed floating constant such as 3.14 has type double. Writing 3.14f makes it a float. In `float x = 3.14;` the double is computed then narrowed — harmless here, but in expressions it can silently promote the whole computation to double precision.

2.5 Character types and ASCII

A char occupies exactly one byte and holds a single character — but what it really holds is a small integer, the character's code. C uses **ASCII** (American Standard Code for Information In-

terchange), which encodes 128 characters in 7 bits.

Range (dec)	Range (hex)	Contents	Useful fact
0–31	00–1F	control characters	<code>\n=10, \t=9, \0=0</code>
32	20	space	
48–57	30–39	'0'–'9'	'7' - '0' == 7
65–90	41–5A	'A'–'Z'	'A' = 65
97–122	61–7A	'a'–'z'	'a' = 97 = 'A' + 32
127	7F	DEL	

Because characters are integers, arithmetic on them works and is genuinely useful:

```

1  #include <stdio.h>
2  int main(void)
3  {
4      char c = 'A';
5      printf("%c has code %d\n", c, c);           /* A has code 65 */
6      printf("lowercase: %c\n", c + 32);         /* a (65+32 = 97) */
7      printf("or portably: %c\n", c - 'A' + 'a'); /* a */
8
9      char digit = '7';
10     int value = digit - '0';                    /* 55 - 48 = 7 */
11     printf("the character '%c' as a number is %d\n", digit, value);
12
13     for (char ch = 'a'; ch <= 'e'; ch++)
14         printf("%c=%d ", ch, ch);
15     printf("\n");
16     return 0;
17 }

```

```

A has code 65
lowercase: a
or portably: a
the character '7' as a number is 7
a=97 b=98 c=99 d=100 e=101

```

Key idea

'A' (single quotes) is a char — one byte, value 65. "A" (double quotes) is a string literal — two bytes, 'A' followed by the terminating '\0'. They are not interchangeable.

✗ Is char signed or unsigned?

Plain char is a *third* type, distinct from both signed char and unsigned char, and whether it is signed is implementation defined — signed on x86, unsigned on ARM. This matters when a char holds a value above 127 or when it is compared against E0F (which is -1); see Chapter 14. Use unsigned char for raw bytes and int for anything that must hold E0F.

2.6 Constants

A **constant** is a read-only value that cannot be changed after it is defined. C provides two mechanisms, and they work at different stages of the build.

2.6.1 The const qualifier

const

```
const <data_type> <var_name> = <value>;
```

```
const double PI = 3.14159265358979;
const int MAX_STUDENTS = 250;

PI = 3.0;    /* error: assignment of read-only variable 'PI' */
```

A **const** object is a real variable: it has a type, occupies storage, has an address, is visible to the debugger, and is checked by the compiler. It must be initialised at the point of declaration, because you can never assign to it afterwards.

2.6.2 The #define directive

define

```
#define <CONST_NAME> <value>
```

```
#define PI 3.14159265358979
#define MAX_STUDENTS 250
```

This is not a variable at all. It is a **macro**: the preprocessor performs blind textual substitution before the compiler runs. There is no type, no storage, no address, and **no semicolon** — writing `#define PI 3.14;` substitutes the semicolon too, which breaks `area = PI * r * r;` into `area = 3.14; * r * r;`.

	const	#define
Handled by	compiler	preprocessor
Has a type	✓ yes, type-checked	✗ no, plain text
Occupies memory	yes (usually .rodata)	no
Has an address	✓ yes, &PI is valid	✗ no
Respects scope	✓ yes, block scoped	✗ no, from #define to end of file
Visible in gdb	✓ yes	✗ no (unless -g3)
Terminated by ;	yes	never

Key idea

Prefer **const** for constants and **enum** for related integer constants. Reserve **#define** for things a **const** cannot express: include guards, conditional compilation, and array sizes in strict C89.

By convention constants are written in **UPPERCASE**, following the same identifier rules as any variable.

2.7 Enumerations

An **enumeration** is a user-defined type whose values are a fixed list of named integer constants.

enum

```
enum <enum_name> { <option>, <option>, ... };
```

enums.c

```
1  #include <stdio.h>
2
3  enum days { SUN, MON, TUE, WED, THU, FRI, SAT };
4  enum status { OK = 200, NOT_FOUND = 404, SERVER_ERROR = 500 };
5  enum level { LOW = 1, MEDIUM, HIGH }; /* 1, 2, 3 */
6
7  int main(void)
8  {
9      enum days today = WED;
10     printf("today = %d\n", today); /* 3 */
11     printf("SAT = %d\n", SAT); /* 6 */
12     printf("MEDIUM = %d\n", MEDIUM); /* 2 */
13     printf("sizeof(enum days) = %zu\n", sizeof(enum days));
14     printf("sizeof(today) = %zu\n", sizeof(today));
15     return 0;
16 }
```

```
today = 3
SAT = 6
MEDIUM = 2
sizeof(enum days) = 4
sizeof(today) = 4
```

Rules and behaviour:

- Values start at 0 and increment by one unless you assign explicitly. After an explicit assignment, counting resumes from there (hence `MEDIUM == 2`).
- The names must be unique *within their scope* — they are ordinary identifiers in the same namespace as variables, not members of the enum. You cannot have `RED` in two different enums in the same file.
- **What is the size of an enum?** It is implementation defined; the compiler picks an integer type large enough to hold every enumerator. In practice gcc uses `int`, so 4 bytes.
- An enum variable can be assigned any integer — C does not enforce that the value is one of the listed ones. The type documents intent; it does not police it.

 Key idea

Use enums instead of magic numbers. `if (state == RUNNING)` tells the reader what is happening; `if (state == 2)` does not. Enums also make `switch` statements self-documenting, and gcc `-Wall` will warn if you forget to handle one of the enumerators.

2.8 typedef: naming types

`typedef` creates an *alias* for an existing type. It defines no new type — the alias and the original are completely interchangeable.

typedef

```
typedef <existing_type> <new_name>;
```

typedefs.c

```

1  #include <stdio.h>
2
3  typedef int Integer;
4  typedef unsigned char byte;
5
6  typedef enum days { SUN, MON, TUE, WED, THU, FRI, SAT } TEST;
7
8  typedef struct {
9      char name[40];
10     int index;
11 } Student;
12
13 int main(void)
14 {
15     Integer x = 42;
16     byte b = 255;
17     TEST d = FRI;           /* no 'enum' keyword needed */
18     Student s = { "Nimal", 7 };
19
20     printf("%d %u %d %s/%d\n", x, b, d, s.name, s.index);
21     return 0;
22 }

```

```
42 255 5 Nimal/7
```

The value of typedef grows with the complexity of the type. Compare:

```

int (*table[10])(double, char);           /* what is this? */

typedef int (*Handler)(double, char);      /* a Handler is a function pointer */
Handler table[10];                        /* an array of 10 of them */

```

Note

typedef obeys the same syntax as a variable declaration — you write the declaration you *would* write for a variable of that type, and put typedef in front. That is why the new name appears where the variable name would go, which looks backwards until you know the rule.

2.9 Type conversions

When an operator has operands of different types, they are converted to a common type before the operation. This is **type conversion**, also called **typecasting**. There are two kinds.

2.9.1 Automatic (implicit) conversion

The compiler inserts conversions silently, following the **usual arithmetic conversions**: operands are promoted *up* the following ladder until they match.

char / short → int → unsigned → long → long long → float → double → long double

widening: safe, no information lost

Conversions in this direction (narrow to wide) are safe: int → float, float → double. No infor-

mation is lost and the compiler says nothing.

2.9.2 Explicit conversion (the cast)

Converting the other way — wide to narrow — is *not* illegal, but it can lose information. The compiler will warn; a **cast** tells it you meant it.

Cast

```
(<type-name>) <expression>
```

casts.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int    a = 7, b = 2;
5     double bad  = a / b;           /* integer division FIRST -> 3.0 */
6     double good = (double) a / b; /* a widened -> 3.5 */
7
8     double pi   = 3.99;
9     int    lost = (int) pi;        /* truncates toward zero -> 3 */
10
11    int    big   = 300;
12    char   sm    = (char) big;     /* 300 does not fit -> 44 */
13
14    printf("bad=%.2f good=%.2f lost=%d sm=%d\n", bad, good, lost, sm);
15    return 0;
16 }
```

```
bad=3.00 good=3.50 lost=3 sm=44
```

Line by line: `a / b` is `int / int`, so integer division happens *before* the result is widened to double — assigning to a double does not retroactively fix it. Casting *one* operand forces the other to be promoted too. `(int) 3.99` truncates towards zero (it does not round). And `(char) 300` keeps only the low 8 bits: $300 - 256 = 44$.

2.9.3 Why conversions matter — and where they hurt

Why they are useful

Type safety — an explicit cast documents that a narrowing was intended, so reviewers and compilers can distinguish deliberate from accidental.

Readability — `(double) count / total` states the intent to do real division.

Data manipulation — mixed-type arithmetic, generic code via `void *`, and interfacing with libraries all require conversion.

What can go wrong

Loss of precision — `double → float` discards significant digits; `float → int` discards the fraction entirely.

Overflow / underflow — a value that does not fit the narrower type wraps to something unrelated, as with `(char) 300`.

Unexpected behaviour — the signed/unsigned trap in Section 2.3, and integer division appearing where real division was intended.

✓ Exam focus

Two conversion facts are near-certain MCQ material: (i) `int / int` is always integer division regardless of what it is assigned to; and (ii) mixing signed and unsigned converts the signed operand to unsigned, not the reverse.

2.10 Mathematical functions: <math.h>

<math.h> declares the mathematical functions. All trigonometric functions work in **radians**, not degrees.

Function	Meaning	Function	Meaning
sqrt(x)	$\sqrt{x}$	sin(x), cos(x), tan(x)	radians
pow(x,y)	x^y	asin(x), acos(x), atan(x)	inverse, radians
fabs(x)	x for doubles	atan2(y,x)	angle of (x,y), correct quadrant
ceil(x)	round up	exp(x)	e^x
floor(x)	round down	log(x), log10(x)	natural, base 10
round(x)	nearest	fmod(x,y)	floating-point remainder
hypot(x,y)	$\sqrt{x^2 + y^2}$	M_PI	π (a POSIX extension)

✖ You must link the maths library

math.h declares the functions; libm defines them. The compiler therefore succeeds and the *linker* fails:

```
$ gcc -o calc calc.c
/usr/bin/ld: /tmp/cc8Hn2.o: undefined reference to `sqrt'
collect2: error: ld returned 1 exit status

$ gcc -o calc calc.c -lm      # -lm goes AFTER the source files
$ ./calc
```

The position matters: the linker processes its inputs left to right, so a library must appear after the files that use it.

trig.c --- degrees to radians (Exercise II, 6(e))

```
1 #include <stdio.h>
2 #include <math.h>
3
4 #define PI 3.14159265358979323846
5
6 double to_radians(double degrees) { return degrees * PI / 180.0; }
7 double to_degrees(double radians) { return radians * 180.0 / PI; }
8
9 int main(void)
10 {
11     double deg = 45.0;
12     double rad = to_radians(deg);
13     printf("%.1f deg = %.6f rad\n", deg, rad);
14     printf("sin = %.6f cos = %.6f tan = %.6f\n",
15           sin(rad), cos(rad), tan(rad));
16     printf("back again: %.1f deg\n", to_degrees(rad));
17     printf("fmod(10.5, 3.0) = %.4f\n", fmod(10.5, 3.0));
18     return 0;
19 }
```

```
45.0 deg = 0.785398 rad
sin = 0.707107 cos = 0.707107 tan = 1.000000
back again: 45.0 deg
fmod(10.5, 3.0) = 1.5000
```

2.11 Timing: <time.h> and clock()

<time.h> declares types and functions for dates and times. The one this course needs is `clock()`, which returns the **processor time** consumed by the program since it started.

timing.c

```

1  #include <stdio.h>
2  #include <time.h>
3
4  int main(void)
5  {
6      clock_t start = clock();
7
8      volatile double sum = 0.0;          /* volatile: stop the optimiser
9                                          deleting the whole loop */
10     for (long i = 1; i <= 50000000L; i++)
11         sum += 1.0 / i;
12
13     clock_t end = clock();
14     double secs = (double)(end - start) / CLOCKS_PER_SEC;
15
16     printf("sum      = %f\n", sum);
17     printf("ticks    = %ld\n", (long)(end - start));
18     printf("CLOCKS_PER_SEC = %ld\n", (long) CLOCKS_PER_SEC);
19     printf("elapsed  = %.4f s\n", secs);
20     return 0;
21 }
```

```

sum      = 18.997896
ticks    = 175163
CLOCKS_PER_SEC = 1000000
elapsed  = 0.1752 s
```

Key points:

- `clock_t` is an arithmetic type (usually `long`) holding a count of implementation-defined “clock ticks”. **Always** divide by `CLOCKS_PER_SEC` to get seconds; the raw tick count is meaningless across systems.
- It measures *CPU* time, not wall-clock time. A program that sleeps or waits for input accumulates almost no CPU time. For wall-clock elapsed time use `time()` or `clock_gettime()`.
- The resolution is limited, so timing a fast operation once gives noise. Time a loop of many repetitions and divide — this is exactly the experiment Exercise II question 6(f) asks you to run and plot, and its point is to show you why measured time is a poor basis for comparing algorithms. Chapter 26 develops the alternative.

✓ Exam focus

`clock_t` is the type returned by `clock()`; the value must be divided by the macro `CLOCKS_PER_SEC` to yield seconds. Be able to state both.

2.12 Chapter summary

- Types answer two questions: how many bytes, and how to interpret them. Three families: primitive, derived, user-defined.
- The standard guarantees minimum *ranges*, not sizes. Use `sizeof` and `<limits.h>/<float.h>` rather than assuming.

-
- Signed integers use two's complement; the range is asymmetric; unsigned overflow wraps but signed overflow is undefined behaviour.
 - Floating point is approximate. Never compare with `==`.
 - `char` is a one-byte integer holding an ASCII code, so character arithmetic works.
 - `const` is a typed, scoped, addressable read-only variable; `#define` is untyped preprocessor text substitution with no semicolon.
 - `enum` defines named integer constants starting at 0; its size is implementation defined (4 bytes with `gcc`).
 - `typedef` creates an alias, not a new type.
 - Conversions widen automatically and narrow only with an explicit cast; `int / int` is integer division no matter the destination.
 - `math.h` needs `-lm` at link time; trigonometry is in radians.
 - `clock()` returns `clock_t` ticks; divide by `CLOCKS_PER_SEC`.

Formatted Input and Output

Almost every program in this course communicates through two functions, `printf` and `scanf`. They are deceptively deep: the format string is a small language of its own, and most beginners' mysterious bugs — output that looks nothing like what was intended, input that silently refuses to work — come from misunderstanding it.

3.1 Streams

C models all input and output as **streams**: sequences of bytes flowing to or from your program. Three streams are opened automatically:

Stream	Name	Default	Used by
Standard input	<code>stdin</code>	keyboard	<code>scanf</code> , <code>getchar</code> , <code>fgets</code>
Standard output	<code>stdout</code>	screen (line buffered)	<code>printf</code> , <code>putchar</code> , <code>puts</code>
Standard error	<code>stderr</code>	screen (unbuffered)	<code>fprintf(stderr, ...)</code> , <code>perror</code>

The separation of `stdout` from `stderr` is what lets a shell redirect results without losing error messages:

```
$ ./prog > results.txt           # results to the file, errors still on screen
$ ./prog > results.txt 2>&1      # both to the file
```

💡 Why your `printf` sometimes does not appear

`stdout` is *line buffered* when attached to a terminal: output is held until a newline arrives. If your program crashes before printing `\n`, that text is lost — which makes “printf debugging” lie to you about where the crash happened. Either end every debug line with `\n`, call `fflush(stdout)`, or print to the unbuffered `stderr`.

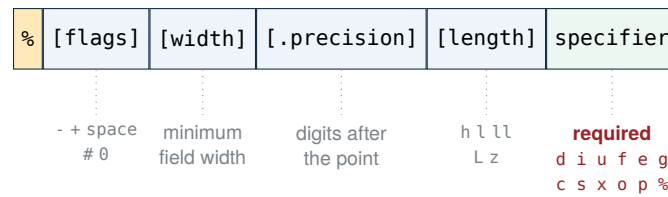
3.2 `printf`: the anatomy of a format string

printf

```
int printf(const char *format, ...);
```

`printf` copies the format string to `stdout` verbatim, except that each **conversion specification** — introduced by `%` — is replaced by the next argument, formatted as directed. It returns the number of characters written.

Every conversion specification has this shape, and only the specifier is mandatory:



3.2.1 Specifiers

Spec	Argument type	Prints
%d, %i	int	signed decimal
%u	unsigned int	unsigned decimal
%o	unsigned int	octal (base 8), no leading 0
%x, %X	unsigned int	hexadecimal, lower / UPPER case
%f, %F	double	decimal notation, 6 decimals by default
%e, %E	double	scientific: 1.234560e+02
%g, %G	double	%e or %f, whichever is shorter; trailing zeros removed
%a, %A	double	hexadecimal floating point (exact, C99)
%c	int (as char)	a single character
%s	char *	characters up to the '\0'
%p	void *	a pointer value, implementation-defined format
%%	—	a literal %
%n	int *	stores the count written so far (avoid: a security hazard)

3.2.2 Flags

Flag	Effect	Example
-	left-justify within the field width	%-10d → "76 "
+	always show a sign, even for positives	%+d → "+76"
space	print a space where + would go	% d → " 76"
#	alternate form: 0 before octal, 0x/0X before hex, always show the decimal point for floats	%#x → "0x4c"
0	pad with leading zeros rather than spaces	%010d → "0000000076"

3.2.3 Width

Width sets the *minimum* number of characters printed.

- If the value needs fewer characters, it is padded (with spaces by default, on the left; with - on the right; with 0 if the 0 flag is given).
- If the value needs *more* characters, the width is ignored entirely — **output is never truncated by width**. This is the single most-missed detail about width.
- A width of * takes the width from the argument list: `printf("%*d", 8, x)`.

3.2.4 Precision

Precision — a dot followed by a number — means different things for different specifiers:

With	.n means
%d %i %u %o %x	the <i>minimum</i> number of digits; the value is zero-padded on the left to reach it. %.5d of 76 → 00076.
%f %e	the number of digits <i>after</i> the decimal point (default 6). %.3f of 125.4567 → 125.457 — note it rounds .
%g	the total number of <i>significant</i> digits.
%s	the <i>maximum</i> number of characters to print — strings, unlike numbers, <i>are</i> truncated. %.3s of "Hello" → Hel.

3.2.5 Length modifiers

The length modifier tells `printf` the actual size of the argument. Getting it wrong is undefined behaviour, because `printf` reads the wrong number of bytes off the argument list.

Modifier	With d/i/u/x/o	With f/e/g
hh	signed char / unsigned char	—
h	short	—
(none)	int	double
l	long	— (%f already takes a double)
ll	long long	—
L	—	long double
z	size_t	—
t	ptrdiff_t	—

✗ %lf in printf vs scanf

In `printf`, float arguments are automatically promoted to double, so `%f` handles both and `%lf` is redundant (though legal since C99). In `scanf` there is no promotion, and the distinction is critical: `%f` means “write to a float *” and `%lf` means “write to a double *”. Using `%f` to read into a double corrupts memory.

3.3 Worked example: Exercise I, question 5

This exercise is entirely about the format string, and it is the best single piece of `printf` practice in the course. Every output below was produced by the program shown.

(a) `double x = 125.456734516547332453425;`

	Required output	Format	Why
i	125.456735	%f	default precision is 6 decimals; the 7th digit rounds the 6th up
ii	125.457	%.3f	three decimals, rounded
iii	+1e+02	%+.0e	scientific, zero decimals, forced sign
iv	1.254567E+02	%.6E	scientific with capital E, six decimals
v	0X1.F5D3B2368E19CP+6	%A	hexadecimal float: exact bit pattern, no rounding

Note (v): `%A` shows the value the machine actually stores. Reading it, $0x1.F5D3B2368E19C \times 2^6 = 1.9602... \times 64 = 125.4567...$ — the mantissa and exponent of Section 2.4 in printable form.

(b) int y = 76;

	Output	Format	Why
i	76	%d	plain decimal
ii	4c	%x	$76 = 4 \times 16 + 12$, and $12 = c$
iii	4C	%X	same, uppercase digits
iv	114	%o	$76 = 1 \times 64 + 1 \times 8 + 4$
v	0x4c	%#x	# adds the 0x prefix
vi	0x4C	0x%X	%#X would give 0X4C with a capital X; to get a lowercase x with uppercase digits, write the
vii	0114	%#o	# adds the leading 0
viii	0000000076	%010d	width 10, zero-padded
ix	76	%10d	width 10, space-padded: 8 spaces then 2 digits

Item (vi) is the interesting one — it is the only case that cannot be done with flags alone, because # ties the case of the prefix to the case of the digits.

(c) date-time from day=30, month=4, year=2024, hour=12, minute=57

	Output	Format string
i	2024-04-30 12:57	"%04d-%02d-%02d %02d:%02d"
ii	30/04/2024 12:57 Hrs	"%02d/%02d/%04d %02d:%02d Hrs"
iii	4.30.2024 12 Hours 57 Minutes	"%d.%d.%d %d Hours %d Minutes"
iv	7e8430-012057	"%x%x%d-%03d%03d"

Format (iv) is a puzzle worth decoding: $2024_{10} = 7e8_{16}$ so the year is %x; the month 4 is also %x (identical to decimal for a single digit); the day 30 is plain %d (as hex it would be 1e); then hour and minute are printed with %03d, giving 012 and 057. Concatenated: 7e8+4+30+ - +012+057.

(d) char c = 'A';

%c prints A; %d prints 65. The same byte, read two ways — which is the entire point of the question.

ex1q5.c --- the complete program

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      double x = 125.456734516547332453425;
6      printf("%f\n%.3f\n%+.0e\n%.6E\n%A\n", x, x, x, x, x);
7
8      int y = 76;
9      printf("%d\n%x\n%X\n%o\n%#x\n0x%X\n%#o\n%010d\n%10d\n",
10             y, y, y, y, y, y, y, y, y);
11
12     int day = 30, month = 4, year = 2024, hour = 12, minute = 57;
13     printf("%04d-%02d-%02d %02d:%02d\n", year, month, day, hour, minute);
14     printf("%02d/%02d/%04d %02d:%02d Hrs\n", day, month, year, hour, minute);
15     printf("%d.%d.%d %d Hours %d Minutes\n", month, day, year, hour, minute);
16     printf("%x%x%d-%03d%03d\n", year, month, day, hour, minute);
17

```

```

18 char c = 'A';
19 printf("%c\n%d\n", c, c);
20 return 0;
21 }

```

3.4 Escape sequences

Inside a character or string literal, a backslash introduces an **escape sequence**.

Escape	Name	Effect
\n	newline	Moves the cursor to the beginning of the next line.
\t	horizontal tab	Moves the cursor to the next tab stop.
\r	carriage return	Moves to the start of the <i>current</i> line without advancing — useful for progress indicators.
\a	alert / bell	Produces an audible or visible alert. Does not move the cursor.
\b	backspace	Moves the cursor back one position.
\\	backslash	Prints one literal backslash.
\"	double quote	A " inside a string literal.
\'	single quote	A ' inside a character literal.
\0	null character	Value 0 — the string terminator.
\101	octal escape	The character with octal code 101 (= 'A').
\x41	hex escape	The character with hex code 41 (= 'A').

Backslash and double quote need escaping precisely because they already have a meaning inside a literal — the backslash starts an escape, and the quote would end the string.

```

printf("A tab:\there\n");
printf("A quote: \" and a backslash: \\ \n");
printf("Path: C:\\Users\\ravin\\main.c\n");
printf("Percent sign: 100%\n");      /* %% not \% */

```

✖ Common pitfall

The escape for a literal percent sign is `%%`, not `\%`. `%` is special to `printf`, not to the C compiler — so it is escaped by `printf`'s rules, not by the compiler's.

3.5 puts and putchar

Two lighter-weight output functions. Exercise II's self-study task asks about `puts` specifically.

<code>int puts(const char *s)</code>	Writes the string <code>s</code> to stdout and appends a newline . Returns a non-negative value on success, EOF on failure. Does no format processing at all.
<code>int putchar(int c)</code>	Writes the single character <code>c</code> . Returns the character written.

```

puts("Hello World");          /* identical output to ... */
printf("Hello World\n");      /* ... this */

puts("100% sure");            /* safe: no format processing */
printf("100% sure");          /* BROKEN: % s is a conversion spec */

```

Similarities with printf: both write to stdout, both are declared in `<stdio.h>`, both return an `int` status. **Differences:** `puts` takes exactly one string argument and never interprets it, whereas `printf` is variadic and interprets `%`. That makes `puts` both faster and safer for fixed messages — which is exactly why the optimiser rewrote `printf("Hello World\n")` into `puts` in Section 1.8.

✗ Never pass user data as a format string

```
printf(user_input);           /* format string vulnerability */
printf("%s", user_input);     /* correct */
```

If `user_input` contains `%s` or `%n`, the first form makes `printf` read arguments that were never passed — reading arbitrary memory, or with `%n`, writing it.

3.6 scanf: formatted input

scanf

```
int scanf(const char *format, &variable, ...);
```

`scanf` — “scan formatted” — reads from `stdin` and stores converted values into the objects its pointer arguments point at. Two things about that sentence do all the damage:

1. You must pass the *address* of each variable, hence `&`. The exception is arrays (including strings), whose names already are addresses.
2. The format string does not describe what you *want*; it describes what `scanf` *expects to find*. Anything that does not match stops the scan.

3.6.1 The return value

`scanf` returns the number of items successfully converted and assigned. Checking it is not optional in correct code.

Returns	Meaning
> 0	The number of values converted and assigned. If you asked for three and got two, the third did not match.
= 0	No value was assigned — the input did not match the format from the very first conversion. The offending characters are still in the buffer.
< 0 (EOF)	A read error occurred, or end-of-file was reached before any conversion.

readint.c --- Exercise IV, question 1(b)

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n;
6     int result = scanf("%d", &n);
7
8     if (result == 1)    printf("You entered %d\n", n);
9     else if (result == 0) printf("Error: that was not an integer\n");
10    else                printf("Error: end of input or read error\n");
11    return 0;
12 }
```

```
$ echo "42" | ./readint
You entered 42
$ echo "hello" | ./readint
Error: that was not an integer
$ ./readint < /dev/null
Error: end of input or read error
```

3.6.2 The two great scanf traps

✖ Trap 1: the leftover newline

```
int age;    char grade;
scanf("%d", &age);    /* you type: 20 <Enter> */
scanf("%c", &grade);  /* reads the '\n' left behind, not your letter */
```

%d consumes the digits but leaves the newline in the buffer. %c — unlike every numeric specifier — does *not* skip whitespace, so it happily reads that newline. **Fix:** put a space before %c. A space in a scanf format means “skip any amount of whitespace here”.

```
scanf(" %c", &grade); /* the leading space fixes it */
```

✖ Trap 2: the infinite loop on bad input

```
while (scanf("%d", &n) != 1)
    printf("Please enter a number: "); /* spins forever */
```

When conversion fails, scanf leaves the offending characters in the buffer. The next call sees the same "abc" and fails again, immediately, forever. You must drain the buffer yourself:

```
int n, c;
while (scanf("%d", &n) != 1) {
    while ((c = getchar()) != '\n' && c != EOF)
        ; /* discard the rest of the line */
    printf("Please enter a number: ");
}
```

3.6.3 Reading the various bases — Exercise IV, question 1(a)

formats.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a;    /* decimal integer */
6     float b;  /* floating point */
7     char c;   /* single character */
8     int d;    /* octal */
9     int e;    /* hexadecimal */
10
11     printf("Enter int float char octal hex: ");
12     int got = scanf("%d %f %c %o %x", &a, &b, &c, &d, &e);
13
14     if (got != 5) { printf("Expected 5 values, converted %d\n", got); return 1; }
```

```

15
16     printf("as given   : %d %f %c %o %x\n", a, b, c, d, e);
17     printf("as integers: %d %d %d %d %d\n", a, (int) b, c, d, e);
18     return 0;
19 }

```

```

$ echo "42 3.5 Z 17 ff" | ./formats
Enter int float char octal hex: as given   : 42 3.500000 Z 17 ff
as integers: 42 3 90 15 255

```

Two things to notice. First, the conversion happens on *input*: `%o` read the characters 17 as octal, giving the integer 15, and `%x` read `ff`, giving 255. Printing them back with `%o` and `%x` converts in the opposite direction, which is why the first output line looks like the input.

Second, the explicit space before `%c` in the format string is what makes this work. `%f` stops as soon as it meets a non-numeric character, leaving the space before `Z` in the buffer. Without the space directive, `%c` would read that space into `c` and everything after it would shift by one. With it, whitespace is skipped and `c` receives `'Z'` as intended.

3.6.4 Scansets

A **scanset** `%[...]` reads only characters belonging to a specified set, stopping at the first character that is not in it. This makes `scanf` into a crude but useful tokeniser.

Scanset	Reads
<code>%[A-Z]</code>	only capital letters; stops at the first non-capital
<code>%[^a]</code>	everything <i>until</i> the first <code>a</code> ; the <code>a</code> is not stored
<code>%[A-Z_a,b,c]</code>	capitals, underscore, and the literal characters <code>a</code> , <code>b</code> , <code>c</code> , <code>,</code>
<code>%[0-9]</code>	only digit characters
<code>%[^\\n]</code>	everything up to the newline — i.e. read a whole line

```

scanset.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char caps[64] = {0}, rest[64] = {0}, line[128] = {0};
6
7      scanf("%[A-Z]", caps);           /* stops at first non-capital */
8      scanf("%[^\\n]", rest);          /* the remainder of the line */
9      scanf("%*c");                   /* discard the newline (* = suppress) */
10     scanf("%[^\\n]", line);          /* a whole second line */
11
12     printf("caps = [%s]\\n", caps);
13     printf("rest = [%s]\\n", rest);
14     printf("line = [%s]\\n", line);
15     return 0;
16 }

```

```

$ printf 'HELL0world123\\nsecond line here\\n' | ./scanset
caps = [HELL0]
rest = [world123]
line = [second line here]

```

The `*` in `%c` is the **assignment suppression** flag: the item is read and discarded, and it does not count towards the return value.

✖ **%s and scansets have no length limit by default**

`scanf("%s", buf)` will write past the end of `buf` if the input is longer — a buffer overflow. Always give a maximum field width, one less than the array size:

```
char buf[32];
scanf("%31s", buf);      /* at most 31 chars + '\0' */
scanf("%31[^\n]", buf);  /* same protection for a scanset */
```

3.7 Reading lines: fgets is the safe choice

For interactive input, most experienced C programmers avoid `scanf` entirely and read a whole line, then parse it.

safeinput.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     char line[128];
7
8     printf("Enter a number: ");
9     if (fgets(line, sizeof line, stdin) == NULL) {
10         printf("no input\n");
11         return 1;
12     }
13
14     char *end;
15     long value = strtol(line, &end, 10);
16
17     if (end == line) printf("that was not a number at all\n");
18     else             printf("value = %ld\n", value);
19     return 0;
20 }
```

This is immune to both `scanf` traps: `fgets` consumes the entire line including the newline, so nothing is left behind, and it can never overflow the buffer because you told it the size.

✖ **Never use gets**

`gets(buf)` reads a line with *no* length limit. It cannot be used safely under any circumstances and was removed from the language in C11. If you see it in old code or on a website, replace it with `fgets`.

3.8 Chapter summary

- Three streams open automatically: `stdin`, `stdout` (line buffered), `stderr` (unbuffered).
- A conversion specification is `[%[flags][width][.precision][length]specifier]`; only the specifier is required.
- Width is a *minimum* and never truncates numbers; precision means decimals for `%f`, minimum digits for `%d`, and a *maximum* for `%s`.

- `puts` writes a string plus a newline and does no format processing; never pass user data to `printf` as a format string.
- `scanf` needs `&` for scalars, returns the count of successful conversions, leaves the newline behind (fix: " %c"), and leaves bad input in the buffer (fix: drain with `getchar`).
- Scansets `%[...]` read only characters from a set; `/*` suppresses assignment; always bound `%s` with a width.
- `fgets` plus `strtoul` is the robust way to read numbers interactively. Never use `gets`.

Operators and Expressions

An **expression** is anything that has a value. An **operator** combines expressions into larger ones. C has an unusually rich operator set — richer than most languages of its era — and a precedence table that repays memorising.

4.1 Assignment versus equality

In C, `=` is the **assignment operator**, not a statement of fact.

Assignment

```
<L.H.S> = <R.H.S>
```

The right-hand side is evaluated and the result stored in the object named on the left. `int a = 5;` puts the value 5 into `a`. To test whether two values are equal, use the **relational operator** `==`.

✗ The single most costly typo in C

```
if (x = 5) { ... }    /* assigns 5 to x, then tests 5, which is true.
                      The branch ALWAYS runs and x is destroyed. */
if (x == 5) { ... }  /* tests whether x equals 5 */
```

This compiles without error because assignment is an expression with a value. `gcc -Wall` warns suggest parentheses around assignment used as truth value. Some programmers write the constant first — `if (5 == x)` — so that the typo becomes a compile error. These are called *Yoda conditions*.

Assignment is an expression, so it has a value: the value assigned. That is why `x = y = z = 4` works — it associates right to left, so `z = 4` evaluates to 4, which is assigned to `y`, and so on.

4.2 Arithmetic operators

Kind	Operators	Notes
Binary	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>	Two operands. <code>%</code> is the remainder (modulo) operator.
Unary	<code>++</code> <code>--</code> <code>+</code> <code>-</code>	One operand. Unary <code>-</code> negates; unary <code>+</code> does nothing except force integer promotion.

4.2.1 Integer division

🔑 Key idea

When *both* operands of `/` are integers, C performs **integer division**: the result is the mathematical quotient with the fractional part discarded (truncated *towards zero*), and the result type is an integer.

```

7 / 2  == 3      /* not 3.5 */
-7 / 2  == -3    /* truncation is toward zero, not toward -infinity */
7 % 2   == 1
-7 % 2   == -1   /* sign of the result follows the DIVIDEND */
7 % -2   == 1
7.0 / 2  == 3.5  /* one operand is double -> real division */

```

The identity that always holds is $a = (a/b) \times b + (a\%b)$, which forces % to take the sign of the left operand.

✖ Common pitfall

% requires **integer** operands. `7.5 % 2` is a compile error. For floating-point remainder use `fmod()` from `<math.h>`.

4.2.2 Increment and decrement

Form	Name	Meaning
<code>++x</code>	prefix (pre-increment)	Increment <i>first</i> , then yield the new value.
<code>x++</code>	postfix (post-increment)	Yield the <i>old</i> value, then increment.

```

int i = 5, j;
j = ++i;    /* i becomes 6, j is 6 */
i = 5;
j = i++;    /* j is 5, then i becomes 6 */

```

As a standalone statement, `i++;` and `++i;` are identical — the value is discarded either way. The distinction only matters when the result is used.

4.3 Relational and logical operators

These operators, placed in expressions, yield the logical truth or falsity of the expression — specifically the `int` value 1 for true and 0 for false.

Relational	Logical
<code>></code> greater than	<code>&&</code> logical AND
<code>>=</code> greater than or equal	<code> </code> logical OR
<code><</code> less than	<code>!</code> logical NOT (unary)
<code><=</code> less than or equal	
<code>==</code> equal to	
<code>!=</code> not equal to	

📖 Truth in C

There is no dedicated boolean type in classic C. **Zero is false; every non-zero value is true.** `-1`, `0.5` and `'a'` are all true. Operators that *produce* a truth value always produce exactly 0 or 1, but operators that *consume* one accept any value.

C99 added `<stdbool.h>`, which defines `bool`, `true` and `false` as macros over the built-in `_Bool`. Using it is good style, but it changes nothing underneath.

4.3.1 Short-circuit evaluation

`&&` and `||` evaluate their left operand first, and evaluate the right operand **only if necessary**.

- `A && B` — if `A` is false, the result must be false, so `B` is never evaluated.
- `A || B` — if `A` is true, the result must be true, so `B` is never evaluated.

This is not an optimisation; it is guaranteed by the standard, and there is a sequence point between the two operands. Programs rely on it:

```
if (p != NULL && p->value > 0)    /* safe: p is only dereferenced if non-NULL */
if (i < n && arr[i] == target)    /* safe: arr[i] only read if i is in range */
if (denominator != 0 && total / denominator > 5)
```

Reverse the order in any of those and the program crashes.

✗ Side effects hidden behind a short circuit

```
if (fast_check() || expensive_update())    /* update may never happen! */
```

If the right operand does real work, that work is conditional. Do not put side effects on the right of `&&` or `||` unless you mean them to be conditional.

4.4 Bitwise operators

C provides six operators for manipulating individual bits. They may be applied only to integer types — `char`, `short`, `int`, `long` and their unsigned variants — never to `float` or `double`.

Op	Name	Result bit is 1 when...
<code>&</code>	bitwise AND	both input bits are 1
<code> </code>	bitwise OR	at least one input bit is 1
<code>^</code>	bitwise XOR	exactly one input bit is 1
<code>~</code>	complement (unary)	the input bit is 0 — every bit is flipped
<code><<</code>	left shift	bits move left; zeros fill in from the right
<code>>></code>	right shift	bits move right

<code>a = 44</code>	0	0	1	0	1	1	0	0
<code>b = 26</code>	0	0	0	1	1	0	1	0
<code>a & b = 8</code>	0	0	0	0	1	0	0	0
<code>a b = 62</code>	0	0	1	1	1	1	1	0
<code>a ^ b = 54</code>	0	0	1	1	0	1	1	0
<code>~a = -45</code>	1	1	0	1	0	0	1	1
<code>a << 1 = 88</code>	0	1	0	1	1	0	0	0
<code>a >> 1 = 22</code>	0	0	0	1	0	1	1	0

4.4.1 Shifts as multiplication and division

`x << n` multiplies by 2^n ; `x >> n` divides by 2^n .

```
1 << 3 == 8      /* 1 * 2^3 */
5 << 2 == 20     /* 5 * 4 */
40 >> 3 == 5     /* 40 / 8 */
```

✗ Right shift of a negative number

For *unsigned* values, `>>` shifts in zeros (a **logical shift**). For *signed negative* values, the standard leaves it implementation-defined; every mainstream compiler performs an **arithmetic shift**, copying the sign bit inwards, which preserves the sign:

```
-8 >> 3 == -1     /* gcc: sign-preserving arithmetic shift */
```

Left-shifting a negative number, or shifting by an amount $\geq$ the type's width, is undefined behaviour. Do bit manipulation on unsigned types.

4.4.2 The standard bit idioms

Idiom	Meaning
<code>x & (1u << n)</code>	test bit <i>n</i> (non-zero if set)
<code>x = (1u << n)</code>	set bit <i>n</i> to 1
<code>x &= ~(1u << n)</code>	clear bit <i>n</i> to 0
<code>x ^= (1u << n)</code>	toggle bit <i>n</i>
<code>x & 1</code>	test whether <i>x</i> is odd
<code>x & (x - 1)</code>	clear the lowest set bit; <code>== 0</code> iff <i>x</i> is a power of two

✗ `&` versus `&&`

`&` operates on every bit and always evaluates both operands. `&&` produces 0 or 1 and short-circuits. They are not interchangeable: `2 & 1` is 0 (false!) but `2 && 1` is 1 (true). This distinction is the whole point of Exercise II question 2.

4.5 Assignment operators

Every binary arithmetic and bitwise operator has a compound assignment form: `+=` `-=` `*=` `/=` `%=` `<=<=` `>=>=` `&=` `^=` `|=`.

`x op= y` is equivalent to `x = x op (y)` — with two refinements that matter:

1. *x* is evaluated only once. This is significant when *x* is complicated: `arr[f(i)] += 1`; calls *f* once, `arr[f(i)] = arr[f(i)] + 1`; calls it twice.
2. The right-hand side behaves as though parenthesised. So `x *= 3 + 2` means `x = x * (3 + 2)`, i.e. `x * 5` — **not** `x * 3 + 2`. This is the first line of Exercise II question 1.

4.6 Operator precedence and associativity

Precedence decides which operator binds more tightly when operators of different kinds meet. **Associativity** decides the order among operators of equal precedence.

Level	Operators	Description	Associativity
1	() [] . -> ++ -- (postfix)	call, subscript, member, postfix	left → right
2	++ -- (prefix) + - (unary) ! ~ * & sizeof (cast)	unary	right → left
3	* / %	multiplicative	left → right
4	+ -	additive	left → right
5	<< >>	shift	left → right
6	< <= > >=	relational	left → right
7	== !=	equality	left → right
8	&	bitwise AND	left → right
9	^	bitwise XOR	left → right
10		bitwise OR	left → right
11	&&	logical AND	left → right
12		logical OR	left → right
13	?:	conditional	right → left
14	= += -= *= /= %= &= ^= = <<= >>=	assignment	right → left
15	,	comma	left → right

✓ The five orderings worth memorising

1. Unary binds tighter than *everything* binary.
2. * / % > + - > shifts > relational > equality.
3. **Bitwise &, ^, | bind looser than the relational and equality operators.** This is a historical accident and a famous source of bugs: `x & 1 == 0` parses as `x & (1 == 0) = x & 0 = 0`. Always parenthesise: `(x & 1) == 0`.
4. && binds tighter than ||.
5. Assignment and the unary operators associate *right to left*; nearly everything else associates left to right.

Statement (f) of Exercise II question 4 — “all operators have left-to-right associativity” — is therefore **false**.

4.7 Sequence points and undefined behaviour

Precedence tells you how an expression is *parsed*. It does **not** tell you the order in which the operands are *evaluated*. C deliberately leaves that order unspecified so compilers can optimise.

Key idea

If a scalar object is modified twice, or is modified once and separately read for a purpose unrelated to computing its new value, without an intervening **sequence point**, the behaviour is **undefined**. (C11 §6.5p2)

Sequence points occur at the end of a full expression (the `;`), between the operands of `&&`, `||`, `?:` and the comma operator, and before a function call is entered.

✗ The lecture's own precedence examples are undefined behaviour

The three expressions posed in Lecture 2 all violate this rule. Compile them and gcc says so:

```
$ gcc -Wall -Wextra -std=c18 -o ex ex.c
ex.c:3:27: warning: operation on 'x' may be undefined [-Wsequence-point]
ex.c:4:26: warning: operation on 'a' may be undefined [-Wsequence-point]
ex.c:5:43: warning: operation on 'x' may be undefined [-Wsequence-point]
```

They still have a defined *parse*, and any particular compiler produces some particular answer, so they are legitimate exercises in precedence. But the answers below are *what gcc 11.4 on x86-64 produced* — they are not guaranteed by the language, and a different compiler may legitimately print something else. Never write code like this.

4.7.1 Lecture 2, Example 1

```
int x = 4, y = 5, z = 2;
int result = x++ * --y + z++ / x;
```

Parse: $((x++) * (--y)) + ((z++) / x)$, since $*$ and $/$ bind tighter than $+$.

Step	Sub-expression	Yields	State after
1	$x++$	4 (old value)	$x=5$
2	$--y$	4 (new value)	$y=4$
3	$4 * 4$	16	
4	$z++$	2 (old value)	$z=3$
5	$2 / x$	$2 / 5 = 0$	(gcc reads the <i>updated</i> x)
6	$16 + 0$	16	

Result: $x=5, y=4, z=3, result=16$. Step 5 is the undefined part: whether x is read as 4 or 5 is exactly what the standard refuses to specify.

4.7.2 Lecture 2, Example 2

```
int a = 3, b = 4, c = 2;
int result = ++a * b-- + c++ - a;
```

Parse: $((++a) * (b--)) + (c++) - a$.

Step	Sub-expression	Yields	State after
1	$++a$	4	$a=4$
2	$b--$	4 (old)	$b=3$
3	$4 * 4$	16	
4	$c++$	2 (old)	$c=3$
5	$16 + 2$	18	
6	$18 - a$	$18 - 4 = \mathbf{14}$	

Result: $a=4, b=3, c=3, result=14$.

4.7.3 Lecture 2, Example 3

```
int x = 5, y = 10, z = 4;
int result = x++ & y | z++ ^ --x;
```

Precedence among the bitwise operators is $& > ^ > |$, so the parse is $((x++) \& y) | ((z++) ^ (--x))$.

Step	Sub-expression	Binary	Yields
1	$x++ \rightarrow 5, x=6$	0101	5
2	$5 \& 10$	$0101 \& 1010$	0
3	$z++ \rightarrow 4, z=5$	0100	4
4	$--x \rightarrow x=5, \text{yields } 5$	0101	5
5	$4 ^ 5$	$0100 ^ 0101$	1
6	$0 1$		1

Result: $x=5, y=10, z=5, \text{result}=1$. Note x ends at 5: it was incremented by $x++$ to 6 and then decremented by $--x$ back to 5.

4.8 Worked example: Exercise II, question 1

ex2q1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x = 2, y, z;
6
7     x *= 3 + 2;
8     printf("%d\n", x);
9
10    x *= y = z = 4;
11    printf("%d\n", x);
12
13    x = y = z;
14    printf("%d\n", x);
15
16    x = x && (y = z);
17    printf("%d\n", x);
18    return 0;
19 }
```

```
10
40
4
1
```

Trace

Line 1

$x *= 3 + 2$. Compound assignment parenthesises its right-hand side, so this is $x = x * (3 + 2) = 2 * 5 = 10$. Beginners expect $2 \times 3 + 2 = 8$; the answer is 10.

Line 2

$x *= y = z = 4$. Assignment associates right to left: $z = 4$ (value 4), then $y = 4$ (value 4), then $x = x * 4 = 10 * 4 = 40$.

Line 3

$x = y = z$. z is 4, so y becomes 4 and x becomes 4.

Line 4

$x = x \ \&\& \ (y = z)$. x is 4, which is non-zero, so the left operand is *true*. $\&\&$ therefore evaluates the right operand: $y = z$ assigns 4, and 4 is also true. True $\&\&$ true yields the int 1. Note the result is 1, not 4 — logical operators normalise to 0 or 1.

4.9 Worked example: Exercise II, question 2

ex2q2.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x, y, z;
6      x = 03; y = 02; z = 01;          /* octal literals: 3, 2, 1 */
7      printf("%d\n", x | y & z );
8      printf("%d\n", x | y & ~z);
9      printf("%d\n", x ^ y & ~z);
10     printf("%d\n", x & y && z);
11
12     x = 1; y = -1;
13     printf("%d\n", !x | x);
14     printf("%d\n", ~x | x);
15     printf("%d\n", x ^ x);
16
17     x <= 3; printf("%d\n", x);
18     y <= 3; printf("%d\n", y);
19     y >= 3; printf("%d\n", y);
20     return 0;
21 }
```

```

3
3
1
1
1
-1
0
8
-8
-1
```

Trace

First note the literals: a leading 0 makes an **octal** constant, so 03, 02, 01 are just 3, 2, 1 — the notation is a red herring designed to make you check.

Expression	Parse	Working	Result
$x \mid y \ \& \ z$	$x \mid (y \ \& \ z)$	$2\&1 = 010\&001 = 0; 3 \mid 0$	3
$x \mid y \ \& \sim z$	$x \mid (y \ \& \ (\sim z))$	$\sim 1 = -2 = \dots 11110; 2\&-2 = 2; 3 \mid 2$	3
$x \wedge y \ \& \sim z$	$x \wedge (y \ \& \ (\sim z))$	$2\&-2 = 2; 3\wedge 2 = 011\wedge 010$	1
$x \ \& \ y \ \&\& \ z$	$(x \ \& \ y) \ \&\& \ z$	$3\&2 = 2; 2 \ \&\& \ 1 = \text{true}$	1
$!x \mid x$	$(!x) \mid x$	$!1 = 0; 0 \mid 1$	1
$\sim x \mid x$	$(\sim x) \mid x$	$\sim 1 = -2 = \dots 11110; \text{OR with } \dots 00001 = \text{all ones}$	-1
$x \wedge x$		anything XOR itself	0
$x \ll= 3$	$x = x \ll 3$	1×2^3	8
$y \ll= 3$	$y = y \ll 3$	-1×2^3	-8
$y \gg= 3$	$y = y \gg 3$	arithmetic shift preserves the sign: $-8/8$	-1

The fourth line is the one to study: $x \ \& \ y$ gives 2, and although 2 is not 1, it is non-zero and therefore *true*, so $\&\&$ yields 1. Bitwise gives you a bit pattern; logical gives you a truth value.

4.10 Worked example: Exercise II, question 3 — short circuiting

ex2q3.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x, y, z;
6      x = y = z = 1;  ++x || ++y && ++z;
7      printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
8
9      x = y = z = 1;  ++x && ++y || ++z;
10     printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
11
12     x = y = z = 1;  ++x && ++y && ++z;
13     printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
14
15     x = y = z = -1;  ++x && ++y || ++z;
16     printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
17
18     x = y = z = -1;  ++x || ++y && ++z;
19     printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
20
21     x = y = z = -1;  ++x && ++y && ++z;
22     printf("X = %d\t, Y = %d\t, Z = %d\n", x, y, z);
23     return 0;
24 }
```

```

X = 2^^I, Y = 1^^I, Z = 1
X = 2^^I, Y = 2^^I, Z = 1
X = 2^^I, Y = 2^^I, Z = 2
X = 0^^I, Y = -1^^I, Z = 0
X = 0^^I, Y = 0^^I, Z = -1
X = 0^^I, Y = -1^^I, Z = -1
```

Trace

Two facts drive everything: `&&` binds tighter than `||`, and starting from `-1` makes `++` produce `0`, which is *false*.

#	Parse	What happens
1	<code>((++x) ((++y) && (++z)))</code>	<code>++x = 2</code> , true. <code> </code> short-circuits: the whole right side is skipped, so <code>y</code> and <code>z</code> are untouched. $\rightarrow 2, 1, 1$
2	<code>((++x) && (++y)) (++z)</code>	<code>++x=2</code> true, so <code>++y=2</code> is evaluated, also true. The <code>&&</code> is true, so <code> </code> short-circuits and <code>z</code> is untouched. $\rightarrow 2, 2, 1$
3	<code>((++x) && (++y)) && (++z)</code>	All three are non-zero after incrementing, so all three are evaluated. $\rightarrow 2, 2, 2$
4	<code>((++x) && (++y)) (++z)</code>	<code>++x = 0</code> , <i>false</i> , so <code>&&</code> short-circuits and <code>y</code> stays <code>-1</code> . The left of <code> </code> is false, so <code>++z</code> is evaluated $\rightarrow 0$. $\rightarrow 0, -1, 0$
5	<code>((++x) ((++y) && (++z)))</code>	<code>++x = 0</code> , false, so the right side is evaluated. Inside it <code>++y = 0</code> , false, so that <code>&&</code> short-circuits and <code>z</code> is untouched. $\rightarrow 0, 0, -1$
6	<code>((++x) && (++y)) && (++z)</code>	<code>++x = 0</code> , false. Everything else is skipped. $\rightarrow 0, -1, -1$

✓ Exam focus

To trace short-circuit questions on paper: (1) parenthesise using precedence — `&&` before `||`; (2) evaluate strictly left to right; (3) at each `&&`, if the left is `0`, stop and skip the right; at each `||`, if the left is non-zero, stop and skip the right; (4) note which increments were therefore never executed.

4.11 The conditional (ternary) operator

C's only three-operand operator.

Ternary

```
<condition> ? <value-if-true> : <value-if-false>
```

```
int max = (a > b) ? a : b;
printf("%d item%s\n", n, (n == 1) ? "" : "s");
```

The condition is evaluated; exactly one of the two branches is then evaluated and becomes the value of the whole expression. There is a sequence point after the condition.

📌 Key idea

The ternary operator is an *expression* — it produces a value and can be used anywhere a value is expected, including inside a `printf` argument or a variable initialiser. `if/else` is a *statement* — it produces no value. That is the real difference, and it is why the ternary operator is not a replacement for `if`.

Used well it is concise; used badly it is unreadable. Nested ternaries in particular should be avoided:

```
/* Do not do this */
grade = (m >= 75) ? 'A' : (m >= 65) ? 'B' : (m >= 55) ? 'C' : (m >= 45) ? 'S' : 'F';
```

4.12 The comma operator

The comma operator evaluates its left operand, discards the result, then evaluates and yields its right operand. There is a sequence point between them. Its only common use is in `for` loops:

```
for (i = 0, j = n - 1; i < j; i++, j--) { ... } /* two counters */
```

Do not confuse it with the commas separating function arguments or declarators — those are punctuation, not operators.

4.13 Chapter summary

- `=` assigns and yields the assigned value; `==` compares. `if (x = 5)` is a bug that compiles.
- `int / int` is integer division, truncating toward zero; `%` takes the sign of the dividend and requires integers.
- Prefix `++x` yields the new value, postfix `x++` the old.
- Zero is false, non-zero is true; logical operators yield exactly 0 or 1.
- `&&` and `||` short-circuit — guaranteed, and relied upon for NULL and bounds guards.
- Bitwise `&`, `|`, `^` act on every bit and bind *looser* than `==`; always parenthesise.
- `x op= y` means `x = x op (y)` with `x` evaluated once.
- Precedence governs parsing, not evaluation order. Modifying an object twice in one expression, or modifying and reading it, is undefined behaviour — `gcc -Wall` reports it as `-Wsequence-point`.
- The ternary operator is an expression; `if` is a statement.

Control Flow

By default a C program executes statements in order, top to bottom. Control flow constructs break that order: they choose between alternatives, repeat work, and jump. Everything a program can do is built from sequence, selection and iteration.

5.1 Statements and blocks

A **statement** is a line of code commanding a specific task. In C every statement is terminated by a semicolon.

```
total = price * quantity;    /* an expression statement */
printf("done\n");           /* a function-call statement */
;                            /* the null statement -- legal, does nothing */
```

A **block** (or compound statement) is a set of statements and declarations grouped by braces so that, syntactically, the whole group counts as a single statement.

```
{
    int temp = a;           /* declarations may appear inside a block */
    a = b;
    b = temp;
}                           /* NO semicolon after the closing brace */
```

Two consequences follow, and both matter:

- Anywhere the grammar allows one statement, a block may be used instead. This is why `if`, `while` and `for` can control either a single statement or many.
- A block introduces a **scope**. Names declared inside it vanish at the closing brace, and may shadow names from outside.

✗ The stray semicolon

```
if (x > 0);                 /* <- this semicolon is the entire if-body */
    printf("positive\n");   /* always executes, whatever x is */

for (i = 0; i < 10; i++);   /* loops doing nothing, ten times */
    sum += i;              /* executes ONCE, after the loop */
```

The null statement is legal, so the compiler cannot object. `gcc -Wall` catches the second case with `-Wempty-body`; consistent indentation catches both.

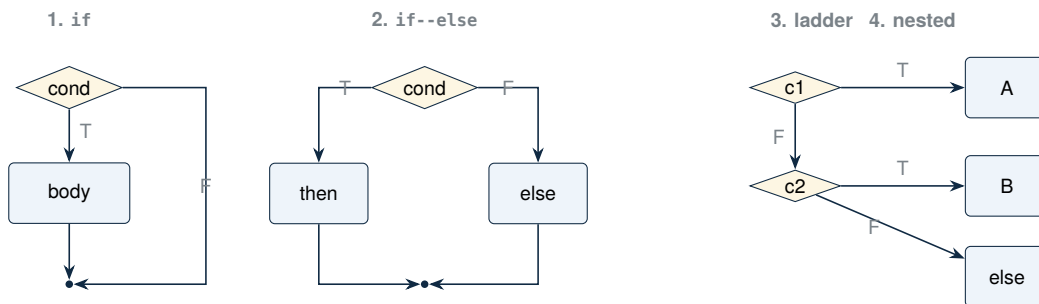
5.2 The if-else statement

if-else

```
if (<expression>)
{
    // code when the expression is true
}
[else
{
    // code when the expression is false
}]
```

The expression is evaluated. If it yields a non-zero value (true), the if block runs; if it yields zero (false), the else block runs if one is present. The else part is optional — a bare if is perfectly correct.

There are four variants.



ladder.c --- the if-else ladder

```
1  #include <stdio.h>
2
3  char grade_of(int marks)
4  {
5      if (marks >= 75) return 'A';
6      else if (marks >= 65) return 'B';
7      else if (marks >= 55) return 'C';
8      else if (marks >= 45) return 'S';
9      else return 'F';
10 }
11
12 int main(void)
13 {
14     int marks[] = { 92, 70, 58, 47, 12 };
15     for (int i = 0; i < 5; i++)
16         printf("%3d -> %c\n", marks[i], grade_of(marks[i]));
17     return 0;
18 }
```

```
92 -> A
70 -> B
58 -> C
47 -> S
12 -> F
```

The ladder is tested top to bottom and stops at the first match, which is why each test only needs a lower bound — reaching `marks >= 65` already guarantees `marks < 75`.

✖ The dangling else

```
if (a > 0)
    if (b > 0)
        printf("both positive\n");
else
    printf("a is not positive\n");    /* misleading indentation! */
                                     /* WRONG: binds to the INNER if */
```

An else always binds to the nearest unmatched if, regardless of indentation. Braces remove the ambiguity:

```
if (a > 0) {
    if (b > 0) printf("both positive\n");
} else {
    printf("a is not positive\n");
}
```

🔧 Always use braces

Even for a one-line body. The cost is two characters; the benefit is that adding a second statement later cannot silently change the meaning of the program. This is not a style preference — it is the cause of real security bugs, most famously Apple’s 2014 “goto fail” TLS vulnerability.

5.2.1 Uses and problems

General use	Possible problems
Decision-making on an arbitrary condition — any expression, any types, ranges as well as equality.	Deep nesting destroys readability; each level adds indentation and a case the reader must hold in mind.
Complex decisions by combining conditions with &&, , ! and by chaining into a ladder.	Code duplication when the same logic appears in several branches — a sign the structure is wrong.

A useful antidote to nesting is the **guard clause**: handle the exceptional cases first and return early, leaving the main path unindented.

```
/* nested */
if (fp != NULL) {
    if (n > 0) {
        if (buf != NULL) {
            process(fp, buf, n);
        }
    }
}

/* guard clauses */
if (fp == NULL) return -1;
if (n <= 0) return -2;
if (buf == NULL) return -3;
process(fp, buf, n);
return 0;
```

5.3 The switch statement

switch is a multi-way decision that tests whether an expression matches one of several *constant integer* values.

switch

```

switch (<expression>)
{
    case constant-expression: statements
    case constant-expression: statements
    default: statements
}

```

Rules:

- The controlling expression must have an **integer** type — int, char, enum. Not float, not double, not a string.
- Each case label must be a *compile-time constant*. You cannot write case x: or case n > 5:.
- Labels must be distinct.
- default handles everything unmatched. It is optional and may appear anywhere, though it is conventionally last.

5.3.1 Fall-through

Key idea

If the expression matches a case, execution starts at that case and **falls through** every subsequent case until a break (or the end of the switch) is reached. The case labels are entry points, not mutually exclusive branches.

fallthrough.c

```

1  #include <stdio.h>
2
3  void classify(char c)
4  {
5      switch (c) {
6          case 'a': case 'e': case 'i': case 'o': case 'u':
7              printf("%c: vowel\n", c);
8              break;
9
10         case '0': case '1': case '2': case '3': case '4':
11         case '5': case '6': case '7': case '8': case '9':
12             printf("%c: digit\n", c);
13             break;
14
15         case ' ':
16             printf("space\n");
17             /* falls through deliberately */
18         case '\t':
19             printf(" (whitespace)\n");
20             break;
21
22         default:
23             printf("%c: something else\n", c);
24     }
25 }
26
27 int main(void)
28 {
29     classify('e'); classify('7'); classify(' '); classify('z');
30     return 0;

```

31 }

```
e: vowel
7: digit
space
  (whitespace)
z: something else
```

Grouping several labels with no code between them (the vowels, the digits) is the good use of fall-through. Falling through *after* executing code is the dangerous use — always mark it with a comment, since a reader cannot otherwise tell a deliberate fall-through from a forgotten break.

✖ The forgotten break

This is the classic switch bug and near-certain examination material:

```
switch (choice) {
  case 1: printf("one\n");      /* no break! */
  case 2: printf("two\n"); break;
}
```

Input 1 prints *both* one and two. Compile with `-Wimplicit-fallthrough` to be warned.

✖ Declarations inside a switch

```
switch (n) {
  case 1:
    int x = 5;      /* error: a label can only be part of a statement */
    break;
}
```

A case label must be followed by a statement, and a declaration is not a statement. Wrap the case body in a block: `case 1: { int x = 5; ... } break;`

5.3.2 switch versus the if–else ladder

	switch	if–else ladder
Tests	equality against integer constants only	any condition at all
Types	integer and char only	anything, including double and strings via <code>strcmp</code>
Compiled to	often a jump table — $O(1)$ regardless of the number of cases	a sequence of tests — $O(n)$
Readability	excellent for menus and state machines	better for ranges and compound conditions

5.4 Iteration

Iterative statements, or loops, repeat a body. C has three, and all three have the same three components:

Loop condition	the test that decides whether to continue
Loop counter	the variable that changes so that the condition eventually becomes false
Loop body	the work being repeated

5.4.1 The while loop

```

while
while (<expression>)
{
    // code executed while the expression is true
}

```

The condition is tested *before* each iteration, so the body may run **zero** times.

```

int i = 1;                /* initialisation */
while (i <= 5) {          /* condition   */
    printf("%d ", i);
    i++;                  /* update -- forget this and it loops forever */
}

```

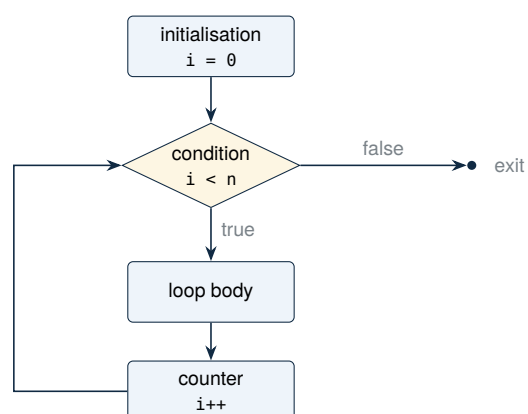
5.4.2 The for loop

```

for
for (<initialization>; <condition>; <counter>)
{
    // loop body
}

```

for gathers the three components into one line, which is why it is preferred whenever the number of iterations is known.



```

for (int i = 0; i < n; i++) { ... } /* C99: declare i in the loop */

for (;;) { ... } /* all three parts optional: infinite */

for (i = 0, j = n-1; i < j; i++, j--) /* comma operator: two counters */

```

Declaring the counter inside the for (a C99 feature) limits its scope to the loop, which prevents a whole class of bugs caused by reusing a stale counter.

5.4.3 The do-while loop

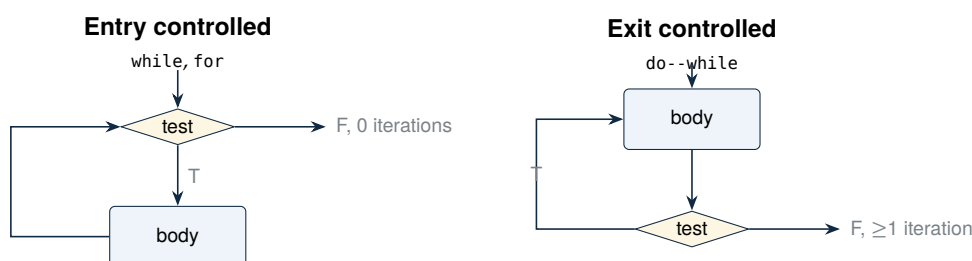
do-while

```
do
{
    // loop body
} while (<expression>;
```

The condition is tested *after* the body, so the body always runs **at least once**. Note the mandatory semicolon after the closing parenthesis — the only loop that has one.

```
int choice;
do {
    print_menu();
    scanf("%d", &choice);
    handle(choice);
} while (choice != 15);    /* menu is shown at least once */
```

5.4.4 Entry-controlled and exit-controlled loops



✓ Exam focus

while and for are **entry controlled** — the test comes first, so the body may execute zero times. do-while is **exit controlled** — the test comes last, so the body executes at least once. This distinction is standard examination material.

5.4.5 Choosing a loop

Use	When
for	the number of iterations is known, or you are walking an index or a pointer over a range
while	you repeat until a condition changes, and it may already be satisfied — reading until EOF, searching, converging
do-while	the body must run at least once — menus, input validation, retry loops

✗ Off-by-one and the fencepost

```
int a[10];
for (int i = 0; i <= 10; i++) a[i] = 0;    /* writes a[10] -- out of bounds */
for (int i = 0; i < 10; i++) a[i] = 0;    /* correct */
```

A C array of size n has valid indices 0 to $n-1$. The idiom `for (i = 0; i < n; i++)` is correct precisely because `<` excludes n . C does no bounds checking, so the first loop compiles, runs, and quietly corrupts whatever follows `a` in memory.

5.5 Jump statements

C has four jump statements: `break`, `continue`, `goto` and `return`.

5.5.1 `break`

`break` terminates the **immediately enclosing** loop or switch block. Control passes to the statement following that block. Outside a loop or switch it is a compile error.

```
for (int i = 0; i < n; i++) {
    if (arr[i] == target) { found = i; break; }    /* stop searching */
}
```

✗ `break` escapes only one level

```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (grid[i][j] == target) break;    /* leaves the INNER loop only */
```

C has no labelled `break`. Use a flag, factor the search into a function and `return`, or — the one respectable use of `goto` — jump to a label after both loops.

5.5.2 `continue`

`continue` skips the rest of the current iteration and transfers control to the next one. Code after the `continue` in the loop body is not executed.

```
for (int i = 0; i < n; i++) {
    if (arr[i] < 0) continue;    /* skip negatives */
    sum += arr[i];
}
```

✗ `continue` in a `while` skips the update

```
int i = 0;
while (i < n) {
    if (arr[i] < 0) continue;    /* i is never incremented -> infinite loop */
    sum += arr[i];
    i++;
}
```

In a `for` loop, `continue` still runs the counter expression, so the equivalent `for` version is safe. This asymmetry is worth remembering.

5.5.3 `goto`

`goto` is an explicit, unconditional jump to a label within the same function.

goto

```
goto <label-name>;
...
<label-name>:
```

```

for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
        if (grid[i][j] == target) goto found;

printf("not found\n");
goto done;

found:
    printf("found at (%d,%d)\n", i, j);
done:
    return 0;

```

goto cannot jump into or out of a function, and cannot jump into the scope of a variable-length array. Its use is heavily discouraged — Dijkstra’s 1968 letter *Go To Statement Considered Harmful* is one of the most cited papers in computing — because arbitrary jumps make it impossible to reason about a program by reading it linearly. Two uses survive in practice: breaking out of nested loops, and the single-exit error cleanup pattern used throughout the Linux kernel:

```

int process(void)
{
    FILE *fp = NULL;  char *buf = NULL;
    fp = fopen("data.txt", "r");
    if (!fp) goto fail;
    buf = malloc(1024);
    if (!buf) goto fail_close;
    /* ... work ... */
    free(buf); fclose(fp);
    return 0;

fail_close: fclose(fp);
fail:      return -1;
}

```

5.5.4 return

return terminates the current function and passes control back to the caller, optionally with a value. It is covered in Chapter 6. return from main ends the program.

Statement	Transfers control to	Legal in
break	the statement after the enclosing loop or switch	loop, switch
continue	the next iteration of the enclosing loop	loop only
goto	a label in the same function	anywhere
return	the caller	any function

5.6 Chapter summary

- Statements end with `;`; blocks are braced groups that count as one statement and introduce a scope. A stray `;` after `if` or `for` silently becomes the body.
- `if-else` has four variants: `if`, `if-else`, `nested`, and the `ladder`. `else` binds to the nearest unmatched `if`. Always use braces.
- `switch` tests integer constants only, and **falls through** from a matched case until a `break`.
- Three loops, each with condition, counter and body. `while` and `for` are entry controlled (possibly zero iterations); `do-while` is exit controlled (at least one).

- Four jump statements: `break` leaves the innermost loop or switch, `continue` starts the next iteration, `goto` jumps to a label, `return` leaves the function.
- Valid array indices are 0 to $n-1$; the standard loop uses $<$, never $<=$.

Functions

A **function** is a named block of functionality that takes inputs and produces an output. Neither is compulsory — some functions take no inputs, some produce no output, some do neither. What every function does provide is a way to give a piece of logic a name and reuse it.

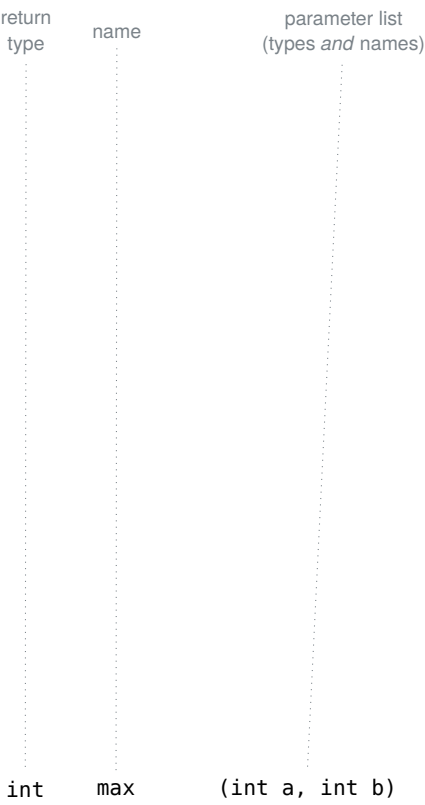
6.1 Why functions exist

- **Reuse.** Write the logic once, call it from many places.
- **Decomposition.** A large problem becomes a set of small ones, each small enough to hold in your head.
- **Abstraction.** The caller needs to know *what* a function does, not *how*. This is the same idea that makes an abstract data type possible (Chapter 19).
- **Testability.** A function with a clear contract can be tested alone.

6.2 Anatomy of a function

Function definition

```
<return_type> <function_name>([<params>,*])
{
    // function body
}
```



```
{ return (a > b) ? a : b; }
```

6.3 The return type

A function may return any type: a primitive, a pointer, a struct, an enum, a typedef name. It may not return an array or a function (though it may return a pointer to either).

When a function returns nothing, its return type is void:

```
void print_banner(void)
{
    printf("=====\n");
    return;           /* optional in a void function */
}
```

The three meanings of void

Context	Example	Meaning
return type	void f(int x)	no value is returned
parameter list	int f(void)	no parameters are accepted
pointer target	void *p	no type — a generic pointer

✗ Falling off the end of a non-void function

```
int bad(int x)
{
    if (x > 0) return 1;
    /* no return when x <= 0 -- the caller receives garbage */
}
```

This is undefined behaviour if the caller uses the value. gcc -Wall reports control reaches end of non-void function. The one special case is main, where reaching the closing brace is defined to be equivalent to return 0; — but write it anyway.

6.4 Parameters and arguments

Key idea

Parameters are the variables named in the function *definition*. **Arguments** are the actual values supplied at the *call*. In this course's terminology, arguments passed by the caller are **actual parameters**, and the variables that receive them inside the function are **formal parameters**.

```
int add(int a, int b) { return a + b; } /* a and b: formal parameters */
...
int s = add(3, x + 1);                 /* 3 and x+1: actual parameters */
```

Two rules that surprise people coming from other languages:

- **Parameters are not mandatory.** int f(void) is a complete, legal function.
- **Parameters cannot be optional.** C has no default arguments and no overloading. A function taking three parameters must be called with exactly three arguments of compatible types.

6.5 Declaration, definition and prototypes

A **definition** provides the body. A **declaration** (or **prototype**) announces the name, return type and parameter types without a body, so the compiler can check calls that appear before the definition.

```

1  prototype.c
2  #include <stdio.h>
3
4  double area(double r);           /* prototype -- note the semicolon */
5  void banner(void);
6
7  int main(void)
8  {
9      banner();
10     printf("%.4f\n", area(2.0));  /* legal: the compiler has seen the prototype */
11     return 0;
12 }
13
14 double area(double r) { return 3.14159265358979 * r * r; } /* definition */
15 void banner(void) { printf("--- areas ---\n"); }
```

```

--- areas ---
12.5664
```

Without the prototype, a call appearing before the definition provokes warning: implicit declaration of function 'area' — and, worse, the compiler assumes it returns `int`, so the `double` return value is misread. In C99 and later this is an outright constraint violation. Prototypes are what header files contain (Chapter 11).

Note

In a prototype the parameter *names* are optional and purely documentation: `double area(double);` is equivalent to `double area(double r);`. Include them anyway — they tell the reader what the arguments mean.

6.6 Call by value and call by reference

This is the most important section in the chapter, and the bridge to Chapter 7.

6.6.1 Call by value

Key idea

C passes **everything** by value. When a function is called, each actual parameter is *copied* into the corresponding formal parameter. Two copies now exist. Changes the function makes affect only its own copy; the caller's variables are untouched.

```

1  swap.c
2  #include <stdio.h>
3
4  void swap_wrong(int a, int b)    /* receives COPIES */
5  {
6      int t = a;  a = b;  b = t;    /* swaps the copies */
7  }
```

```

7
8 void swap_right(int *a, int *b)    /* receives ADDRESSES */
9 {
10     int t = *a; *a = *b; *b = t; /* swaps what they point at */
11 }
12
13 int main(void)
14 {
15     int x = 10, y = 20;
16
17     swap_wrong(x, y);
18     printf("after swap_wrong : x=%d y=%d\n", x, y);
19
20     swap_right(&x, &y);
21     printf("after swap_right: x=%d y=%d\n", x, y);
22     return 0;
23 }

```

after swap_wrong : x=10 y=20
after swap_right: x=20 y=10

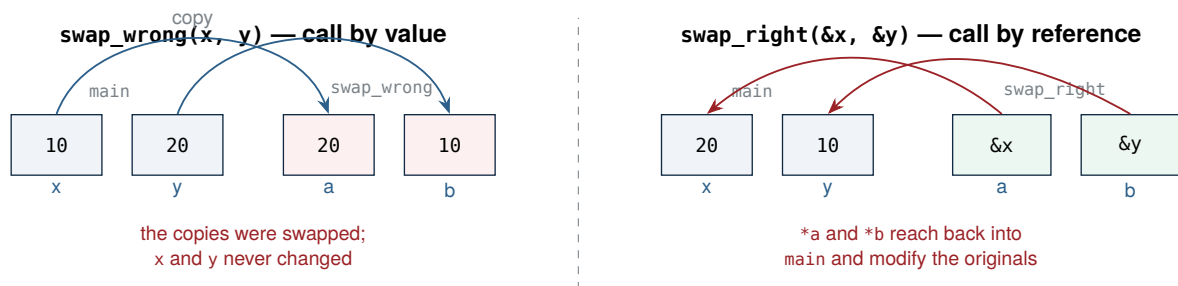


Figure 6.1. Call by value copies the values; call by reference copies the *addresses*, which is what lets the function reach back into the caller.

6.6.2 Call by reference

C has no reference types (those are a C++ feature). What is called “call by reference” in C is call by value applied to a *pointer*: the address is copied, but both copies point at the same object.

	Pass by value	Pass by reference (via pointer)
What is copied	the value itself	the address of the value
Copies in memory	two independent copies exist	one object, two names
Effect on caller	actual variables do not change	actual variables do change if modified
Safety	safer — the function cannot corrupt the caller's data	riskier — requires a valid, non-NULL pointer
Cost	copies the whole object; expensive for large structs	copies one pointer (8 bytes) regardless of object size
Preferred when	the object is small and should not be modified	the object is large, or the function must modify it, or it must return several values

Key idea

Passing a pointer is the *only* way a C function can (a) modify its caller's variables, or (b) return more than one value.

```

/* returning two values through pointer parameters */
void min_max(const int a[], int n, int *min_out, int *max_out)
{
    *min_out = *max_out = a[0];
    for (int i = 1; i < n; i++) {
        if (a[i] < *min_out) *min_out = a[i];
        if (a[i] > *max_out) *max_out = a[i];
    }
}
...
int lo, hi;
min_max(data, 100, &lo, &hi);

```

📌 Arrays are the apparent exception

An array argument does not get copied. The array name *decays* to a pointer to its first element, so the function receives an address — which is why functions can modify array contents without any `&`. The pointer itself is still passed by value. Chapter 8 explains decay in full.

6.7 Worked examples from Exercise IV

6.7.1 Power

power.c --- Exercise IV, 3(a)

```

1  #include <stdio.h>
2
3  long power(int base, int exp)
4  {
5      long result = 1;
6      for (int i = 0; i < exp; i++)
7          result *= base;
8      return result;
9  }
10
11 int main(void)
12 {
13     printf("power(3,4) = %ld\n", power(3, 4));
14     printf("power(2,10) = %ld\n", power(2, 10));
15     return 0;
16 }

```

```

power(3,4) = 81
power(2,10) = 1024

```

📌 Note

The exercise states that 3 to the power 4 should give 81, which it does. Note that `pow()` from `<math.h>` returns a double and can give 80.999999 for integer inputs on some platforms — for integer powers, write the loop.

6.7.2 Odd or even

```
int is_even(int n) { return (n % 2) == 0; }
```

Use $(n \% 2) == 0$ rather than $n \% 2 == 1$: for negative odd numbers, $n \% 2$ is -1 , not 1 , so the second test fails. The bitwise form $!(n \& 1)$ is correct for all values and is what a compiler generates anyway.

6.7.3 Greatest common divisor, three ways

gcd.c --- Exercise IV, 3(c)

```
1  #include <stdio.h>
2
3  /* (i) brute force: try every divisor from 1 to min(a,b) */
4  int gcd_loop(int a, int b)
5  {
6      int g = 1;
7      int m = (a < b) ? a : b;
8      for (int i = 1; i <= m; i++)
9          if (a % i == 0 && b % i == 0)
10             g = i;
11     return g;
12 }
13
14 /* least common multiple, by brute force */
15 int lcm(int a, int b)
16 {
17     int m = (a > b) ? a : b;
18     while (1) {
19         if (m % a == 0 && m % b == 0) return m;
20         m++;
21     }
22 }
23
24 /* (ii) via the identity GCD(a,b) = (a*b) / LCM(a,b) */
25 int gcd_via_lcm(int a, int b) { return (a * b) / lcm(a, b); }
26
27 /* the classical algorithm, for comparison */
28 int gcd_euclid(int a, int b) { return (b == 0) ? a : gcd_euclid(b, a % b); }
29
30 /* (iii) GCD of an arbitrary number of values */
31 int gcd_many(const int v[], int n)
32 {
33     int g = v[0];
34     for (int i = 1; i < n; i++)
35         g = gcd_euclid(g, v[i]);
36     return g;
37 }
38
39 int main(void)
40 {
41     printf("gcd_loop(48,18)      = %d\n", gcd_loop(48, 18));
42     printf("lcm(48,18)          = %d\n", lcm(48, 18));
43     printf("gcd_via_lcm(48,18)   = %d\n", gcd_via_lcm(48, 18));
44     printf("gcd_euclid(48,18)    = %d\n", gcd_euclid(48, 18));
45
46     int v[] = { 48, 18, 30, 60 };
47     printf("gcd_many            = %d\n", gcd_many(v, 4));
48     return 0;
49 }
```

```

gcd_loop(48,18)    = 6
lcm(48,18)        = 144
gcd_via_lcm(48,18) = 6
gcd_euclid(48,18)  = 6
gcd_many          = 6

```

The three methods illustrate a theme this course returns to in Chapter 26. `gcd_loop` performs $\min(a,b)$ iterations — $O(\min(a,b))$. `gcd_via_lcm` is worse, because `lcm` itself searches. `gcd_euclid` finishes in $O(\log \min(a,b))$ steps. For `gcd(1000000, 999999)` that is the difference between a million iterations and about thirty.

✖ `gcd_via_lcm` overflows

`(a * b)` is computed as `int`. For `a` and `b` near 46,341 the product exceeds `INT_MAX` and the answer is nonsense. The robust form divides first: `(a / gcd(a,b)) * b`.

6.8 Scope, lifetime and recursion

A variable declared inside a function is **local** to it: invisible outside, created when the function is entered and destroyed when it returns. Two calls to the same function have entirely separate locals — which is precisely what makes recursion possible.

```

int factorial(int n)
{
    if (n <= 1) return 1;           /* base case */
    return n * factorial(n - 1);    /* recursive step */
}

```

Each nested call gets its own `n` on the stack. Recursion, stack frames and the cost of both are the subject of Chapter 20; storage classes and lifetime are covered in Chapter 12.

6.9 Chapter summary

- A function is a named block taking parameters and optionally returning a value. `void` means no value, no parameters, or no type depending on where it appears.
- Declare before use. A prototype gives the compiler the return and parameter types so it can check calls; this is what headers contain.
- C has no default arguments and no overloading: the call must match the parameter list exactly.
- **C passes everything by value.** A function receives copies and cannot modify the caller's variables.
- "Call by reference" means passing a pointer. It is the only way to modify the caller's data or return more than one value, and the only efficient way to pass large objects.
- Falling off the end of a non-void function is undefined behaviour, except in `main`.

PART II

Pointers, Memory and Aggregates

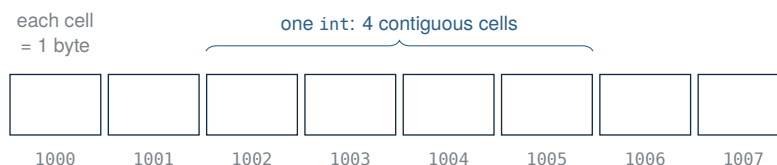
Pointers

Pointers are the reason C has a reputation for difficulty, and they are the reason C is worth learning. Everything else in this course — dynamic allocation, strings, structures, linked lists, trees — is built on them. This chapter goes slowly.

7.1 Memory as numbered cells

Key idea

A running program sees memory as a very long row of consecutively numbered cells, each one byte in size. The number of a cell is its **address**. Cells can be manipulated individually or as contiguous groups.



When you declare `int x = 10;` the compiler reserves four adjacent cells and remembers that the name `x` refers to the first of them. The *value* 10 lives in those cells; the *address* of the first cell is where they are.

7.1.1 Logical and physical memory

The addresses your program sees are not the addresses on the memory chips.

Logical (virtual) memory	The memory as seen by the program. Addresses are generated by the CPU and are what the programmer manipulates. Every process gets its own logical address space. The set of all logical addresses a CPU generates for a process is that process's logical address space .
Physical memory	The actual RAM in the machine. Addresses here are shared by every process and managed by the operating system.

The Memory Management Unit translates one to the other on every access. Two consequences matter to you:

- Your program cannot touch another program's memory, and cannot corrupt the OS — an invalid access raises a **segmentation fault** instead.
- The addresses you print will differ on every run on a modern system, because of **address space layout randomisation**. Do not be alarmed that `%p` shows a different number each time.

7.2 The address-of operator &

The unary operator & yields the address of an object.

```
int x = 10;
printf("%p\n", (void *) &x);      /* 0x7ffd9bc81fe8 */
```

✖ Common pitfall

& as a *unary* operator means “address of”. & as a *binary* operator means bitwise AND. They are unrelated; the compiler tells them apart by whether there is an operand on the left. Likewise * is either multiplication or dereference.

& may only be applied to something that has an address — a variable, an array element, a struct member. It cannot be applied to a constant (&5 is meaningless), to an expression (&(a+b)), or to a register variable.

7.3 Declaring and using a pointer

Key idea

A **pointer** is a variable capable of holding the memory address of another object. It is declared with the unary *, called the **indirection** or **dereference** operator.

Pointer declaration and use

```
<data type> *ptr;           // declare
ptr = &<object>;            // assign an address
<variable> = *ptr;          // read the value pointed at
*ptr = <value>;             // store a value there
```

basics.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 10;
6      int *p;           /* p is a pointer to int; currently uninitialised */
7
8      p = &x;           /* p now holds the address of x */
9
10     printf("x = %d\n", x);
11     printf("&x = %p\n", (void *) &x);
12     printf("p = %p\n", (void *) p);      /* same as &x */
13     printf("*p = %d\n", *p);             /* 10 -- the value at that address */
14     printf("&p = %p\n", (void *) &p);     /* p is itself a variable somewhere */
15
16     *p = 25;                          /* write through the pointer */
17     printf("x is now %d\n", x);         /* 25 -- x changed! */
18     return 0;
19 }
```

```
x = 10
&x = 0x7ffd9bc81fe8
```

```
p = 0x7ffd9bc81fe8
*p = 10
&p = 0x7ffd9bc81ff0
x is now 25
```

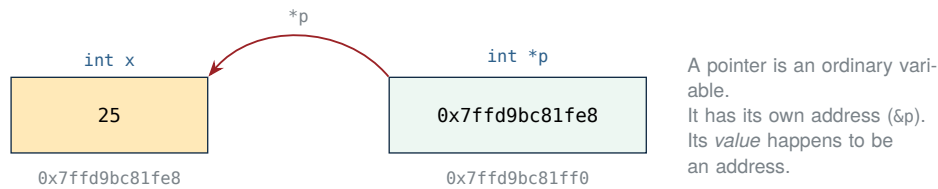


Figure 7.1. `p` holds the address of `x`. `*p` means “the object at the address stored in `p`”.

🔍 Read declarations right to left

`int *p;` reads as “`p` is a pointer to `int`”. The `*` binds to the *variable*, not to the type, which is why

```
int* a, b;    /* a is a pointer to int; b is a PLAIN int */
int *a, *b;   /* both are pointers -- what you probably meant */
```

For this reason many C programmers write the star next to the name.

7.4 The size of a pointer

A pointer stores an address, and every address on a given machine is the same size. So all pointers on a platform are the same size, regardless of what they point to.

```
sizeof(int *)    == 8    /* on 64-bit systems */
sizeof(char *)   == 8
sizeof(double *) == 8
sizeof(void *)   == 8
```

Architecture	Pointer size	Addressable memory
32-bit (x86, ARM32)	4 bytes	$2^{32} = 4 \text{ GiB}$
64-bit (x86-64, ARM64)	8 bytes	2^{64} (in practice 48 bits are used)

The size depends on the operating system and CPU architecture, not on the pointed-to type.

7.4.1 So why does a pointer need a type at all?

This is the question Lecture 4 poses, and it has a precise three-part answer.

1. **To know how many bytes to read or write when dereferencing.** `*p` on an `int *` loads four bytes and interprets them as a two’s-complement integer. On a `char *` it loads one byte. On a `double *`, eight bytes interpreted as IEEE 754. The address alone does not say how much to fetch or how to decode it.
2. **To scale pointer arithmetic.** `p + 1` means “the next object”, not “the next byte”. The compiler multiplies by `sizeof(*p)`.
3. **To let the compiler type-check.** Assigning an `int *` to a `double *` is a diagnosable error precisely because the types differ.

Key idea

All pointers are the same size, but they are not the same type. The size tells you how much space the *pointer* occupies; the type tells the compiler how to interpret the memory it *points at*.

7.5 Pointer arithmetic

Key idea

Adding n to a pointer advances it by $n \times \text{sizeof}(*p)$ bytes — that is, it moves forward by n objects, not n bytes.

arith.c --- Exercise IV, 2(c)

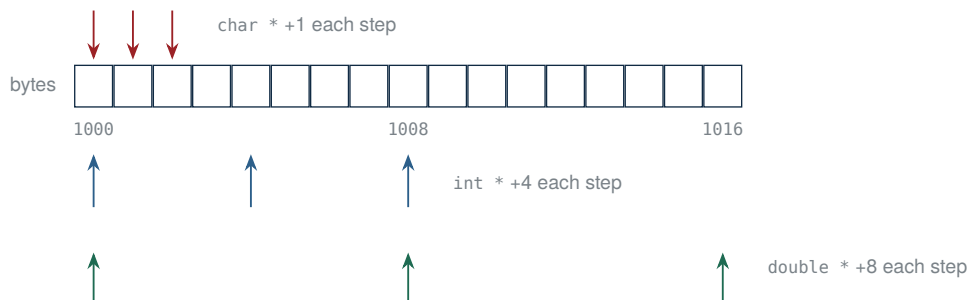
```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char *pc = (char *) 1000;    /* pretend addresses, for clarity */
6      int *pi = (int *) 1000;
7      double *pd = (double *) 1000;
8
9      for (int i = 0; i < 3; i++) {
10         printf("step %d: char* = %lu  int* = %lu  double* = %lu\n",
11             i, (unsigned long) pc, (unsigned long) pi, (unsigned long) pd);
12         pc++; pi++; pd++;
13     }
14     return 0;
15 }
```

```

step 0: char* = 1000  int* = 1000  double* = 1000
step 1: char* = 1001  int* = 1004  double* = 1008
step 2: char* = 1002  int* = 1008  double* = 1016
```

Explanation of the behaviour (this is exactly what the exercise asks for): each pointer advances by the size of the type it points to. The `char *` moves one byte at a time because `sizeof(char)` is 1; the `int *` moves four; the `double *` moves eight. All three started at the same address and diverge immediately.



7.5.1 What is and is not allowed

Operation		Meaning
$p + n, p - n$	✓	move n objects forward / back
$p++, --p$	✓	move to the next / previous object
$p - q$	✓	<i>number of objects</i> between two pointers into the same array; type <code>ptrdiff_t</code>
$p == q, p < q$	✓	compare, if both point into the same array
$p + q$	✗	adding two addresses is meaningless
$p * 2, p / 2$	✗	multiplication and division are meaningless
$p + 1.5$	✗	the offset must be an integer

```
int a[5] = {10, 20, 30, 40, 50};
int *s = &a[0], *e = &a[4];
printf("%ld\n", (long)(e - s));    /* 4 -- four ELEMENTS apart, not 16 bytes */
```

✗ Common pitfall

Arithmetic that takes a pointer outside the array it points into — other than to the one-past-the-end position, which is legal to compute but not to dereference — is undefined behaviour, even if you never dereference the result.

7.5.2 Arithmetic on the dereferenced value

Lecture 4 notes that arithmetic can be performed on the *dereferenced* pointer. This is a different thing entirely:

```
int x = 10; int *p = &x;

*p + 1    /* 11 -- read x, add 1. x is unchanged. */
*p += 1   /* x becomes 11 */
(*p)++    /* x becomes 12; yields 11. Parentheses are ESSENTIAL */
*p++      /* yields the value at p, THEN advances the POINTER */
++*p      /* x becomes 13; yields 13 */
```

✗ *p++ versus (*p)++

Postfix `++` binds tighter than unary `*`. So `*p++` parses as `*(p++)`: it dereferences `p`, then increments *the pointer*. To increment the *pointed-to value*, you must write `(*p)++`. This is a standard examination question and a standard source of bugs.

7.6 Pointers are variables too

Because a pointer is an ordinary variable, everything you can do with a variable you can do with a pointer:

```
int x = 5, y = 9;
int *p = &x;
int *q;

q = p;    /* copy a pointer -- q and p now point at the same object */
p = &y;    /* repoint p; q still points at x */
```

```
printf("%p\n", (void *) &p); /* the address OF the pointer variable */
```

What is the address of a pointer? It is the location of the pointer variable itself in memory — obtained with `&p`, and having type `int **`, “pointer to pointer to `int`”. That is the subject of Section 7.8.6.

7.7 Pointers as function parameters

This is the payoff. Section 6.6 showed that C copies arguments; passing a pointer copies an *address*, so the function can reach the original.

addptr.c --- Exercise IV, 2(a)

```
1  #include <stdio.h>
2
3  /* add two numbers and store the result in the first operand's location */
4  void add_into_first(int *a, int *b)
5  {
6      *a = *a + *b;
7  }
8
9  int main(void)
10 {
11     int x = 7, y = 5;
12     printf("before: x=%d y=%d\n", x, y);
13     add_into_first(&x, &y);
14     printf("after : x=%d y=%d\n", x, y);
15     return 0;
16 }
```

```
before: x=7 y=5
after : x=12 y=5
```

7.8 The pointer taxonomy

Pointers are classified by how they are declared and what they point to. Lecture 7 lists eight kinds; each is covered below.

Kind	Characteristic
Null pointer	points to nothing; guaranteed distinct from any valid address
Void pointer	has no associated type; can hold any object address
Wild pointer	uninitialised; points somewhere arbitrary
Dangling pointer	points to memory that has been freed or has gone out of scope
Constant pointer	the pointer itself cannot be repointed
Pointer to constant	the pointed-to data cannot be modified through it
Array of pointers	an array whose elements are pointers
Function pointer	holds the address of executable code
Pointer to pointer	holds the address of another pointer

7.8.1 Null pointers

A **null pointer** does not point to any object. It is obtained by assigning the constant `0` or, preferably, the macro `NULL` defined in `<stddef.h>` and several other headers.

```
int *p = NULL;
if (p != NULL) { *p = 5; }      /* the standard guard */
if (p)          { *p = 5; }     /* idiomatic equivalent */
```

Uses:

- To initialise a pointer that has no address to hold yet — so it is not wild.
- As a sentinel to check before dereferencing.
- As a “no object” argument or return value: malloc returns NULL on failure, fopen returns NULL if the file cannot be opened, a linked list’s last next is NULL.

Dereferencing a null pointer is undefined behaviour; on a modern OS it reliably produces a segmentation fault, which is a mercy — the failure is loud and immediate. **Printing** a null pointer with %p is implementation-defined and varies across machines: gcc/glibc prints (nil), others print 0x0 or 00000000.

	Null pointer	Void pointer
Is it a type?	No — it is a <i>value</i>	Yes — void * is a type
Points to memory?	No, by definition	Yes, to an address of unknown type
Are they all the same?	Yes, all null pointers compare equal	No — different void * values point to different places

7.8.2 Wild pointers

A **wild pointer** is one that has never been initialised. It contains whatever bit pattern happened to be in that memory, so it points at an arbitrary location.

```
int *p;          /* WILD -- contains garbage */
*p = 10;         /* writes to a random address: crash, or silent corruption */
```

The second outcome is far worse than the first. A wild write may land in memory your program legitimately owns, corrupting an unrelated variable and producing a bug that appears thousands of instructions later. **Always initialise pointers**, to NULL if you have nothing better.

7.8.3 Void pointers

A **void pointer** has no data type associated with it. It can hold the address of an object of any type, and can be cast to any pointer type.

```
void pointer
void *<name>;
(<data_type> *) <void_pointer_name>    // to cast it back
```

```
voidptr.c
1 #include <stdio.h>
2
3 void print_any(void *p, char type)
4 {
5     switch (type) {
6         case 'i': printf("int    %d\n", *(int *)  p); break;
7         case 'd': printf("double %.2f\n", *(double *) p); break;
8         case 'c': printf("char   %c\n", *(char *)  p); break;
9     }
```

```

10 }
11
12 int main(void)
13 {
14     int    i = 42;
15     double d = 3.5;
16     char   c = 'Z';
17     print_any(&i, 'i');
18     print_any(&d, 'd');
19     print_any(&c, 'c');
20     return 0;
21 }

```

```

int    42
double 3.50
char   Z

```

Rules and consequences:

- **A void pointer cannot be dereferenced** without a cast. `*p` on a `void *` is an error, because the compiler does not know how many bytes to read.
- **Pointer arithmetic on void * is not allowed** by the standard, for the same reason. GNU C permits it as an extension, treating `sizeof(void)` as 1 — so it behaves like `char *`. Do not rely on this; -pedantic rejects it.
- Conversion to and from `void *` needs no cast in C (unlike C++), which is why `int *p = malloc(n);` compiles.

Why they exist: void pointers enable **generic** functions — code that works on any type. `malloc` returns `void *` because it does not know what you will store. `qsort` takes a `void *` array and a comparison function. Generic data structures (a stack that can hold anything) are written this way.

The cost: type safety is discarded. Nothing stops you casting a `void *` that points at a double into an `int *`. The compiler cannot help; you must track types manually.

7.8.4 const and pointers

There are three combinations, distinguished by *which side of the ** the `const` appears on.

Declaration	Name	Can repoint?	Can modify data?
<code>const int *p;</code>	pointer to constant	✓	✗
<code>int const *p;</code>	same thing	✓	✗
<code>int *const p = &x;</code>	constant pointer	✗	✓
<code>const int *const p = &x;</code>	constant pointer to constant	✗	✗

```

int x = 1, y = 2;

const int *p = &x;      /* pointer to constant */
*p = 5;                 /* ERROR: read-only location */
p = &y;                  /* fine: the pointer may be repointed */

int *const q = &x;      /* constant pointer -- MUST be initialised here */
*q = 5;                 /* fine: the data may be changed */
q = &y;                  /* ERROR: assignment of read-only variable 'q' */

```

```
const int *const r = &x; /* both fixed -- must be initialised */
```

🔗 The right-to-left rule

Read the declaration backwards from the variable name. `int *const p` → “`p` is a `const` pointer to `int`”. `const int *p` → “`p` is a pointer to `int` that is `const`”. Whatever `const` sits immediately to the left of is what is frozen.

The *practical* use is in function parameters. `const char *` says “I will read this string but not modify it” — a contract the compiler enforces:

```
size_t strlen(const char *s);           /* promises not to modify s */
char *strcpy(char *dst, const char *src); /* modifies dst, reads src */
```

7.8.5 Arrays of pointers

A **pointer array** is a homogeneous collection of pointer variables indexed as an array.

Array of pointers

```
pointer_type *array_name[array_size];
```

ptrarray.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     const char *names[4] = { "Nimal", "Kamala", "Sunil", "Amara" };
6
7     for (int i = 0; i < 4; i++)
8         printf("%d: %s (at %p)\n", i, names[i], (const void *) names[i]);
9
10    /* the array holds 4 pointers, not 4 strings */
11    printf("sizeof(names) = %zu (4 pointers x 8 bytes)\n", sizeof names);
12    return 0;
13 }
```

```
0: Nimal (at 0x55a4f8c02004)
1: Kamala (at 0x55a4f8c0200a)
2: Sunil (at 0x55a4f8c02011)
3: Amara (at 0x55a4f8c02017)
sizeof(names) = 32 (4 pointers x 8 bytes)
```

Each element is an address; the strings themselves live elsewhere. This is exactly how `argv` works (Chapter 15) and how ragged two-dimensional data is represented.

✗ `int *a[10]` versus `int (*a)[10]`

<code>int *a[10];</code>	an array of 10 pointers to <code>int</code> (80 bytes)
<code>int (*a)[10];</code>	a pointer to an array of 10 <code>int</code> (8 bytes)

`[]` binds tighter than `*`, so parentheses are needed for the second.

7.8.6 Pointer to pointer

Since a pointer is a variable with an address, that address can be stored in another pointer.

doubleptr.c

```

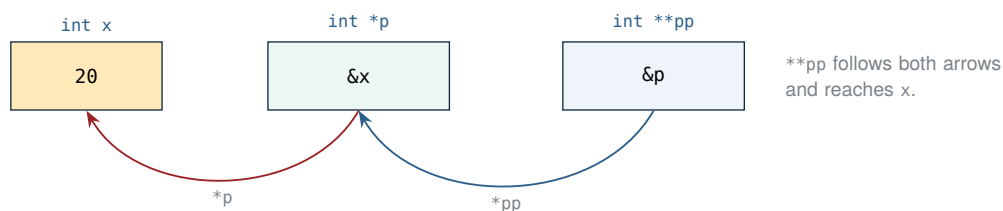
1  #include <stdio.h>
2
3  /* to modify a caller's POINTER, you must pass the pointer's address */
4  void allocate(int **out)
5  {
6      static int storage = 99;
7      *out = &storage;
8  }
9
10 int main(void)
11 {
12     int x = 10;
13     int *p = &x;
14     int **pp = &p;
15
16     printf("x = %d\n", x);
17     printf("*p = %d\n", *p);
18     printf("**pp = %d\n", **pp);
19
20     **pp = 20;          /* two hops: pp -> p -> x */
21     printf("x is now %d\n", x);
22
23     int *fresh = NULL;
24     allocate(&fresh);
25     printf("fresh points at %d\n", *fresh);
26     return 0;
27 }

```

```

x = 10
*p = 10
**pp = 10
x is now 20
fresh points at 99

```



Double pointers are needed whenever a function must change the caller's *pointer* rather than the pointed-to data — allocating memory on the caller's behalf, or inserting at the head of a linked list (Chapter 23).

7.8.7 Function pointers

Code lives in memory too, so a function has an address: the start of its executable instructions. A **function pointer** holds it.

Function pointer

```
return_type (*ptr_name)(param1_type, param2_type, ...);
```

funcptr.c

```

1  #include <stdio.h>
2
3  double add(double a, double b) { return a + b; }
4  double sub(double a, double b) { return a - b; }
5  double mul(double a, double b) { return a * b; }
6
7  /* a function taking a function: the basis of callbacks */
8  void apply(const char *label, double (*op)(double, double), double x, double y)
9  {
10     printf("%-10s %.2f\n", label, op(x, y));
11 }
12
13 int main(void)
14 {
15     double (*p)(double, double) = add;  /* & is optional on a function name */
16
17     printf("via pointer: %.2f\n", p(3, 4));
18     printf("also legal : %.2f\n", (*p)(3, 4));
19
20     /* a dispatch table -- an array of function pointers */
21     double (*ops[3])(double, double) = { add, sub, mul };
22     const char *names[3] = { "add", "sub", "mul" };
23
24     for (int i = 0; i < 3; i++)
25         apply(names[i], ops[i], 10, 4);
26     return 0;
27 }

```

```

via pointer: 7.00
also legal : 7.00
add         14.00
sub         6.00
mul         40.00

```

What they enable:

- **Generic algorithms.** qsort sorts anything because you hand it a comparison function.
- **Callbacks** — a reference to a function passed as a parameter to another function, so it can be called back through. Event handlers, signal handlers and the whole of the C standard library's customisation points work this way.
- **Dispatch tables** — replacing a long switch with an array lookup, as in the menu-driven calculator of Exercise VI question 1.
- **Wrappers** around existing functions, for logging or timing.

The **cost** is declaration syntax that becomes hard to read very quickly. typedef is the cure:

```

typedef double (*BinaryOp)(double, double);
BinaryOp ops[3] = { add, sub, mul };          /* much clearer */
void apply(const char *label, BinaryOp op, double x, double y);

```

7.8.8 Reading complex declarations: the spiral rule

Lecture 7 gives the **spiral** or **clockwise** rule for decoding declarations. Start at the identifier and spiral outwards, clockwise, translating each token as you meet it:

Token	Read as
[x] or []	“array of <i>x</i> ” or “array of undefined size of”
(a, b)	“function taking <i>a</i> and <i>b</i> and returning”
*	“pointer to”

Resolve parentheses first, and repeat until every token is consumed.

Five declarations decoded

<code>int *p[5];</code>	Start at <i>p</i> . Spiralling right meets [5]: “array of 5”. Spiralling left meets *: “pointer to”. Then <code>int</code> . $\Rightarrow$ <i>p</i> is an array of 5 pointers to <code>int</code> .
<code>int (*p)[5];</code>	Parentheses first: <code>*p</code> $\Rightarrow$ “ <i>p</i> is a pointer to”. Now outside: [5] then <code>int</code> . $\Rightarrow$ <i>p</i> is a pointer to an array of 5 <code>ints</code> .
<code>int *f(void);</code>	<code>f</code> , then <code>(void)</code> : “function taking no arguments returning”, then *: “pointer to”, then <code>int</code> . $\Rightarrow$ <i>f</i> is a function returning a pointer to <code>int</code> .
<code>int (*f)(void);</code>	Parentheses: <code>*f</code> $\Rightarrow$ “ <i>f</i> is a pointer to”. Outside, <code>(void)</code> then <code>int</code> . $\Rightarrow$ <i>f</i> is a pointer to a function taking no arguments and returning <code>int</code> .
<code>char *(*fp[10])(int);</code>	<code>fp</code> , [10]: “array of 10”, *: “pointer to”, <code>(int)</code> : “function taking <code>int</code> returning”, <code>char *</code> . $\Rightarrow$ <i>fp</i> is an array of 10 pointers to functions taking an <code>int</code> and returning a pointer to <code>char</code> .

Note

The command-line tool `cdecl` does this mechanically: `cdecl explain "int (*p)[5]"` prints *declare p as pointer to array 5 of int*. Useful for checking your work.

7.9 Chapter summary

- Memory is numbered byte cells; an address is a cell number. Programs see logical (virtual) addresses, which the MMU maps to physical RAM.
- `&x` yields the address of *x*; `*p` yields the object at the address in *p*.
- All pointers on a platform are the same size (8 bytes on 64-bit), but the type is still needed: to know how many bytes to dereference, to scale arithmetic, and for type checking.
- `p + n` advances by $n \times \text{sizeof}(*p)$ bytes. `p - q` yields the number of elements between them.
- `*p++` increments the pointer; `(*p)++` increments the value.
- `NULL` points nowhere and is the standard sentinel; wild (uninitialised) pointers are the dangerous case — always initialise.
- `void *` is generic, cannot be dereferenced without a cast, and has no standard arithmetic.
- `const int *p` freezes the data; `int *const p` freezes the pointer.
- `int **` is needed to modify a caller’s pointer.
- Function pointers enable callbacks, dispatch tables and generic algorithms; decode declarations with the spiral rule and tame them with `typedef`.

Arrays and Their Relationship with Pointers

8.1 What an array is

Key idea

An **array** is a group of elements of the *same* data type stored *contiguously* in memory. Arrays are therefore called a **homogeneous** data structure, and they are fixed in size.

Array declaration

```
<data_type> <array_name>[<size>];
<data_type> <array_name>[] = {<value 1>, ... <value n>;}
```

```
int marks[5]; /* 5 uninitialised ints */
int primes[5] = {2, 3, 5, 7, 11}; /* size deduced as 5 if [] is empty */
int zeros[100] = {0}; /* first element 0, REST also 0 */
int partial[5] = {1, 2}; /* {1, 2, 0, 0, 0} */
char vowels[] = {'a', 'e', 'i', 'o', 'u'}; /* size 5 -- NOT a string, no '\0' */
```

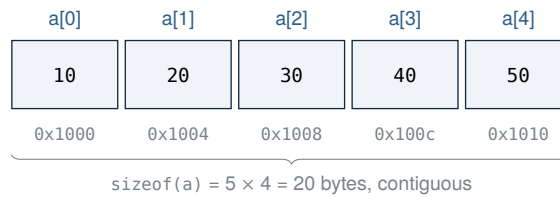
Note

`= {0}` zero-initialises the *entire* array: any element you do not list is set to zero. `= {1}` sets the first to 1 and the rest to 0, which is often a surprise. Without any initialiser, elements of a local array contain garbage.

8.2 Properties of arrays

Property	Consequence
Homogeneous	every element has the same type and therefore the same size
Contiguous	element i is at $\text{base} + i * \text{sizeof}(\text{element})$
Fixed size	the size is part of the type and cannot change at run time
Zero-based indexing	valid indices are $0 \dots n-1$
Linear	elements have a natural predecessor and successor
Random access	any element is reachable in $O(1)$ by computing its address
Sequential access	the same array can also be walked in order
Row-major or column-major	for multi-dimensional arrays, the choice of layout

The contiguity property is what buys $O(1)$ random access, and it is the single most important difference from a linked list (Chapter 23).



8.3 Finding the length of an array

```
int a[5] = {10, 20, 30, 40, 50};
size_t n = sizeof(a) / sizeof(a[0]);    /* 20 / 4 = 5 */
```

This idiom divides the total size by the size of one element. It is computed at compile time and costs nothing at run time.

✗ The idiom fails inside a function

```
void f(int a[]) {
    size_t n = sizeof(a) / sizeof(a[0]);    /* 8 / 4 = 2, ALWAYS WRONG */
}
```

Inside the function, `a` is not an array — it is a pointer. `sizeof(a)` is the size of a pointer, 8. This is the direct consequence of decay, explained next. **A C function can never determine the length of an array passed to it; you must pass the length as a separate parameter.**

8.4 Array–pointer duality

📖 Decay

In almost every expression, an array name is automatically converted to — “decays to” — a pointer to its first element. `a` becomes `&a[0]`.

The two exceptions are `sizeof(a)`, which gives the size of the whole array, and `&a`, which gives a pointer to the array as a whole.

From decay, one identity follows, and it explains everything else:

`a[i]` is defined as `*(a + i)`

duality.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a[5] = {10, 20, 30, 40, 50};
6
7     printf("a      = %p\n", (void *) a);
8     printf("&a[0]  = %p\n", (void *) &a[0]);    /* identical */
9     printf("a + 1  = %p (4 bytes further)\n", (void *) (a + 1));
10
11     printf("a[2]    = %d\n", a[2]);
12     printf("*(a + 2) = %d\n", *(a + 2));
13     printf("2[a]    = %d\n", 2[a]);            /* legal, and horrible */
```

```

14     printf("(2 + a) = %d\n", *(2 + a));
15
16     int *p = a;                /* no & needed: a already IS an address */
17     printf("p[3] = %d    *(p+3) = %d\n", p[3], *(p + 3));
18     return 0;
19 }

```

```

a      = 0x7ffd9bc82020
&a[0]  = 0x7ffd9bc82020
a + 1   = 0x7ffd9bc82024  (4 bytes further)
a[2]    = 30
*(a + 2) = 30
2[a]    = 30
*(2 + a) = 30
p[3] = 40    *(p+3) = 40

```

`2[a]` works because subscripting is defined as `*(a + 2)`, and addition is commutative, so `*(2 + a)` is the same thing. It is a curiosity, not a technique — but knowing *why* it compiles means you have understood subscripting.

8.4.1 Where arrays and pointers differ

They are not the same thing, and treating them as identical causes bugs.

	<code>int a[5];</code>	<code>int *p;</code>
What it is	5 ints, contiguous	one variable holding an address
<code>sizeof</code>	20 (whole array)	8 (the pointer)
Assignable?	✗ <code>a = something</code> is an error	✓ <code>p = a</code> is fine
Storage	the elements themselves	only the address; the target is elsewhere
<code>&name</code>	pointer to the whole array, <code>int (*)[5]</code>	<code>int **</code>
Increment	✗ <code>a++</code> is an error	✓ <code>p++</code>

8.5 Traversing an array

Three equivalent idioms:

```
|  |  |
| --- | --- |
| traverse.c | |

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a[5] = {10, 20, 30, 40, 50};
6      size_t n = sizeof a / sizeof a[0];
7
8      /* 1. index */
9      for (size_t i = 0; i < n; i++) printf("%d ", a[i]);
10     printf("\n");
11
12     /* 2. pointer with index arithmetic */
13     for (size_t i = 0; i < n; i++) printf("%d ", *(a + i));
14     printf("\n");
15
16     /* 3. moving pointer -- Exercise IV, 4(d) */

```

```

17     for (int *p = a; p < a + n; p++) printf("%d ", *p);
18     printf("\n");
19     return 0;
20 }

```

```

10 20 30 40 50
10 20 30 40 50
10 20 30 40 50

```

The third form — comparing against the one-past-the-end pointer `a + n` — is the idiomatic C loop and the ancestor of the C++ iterator.

8.6 Passing arrays to functions

Because of decay, an array parameter is really a pointer parameter. The two forms are *exactly* equivalent:

Two ways to write the same parameter

```

<return type> function(array_type name[size], ...)
<return type> function(array_type *name, ...)

```

The size in the first form is documentation only — the compiler ignores it. Since the length is lost, it must be passed explicitly.

stats.c --- Exercise IV, 4(e)

```

1  #include <stdio.h>
2  #include <math.h>
3
4  double average(const int a[], int n)
5  {
6      long sum = 0;
7      for (int i = 0; i < n; i++) sum += a[i];
8      return (double) sum / n;
9  }
10
11 int minimum(const int a[], int n)
12 {
13     int m = a[0];
14     for (int i = 1; i < n; i++) if (a[i] < m) m = a[i];
15     return m;
16 }
17
18 int maximum(const int a[], int n)
19 {
20     int m = a[0];
21     for (int i = 1; i < n; i++) if (a[i] > m) m = a[i];
22     return m;
23 }
24
25 double std_dev(const int a[], int n)
26 {
27     double mean = average(a, n), acc = 0.0;
28     for (int i = 0; i < n; i++) acc += (a[i] - mean) * (a[i] - mean);
29     return sqrt(acc / n);          /* population standard deviation */
30 }
31
32 int main(void)

```

```

33 {
34     int data[10] = { 42, 7, 93, 18, 55, 61, 3, 77, 29, 84 };
35     int n = 10;
36     printf("average = %.4f\n", average(data, n));
37     printf("minimum = %d\n",    minimum(data, n));
38     printf("maximum = %d\n",    maximum(data, n));
39     printf("std dev = %.4f\n", std_dev(data, n));
40     return 0;
41 }

```

```

average = 46.9000
minimum = 3
maximum = 93
std dev = 30.6119

```

Note `const int a[]`: the functions promise not to modify the array, and the compiler enforces it. Get into this habit.

Key idea

Arrays are always effectively passed by reference, because what is copied is the address of the first element. A function *can* modify the caller's array, and it costs nothing to pass a million-element array.

8.7 Multi-dimensional arrays

An array may have more than one subscript.

Multi-dimensional array

```
<data_type> <name>[size][size]...[size];
```

```

int grid[3][4];           /* 3 rows, 4 columns */
int m[2][3] = { {1,2,3}, {4,5,6} }; /* braced by row */
int m2[2][3] = {1,2,3,4,5,6}; /* same thing -- inner braces optional */

```

A 2-D array in C is really an *array of arrays*: `grid` is an array of 3 objects, each of which is an array of 4 `int`. Memory, however, is one-dimensional, so the compiler must flatten it.

8.7.1 Row-major versus column-major order

logical view

row 0	1	2	3	4
row 1	5	6	7	8
row 2	9	10	11	12

Row-major (C): rows stored one after another

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

Column-major (Fortran, MATLAB): columns stored one after another

1	5	9	2	6	10	3	7	11	4	8	12
---	---	---	---	---	----	---	---	----	---	---	----

Key idea

C uses row-major order. The element `a[i][j]` of an `a[ROWS][COLS]` array is at byte offset $(i \times \text{COLS} + j) \times \text{sizeof}(\text{element})$ from the base.

rowmajor.c

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int m[2][3] = { {1,2,3}, {4,5,6} };
6     int *flat = &m[0][0];
7
8     printf("as stored in memory: ");
9     for (int i = 0; i < 6; i++) printf("%d ", flat[i]);
10    printf("\n");
11
12    printf("m[1][2] = %d, flat[1*3+2] = %d\n", m[1][2], flat[1*3 + 2]);
13    printf("sizeof(m)    = %zu\n", sizeof m);          /* 24 */
14    printf("sizeof(m[0]) = %zu\n", sizeof m[0]);        /* 12 -- one row */
15    return 0;
16 }
```

```

as stored in memory: 1 2 3 4 5 6
m[1][2] = 6, flat[1*3+2] = 6
sizeof(m)    = 24
sizeof(m[0]) = 12
```

Why the layout matters for performance

Modern CPUs fetch memory in *cache lines* of 64 bytes. Walking a row-major array row by row touches consecutive bytes and every fetched line is fully used. Walking it column by column jumps by a whole row each step, so each fetched line supplies one useful value. On a large matrix the row-wise loop can be several times faster — same operations, same big-O, different constant. This is one of the reasons Chapter 26 insists that measured time and asymptotic complexity are different things.

8.7.2 Passing a 2-D array

Only the *first* dimension may be omitted, because the compiler needs the row length to compute offsets:

```
void print(int m[][3], int rows);      /* OK: column count required */
void print(int (*m)[3], int rows);    /* the same thing */
void print(int m[2][3], int rows);    /* also fine */
void print(int **m, int rows);        /* WRONG -- a different type entirely */
```

8.8 Arrays as the basis of other data structures

Arrays are the substrate on which abstract data types are built. Because they give $O(1)$ indexed access and perfect locality, they are the natural implementation for:

- **Stacks** — one index (top) is all the bookkeeping needed (Chapter 21).
- **Queues** — two indices, plus the circular trick (Chapter 22).
- **Strings** — an array of char with a terminator (Chapter 9).
- **Hash tables, heaps, matrices** and much else.

The limitation is the fixed size, which is what motivates dynamic allocation (Chapter 13) and linked structures (Chapter 23).

8.9 Worked examples from Exercise IV

8.9.1 Question 4(f) — tracing an array

```
ex4qf.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int arr[10];
6      arr[1] = 97;
7      arr[2] = 75;
8      arr[7] = 50;
9      arr[9] = 3;
10     arr[0] = arr[1] * arr[9] - arr[2];
11     arr[3] = 5 * arr[7];
12     arr[4] = 1;
13     arr[8] = arr[4] + arr[4];
14
15     printf("%d\n", arr[9]);
16
17     for (short i = 9; i >= 0; i--)
18         printf("%d = %d\n", i, arr[i]);
19     return 0;
20 }
```

Trace

Assignments in order: $\text{arr}[1]=97, \text{arr}[2]=75, \text{arr}[7]=50, \text{arr}[9]=3$. Then $\text{arr}[0] = 97 * 3 - 75 = 291 - 75 = 216; \text{arr}[3] = 5 * 50 = 250; \text{arr}[4] = 1; \text{arr}[8] = 1 + 1 = 2$.

Elements 5 and 6 are never assigned. arr is a local array, so they contain whatever was on the stack — *indeterminate values*. The program's output for those two lines is not predictable, and reading them

is technically undefined behaviour. On one run of gcc 11.4 the output was:

```

          3
9 = 3  8 = 2  7 = 50  6 = 529267711  5 = 0  4 = 1
          3 = 250  2 = 75  1 = 97  0 = 216

```

Elements 5 and 6 are pure luck: on this run arr[5] happened to be 0 while arr[6] was 529267711 — leftover bytes from whatever used that stack space before. Run the same binary again and the value may differ. **That is the lesson of the question:** the defined values are 216, 97, 75, 250, 1, ?, ?, 50, 2, 3, and the two question marks are not zero just because arrays in some other language would be.

8.9.2 Question 4(g) — a char array that is not a string

ex4qg.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      char welcome[] = {'H', 'E', 'L', 'L', '0'};
6
7      printf("%c\n", welcome[3]);
8      welcome[0] = 'Y';
9      printf("%c\n", welcome[3] + 2);
10
11     for (unsigned short j = 0; j < 3; j++)
12         printf("%c\n", welcome[j]);
13     return 0;
14 }

```

```

L
N
Y
E
L

```

Trace

- welcome[3] is 'L'.
- welcome[0] = 'Y' changes the array to Y E L L 0; note that welcome[3] is unaffected.
- welcome[3] + 2: 'L' is 76, so the sum is 78. But %c converts 78 back to a character: 'N'. *The addition happens on the integer code.*
- The loop prints indices 0, 1, 2 of the modified array: Y, E, L.

Critically, this array is not a string — there is no '\0'. printf("%s", welcome) would run off the end of the array printing garbage until it happened to meet a zero byte.

8.9.3 Question 4(c) — palindrome

palindrome.c

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4
5  int is_palindrome(const char *s)
6  {
7      int i = 0, j = (int) strlen(s) - 1;

```

```

8   while (i < j) {
9       while (i < j && !isalnum((unsigned char) s[i])) i++; /* skip spaces */
10      while (i < j && !isalnum((unsigned char) s[j])) j--;
11      if (toupper((unsigned char) s[i]) != toupper((unsigned char) s[j]))
12          return 0;
13      i++; j--;
14  }
15  return 1;
16  }
17
18  int main(void)
19  {
20      const char *tests[] = { "RACECAR", "WOW", "TACO CAT", "HELLO", "A" };
21      for (int i = 0; i < 5; i++)
22          printf("%-10s -> %s\n", tests[i],
23                is_palindrome(tests[i]) ? "palindrome" : "not a palindrome");
24      return 0;
25  }

```

```

RACECAR    -> palindrome
WOW        -> palindrome
TACO CAT   -> palindrome
HELLO      -> not a palindrome
A          -> palindrome

```

The two-pointer technique — one index walking forward, one backward, meeting in the middle — runs in $O(n)$ time and $O(1)$ extra space, and is worth recognising. Note the `(unsigned char)` casts: the `<ctype.h>` functions have undefined behaviour on negative char values.

8.9.4 Question 4(b) — binary to decimal

```

bits.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int bits[8];
6      int value = 0;
7
8      printf("Enter 8 bits (0 or 1), most significant first: ");
9      for (int i = 0; i < 8; i++) {
10         if (scanf("%d", &bits[i]) != 1 || (bits[i] != 0 && bits[i] != 1)) {
11             printf("invalid bit at position %d\n", i);
12             return 1;
13         }
14         value = value * 2 + bits[i]; /* Horner's method */
15     }
16
17     printf("binary  ");
18     for (int i = 0; i < 8; i++) printf("%d", bits[i]);
19     printf("\ndecimal %d\n", value);
20     return 0;
21 }

```

```

$ echo "1 0 1 0 1 1 0 0" | ./bits
Enter 8 bits (0 or 1), most significant first: binary  10101100
decimal 172

```

`value = value * 2 + bit` is **Horner's method**: it converts in a single pass with no powers and no `pow()` call.

8.10 Chapter summary

- An array is a fixed-size, contiguous, homogeneous, zero-indexed, randomly-accessible linear structure.
- `sizeof(a)/sizeof(a[0])` gives the length — but only where `a` is a real array, never inside a function.
- Array names decay to `&a[0]` in nearly every expression; the exceptions are `sizeof` and `&`.
- `a[i]` is defined as `*(a + i)`, which is why `i[a]` compiles.
- Arrays are effectively passed by reference; the length must be passed separately.
- C stores multi-dimensional arrays in **row-major** order; `a[i][j]` is at offset $(i \times \text{COLS} + j)$.
- When passing a 2-D array, only the first dimension may be omitted.
- C performs **no bounds checking whatsoever**. Reading or writing outside an array is undefined behaviour and the most common cause of crashes and security holes in C programs.

Strings

9.1 C has no string type

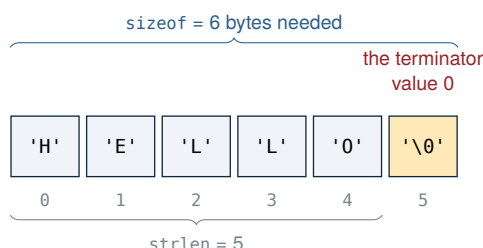
Key idea

The C programming language has no built-in string data type. A **string** is a sequence of characters terminated by the null character `'\0'`. Declaring a string is exactly the same as declaring a one-dimensional array of `char`.

String declaration

```
char <string_name>[array_size];
```

The terminator is not decoration — it is the only way any function can know where the string ends, because the array's length is not carried with it.



Key idea

To store a string of n visible characters, the array must have size at least $n + 1$.

9.2 Four ways to make a string

decl.c

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void)
5 {
6     char str1[] = "SCS 1301 - DSA in C"; /* size deduced: 19 + 1 = 20 */
7     char str2[25] = "SCS 1301 - DSA in C"; /* 20 used, 5 spare (zeroed) */
8     char str3[] = {'T', 'E', 'S', 'T', '\0'}; /* manual: YOU add the '\0' */
9     const char *str4 = "read only literal"; /* pointer to a literal */
10
11     printf("str1 [%s] strlen=%zu sizeof=%zu\n", str1, strlen(str1), sizeof str1);
12     printf("str2 [%s] strlen=%zu sizeof=%zu\n", str2, strlen(str2), sizeof str2);
13     printf("str3 [%s] strlen=%zu sizeof=%zu\n", str3, strlen(str3), sizeof str3);
14     printf("str4 [%s] strlen=%zu sizeof=%zu (a pointer!)\n",
15           str4, strlen(str4), sizeof str4);
```

```

16     return 0;
17 }

```

```

str1 [SCS 1301 - DSA in C] strlen=19 sizeof=20
str2 [SCS 1301 - DSA in C] strlen=19 sizeof=25
str3 [TEST] strlen=4 sizeof=5
str4 [read only literal] strlen=17 sizeof=8 (a pointer!)

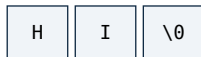
```

Key idea

With double-quote initialisation (`str1`, `str2`) the `'\0'` is supplied automatically. When you assign character by character (`str3`) **you** must insert it, or the result is not a string.

9.2.1 Array versus pointer initialisation

```
char s[] = "HI";
```



a modifiable array of 3 bytes
on the stack; `s[0]='J'` is fine

```
char *p = "HI";
```



a pointer (8 bytes) to a literal in *read-only*
memory;
`p[0]='J' ⇒ segmentation fault`

✗ Modifying a string literal is undefined behaviour

```

char *p = "hello";
p[0] = 'H';           /* CRASH: literals live in read-only .rodata */

char a[] = "hello";
a[0] = 'H';           /* fine: a is a copy on the stack */

```

Recall the assembly in Section 1.8: the literal was placed in the `.rodata` section. Always declare pointers to literals as `const char *` so the compiler catches the mistake at compile time.

9.3 The <string.h> library

Function	Purpose	Complexity
strlen(s)	number of characters before the '\0'	$O(n)$
strcpy(dst, src)	copy src into dst, including the terminator	$O(n)$
strncpy(dst, src, n)	copy at most n bytes — see the warning below	$O(n)$
strcat(dst, src)	append src to the end of dst	$O(n + m)$
strncat(dst, src, n)	append at most n bytes	$O(n + m)$
strcmp(a, b)	< 0, 0, > 0 as a is less than, equal to, or greater than b	$O(n)$
strncmp(a, b, n)	compare the first n bytes	$O(n)$
strchr(s, c)	pointer to the first c in s, else NULL	$O(n)$
strrchr(s, c)	pointer to the <i>last</i> c	$O(n)$
strstr(hay, need)	pointer to the first occurrence of the substring	$O(nm)$
strtok(s, delim)	split into tokens — <i>modifies s</i>	$O(n)$
memcpy(d, s, n)	copy n bytes; regions must not overlap	$O(n)$
memmove(d, s, n)	copy n bytes; overlap is safe	$O(n)$
memset(d, c, n)	fill n bytes with c	$O(n)$

strdemo.c

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      char a[64] = "Hello";
7      char b[] = " World";
8
9      printf("strlen(a)      = %zu\n", strlen(a));
10     strcat(a, b);
11     printf("after strcat    = %s\n", a);
12
13     char c[64];
14     strcpy(c, a);
15     printf("copy           = %s\n", c);
16
17     printf("strcmp(a,c)       = %d\n", strcmp(a, c));
18     printf("strcmp(\"abc\", \"abd\") = %d\n", strcmp("abc", "abd"));
19     printf("strcmp(\"abd\", \"abc\") = %d\n", strcmp("abd", "abc"));
20
21     char *found = strstr(a, "World");
22     printf("strstr offset    = %ld\n", (long)(found - a));
23     printf("strchr('o')      = %ld\n", (long)(strchr(a, 'o') - a));
24     return 0;
25 }
```

```

strlen(a)      = 5
after strcat    = Hello World
copy           = Hello World
strcmp(a,c)     = 0
strcmp("abc","abd") = -1
strcmp("abd","abc") = 1
strstr offset   = 6
strchr('o')     = 4
```

✗ == does not compare strings

```
char a[] = "hello", b[] = "hello";
if (a == b)           /* FALSE -- compares two ADDRESSES */
if (strcmp(a, b) == 0) /* TRUE  -- compares CONTENT */
```

Similarly `a = b` does not copy a string; you cannot assign arrays. Use `strcpy`.

✗ strcpy and strcat have no idea how big the destination is

```
char small[8];
strcpy(small, "this is far too long"); /* overruns -- corrupts the stack */
```

This is the classic buffer overflow, the mechanism behind decades of security vulnerabilities. Bound the copy yourself, and note that `strncpy` has a trap of its own: if the source is exactly `n` bytes or longer it does *not* write a terminator.

```
strncpy(small, src, sizeof small - 1);
small[sizeof small - 1] = '\0'; /* always terminate manually */

snprintf(small, sizeof small, "%s", src); /* simpler and always terminates */
```

9.4 Implementing the library functions yourself

Writing these is the best way to be certain you understand both strings and pointers.

myimpl.c

```
1  #include <stdio.h>
2
3  size_t my_strlen(const char *s)
4  {
5      const char *start = s;
6      while (*s) s++;           /* stop at the '\0', which is 0 = false */
7      return (size_t)(s - start); /* pointer difference = element count */
8  }
9
10 char *my_strcpy(char *dst, const char *src)
11 {
12     char *out = dst;
13     while ((*dst++ = *src++)) /* copy, then test the copied byte */
14         ;
15     return out;
16 }
17
18 int my_strcmp(const char *a, const char *b)
19 {
20     while (*a && (*a == *b)) { a++; b++; }
21     return (unsigned char)*a - (unsigned char)*b;
22 }
23
24 char *my_strcat(char *dst, const char *src)
25 {
26     char *out = dst;
27     while (*dst) dst++; /* walk to the terminator of dst */
28     while ((*dst++ = *src++))
```

```

29     ;
30     return out;
31 }
32
33 int main(void)
34 {
35     char buf[64];
36     printf("my_strlen = %zu\n", my_strlen("Hello World"));
37     my_strcpy(buf, "Hello");
38     my_strcat(buf, " World");
39     printf("buf      = %s\n", buf);
40     printf("my_strcmp = %d %d %d\n",
41           my_strcmp("abc", "abc"), my_strcmp("abc", "abd"), my_strcmp("abd", "abc"));
42     return 0;
43 }

```

```

my_strlen = 11
buf       = Hello World
my_strcmp = 0 -1 1

```

while ((*dst++ = *src++)) is a famous line of C. Read it as: copy the byte src points at into the byte dst points at; advance both pointers; the value of the assignment expression is the byte copied, which is zero exactly when the terminator was copied, ending the loop. The extra parentheses tell the compiler the assignment is deliberate and not a mistyped ==.

9.5 Reading and printing strings

9.5.1 With printf

%s prints characters until it meets '\0'. It can be combined with width and precision like any other specifier:

```

printf("[%s]", "Hello"); /* [Hello] */
printf("[%10s]", "Hello"); /* [ Hello] right justified in 10 */
printf("[% -10s]", "Hello"); /* [Hello ] left justified */
printf("[% .3s]", "Hello"); /* [Hel] precision TRUNCATES strings */
printf("[%10.3s]", "Hello"); /* [ Hel] */

```

9.5.2 With scanf and scansets

%s reads a single whitespace-delimited word and appends the terminator. It stops at the first space, which is why it cannot read a full name.

```

char word[32];
scanf("%31s", word); /* one word, bounded */

char line[80];
scanf("%79[^\n]", line); /* a whole line, bounded */

```

A **scanset** %[...] processes only those characters in the set. Reproducing the examples from Lecture 7:

<code>%[A-Z]</code>	stores only capital letters; stops at the first non-capital
<code>%[^a]</code>	stores characters until the first 'a'; the 'a' is not stored
<code>%[A-Z_a,b,c]</code>	stores capitals, underscore and the literals a, b, c, ,
<code>%[0-9]</code>	stores only digit characters

Note

A `^` immediately after the opening bracket *negates* the set. A `]` as the first character in the set is a literal `]`. A `-` between two characters denotes a range; at the start or end it is a literal hyphen.

9.5.3 Character classification: <ctype.h>

<code>isalpha(c)</code>	letter	<code>isupper(c)</code>	uppercase
<code>isdigit(c)</code>	'0'–'9'	<code>islower(c)</code>	lowercase
<code>isalnum(c)</code>	letter or digit	<code>toupper(c)</code>	convert to uppercase
<code>isspace(c)</code>	space, tab, newline...	<code>tolower(c)</code>	convert to lowercase
<code>ispunct(c)</code>	printable, not alphanumeric or space	<code>isxdigit(c)</code>	hex digit

✗ Common pitfall

Every <ctype.h> function takes an `int` whose value must be representable as unsigned `char`, or be `E0F`. Passing a plain `char` that is negative — which happens for any byte above 127 on a platform where `char` is signed — is undefined behaviour. Cast: `isalpha((unsigned char) s[i])`.

9.5.4 Exercise IV, 3(g) — swap the case of a character**swapcase.c**

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4  char opposite_case(char c)
5  {
6      if (islower((unsigned char) c)) return (char) toupper((unsigned char) c);
7      if (isupper((unsigned char) c)) return (char) tolower((unsigned char) c);
8      return c;                      /* leave non-letters alone */
9  }
10
11 int main(void)
12 {
13     const char *tests = "aZ3_q";
14     for (const char *p = tests; *p; p++)
15         printf("%c -> %c\n", *p, opposite_case(*p));
16     return 0;
17 }
```

```

a -> A
Z -> z
3 -> 3
_ -> _
q -> Q
```

Without <ctype.h> the arithmetic version is `c ^ 32`, which flips the single bit that distinguishes

upper from lower case in ASCII — but that trick only works for letters, so the classification test is still needed.

9.6 Arrays of strings

Two representations, with different trade-offs.

```

1  strarrays.c
2  #include <stdio.h>
3
4  int main(void)
5  {
6      /* 1. array of pointers: ragged, each string is exactly as long as needed,
7       * but the strings are literals and must not be modified */
8      const char *days[] = { "Sunday", "Monday", "Tuesday" };
9
10     /* 2. 2-D array: rectangular, wastes space, but every string is modifiable */
11     char months[3][10] = { "January", "February", "March" };
12
13     for (int i = 0; i < 3; i++) printf("%s ", days[i]);
14     printf("\n");
15     months[0][0] = 'j';          /* legal */
16     for (int i = 0; i < 3; i++) printf("%s ", months[i]);
17     printf("\n");
18
19     printf("sizeof(days)   = %zu (3 pointers)\n", sizeof days);
20     printf("sizeof(months) = %zu (3 x 10 chars)\n", sizeof months);
21     return 0;
22 }
```

```

Sunday Monday Tuesday
january February March
sizeof(days)   = 24 (3 pointers)
sizeof(months) = 30 (3 x 10 chars)
```

9.7 Chapter summary

- A C string is a char array terminated by '\0'. An n -character string needs an array of at least $n + 1$.
- Double-quote initialisation supplies the terminator; character-by-character initialisation does not.
- `char a[] = "x"` is a modifiable copy; `char *p = "x"` points at a read-only literal — declare it `const char *`.
- `strlen` counts up to but excluding the terminator; `sizeof` counts the whole array including it.
- Compare with `strcmp`, not `==`; copy with `strcpy`, not `=`.
- `strcpy` and `strcat` do not know the destination size — prefer `snprintf`, and remember `strncpy` may not terminate.
- Scansets `%[...]` filter input by character class; `^` negates the set.
- Cast to unsigned char before calling any `<ctype.h>` function.

Structures, Unions and Bit-fields

Arrays group items of the *same* type. Structures group items of *different* types. Between them they cover almost all of data representation.

10.1 Structures

Key idea

A **structure** is a user-defined type that logically groups related *heterogeneous* items under a single unit, where each item can still be accessed separately. The items are called **members**.

struct definition

```
struct <name>
{
    <data_type 1> <variable_name 1>;
    <data_type 2> <variable_name 2>;
    ...
} [variable name 1, variable name 2 ...];
```

```
struct Student {
    int id;
    char name[20];
    float gpa;
}; /* NOTE the semicolon -- a struct is a declaration */

struct Student s1; /* declare a variable */
struct Student s2 = {102, "Kamala", 3.9f}; /* initialise in member order */
struct Student s3 = {.gpa = 3.2f, .id = 103}; /* C99 designated initialiser */
```

✖ Common pitfall

The semicolon after the closing brace of a struct definition is mandatory. Omitting it produces a cascade of confusing errors on the *following* lines, because the compiler thinks the next declaration is part of the struct.

10.1.1 Accessing members

Member access

```
<variable_name>.<member_name>    // through a variable
<pointer_name>-><member_name>    // through a pointer
```

structs.c

```

1  #include <stdio.h>
2  #include <string.h>
3
4  struct Student {
5      int    id;
6      char   name[20];
7      float  gpa;
8  };
9
10 int main(void)
11 {
12     struct Student s = {101, "Nimal", 3.75f};
13     struct Student *p = &s;           /* pointer to a structure */
14
15     printf("dot   : %d %s %.2f\n", s.id, s.name, (double) s.gpa);
16     printf("arrow : %d %s %.2f\n", p->id, p->name, (double) p->gpa);
17     printf("ugly  : %d %s %.2f\n", (*p).id, (*p).name, (double) (*p).gpa);
18
19     strcpy(s.name, "Nimal Perera");    /* arrays are assigned with strcpy */
20     s.gpa = 3.80f;
21     printf("after : %d %s %.2f\n", s.id, s.name, (double) s.gpa);
22     return 0;
23 }

```

```

dot   : 101 Nimal 3.75
arrow : 101 Nimal 3.75
ugly  : 101 Nimal 3.75
after : 101 Nimal Perera 3.80

```

 Key idea

`p->member` is exactly `(*p).member`. The parentheses in the second form are essential, because `.` binds tighter than `*` — so `*p.member` would mean `*(p.member)`. The `->` operator exists precisely so nobody has to write that.

10.1.2 What you can and cannot do with a struct

Operation	Notes
Assign one struct to another	✓ <code>s2 = s1</code> ; performs a member-by-member copy — unlike arrays, which cannot be assigned
Pass to a function by value	✓ the whole struct is copied; expensive for large structs
Return from a function	✓
Take the address	✓ <code>&s</code>
Have an array of structs	✓ <code>struct Student class[50];</code>
Contain an array or another struct	✓ nesting is unlimited
Compare with <code>==</code>	✗ not allowed — padding bytes have indeterminate values, so a bitwise comparison would be meaningless. Compare member by member.
Contain an instance of itself	✗ infinite size — but it <i>may</i> contain a <i>pointer</i> to itself

10.1.3 Nested and self-referential structures

```

struct Date { int day, month, year; };

struct Employee {
    char    name[40];
    struct Date joined;      /* a struct inside a struct */
    double   salary;
};

struct Employee e;
e.joined.year = 2026;      /* chain the dots */

/* self-referential: legal, because a POINTER to an incomplete type is fine */
struct Node {
    int      data;
    struct Node *next;      /* the foundation of every linked structure */
};

```

Key idea

`struct Node { struct Node next; };` is impossible — the type would have infinite size. `struct Node *next;` is fine, because a pointer has a known size (8 bytes) regardless of what it points to. This single fact makes linked lists, trees and graphs possible (Chapter 23).

10.2 Memory layout, padding and alignment

Key idea

A structure allocates memory for each data member, and all members lie in a single contiguous block. However, `sizeof` a structure is often *larger* than the sum of its members' sizes, because of the computer's **data alignment** requirements.

CPUs read memory most efficiently when an n -byte object sits at an address that is a multiple of n . The compiler therefore inserts unnamed **padding** bytes so that every member is correctly aligned, and pads the struct as a whole so that arrays of it stay aligned.

padding.c

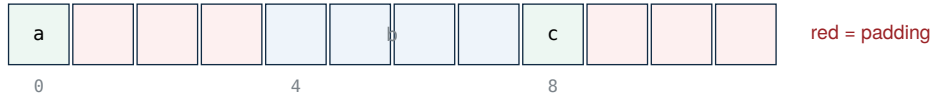
```

1  #include <stdio.h>
2  #include <stddef.h>
3
4  struct Padded { char a; int b; char c; }; /* 1 + 4 + 1 = 6 bytes of data */
5  struct Packed { char a; char c; int b; }; /* the same members, reordered */
6
7  int main(void)
8  {
9      printf("sizeof(Padded) = %zu\n", sizeof(struct Padded));
10     printf("sizeof(Packed) = %zu\n", sizeof(struct Packed));
11     printf("offsets in Padded: a=%zu b=%zu c=%zu\n",
12           offsetof(struct Padded, a),
13           offsetof(struct Padded, b),
14           offsetof(struct Padded, c));
15     return 0;
16 }

```

```
sizeof(Padded) = 12
sizeof(Packed) = 8
offsets in Padded: a=0 b=4 c=8
```

struct Padded { char a; int b; char c; } = 12 bytes



struct Packed { char a; char c; int b; } = 8 bytes

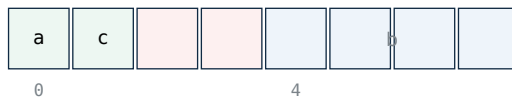


Figure 10.1. The same three members, reordered, cost 12 bytes or 8. Declaring members in decreasing order of size minimises padding.

✓ Exam focus

Be ready to explain why `sizeof` a struct may exceed the sum of its members: **data alignment**. The compiler inserts padding so each member begins at an address that is a multiple of its own size, and pads the total so that an array of the struct keeps every element aligned. `offsetof` from `<stddef.h>` reveals the actual layout.

10.3 Unions

📌 Key idea

A **union** is a user-defined type that allows several different data types to be stored *in the same memory location*. Because all members share that location, only one member holds meaningful data at any instant.

union definition

```
union <union_name>
{
    <data_type> <member_name1>;
    <data_type> <member_name2>;
    ...
} [variable_name];
```

unions.c

```
1 #include <stdio.h>
2
3 union Value {
4     int i;
5     float f;
6     char s[8];
7 };
8
9 int main(void)
10 {
```

```

11  union Value v;
12
13  printf("sizeof(union Value) = %zu (largest member)\n", sizeof v);
14
15  v.i = 1078530011;
16  printf("stored as int    : v.i = %d\n", v.i);
17  printf("read back as float: v.f = %f\n", (double) v.f);  /* same bits! */
18
19  v.f = 2.5f;
20  printf("after v.f = 2.5, v.i is now %d (the old value is gone)\n", v.i);
21  return 0;
22  }

```

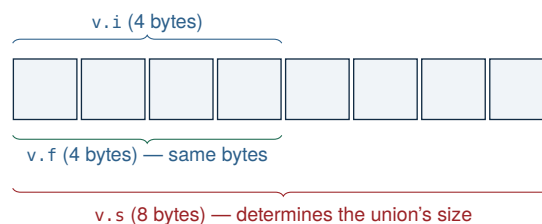
```

sizeof(union Value) = 8 (largest member)
stored as int    : v.i = 1078530011
read back as float: v.f = 3.141593
after v.f = 2.5, v.i is now 1075838976 (the old value is gone)

```

The bit pattern 1078530011 interpreted as an IEEE 754 float is π — a vivid demonstration that a union stores bits, not values.

union Value — one block, three views



Properties:

- **The size of a union is that of its largest member** (plus any padding needed for alignment).
- All members cannot be initialised at once; only the first member may be given an initialiser in a brace list.
- Changing one member changes all the others, because they occupy the same bytes.
- Member access uses the same `.` and `->` syntax as structures.
- The primary application is **saving memory** when a value can be one of several types but only ever one at a time.

10.3.1 The tagged union

Because a union does not remember which member is valid, the standard technique is to pair it with a tag:

```

struct Variant {
    enum { IS_INT, IS_FLOAT, IS_STRING } kind;  /* the tag */
    union { int i; float f; char s[32]; } data; /* the payload */
};

void print(const struct Variant *v)
{
    switch (v->kind) {
        case IS_INT:    printf("%d\n", v->data.i); break;
        case IS_FLOAT:  printf("%f\n", (double) v->data.f); break;
    }
}

```

```

        case IS_STRING: printf("%s\n", v->data.s); break;
    }
}

```

This is how interpreters represent values, how network protocol parsers handle variable message types, and how a generic container can hold mixed data.

10.3.2 A legitimate trick: detecting endianness

```

1  #include <stdio.h>
2
3  union Probe { int i; unsigned char b[4]; };
4
5  int main(void)
6  {
7      union Probe p;
8      p.i = 1;
9      printf("bytes: %d %d %d %d\n", p.b[0], p.b[1], p.b[2], p.b[3]);
10     printf("%s-endian\n", p.b[0] == 1 ? "little" : "big");
11     return 0;
12 }

```

```

bytes: 1 0 0 0
little-endian

```

x86 and ARM store the least significant byte first — **little-endian**. Network protocols use big-endian. This matters whenever bytes cross a machine boundary.

10.3.3 Structures versus unions

	struct	union
Memory	each member has its own storage	all members share one location
Size	sum of members plus padding	size of the largest member
Valid at once	all members	exactly one
Initialiser	every member may be given one	only the first
Purpose	group related data	save memory / reinterpret bits

10.4 Bit-fields

A structure member may be declared with an explicit width in bits.

```

1  #include <stdio.h>
2
3  struct Flags {
4      unsigned int is_active   : 1;    /* 1 bit */
5      unsigned int permissions : 3;    /* 3 bits, values 0-7 */
6      unsigned int user_id    : 12;   /* 12 bits, values 0-4095 */
7  };
8
9  int main(void)
10 {
11     struct Flags f = {1, 5, 2048};

```

```

12     printf("active=%u perms=%u uid=%u\n", f.is_active, f.permissions, f.user_id);
13     printf("sizeof(struct Flags) = %zu bytes for 16 bits of data\n", sizeof f);
14
15     f.permissions = 9;           /* 9 does not fit in 3 bits: 9 & 7 = 1 */
16     printf("perms after assigning 9 = %u\n", f.permissions);
17     return 0;
18 }

```

```

active=1 perms=5 uid=2048
sizeof(struct Flags) = 4 bytes for 16 bits of data
perms after assigning 9 = 1

```

Bit-fields are used for hardware registers, protocol headers and dense flag sets. Their layout — which end the compiler starts packing from, whether fields may straddle a storage unit — is implementation defined, so they are not portable across compilers. You cannot take the address of a bit-field.

10.5 typedef with structures

Writing `struct Student` everywhere is tiresome. `typedef` removes the keyword:

```

typedef struct {
    int id;
    char name[20];
} Student;           /* anonymous struct, named by the typedef */

Student s;           /* no 'struct' needed */

/* for self-referential types, the tag is still required inside */
typedef struct Node {
    int data;
    struct Node *next; /* 'Node' is not defined yet at this point */
} Node;

Node *head = NULL;

```

10.6 Arrays of structures and structures as arguments

```

records.c
1  #include <stdio.h>
2
3  typedef struct { int id; char name[16]; float gpa; } Student;
4
5  /* by value: the whole 24-byte struct is copied */
6  void show_copy(Student s) { printf("%3d %-10s %.2f\n", s.id, s.name, (double)s.gpa); }
7
8  /* by pointer: 8 bytes copied, and const documents that we do not modify it */
9  void show_ref(const Student *s) { printf("%3d %-10s %.2f\n", s->id, s->name, (double)s->gpa); }
10
11 float class_average(const Student a[], int n)
12 {
13     float sum = 0;
14     for (int i = 0; i < n; i++) sum += a[i].gpa;
15     return sum / n;
16 }
17

```

```

18 int main(void)
19 {
20     Student class[3] = {
21         {101, "Nimal", 3.75f},
22         {102, "Kamala", 3.90f},
23         {103, "Sunil", 3.20f}
24     };
25
26     for (int i = 0; i < 3; i++) show_ref(&class[i]);
27     show_copy(class[0]);
28     printf("average GPA = %.4f\n", (double) class_average(class, 3));
29     printf("sizeof(Student)=%zu sizeof(class)=%zu\n", sizeof(Student), sizeof class);
30     return 0;
31 }

```

```

101 Nimal      3.75
102 Kamala    3.90
103 Sunil     3.20
101 Nimal      3.75
average GPA = 3.6167
sizeof(Student)=24 sizeof(class)=72

```

Key idea

Prefer `const struct T *` over `struct T` for function parameters. It copies 8 bytes instead of the whole object, and `const` tells both the compiler and the reader that the argument is not modified. Pass by value only for genuinely small structs where the copy is the point.

10.7 Chapter summary

- A struct groups heterogeneous members; each has its own storage; access with `.` for a variable and `->` for a pointer (`p->m` $\equiv$ `(*p).m`).
- Structs may be assigned, passed and returned by value — but never compared with `==`.
- `sizeof` a struct exceeds the sum of its members because of **alignment padding**; ordering members largest-first minimises it.
- A struct may contain a *pointer* to its own type. This is what makes linked structures possible.
- A union stores all members at one address; its size is that of the largest member; only one member is valid at a time. Pair it with an enum tag to make it safe.
- Bit-fields pack members into named bit ranges; layout is implementation defined and their addresses cannot be taken.
- `typedef` removes the need to write struct repeatedly.

The Preprocessor, Macros and Multi-file Programs

The preprocessor runs before the compiler and knows nothing about C. It performs textual substitution, and that is both its power and its danger.

11.1 Directives

Any line whose first non-blank character is # is a preprocessor directive. Directives are not statements and are not terminated by a semicolon.

Directive	Effect
#include	insert the contents of another file
#define	define a macro
#undef	remove a macro definition
#ifdef / #ifndef	compile the following text only if a macro is / is not defined
#if / #elif / #else / #endif	conditional compilation on a constant expression
#error	abort compilation with a message
#pragma	implementation-specific instruction to the compiler

11.2 #include

```
#include <stdio.h>    /* angle brackets: search the SYSTEM include paths */
#include "stack.h"    /* quotes: search the current directory FIRST,
                      then the system paths */
```

The rule of thumb is: angle brackets for standard and third-party library headers, quotes for headers belonging to your own project. Add extra search directories with `gcc -I`.

11.3 Macros

Key idea

A **macro** is a name given to a piece of text. Wherever the preprocessor meets that name in the program, it replaces it with the text defined. Macros are not terminated by a semicolon.

There are two kinds.

11.3.1 Object-like macros

An object-like macro has a defined value and behaves as a named constant.

```
#define PI 3.146
#define MAX_STUDENTS 250
#define GREETING "Hello, world"
```

11.3.2 Function-like macros

A function-like macro takes arguments, which are substituted into the replacement text.

```
#define add(a, b)      (a + b)
#define min(a, b)      ((a) < (b)) ? (a) : (b)
#define area(r)        (PI * (r) * (r))
#define SQUARE(x)      ((x) * (x))
```

The main use case is efficiency: defining a constant or a tiny operation once and reusing it, rather than repeating the same text — with a macro there is no function-call overhead, because the expansion happens before compilation.

11.3.3 Predefined macros

__DATE__	compilation date, "Aug 9 2026"
__TIME__	compilation time, "18:24:07"
__FILE__	current source file name
__LINE__	current line number (an int)
__func__	current function name (C99)
__STDC__	1 in a conforming implementation
__STDC_VERSION__	201710L for C18

These make a serviceable debug macro:

```
#define TRACE(fmt, ...) \
    fprintf(stderr, "[%s:%d %s] " fmt "\n", __FILE__, __LINE__, __func__, __VA_ARGS__)

TRACE("i=%d total=%d", i, total);
/* [main.c:42 process] i=3 total=17 */
```

11.4 The three macro traps

Macros are text, not code. Every macro pitfall follows from that one fact.

✘ Trap 1: missing parentheses around parameters

```
#define SQUARE(x) x * x
int r = SQUARE(2 + 3);    /* expands to 2 + 3 * 2 + 3 = 11, not 25 */
```

Parenthesise *every* parameter:

```
#define SQUARE(x) ((x) * (x))
int r = SQUARE(2 + 3);      /* ((2+3) * (2+3)) = 25 */
```

✖ Trap 2: missing parentheses around the whole body

```
#define HALF(x) (x) / 2
int r = 10 / HALF(4);      /* 10 / (4) / 2 = 1, not 5 */

#define HALF(x) ((x) / 2)  /* correct */
```

✖ Trap 3: multiple evaluation of arguments

```
#define min(a, b) (((a) < (b)) ? (a) : (b))
int m = min(i++, j++);     /* one of them is incremented TWICE */
int n = min(f(), g());     /* one of the functions is CALLED TWICE */
```

A macro cannot avoid this: the argument text is duplicated wherever the parameter appears. **This is the decisive argument for using functions instead of macros** wherever a function will do — and with `inline` (C99), a function costs nothing extra.

	Function-like macro	Function
Processed by	preprocessor (text)	compiler (code)
Type checking	✖ none	✓ full
Arguments evaluated	possibly many times	exactly once
Call overhead	none — expanded inline	small; zero with <code>inline</code>
Debuggable	✖ gdb sees the expansion	✓
Works on any type	✓ (accidentally)	only the declared types
Recursion	✖	✓

11.5 Multi-statement macros

```
#define SWAP(a, b) { int t = a; a = b; b = t; }

if (x > y) SWAP(x, y);      /* expands to if (x>y) { ... }; -- the stray
                           semicolon breaks a following else */
```

The standard fix is the `do { ...} while (0)` idiom, which makes the macro a single statement that *requires* a terminating semicolon:

```
#define SWAP(a, b) do { int t = (a); (a) = (b); (b) = t; } while (0)

if (x > y) SWAP(x, y);
else     printf("already ordered\n");  /* now compiles correctly */
```

11.6 Stringification and token pasting

#x	replace with a string literal containing the argument text
a##b	paste the two tokens together to form one token

```
#define SHOW(x)    printf(#x " = %d\n", (x))
SHOW(a + b);      /* printf("a + b" " = %d\n", (a + b)); */

#define MAKE_VAR(n) int var_##n
MAKE_VAR(1);      /* int var_1; */
```

11.7 Conditional compilation and include guards

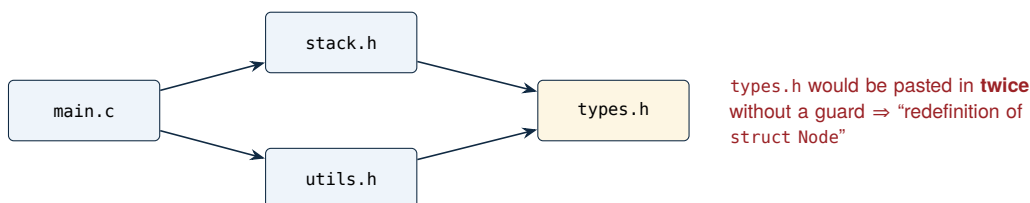
Key idea

When declaring headers it is essential that the same types are not defined twice. A header included from two places would otherwise be pasted in twice, and duplicate type definitions are an error. The preprocessor solves this with an **include guard**.

Include guard

```
#ifndef <HEADER_H>
#define <HEADER_H>
    // definitions
#endif
```

The first time the header is seen, `HEADER_H` is not defined, so the body is processed and the macro gets defined. On every subsequent inclusion the macro *is* defined, so the whole body is skipped.



stack.h --- a guarded header

```
1  #ifndef STACK_H
2  #define STACK_H
3
4  #define STACK_CAPACITY 100
5
6  typedef struct {
7      int data[STACK_CAPACITY];
8      int top;
9  } Stack;
10
11 void init(Stack *s);
12 int push(Stack *s, int value);
13 int pop(Stack *s, int *out);
14 int peek(const Stack *s, int *out);
15 int is_empty(const Stack *s);
16
17 #endif /* STACK_H */
```

Note

`#pragma` once achieves the same effect in one line and is supported by `gcc`, `clang` and `MSVC` — but it is not in the C standard. The `#ifndef` form is universally portable and is what this course expects.

Conditional compilation has other uses:

```
#ifndef DEBUG
    printf("x = %d\n", x);          /* only compiled with gcc -DDEBUG */
#endif

#if 0
    ...any code, including block comments... /* cleanly disabled */
#endif

#ifdef _WIN32
    #include <windows.h>
#else
    #include <unistd.h>
#endif
```

11.8 Writing a C program in multiple files

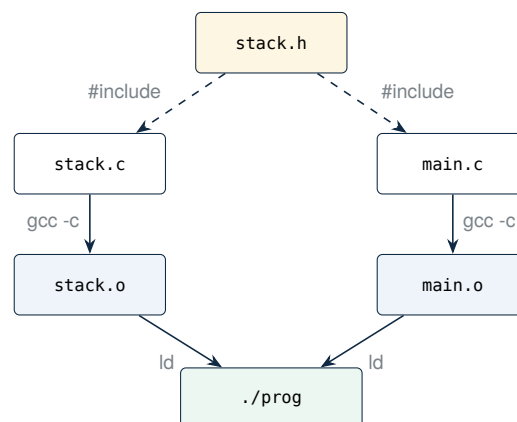
A C program may live in a single file or many. Multiple files come in two forms:

Header files `.h` *Declarations*: type definitions (primitive or user-defined), macros, and function prototypes that the `.c` files will use. Headers describe an interface.

Source files `.c` *Definitions*: the actual code and the actual variables. Source files provide an implementation.

Key idea

Headers say *what exists*; source files say *what it does*. A header should contain no code that allocates storage — no function bodies, no variable definitions — because it will be textually pasted into every file that includes it.



11.8.1 Compiling multiple files

To compile several files, list all the `.c` files:

```
$ gcc -Wall -Wextra -std=c18 -o prog main.c stack.c

# or separately, which is what make automates:
$ gcc -c stack.c          # -> stack.o
$ gcc -c main.c           # -> main.o
$ gcc -o prog main.o stack.o
```

Never compile a `.h` file, and never `#include` a `.c` file. Headers are pasted in; source files are compiled separately and joined by the linker.

11.9 extern: sharing across files

Key idea

`extern` increases the visibility of C variables and functions. It *declares* without *defining* — **no memory is allocated** — telling the compiler “this exists somewhere; the linker will find it”.

extern

```
extern <data_type> <variable_name>;
```

```
/* ---- counter.c : the single DEFINITION, allocates storage ---- */
int global_count = 0;

/* ---- counter.h : the DECLARATION, allocates nothing ---- */
#ifndef COUNTER_H
#define COUNTER_H
extern int global_count;    /* every including file now knows it exists */
void bump(void);
#endif
```

Functions are `extern` by default, which is why a prototype in a header needs no keyword.

✗ Defining a variable in a header

```
/* in shared.h -- WRONG */
int counter = 0;
```

Every `.c` file that includes this gets its own definition, and the linker reports multiple definition of `'counter'`. Put `extern int counter;` in the header and `int counter = 0;` in exactly one `.c` file.

11.9.1 static at file scope: the opposite of extern

Applying `static` to a global variable or a function restricts it to the file in which it is declared — it gets **internal linkage** and is invisible to the linker.

static function

```
static <return_type> <function_name>([params] ...)
{
    ...
}
```

```
static int helper_count = 0;          /* private to this file */
static void internal_helper(void) { ... } /* not callable from elsewhere */
```

The main use is exactly the one Lecture 7 gives: allowing the same function name to be used in different files without collision. More generally, marking everything that is not part of a module's interface as static is what gives C a form of encapsulation. Everything not static is part of your public API whether you intended it or not.

11.10 A complete multi-file example

counter.h

```
1 #ifndef COUNTER_H
2 #define COUNTER_H
3
4 extern int global_count;      /* declaration -- no storage */
5
6 void bump(void);
7 int current(void);
8
9 #endif /* COUNTER_H */
```

counter.c

```
1 #include "counter.h"
2
3 int global_count = 0;          /* the one definition */
4 static int internal_calls = 0; /* invisible outside this file */
5
6 static void log_call(void) { internal_calls++; }
7
8 void bump(void) { log_call(); global_count++; }
9 int current(void) { return global_count; }
```

main.c

```
1 #include <stdio.h>
2 #include "counter.h"
3
4 int main(void)
5 {
6     for (int i = 0; i < 5; i++) bump();
7     printf("global_count = %d\n", global_count);
8     printf("current()    = %d\n", current());
9     return 0;
10 }
```

```
$ gcc -Wall -Wextra -std=c18 -o prog main.c counter.c
$ ./prog
global_count = 5
current()    = 5

$ nm counter.o
000000000000001a T bump
0000000000000039 T current
0000000000000000 B global_count
0000000000000004 b internal_calls
0000000000000000 t log_call
```

The nm output shows the linkage difference directly. bump, current and global_count carry **upper-**

case letters — T for the text segment, B for bss — meaning they are global symbols the linker can match against references in other object files. `internal_calls` and `log_call` carry **lowercase** b and t, meaning they are local to this object file and the linker will never match them from outside. That is precisely what `static` did.

Note also that `global_count` lands in B (bss) rather than D (initialised data), because it was initialised to zero — zero-valued globals need no bytes in the executable, only a reservation. Chapter 12 returns to this.

11.11 Chapter summary

- The preprocessor performs textual substitution before compilation. Its directives start with # and take no semicolon.
- `#include <>` searches system paths, `#include ""` searches locally first.
- Macros are text. Parenthesise every parameter *and* the whole body, and beware multiple evaluation of arguments. Prefer functions.
- Use `do {...} while (0)` for multi-statement macros.
- Include guards (`#ifndef/#define/#endif`) prevent a header being pasted in twice.
- Headers hold declarations; source files hold definitions. Compile all .c files; never compile or include a .h as source.
- `extern` declares without allocating, sharing a symbol across files; `static` at file scope hides a symbol from other files.

Storage Classes and the Memory Layout of a Program

Every variable in C has three independent attributes: **scope** (where the name is visible), **lifetime** (how long the storage exists) and **linkage** (whether other translation units can see it). Storage class specifiers set them.

12.1 The four storage classes

Class	Stored in	Lifetime	Scope
auto	stack	until the block exits	local to the block
static	data segment	until the program ends	local, or file-scope with internal linkage
extern	data segment	until the program ends	across multiple files
register	a CPU register (if the compiler agrees)	until the block exits	local to the block

12.1.1 auto

auto

```
auto <data_type> <variable_name>;
```

`auto` specifies automatic storage duration: the memory is allocated from the stack when the block is entered and released when it exits. Since this is already the default for every local variable, **the `auto` keyword has no practical use in C**. You will essentially never see it written.

✗ `auto` means something completely different in C++

In C, `auto` is a storage class with no practical effect. In classic C++ (before C++11) it meant the same. In modern C++ (C++11 onwards) `auto` means *deduce this variable's type from its initialiser* — an entirely unrelated feature. Do not carry the C++ meaning into C.

12.1.2 static

`static` has two distinct meanings depending on where it is used. Both are examinable.

On a local variable it preserves the value even after the variable is no longer in scope.

static variable

```
static <data_type> <var_name> = value;
```

```

1  staticdemo.c
2  #include <stdio.h>
3
4  void counter_static(void)
5  {
6      static int c = 0;          /* initialised ONCE, at program start */
7      c++;
8      printf("static c = %d\n", c);
9  }
10
11 void counter_auto(void)
12 {
13     int c = 0;                  /* re-created and re-initialised every call */
14     c++;
15     printf("auto c = %d\n", c);
16 }
17
18 static int uninit_global;      /* not initialised -> automatically zero */
19 static int init_global = 42;
20
21 int main(void)
22 {
23     for (int i = 0; i < 3; i++) { counter_static(); counter_auto(); }
24     printf("uninitialised static global = %d (auto-zeroed)\n", uninit_global);
25     printf("initialised static global = %d\n", init_global);
26     return 0;
27 }

```

```

static c = 1
auto c = 1
static c = 2
auto c = 1
static c = 3
auto c = 1
uninitialised static global = 0 (auto-zeroed)
initialised static global = 42

```

Properties of static variables:

- They remain in memory as long as the program is running.
- They live in the **data segment**, not on the stack.
- They are **automatically initialised to zero** if you do not initialise them — unlike automatic variables, which contain garbage.
- The initialiser must be a constant expression, and runs once, before `main`.
- A static variable should not be declared inside a structure. (A *static structure variable* is perfectly acceptable — the restriction is on members.)

On a function or a file-scope variable it restricts visibility to the file in which it is declared — see Section 11.9. This is used mainly so that the same function name can be reused in other files without collision.

12.1.3 `extern`

Covered in Chapter 11: it declares a name defined elsewhere and allocates no memory. Its lifetime is the whole program and its scope spans files.

12.1.4 register

Registers are memory units built directly into the CPU, and are far faster to access than main memory. It is therefore a sensible practice to put frequently-used variables in registers.

register

```
register <data_type> <var_name> = <value>;
```

```
for (register int i = 0; i < 1000000; i++) { ... }
```

Facts to note:

- It is only a *request*. The compiler ultimately decides whether the variable goes in a register, and modern optimisers make far better decisions than a programmer can. In practice the keyword is obsolete.
- **You cannot take the address of a register variable.** `&r` is an error — registers have no memory address. This is the one remaining semantic effect of the keyword.
- Register storage cannot be static or global; it is only meaningful in a local context.

12.2 Scope, lifetime and linkage together

Declaration	Scope	Lifetime	Linkage
<code>int x;</code> inside a function	block	automatic	none
<code>static int x;</code> inside a function	block	static	none
<code>int x;</code> at file scope	file onwards	static	external
<code>static int x;</code> at file scope	file onwards	static	internal
<code>extern int x;</code>	file onwards	static	external (defined elsewhere)
<code>void f(void)</code>	program	—	external
<code>static void f(void)</code>	file	—	internal

Key idea

Scope is a *compile-time* property: which names the compiler can see. Lifetime is a *run-time* property: how long the storage exists. A static local has block scope but static lifetime — it is invisible outside the function, yet it survives between calls.

12.3 The memory layout of a running program

A C program's memory image consists of five sections.

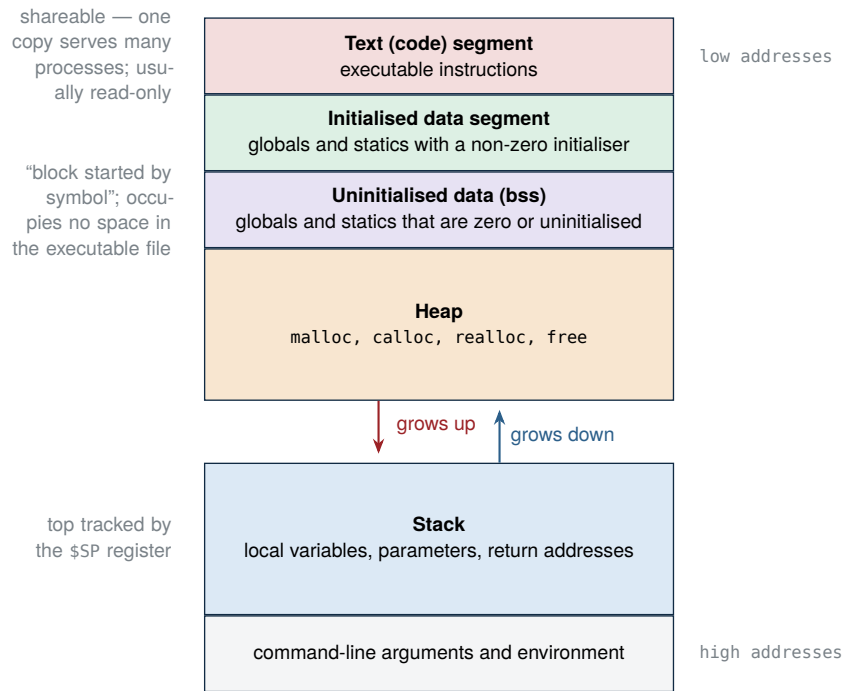


Figure 12.1. The five sections of a C program's address space.

Text segment

Contains the executable instructions. It is **shareable**, so only a single copy is required in physical memory even when many processes run the same program. It is usually marked read-only, so that a program cannot accidentally overwrite its own code.

Initialised data segment

Global and static variables that are initialised to a non-zero value. These values are stored in the executable file itself.

Uninitialised data segment (bss)

Named “block started by symbol”, a term inherited from an old assembler. It holds global and static variables that are uninitialised or explicitly set to zero. Because the value is known to be zero, the executable file need only record the *size* — which is why a program with a 100 KB zero-filled array is no larger on disk than one without it. The loader zeroes the region at start-up, which is exactly why static variables are guaranteed to start at zero.

Heap

Where dynamic memory allocation takes place, managed through `malloc`, `calloc`, `realloc` and `free` (Chapter 13). It grows towards higher addresses.

Stack

Contains the program stack: local variables, function parameters and return addresses. It grows towards lower addresses, and the top is tracked by the special-purpose stack pointer register, `$SP`. The set of values pushed for one function call is a **stack frame**.

12.3.1 Seeing the segments: the `size` command

The `size` command reports the sizes in bytes of the text, data and bss segments of an executable.

```

1  #include <stdio.h>
2
3  int initialised_global = 12345;    /* -> data */

```

```

4 int uninitialised_global;          /* -> bss */
5 static char big_buffer[100000];    /* -> bss */
6
7 int main(void) { printf("hi\n"); return 0; }

```

```

$ gcc -std=c18 -o sz sz.c
$ size sz
   text^I  data^I  bss^I   dec^I   hex^Ifilename
  1365^I   604^I 100064^I 102033^I 18e91^Isz

$ echo 'int main(void){return 0;}' > sz2.c && gcc -o sz2 sz2.c
$ size sz2
   text^I  data^I  bss^I   dec^I   hex^Ifilename
  1228^I   544^I    8^I   1780^I   6f4^Isz2

$ ls -l sz sz2
-rwxr-xr-x 1 user user 16072 sz
-rwxr-xr-x 1 user user 15776 sz2

```

The 100 KB buffer added 100,000 bytes to bss, yet the executable on disk grew by under 300 bytes. That is the whole point of the bss segment: the file records only “reserve 100,064 zero bytes”, and the loader supplies them at start-up.

12.4 The stack and stack frames

Each function call pushes a **stack frame** (also called an activation record) containing everything that call needs.

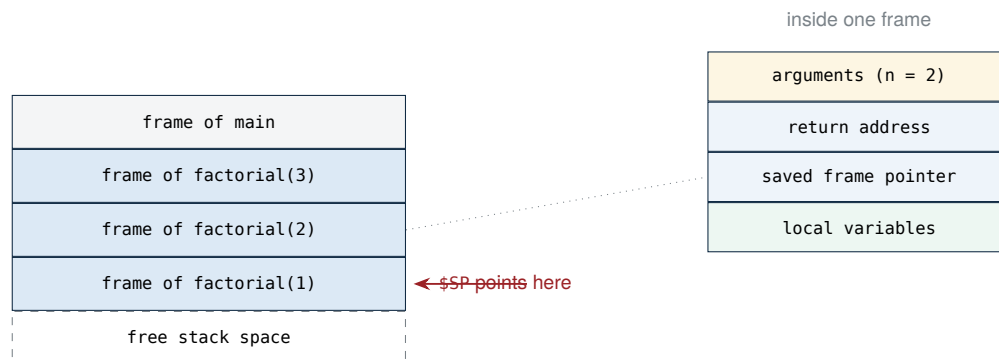


Figure 12.2. The call stack during factorial(3), and the contents of a single frame.

Because the frame is destroyed when the function returns, two rules follow, and both are absolute:

✗ Never return a pointer to a local variable

```

int *broken(void)
{
    int x = 42;
    return &x;          /* x dies when this function returns */
}
...
int *p = broken();
printf("%d\n", *p);     /* reads memory that has been reclaimed */

```

gcc -Wall warns: function returns address of local variable. The program may even print 42 — the bytes have not been overwritten *yet* — which makes this a bug that hides until the worst possible moment. Chapter 13 shows the correct alternatives.

✗ The stack is small and finite

Typically 8 MB on Linux. Two ways to exhaust it:

```
int huge[100000000];           /* 40 MB local array -> instant crash */
void f(void) { f(); }          /* infinite recursion -> stack overflow */
```

Both produce a segmentation fault. Large or long-lived data belongs on the heap.

12.5 Where does each variable live?

```
#include <stdlib.h>

int g_init = 5;           /* DATA segment */
int g_zero;               /* BSS segment */
static int s_file = 7;    /* DATA, internal linkage */
const char *msg = "hello"; /* msg -> DATA; "hello" -> TEXT (.rodata) */

void f(int param)         /* param -> STACK */
{
    int local = 1;        /* STACK */
    static int keeper = 2; /* DATA -- survives between calls */
    int *dyn = malloc(sizeof(int)); /* dyn -> STACK,
                                     *dyn -> HEAP */

    free(dyn);
}
```

✓ Exam focus

Be able to place any variable in the right segment on sight. The test is mechanical: is it local and non-static? Stack. Is it malloc'd? Heap. Is it global or static *with* a non-zero initialiser? Data. Global or static *without*, or zero? Bss. Is it a string literal or code? Text.

12.6 Chapter summary

- Four storage classes: auto (default, stack, no practical use), static (data segment, persists, zero-initialised), extern (declares a name defined elsewhere, allocates nothing), register (a hint; & is not allowed on it).
- Scope is compile time, lifetime is run time. A static local has block scope and program lifetime.
- Five memory sections: text (shareable, read-only code), initialised data, bss (zero/uninitialised, costs no file space), heap (grows up, malloc), stack (grows down, \$SP, one frame per call).
- size reports text, data and bss sizes.
- A stack frame holds arguments, return address, saved frame pointer and locals, and is destroyed on return — so never return a pointer to a local, and never put huge arrays on the stack.

Dynamic Memory Allocation

13.1 Why static allocation is not enough

Everything so far has been allocated at compile time. `int a[100];` reserves exactly one hundred integers, decided when you typed the number, and that decision cannot change while the program runs. Two problems follow:

- If you allocate for the worst case, you waste memory in every other case.
- If the real requirement exceeds your guess, the program simply cannot cope — and in C it will not tell you, it will corrupt memory.

Key idea

Dynamic memory allocation is the process of assigning memory to a program at *run time*, giving the programmer the control and capability to work with data structures of varying size. It increases both flexibility and efficiency.

Dynamic allocation occurs in both the stack and the heap. Memory for function calls comes from the stack automatically. When memory is allocated for non-function-call purposes — when *you* ask for it — it is assigned from the **heap**.

13.2 size_t

`size_t` is defined in `<stddef.h>` and used throughout `<stdio.h>`, `<stdlib.h>`, `<time.h>` and `<string.h>`. It is an unsigned integer type guaranteed to be big enough to hold the size of the largest possible object on the host system.

- On a 32-bit system it is typically equivalent to unsigned `int`; on a 64-bit system to unsigned `long` (or unsigned `long long`).
- `size_t` is **never negative**.
- Using it makes code portable, states clearly to the reader that the value is a size rather than an ordinary integer, and integrates cleanly with the many library functions and APIs that use it.
- Print it with `%zu`.

✗ Counting down with `size_t`

```
for (size_t i = n - 1; i >= 0; i--) { ... } /* INFINITE LOOP */
```

An unsigned value is never negative, so `i >= 0` is always true. When `i` is 0 and decrements, it wraps to a huge number. Either use a signed index for downward loops, or write `for (size_t i = n; i-- > 0; )`.

13.3 The four allocation functions

All four are declared in `<stdlib.h>`.

13.3.1 malloc

malloc

```
void *malloc(size_t size);
```

`malloc` — *memory allocation* — reserves a block of the specified size from the heap and returns a pointer to it.

- The size parameter is in **bytes**. Always compute it with `sizeof`.
- It returns a `void *`, which can be assigned to a pointer of any type without a cast in C.
- **It does not initialise the memory.** Whatever garbage was previously at that address is the initial content.
- If there is not enough space, it returns `NULL`.

```
int *p = malloc(5 * sizeof(int));    /* room for 5 ints */
if (p == NULL) { fprintf(stderr, "out of memory\n"); exit(1); }
```

 **Write `sizeof *p`, not `sizeof(int)`**

```
int *p = malloc(n * sizeof *p);    /* stays correct if p's type changes */
```

If the declaration later becomes `long *p`, the first form updates itself and the second silently allocates half the memory needed.

13.3.2 calloc

calloc

```
void *calloc(size_t nelem, size_t elsize);
```

`calloc` — *contiguous allocation* — differs from `malloc` in three ways:

- It takes **two** parameters: the number of elements and the size of each.
- It **initialises every byte to zero**.
- It assigns a specified number of blocks to a single pointer, whereas `malloc` creates one single block.

It also returns a `void *`, and `NULL` on failure. Because of the zeroing, `malloc` is faster than `calloc`.

```
int *a = calloc(5, sizeof *a);    /* 5 ints, all guaranteed 0 */
```

Note

`calloc` also checks `nelem * elsize` for overflow, which `malloc(n * size)` does not. For allocations whose count comes from untrusted input, that check matters.

13.3.3 realloc

realloc

```
void *realloc(void *ptr, size_t size);
```

realloc resizes a previously allocated block when the memory obtained from malloc or calloc proves insufficient. It preserves the existing contents up to the smaller of the old and new sizes, and returns NULL if the space cannot be allocated.

It may either extend the block in place or allocate a new block, copy the data and free the old one — so the returned pointer may differ from the one passed in.

✖ The realloc trap

```
p = realloc(p, newsize);      /* DANGEROUS */
```

If realloc fails it returns NULL — and you have just overwritten your only pointer to the original block, which is still allocated. The block is leaked and the data is unreachable. Always use a temporary:

```
int *tmp = realloc(p, newsize);
if (tmp == NULL) { /* handle failure; p is still valid */ }
else             { p = tmp; }
```

Two special cases: realloc(NULL, n) behaves as malloc(n), and realloc(p, 0) is equivalent to free(p) in most implementations (the standard leaves it unspecified — do not rely on it).

13.3.4 free

free

```
void free(void *ptr);
```

free deallocates memory that was allocated by malloc, calloc or realloc, returning it to the heap for reuse. It is strongly recommended practice to free everything you allocate.

- free(NULL) is explicitly safe and does nothing.
- Freeing a pointer twice, or freeing something not returned by an allocator, is undefined behaviour and typically aborts the program.
- free does not change the pointer variable — after freeing, the pointer still holds the old address and is now **dangling**. Set it to NULL.

13.4 A complete demonstration

dynamic.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     /* malloc: uninitialised */
7     int *a = malloc(5 * sizeof *a);
```

```

8     if (a == NULL) { fprintf(stderr, "allocation failed\n"); return 1; }
9
10    /* calloc: zeroed */
11    int *b = calloc(5, sizeof *b);
12    if (b == NULL) { free(a); return 1; }
13
14    printf("calloc'd values: ");
15    for (int i = 0; i < 5; i++) printf("%d ", b[i]);
16    printf("\n");
17
18    for (int i = 0; i < 5; i++) a[i] = i * i;
19
20    /* realloc: grow to 10, keeping the first 5 */
21    int *tmp = realloc(a, 10 * sizeof *a);
22    if (tmp == NULL) { free(a); free(b); return 1; }
23    a = tmp;
24
25    for (int i = 5; i < 10; i++) a[i] = i * i;
26
27    printf("after realloc: ");
28    for (int i = 0; i < 10; i++) printf("%d ", a[i]);
29    printf("\n");
30
31    free(a); a = NULL;
32    free(b); b = NULL;
33    return 0;
34 }

```

```

calloc'd values: 0 0 0 0 0
after realloc:   0 1 4 9 16 25 36 49 64 81

```

	malloc	calloc	realloc
Parameters	one: total bytes	two: count and element size	pointer and new size
Initialises?	✗ garbage	✓ all zero	only the newly added part is uninitialised
Speed	faster	slower (must zero)	varies
On failure	NULL	NULL	NULL, original block untouched

13.5 Dynamic arrays

13.5.1 One dimension

```

int n;
scanf("%d", &n);                      /* size known only at run time */
int *arr = malloc(n * sizeof *arr);
if (!arr) { ... }

for (int i = 0; i < n; i++) arr[i] = i; /* index it exactly like an array */

free(arr);

```

13.5.2 Two dimensions

```

dyn2d.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int rows = 3, cols = 4;
7
8      /* an array of row pointers, each pointing at a row */
9      int **m = malloc(rows * sizeof *m);
10     for (int i = 0; i < rows; i++)
11         m[i] = malloc(cols * sizeof **m);
12
13     for (int i = 0; i < rows; i++)
14         for (int j = 0; j < cols; j++)
15             m[i][j] = i * cols + j;
16
17     for (int i = 0; i < rows; i++) {
18         for (int j = 0; j < cols; j++) printf("%3d", m[i][j]);
19         printf("\n");
20     }
21
22     /* free in reverse order: rows first, then the array of pointers */
23     for (int i = 0; i < rows; i++) free(m[i]);
24     free(m);
25     return 0;
26 }

```

```

0  1  2  3
4  5  6  7
8  9 10 11

```

Note

This “array of pointers” layout is *not* contiguous — each row lives somewhere different on the heap, so it loses the cache benefit of a true 2-D array. The alternative is one flat block with manual index arithmetic:

```

int *flat = malloc(rows * cols * sizeof *flat);
flat[i * cols + j] = value;          /* row-major by hand */
free(flat);                          /* one free, not rows+1 */

```

13.6 Memory leaks

Key idea

A **memory leak** occurs when the programmer allocates memory in the heap and forgets to delete it, leaving that memory allocated for as long as the program runs.

```

void leaky(void)
{
    int *p = malloc(1000 * sizeof *p);
    /* ... use p ... */
}

```

```

    return;                /* p goes out of scope; the 4000 bytes do not */
}

```

When `leaky` returns, the pointer variable `p` is destroyed along with the stack frame — but the heap block it referred to is still marked as allocated, and now nothing in the program knows its address. It can never be freed.

Consequences:

- For a short-lived program, little harm: the OS reclaims everything on exit.
- For a program that never terminates — a server, an operating system, an embedded controller — this is a major problem. Leaks accumulate, reduce the usable heap, and eventually cause allocation failures, thrashing, or a crash.

13.6.1 Detecting leaks

Leaks *can* be found manually, for instance during code review where each allocation and deallocation is checked — but this involves significant effort and time, and scales badly. Automated tools make detection far easier in large code bases. **Valgrind** is the standard choice.

```

$ sudo apt install valgrind
$ gcc -g -O0 -o leaky leaky.c
$ valgrind --leak-check=full ./leaky

```

```

==12345== HEAP SUMMARY:
==12345==    in use at exit: 4,000 bytes in 1 blocks
==12345== total heap usage: 1 allocs, 0 frees, 4,000 bytes allocated
==12345==
==12345== 4,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4848899: malloc (vg_replace_malloc.c:381)
==12345==    by 0x109151: leaky (leaky.c:5)
==12345==    by 0x10916E: main (leaky.c:11)
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 4,000 bytes in 1 blocks

```

Valgrind names the file and line of the allocation that leaked. Chapter 18 covers it in detail.

The allocation contract

Every `malloc` must have exactly one matching `free`. Decide, when you write the allocation, *who owns this memory and where will it be freed* — and write that down in a comment if it is not obvious.

13.7 Aliasing

Aliasing occurs when the same memory location is accessed using different names.

```

aliasing.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = malloc(sizeof *a);

```

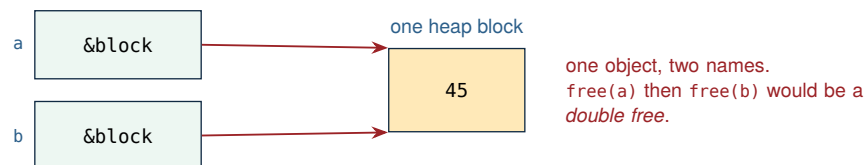
```

7  int *b = a;           /* b is an ALIAS for the same block */
8
9  *a = 25;
10 *b = 45;
11 printf("%d\n", *a);    /* ?? */
12
13 free(a);
14 return 0;
15 }

```

45

The answer is 45, not 25. There is only *one* integer; a and b are two names for it, so the second assignment overwrites the first.



Aliasing is not a bug in itself — it is how linked structures work — but it means you must be clear about *ownership*: which alias is responsible for freeing.

13.8 Dangling pointers

Key idea

When a pointer points to a deleted or freed memory location, it is called a **dangling pointer**. Dangling pointers lead to unpredictable program behaviour.

They arise in two ways.

13.8.1 From memory deallocation

```

int *p = malloc(sizeof *p);
*p = 42;
free(p);
printf("%d\n", *p);    /* DANGLING: use after free */
free(p);               /* DANGLING: double free */

```

The fix is one line, and it costs nothing:

```

free(p);
p = NULL;              /* now a mistake CRASHES instead of corrupting */

```

Because `free(NULL)` is safe, this also makes accidental double frees harmless.

13.8.2 From scoping issues

```
int *dangling(void)
{
    int local = 42;
    return &local;    /* the frame is destroyed on return */
}
```

Same disease, different cause. See Chapter 12.

Fault	What it is	Symptom
Memory leak	allocated, never freed	memory use grows; eventual failure
Dangling pointer	points to freed or dead memory	garbage values, crashes, corruption
Double free	the same block freed twice	heap corruption, abort
Wild pointer	never initialised	random crash or silent corruption
Buffer overrun	written past the end of a block	corrupts adjacent data

✓ Exam focus

Be able to define and distinguish *memory leak*, *dangling pointer*, *aliasing* and *wild pointer*, and to state one cause of each. These four terms are the vocabulary of the dynamic memory section of the syllabus.

13.9 Correct allocation patterns

patterns.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 /* returning heap memory: the CALLER becomes the owner and must free it */
6 char *duplicate(const char *src)
7 {
8     size_t n = strlen(src) + 1;    /* +1 for the terminator */
9     char *copy = malloc(n);
10    if (copy == NULL) return NULL;    /* let the caller decide what to do */
11    memcpy(copy, src, n);
12    return copy;
13 }
14
15 /* growing an array as data arrives */
16 int main(void)
17 {
18     size_t cap = 2, len = 0;
19     int *v = malloc(cap * sizeof *v);
20     if (!v) return 1;
21
22     for (int i = 1; i <= 9; i++) {
23         if (len == cap) {    /* full: double the capacity */
24             size_t newcap = cap * 2;
25             int *tmp = realloc(v, newcap * sizeof *v);
26             if (!tmp) { free(v); return 1; }
27             v = tmp; cap = newcap;
28         }
29         v[len++] = i * i;
```

```

30     }
31
32     printf("len=%zu cap=%zu:", len, cap);
33     for (size_t i = 0; i < len; i++) printf(" %d", v[i]);
34     printf("\n");
35     free(v);
36
37     char *d = duplicate("owned by the caller");
38     if (d) { printf("%s\n", d); free(d); }
39     return 0;
40 }

```

```

len=9 cap=16: 1 4 9 16 25 36 49 64 81
owned by the caller

```

The doubling strategy in that loop is worth noting: it is what gives a growable array *amortised* $O(1)$ insertion, analysed in Section 26.10.

13.10 Chapter summary

- Static allocation fixes sizes at compile time; dynamic allocation obtains memory from the heap at run time.
- `size_t` is the unsigned type for sizes; never negative; print with `%zu`; do not count down with it naively.
- `malloc(size)` — uninitialised. `calloc(n, size)` — zeroed, two arguments, slower. `realloc(p, size)` — resize, may move. `free(p)` — release.
- Every allocator returns `NULL` on failure. Check it every time.
- Never write `p = realloc(p, n)`; use a temporary.
- Every `malloc` needs exactly one `free`. Set the pointer to `NULL` afterwards.
- A memory leak is allocated-and-never-freed; a dangling pointer points at freed or dead memory; aliasing is two names for one object.
- Valgrind (`valgrind --leak-check=full`) finds all of these automatically.

File Operations

14.1 Why file handling exists

Everything a program has done so far disappears when it ends. Variables live in memory, and memory is reclaimed the moment the process exits. Files let data **persist** beyond a single run.

Programs need to:

- Save results — logs, records, configuration.
- Read input that is too large to type in every time.
- Share data between two different programs.

14.2 The FILE pointer

```
FILE *fp;
```

A `FILE *` is declared like any other pointer, but with type `FILE`. `FILE` is not a basic type like `int` — it is a `struct` typedef'd inside the standard library header `<stdio.h>`. It stores the information the library needs to manage an open file: the current position in the file, whether an error or end-of-file has occurred, and buffering information.

Recall from Chapter 7 that pointer types matter: an `int *` cannot correctly point at a `float`. A `FILE *` is just a pointer too — instead of pointing at an `int` or a `float`, it points at a special `struct` that keeps track of an open file. That is why its type must always be `FILE *`.

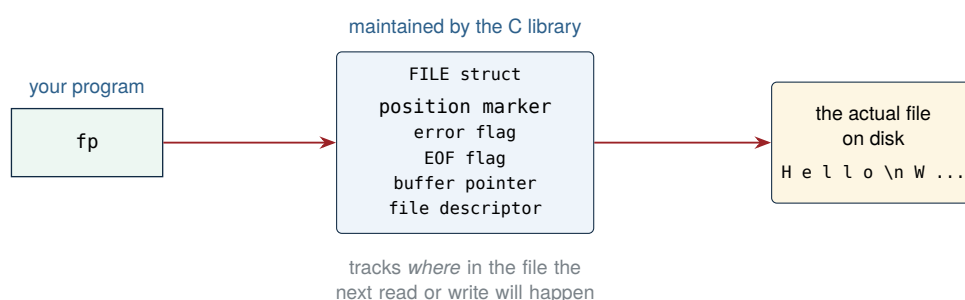


Figure 14.1. Your `FILE *fp` points at a `FILE struct` maintained by the C library; that struct in turn tracks the actual file on disk, plus a position marker.

14.3 Opening and closing

fopen

```
FILE *fopen(const char *filename, const char *mode);
```

`fopen` returns a `FILE *` pointing to the opened file, or `NULL` if the file could not be opened.

```
FILE *fp = fopen("data.txt", "r");
if (fp == NULL) {
    printf("Could not open file\n");
    return 1;
}
```

✗ Always check for NULL

The file may not exist, you may lack permission, the disk may be full, or the path may be wrong. If you skip the check, the first `fgetc(fp)` dereferences a null pointer and the program dies with no useful message. `perror` turns the failure into a readable explanation:

```
FILE *fp = fopen(name, "r");
if (!fp) { perror(name); return 1; }
/* prints e.g. data.txt: No such file or directory */
```

fclose

```
int fclose(FILE *fp);
```

Writes may be **buffered** — held in memory rather than sent to disk immediately, because disk operations are expensive and batching them is far faster. `fclose` flushes that buffer, guaranteeing your data is saved, and releases the OS resource.

Not closing opened files can cause:

- lost or incomplete data — the buffer is discarded rather than written;
- resource leaks in long-running programs — every process has a limit on how many files it may have open at once.

Use `fflush(fp)` if you need the data on disk without closing.

14.4 The six file modes

`fopen` takes a mode as its second argument, which decides to what extent the file can be used.

Single-purpose	Read + write
"r" : read only	"r+" : read and write
"w" : write only	"w+" : write and read
"a" : append (write only)	"a+" : read and append

Mode	If the file exists	If the file is missing	Access
r	opens, keeps content	error (NULL)	read only
w	overwrites content	creates the file	write only
a	keeps, writes at the end	creates the file	write only (append)
r+	keeps content	error (NULL)	read + write
w+	overwrites content	creates the file	read + write
a+	keeps, writes at the end	creates the file	read + write (append)

✘ "w" destroys data immediately

Opening an existing file in "w" or "w+" truncates it to zero length *at the moment of opening* — before you write anything, and even if you write nothing at all. If you meant to add to a file, the mode is "a".

Appending b to any mode ("rb", "wb", "ab+") opens the file in **binary mode**, which is covered in Section 14.10.

14.5 Reading from a file

"r" mode requires that the file already exists. Three functions read from it, at three different granularities.

14.5.1 One character: fgetc

fgetc

```
int fgetc(FILE *fp);
```

Reads a single character and returns it, or the constant EOF when there is nothing left to read.

readchars.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     FILE *fp = fopen("data.txt", "r");
6     if (fp == NULL) { perror("data.txt"); return 1; }
7
8     int ch;                                /* int, NOT char -- see below */
9     while ((ch = fgetc(fp)) != EOF)
10         putchar(ch);
11
12     fclose(fp);
13     return 0;
14 }
```

✘ fgetc returns int, not char

This is the single most important detail in the chapter, and Exercise V question 7(b) is built around it.

EOF is a negative value, conventionally -1 . fgetc must be able to return any of the 256 possible byte values *and* a distinct end-of-file signal — 257 outcomes, which do not fit in one byte. So it returns an int.

If you store the result in a char:

```
char ch;                                /* WRONG */
while ((ch = fgetc(fp)) != EOF) { ... }
```

two things can go wrong. On a platform where char is signed, a legitimate data byte $0xFF$ narrows to -1 and is mistaken for EOF, truncating the file early — which is precisely what happens when copying a binary file such as a JPEG. On a platform where char is unsigned, the real EOF (-1) becomes 255 and never compares equal to EOF, so the loop never ends.

14.5.2 One line: fgets

fgets

```
char *fgets(char *str, int n, FILE *fp);
```

Reads up to $n-1$ characters, or until a newline, whichever comes first, and stores a `'\0'` at the end automatically. Returns `str`, or `NULL` at end of file.

- The newline character **is included** in the string if it was read.
- If the line is longer than the given buffer, `fgets` stops early and the rest of the line is left for the next call.

numbered.c --- Exercise V, 1(c)

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     FILE *fp = fopen("data.txt", "r");
6     if (!fp) { perror("data.txt"); return 1; }
7
8     char line[256];
9     int n = 1;
10    while (fgets(line, sizeof line, fp) != NULL)
11        printf("%d: %s", n++, line);    /* no \n: fgets kept the original */
12
13    fclose(fp);
14    return 0;
15 }
```

```
$ printf 'This is the first line\nThis is the second line\n' > data.txt
$ ./numbered
1: This is the first line
2: This is the second line
```

14.5.3 Formatted data: fscanf

fscanf

```
int fscanf(FILE *fp, const char *format, ...);
```

The same rules as `scanf`, with the stream as an extra first argument. Useful when the file has a predictable, structured format.

```
FILE *fp = fopen("data.txt", "r");
int id; char name[50];
while (fscanf(fp, "%d %49s", &id, name) == 2)
    printf("ID: %d, Name: %s\n", id, name);
fclose(fp);
```

Note

Test `fscanf`'s return value against the *number of items expected*, not against `E0F`. Comparing with `!= E0F` loops forever on malformed input, because a conversion failure returns 0, not `E0F` — the same trap as Chapter 3.

14.6 Writing to a file

"w" mode overwrites an existing file or creates a missing one. Three functions mirror the readers.

<code>int fputc(int ch, FILE *fp)</code>	writes a single character
<code>int fputs(const char *s, FILE *fp)</code>	writes a string — does not add a newline; you must include <code>"\n"</code> yourself
<code>int fprintf(FILE *fp, const char *fmt, ...)</code>	like <code>printf</code> , but to a file

writing.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp = fopen("output.txt", "w");
6      if (!fp) { perror("output.txt"); return 1; }
7
8      fputc('H', fp);                /* one character */
9
10     char str[] = "ello, File!";
11     for (int i = 0; str[i] != '\0'; i++) /* a string, one char at a time */
12         fputc(str[i], fp);
13
14     fputs("\nSecond line\n", fp);    /* a string, with explicit newline */
15     fprintf(fp, "Name: %s\nAge: %d\n", "Anne", 20); /* formatted */
16
17     fclose(fp);
18     return 0;
19 }
```

```

$ ./writing && cat output.txt
Hello, File!
Second line
Name: Anne
Age: 20
```

14.7 Appending

"a" mode adds new data after the existing content, and creates the file if it does not exist. The same three functions are used, but *every write lands at the end* regardless of where the position marker is.

log.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp = fopen("log.txt", "a");
6      if (!fp) { perror("log.txt"); return 1; }
7      fprintf(fp, "New entry\n");    /* the \n matters -- see below */
8      fclose(fp);
9      return 0;
10 }
```

```
$ ./log ; ./log ; ./log ; cat log.txt
New entry
New entry
New entry
```

Without the "\n", each append lands on the same line: New entryNew entryNew entry.

14.8 Moving around a file

14.8.1 fseek

fseek

```
int fseek(FILE *fp, long offset, int position);
```

Moves the file's internal position marker, so that the next read or write happens from a different spot. The third argument says where to count from:

SEEK_SET	from the beginning of the file
SEEK_CUR	from the current position
SEEK_END	from the end of the file

```
fseek(fp, 0, SEEK_SET); /* go to the very start */
fseek(fp, 5, SEEK_CUR); /* move 5 bytes forward from here */
fseek(fp, -3, SEEK_CUR); /* move 3 bytes back */
fseek(fp, 0, SEEK_END); /* go to the very end */
```

14.8.2 ftell

ftell

```
long ftell(FILE *fp);
```

Returns the current position in bytes from the start of the file. Combined with fseek it measures a file's size:

filesize.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     FILE *fp = fopen("data.txt", "r");
6     if (!fp) { perror("data.txt"); return 1; }
7
8     fseek(fp, 0, SEEK_END);
9     long size = ftell(fp);
10    printf("File size: %ld bytes\n", size);
11
12    fclose(fp);
13    return 0;
14 }
```

14.8.3 rewind

`rewind(fp)` resets the position to the beginning. It is equivalent to `fseek(fp, 0, SEEK_SET)`, except that it also clears the error and end-of-file indicators and returns nothing.

14.9 The read/write modes

14.9.1 "r+" — read and write

Opens an *existing* file for both reading and writing. It does not create the file if it is missing, and does not overwrite the existing content. Use `fseek` to move to a position before reading or writing.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp = fopen("data.txt", "r+");
6      if (fp == NULL) {
7          printf("File must already exist for r+\n");
8          return 1;
9      }
10
11     fseek(fp, 5, SEEK_SET);    /* move to the 6th byte */
12     fputs("XXXXX", fp);       /* overwrite 5 characters */
13     fclose(fp);
14     return 0;
15 }

```

```

$ printf 'ABCDEFGHJKLMNOP\n' > data.txt
$ ./overwrite && cat data.txt
ABCDEXXXXKLMNOP

```

Only the bytes at that position are overwritten; everything else in the file stays untouched.

Why you cannot *insert* into a file

Exercise V question 3(b) asks you to insert a string without losing the original content, and to explain why positioning and writing directly is not sufficient.

The reason is that a file is a flat sequence of bytes with no gaps. Writing at position p *replaces* the bytes at p ; there is no operation that pushes the remaining bytes further along. To insert k bytes you must physically move everything after p forward by k :

1. Read and save the tail of the file from position p onwards.
2. Seek back to p and write the new string.
3. Write the saved tail immediately after it.

Part (c) then asks what happens when the string to insert is longer than the buffer set aside for the tail — the answer is that the tail is truncated and data is lost, which is why real implementations either use a temporary file or copy in fixed-size chunks working *backwards* from the end.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>

```

```

4
5 int insert_at(const char *path, long pos, const char *text)
6 {
7     FILE *fp = fopen(path, "r+");
8     if (!fp) { perror(path); return -1; }
9
10    /* how many bytes follow the insertion point? */
11    fseek(fp, 0, SEEK_END);
12    long end = ftell(fp);
13    if (pos > end) { fclose(fp); return -1; }
14    long tail_len = end - pos;
15
16    /* allocate exactly the space needed -- no fixed-size buffer to overflow */
17    char *tail = malloc((size_t) tail_len + 1);
18    if (!tail) { fclose(fp); return -1; }
19
20    fseek(fp, pos, SEEK_SET);
21    size_t got = fread(tail, 1, (size_t) tail_len, fp);
22    tail[got] = '\0';
23
24    fseek(fp, pos, SEEK_SET);
25    fputs(text, fp);          /* the inserted string */
26    fwrite(tail, 1, got, fp); /* the preserved tail */
27
28    free(tail);
29    fclose(fp);
30    return 0;
31 }
32
33 int main(void)
34 {
35     insert_at("data.txt", 5, "[INSERTED]");
36     return 0;
37 }

```

```

$ printf 'ABCDEFGHIIJ\n' > data.txt
$ ./insert && cat data.txt
ABCDE[INSERTED]FGHIIJ

```

14.9.2 "w+" — write and read

Opens a file for both reading and writing, overwriting an existing file or creating it if missing.

- **Difference from "w":** you can also read back what was written.
- **Difference from "r+":** "w+" always truncates or creates, whereas "r+" requires the file to exist and preserves it.

After writing, the position is at the end — trying to read immediately would just hit EOF. `rewind` fixes that.

```

wplus.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     FILE *fp = fopen("data.txt", "w+");
6     if (!fp) { perror("data.txt"); return 1; }
7
8     fputs("Hello World", fp);

```

```

9
10     rewind(fp);                /* position back to the start */
11
12     char buffer[50];
13     fgets(buffer, sizeof buffer, fp);
14     printf("%s\n", buffer);
15
16     fclose(fp);
17     return 0;
18 }

```

Hello World

14.9.3 "a+" — read and append

Opens a file for both reading and appending. If the file exists, its content is preserved and reading is allowed anywhere, but **all writes still go to the end**. If the file does not exist, it is created.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fp = fopen("log.txt", "a+");
6      if (!fp) { perror("log.txt"); return 1; }
7
8      fprintf(fp, "New log entry\n");
9
10     rewind(fp);
11     char buffer[100];
12     while (fgets(buffer, sizeof buffer, fp) != NULL)
13         printf("%s", buffer);
14
15     fclose(fp);
16     return 0;
17 }

```

```

$ rm -f log.txt; ./aplust; ./aplust
New log entry
New log entry
New log entry

```

The second run prints two lines because the first run's entry is still there — demonstrating both the append and the read.

14.10 Binary files

Appending `b` to the mode string — `"rb"`, `"wb"`, `"ab"` — opens the file in binary mode. Binary mode tells the library to transfer the bytes of a file exactly as they are, without any of the line-ending translation that text mode may otherwise perform (on Windows, text mode converts between `\n` and `\r\n`; on Linux there is no difference).

 **Key idea**

The C library places no restriction on what a file may contain. Whether a file is treated as text is a decision made by the *programmer*, not something enforced by the language.

fread / fwrite

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *fp);
```

Both return the number of *items* (not bytes) successfully transferred.

binarycopy.c --- Exercise V, 7(b)

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      if (argc != 3) { fprintf(stderr, "usage: %s src dst\n", argv[0]); return 1; }
6
7      FILE *in = fopen(argv[1], "rb");
8      FILE *out = fopen(argv[2], "wb");
9      if (!in || !out) { perror("open"); return 1; }
10
11     int ch;                                /* int, so 0xFF is not mistaken for EOF */
12     while ((ch = fgetc(in)) != EOF)
13         fputc(ch, out);
14
15     fclose(in); fclose(out);
16     return 0;
17 }
```

Writing structures directly is the other common use of binary mode:

```
typedef struct { int id; char name[20]; float gpa; } Student;

Student s = {101, "Nimal", 3.75f};
FILE *fp = fopen("students.dat", "wb");
fwrite(&s, sizeof s, 1, fp);          /* the raw 28 bytes */
fclose(fp);

fp = fopen("students.dat", "rb");
Student t;
fread(&t, sizeof t, 1, fp);           /* read them straight back */
fclose(fp);
```

✗ Common pitfall

A file written this way is only readable by a program compiled with the same struct layout, the same padding and the same endianness. It is fast and compact but not portable. For data that must cross machines, write a text format.

14.11 Error handling

feof(fp)	non-zero if the end-of-file indicator is set — <i>after</i> a read has already failed
ferror(fp)	non-zero if the error indicator is set
clearerr(fp)	clears both indicators
perror(s)	prints s, a colon, and a description of the last system error

✖ while (!feof(fp)) is always wrong

```
while (!feof(fp)) {                               /* WRONG: processes the last line twice */
    fgets(line, sizeof line, fp);
    printf("%s", line);
}
```

feof becomes true only *after* a read has already hit the end. On the final iteration fgets fails, leaves line unchanged, and the stale contents are printed again. Test the return value of the read itself:

```
while (fgets(line, sizeof line, fp) != NULL) { ... } /* correct */
```

14.12 Two utility programs

14.12.1 Word count — Exercise V, 4(b)

```
WC.C
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main(int argc, char *argv[])
5  {
6      if (argc != 2) { fprintf(stderr, "usage: %s file\n", argv[0]); return 1; }
7      FILE *fp = fopen(argv[1], "r");
8      if (!fp) { perror(argv[1]); return 1; }
9
10     long lines = 0, words = 0, chars = 0;
11     int ch, in_word = 0;
12
13     while ((ch = fgetc(fp)) != EOF) {
14         chars++;
15         if (ch == '\n') lines++;
16         if (isspace(ch)) in_word = 0;
17         else if (!in_word) { in_word = 1; words++; }
18     }
19
20     printf("%7ld %7ld %7ld %s\n", lines, words, chars, argv[1]);
21     fclose(fp);
22     return 0;
23 }
```

```
$ printf 'hello world\nsecond line here\n' > t.txt
$ ./wc t.txt
      2      5      29 t.txt
$ wc t.txt
  2  5 29 t.txt
```

The `in_word` flag is the standard state machine for counting words: a word begins at each transition from whitespace to non-whitespace.

14.12.2 Merging two sorted files — Exercise V, 5(a)

merge.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      FILE *fa = fopen("a.txt", "r");
6      FILE *fb = fopen("b.txt", "r");
7      FILE *out = fopen("merged.txt", "w");
8      if (!fa || !fb || !out) { perror("open"); return 1; }
9
10     int x, y;
11     int have_x = (fscanf(fa, "%d", &x) == 1);
12     int have_y = (fscanf(fb, "%d", &y) == 1);
13
14     /* compare the next unread integer from each file and emit the smaller */
15     while (have_x && have_y) {
16         if (x <= y) { fprintf(out, "%d\n", x); have_x = (fscanf(fa, "%d", &x) == 1); }
17         else      { fprintf(out, "%d\n", y); have_y = (fscanf(fb, "%d", &y) == 1); }
18     }
19     while (have_x) { fprintf(out, "%d\n", x); have_x = (fscanf(fa, "%d", &x) == 1); }
20     while (have_y) { fprintf(out, "%d\n", y); have_y = (fscanf(fb, "%d", &y) == 1); }
21
22     fclose(fa); fclose(fb); fclose(out);
23     return 0;
24 }

```

```

$ printf '1\n4\n7\n9\n' > a.txt ; printf '2\n3\n8\n' > b.txt
$ ./merge && cat merged.txt
1
2
3
4
7
8
9

```

Only one integer from each file is ever held in memory. This is the **merge** step of merge sort (Section 25.2.4), and it is how sorting works on data too large to fit in RAM.

14.13 Chapter summary

- Files make data persist beyond one run of a program.
- `FILE *` points at a library-maintained struct holding the position marker, error and EOF flags, and buffering state.
- `fopen` returns `NULL` on failure — always check. `fclose` flushes the buffer and releases the resource.
- Six modes. `r` needs the file to exist; `w` truncates; `a` appends; the `+` forms add the other access.
- Read with `fgetc` / `fgets` / `fscanf`; write with `fputc` / `fputs` / `fprintf`.
- **`fgetc` returns `int`**, so that all 256 byte values and EOF are distinguishable. Storing it in a `char` breaks binary files.
- `fseek` / `ftell` / `rewind` move and report the position; `SEEK_SET`, `SEEK_CUR`, `SEEK_END`.
- Writing at a position overwrites; inserting requires shifting the tail.
- Never loop on `while (!feof(fp))` — test the read instead.

Command-Line Arguments

15.1 argc and argv

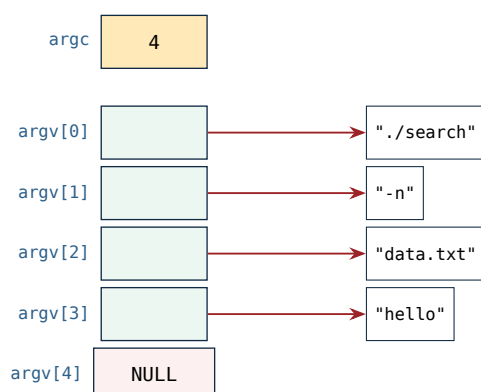
Command-line arguments are handled by the `main` function. To receive them, `main` is declared with two parameters: the *number* of command-line arguments, and the *list* of them.

main with arguments

```
int main(int argc, char *argv[])
{
    ...
}
```

`char *argv[]` is an array of pointers to strings — exactly the structure of Section 7.7.7. `char **argv` means the same thing.

```
$ ./search -n data.txt hello
```



`argv[0]` is the program name as invoked.
The array is always terminated by a NULL pointer.

args.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("argc = %d\n", argc);
6     for (int i = 0; i < argc; i++)
7         printf("argv[%d] = \"%s\"\n", i, argv[i]);
8     printf("argv[%d] = %s\n", argc, argv[argc] == NULL ? "NULL" : "?");
9     return 0;
10 }
```

```
$ ./args -n data.txt hello
argc = 4
argv[0] = "./args"
argv[1] = "-n"
```

```
argv[2] = "data.txt"
argv[3] = "hello"
argv[4] = NULL
```

Key idea

`argc` is always at least 1, because `argv[0]` holds the program name. Arguments are *always* strings — even `./prog 42` gives you the two-character string `"42"`, never the integer 42. Convert them yourself.

15.2 Converting arguments

<code>atoi(s)</code>	string to int. Simple, but returns 0 for invalid input, indistinguishable from a genuine "0".
<code>atof(s)</code>	string to double. Same weakness.
<code>strtol(s, &end, base)</code>	string to long with full error reporting. <code>end</code> points at the first unconverted character.
<code>strtod(s, &end)</code>	string to double, with error reporting.

convert.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4
5  int main(int argc, char *argv[])
6  {
7      if (argc != 2) {
8          fprintf(stderr, "usage: %s <integer>\n", argv[0]);
9          return 1;
10     }
11
12     char *end;
13     errno = 0;
14     long v = strtol(argv[1], &end, 10);
15
16     if (end == argv[1])      fprintf(stderr, "not a number at all\n");
17     else if (*end != '\0')   fprintf(stderr, "trailing junk: \"%s\"\n", end);
18     else if (errno == ERANGE) fprintf(stderr, "out of range\n");
19     else                    printf("value = %ld, squared = %ld\n", v, v * v);
20     return 0;
21 }
```

```
$ ./convert 12
value = 12, squared = 144
$ ./convert 12abc
trailing junk: "abc"
$ ./convert hello
not a number at all
$ ./convert 99999999999999999999
out of range
```

15.3 A realistic argument-handling skeleton

grep-lite.c

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[])
5  {
6      int show_numbers = 0;
7      int first_file = 1;
8
9      /* parse leading option flags */
10     while (first_file < argc && argv[first_file][0] == '-') {
11         if (strcmp(argv[first_file], "-n") == 0) show_numbers = 1;
12         else if (strcmp(argv[first_file], "--help") == 0) {
13             printf("usage: %s [-n] <pattern> <file>\n", argv[0]);
14             return 0;
15         } else {
16             fprintf(stderr, "%s: unknown option %s\n", argv[0], argv[first_file]);
17             return 1;
18         }
19         first_file++;
20     }
21
22     if (argc - first_file != 2) {
23         fprintf(stderr, "usage: %s [-n] <pattern> <file>\n", argv[0]);
24         return 1;
25     }
26
27     const char *pattern = argv[first_file];
28     const char *path = argv[first_file + 1];
29
30     FILE *fp = fopen(path, "r");
31     if (!fp) { perror(path); return 1; }
32
33     char line[512];
34     int n = 0, hits = 0;
35     while (fgets(line, sizeof line, fp)) {
36         n++;
37         if (strstr(line, pattern)) {
38             hits++;
39             if (show_numbers) printf("%d:", n);
40             printf("%s", line);
41         }
42     }
43     fclose(fp);
44     return hits > 0 ? 0 : 1;      /* exit status reports whether we matched */
45 }

```

```

$ printf 'alpha\nbeta\ngamma\nalphabet\n' > words.txt
$ ./grep-lite -n alpha words.txt
1:alpha
4:alphabet
$ echo $?
0
$ ./grep-lite -n zzz words.txt
$ echo $?
1

```

15.4 Exit status

The value returned by `main` — or passed to `exit()` — is the program's **exit status**, available to the shell as `$?`.

0	success (also <code>EXIT_SUCCESS</code>)
non-zero	failure (<code>EXIT_FAILURE</code> is the portable “something went wrong”)

This convention is what makes shell composition work:

```
$ ./grep-lite alpha words.txt && echo "found it"
$ make && ./prog                # only run if the build succeeded
```

Note

`return` from `main` and `exit()` are equivalent at the top level, but `exit()` terminates the program from *anywhere*, flushing buffers and running `atexit` handlers. `_Exit()` skips the flushing. Use `exit(EXIT_FAILURE)` for unrecoverable errors deep in a call chain.

15.5 Redirection and pipes

The shell can reconnect the three standard streams before your program starts. Your program needs no code for this at all — it simply reads `stdin` and writes `stdout`.

```
$ ./prog < input.txt           # stdin comes from a file
$ ./prog > output.txt          # stdout goes to a file (truncates)
$ ./prog >> output.txt          # stdout appended
$ ./prog 2> errors.txt         # stderr only
$ ./prog > out.txt 2>&1         # both to one file
$ ./prog1 | ./prog2            # prog1's stdout becomes prog2's stdin
```

Key idea

Writing errors to `stderr` rather than `stdout` is what makes a program composable: `./prog > results.txt` then captures only the results, while error messages still reach the terminal.

15.6 Chapter summary

- `int main(int argc, char *argv[])` receives the argument count and an array of strings; `argv[0]` is the program name and `argv[argc]` is `NULL`.
- Arguments are always strings; convert with `strtol/strtod` rather than `atoi/atof` when errors matter.
- Return 0 for success and non-zero for failure; the shell reads this as `$?`.
- Diagnostics belong on `stderr`, results on `stdout`.

PART III

The GNU Toolchain

Building Programs with gcc

gcc is the GNU Compiler Collection — strictly a *driver* that runs the preprocessor, compiler, assembler and linker in turn (Section 1.8). This chapter is a working reference for its flags and its error messages.

16.1 The flags that matter

16.1.1 Output and language standard

-o <i>name</i>	name the output file. Without it you get a.out.
-std=c18	compile to the C18 standard. Also c89, c99, c11, c23, and the gnu* variants which add GNU extensions.
-pedantic	warn about anything not permitted by the chosen standard.
-pedantic-errors	make those warnings fatal.

16.1.2 Warnings — turn them all on

-Wall	the common, almost-always-a-bug warnings. Despite the name, it is <i>not</i> all of them.
-Wextra	a further set: unused parameters, signed/unsigned comparison, missing field initialisers.
-Werror	treat every warning as an error. Harsh, and correct for coursework.
-Wshadow	warn when a local variable hides an outer one.
-Wconversion	warn about implicit conversions that may change a value.
-Wfloat-equal	warn about == on floating-point values.

The command line to use for everything in this course

```
$ gcc -Wall -Wextra -std=c18 -g -O0 -o prog prog.c
```

-Wall -Wextra because the compiler finds bugs faster than you do; -g -O0 because you will want the debugger. Add -lm if you use <math.h>.

16.1.3 Debugging and optimisation

-g	include debugging information (variable names, line numbers, types) so gdb can map machine code back to your source.
-g3	as -g, plus macro definitions — lets gdb expand macros.
-O0	no optimisation (the default). Code maps one-to-one onto source. Use this when debugging.
-O1, -O2	increasing optimisation. -O2 is the usual release setting.
-O3	aggressive; may increase code size.
-Os	optimise for size.
-Og	optimise but keep debugging usable — a good middle ground.

✗ Why you must debug at -O0

At -O2 the compiler inlines functions, reorders statements, keeps variables only in registers and deletes ones it can prove are unused. In gdb you then see line numbers jumping backwards and <optimized out> where a variable should be. The program is correct; the *debugging experience* is not. Always build with -g -O0 while debugging.

16.1.4 Preprocessor, include paths and libraries

-DNAME	define a macro, as if #define NAME 1
-DNAME=value	define with a value
-UNAME	undefine a macro
-Idir	add <i>dir</i> to the header search path
-Ldir	add <i>dir</i> to the library search path
-lname	link against libname.so or libname.a. -lm for maths, -lpthread for threads.
-static	link everything statically into the executable

✗ Library order matters

```
$ gcc -lm calc.c -o calc      # FAILS: undefined reference to `sqrt'
$ gcc calc.c -lm -o calc      # works
```

The linker processes its inputs left to right, taking from each library only the symbols already known to be needed. If -lm appears before the file that needs sqrt, nothing needs it yet. **Libraries go last.**

16.1.5 Runtime checking — sanitizers

Modern gcc can instrument your program to catch memory errors as they happen, and they are often faster and more precise than Valgrind.

```
$ gcc -g -fsanitize=address -fno-omit-frame-pointer -o prog prog.c
$ ./prog
```

-fsanitize=address	buffer overflows, use-after-free, double free, leaks
-fsanitize=undefined	signed overflow, bad shifts, misaligned access, null dereference
-fsanitize=address,undefined	both together

overflow.c

```

1 #include <stdlib.h>
2 int main(void)
3 {
4     int *a = malloc(5 * sizeof *a);
5     a[5] = 42;                      /* one past the end */
6     free(a);
7     return 0;
8 }

```

```

=====
==5678==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000000034
WRITE of size 4 at 0x602000000034 thread T0
    #0 0x555555552a1 in main overflow.c:5

0x602000000034 is located 0 bytes to the right of 20-byte region
allocated by thread T0 here:
    #0 0x7ffff768e9cf in malloc
    #1 0x55555555286 in main overflow.c:4

```

The report names the operation, the line, and the allocation the memory came from. This turns a mystery into a two-minute fix.

16.2 Reading compiler messages

Exercise II question 7 asks you to introduce errors deliberately, record the messages, and fix them. That is excellent practice, and this section is the reference key. Consider:

e1.c --- four deliberate mistakes

```

1 #include <stdio.h>
2 int main(void){
3     int x = 5
4     printf("%d\n", y);
5     foo(3);
6     char *s = 5;
7     return 0;
8 }

```

```

$ gcc -Wall -Wextra -std=c18 -c e1.c
e1.c: In function 'main':
e1.c:4:5: error: expected ',' or ';' before 'printf'
    4 |     printf("%d\n", y);
      |     ^~~~~~
e1.c:5:5: warning: implicit declaration of function 'foo'
      [-Wimplicit-function-declaration]
e1.c:6:15: warning: initialization of 'char *' from 'int' makes pointer from
      integer without a cast [-Wint-conversion]
e1.c:6:11: warning: unused variable 's' [-Wunused-variable]
e1.c:3:9: warning: unused variable 'x' [-Wunused-variable]

```

Three lessons about compiler messages are visible here:

1. **The reported line is where the compiler *noticed* the problem, not always where you made it.** The missing semicolon is on line 3; the error is reported on line 4, because the compiler only knows something is wrong when it meets a token that cannot continue the statement. **Always look at the line above.**
2. **Fix the first error first.** One syntax error commonly produces a cascade of nonsense after-

wards. Recompile after each fix.

3. **The bracketed name is the warning's switch** (`-Wunused-variable`), so you can look it up or selectively disable it.

16.2.1 A catalogue of common messages

Message	Cause and fix
expected ';' before 'X'	Missing semicolon — on the <i>previous</i> line.
'x' undeclared (first use in this function)	Typo, or the declaration is missing or out of scope.
implicit declaration of function 'f'	No prototype in scope. Missing <code>#include</code> , or the function is defined below its first use with no declaration.
undefined reference to 'f'	Linker. The prototype existed but no definition. Missing <code>.c</code> file on the command line, or a missing <code>-l</code> flag.
multiple definition of 'x'	Linker. A variable defined in a header, or two files defining the same global. Use <code>extern</code> in the header.
control reaches end of non-void function	A path through the function has no return.
format '%d' expects argument of type 'int', but argument 2 has type 'double'	Mismatched <code>printf</code> specifier.
suggest parentheses around assignment used as truth value	You wrote <code>if (x = 5)</code> and probably meant <code>==</code> .
assignment discards 'const' qualifier	Assigning a <code>const char *</code> to a <code>char *</code> .
comparison of integer expressions of different signedness	Mixing <code>int</code> and <code>unsigned/size_t</code> in a comparison.
operation on 'x' may be undefined [-Wsequence-point]	<code>x</code> modified twice, or modified and read, in one expression.
dereferencing pointer to incomplete type	The struct is declared but not defined — a missing header.
function returns address of local variable	Returning <code>&local</code> . See Chapter 12.
warning: 'x' may be used uninitialised	A path exists where <code>x</code> is read before assignment.

✓ Exam focus

Be able to state which *stage* produced a message. undeclared, expected ';', implicit declaration $\Rightarrow$ compiler. undefined reference, multiple definition $\Rightarrow$ linker. Segmentation fault $\Rightarrow$ neither — that is run time.

16.3 Inspecting what was built

<code>file prog</code>	what kind of file it is: ELF executable, object file, script
<code>size prog</code>	bytes in the text, data and bss segments (Section 12.3)
<code>nm prog.o</code>	the symbol table. T/t = text, D/d = data, B/b = bss, U = undefined. Uppercase means global, lowercase means local.
<code>objdump -d prog</code>	disassemble the machine code
<code>ldd prog</code>	which shared libraries it needs at run time
<code>strings prog</code>	printable strings inside the binary
<code>readelf -h prog</code>	the ELF header

```
$ gcc -o hello hello.c
$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, dynamically linked, not stripped
$ ldd hello
    linux-vdso.so.1
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
    /lib64/ld-linux-x86-64.so.2
$ nm hello | grep -w main
0000000000001149 T main
```

16.4 Static and dynamic linking

	Static (.a, -static)	Dynamic (.so, the default)
When resolved	at link time; library code copied into the executable	at load time, by the dynamic loader
Executable size	large	small
Memory	each process has its own copy	one copy shared by all processes
Updates	must relink to pick up a library fix	a library update benefits every program immediately
Startup	faster	marginally slower
Portability	runs anywhere	needs the right libraries present

```
$ gcc -o hello hello.c && ls -l hello # ~16 KB, dynamic
$ gcc -static -o hello_s hello.c && ls -l hello_s # ~900 KB, static
```

16.5 Building a multi-file project by hand

```
# one step
$ gcc -Wall -Wextra -std=c18 -g -O0 -o prog main.c stack.c queue.c

# separate compilation -- only the changed file is recompiled
$ gcc -Wall -Wextra -std=c18 -g -O0 -c main.c # -> main.o
$ gcc -Wall -Wextra -std=c18 -g -O0 -c stack.c # -> stack.o
$ gcc -Wall -Wextra -std=c18 -g -O0 -c queue.c # -> queue.o
$ gcc -o prog main.o stack.o queue.o # link
```

Separate compilation is why C projects build quickly: change one .c file and only that file is recompiled before relinking. Automating exactly this decision is what make does (Section 18.4).

16.6 Chapter summary

- Use `gcc -Wall -Wextra -std=c18 -g -O0 -o prog prog.c` throughout this course; add `-lm` after the sources if you use maths.
- `-E`, `-S`, `-c` stop after preprocessing, compiling and assembling respectively.
- Debug at `-O0`; optimise only for release.
- Warnings are the cheapest bug detector available. `-Werror` makes them impossible to ignore.
- The reported error line is where the compiler noticed the problem — check the line above, and always fix the first error first.
- undefined reference and multiple definition come from the linker, not the compiler.

- `-fsanitize=address,undefined` catches memory and UB errors at run time with precise reports.

Debugging with gdb

The lecture slides never cover gdb. That is a serious gap, because in C the difference between a productive afternoon and a wasted one is usually whether you can see what the program is actually doing. This chapter is a complete working tutorial. Every transcript in it is real output from gdb 12.1 on gcc 11.4.

17.1 Why a debugger

The alternative — inserting `printf` calls, recompiling, reading output, deleting them again — works, but it is slow, it only shows what you thought to ask for in advance, and it lies to you when the program crashes before the buffer is flushed (Chapter 3).

A debugger lets you stop a running program at any point and inspect *everything*: variables, memory, the call stack, registers. For pointer bugs, which is most C bugs, it is not a convenience but a necessity: you can look at what a pointer actually holds instead of guessing.

Compile for debugging

```
$ gcc -Wall -Wextra -std=c18 -g -O0 -o prog prog.c
```

-g embeds the mapping from machine code back to your source: line numbers, variable names and types. -O0 disables optimisation so that mapping stays faithful. Without -g, gdb can only show you assembly.

17.2 Starting and stopping

<code>gdb ./prog</code>	start the debugger on prog
<code>gdb -q ./prog</code>	quiet — skip the licence banner
<code>gdb --args ./prog a b</code>	start with command-line arguments
<code>run (r)</code>	start the program
<code>run < in.txt</code>	run with input redirected
<code>set args a b</code>	set arguments before running
<code>kill</code>	stop the running program
<code>quit (q)</code>	leave gdb

Throughout this chapter the demonstration program is:

factorial.c

```
1 #include <stdio.h>
2
3 int factorial(int n)
4 {
```

```

5     if (n <= 1) return 1;
6     return n * factorial(n - 1);
7 }
8
9 int sum_to(int n)
10 {
11     int total = 0;
12     for (int i = 1; i <= n; i++)
13         total += i;
14     return total;
15 }
16
17 int main(void)
18 {
19     int a = factorial(5);
20     int b = sum_to(10);
21     printf("factorial(5) = %d\n", a);
22     printf("sum_to(10)   = %d\n", b);
23     return 0;
24 }

```

17.3 Breakpoints

A **breakpoint** tells gdb to stop the program at a chosen place.

<code>break sum_to (b)</code>	stop on entry to <code>sum_to</code>
<code>break factorial.c:13</code>	stop at a line
<code>break 13</code>	stop at line 13 of the current file
<code>break factorial if n == 1</code>	conditional — stop only when the condition holds
<code>tbreak main</code>	temporary: deleted after it is hit once
<code>info breakpoints (i b)</code>	list them, with hit counts
<code>delete 2</code>	delete breakpoint 2; delete alone deletes all
<code>disable 2 / enable 2</code>	switch one off and on without deleting
<code>ignore 2 100</code>	skip the next 100 hits of breakpoint 2

```
gdb -q ./factorial
```

```

(gdb) break sum_to
Breakpoint 1 at 0x1183: file factorial.c, line 11.
(gdb) run
Breakpoint 1, sum_to (n=10) at factorial.c:11
11^I    int total = 0;
(gdb) info args
n = 10

```

Notice that gdb shows the function, its arguments, the file and line, and the source line about to execute — all before you ask anything.

Conditional breakpoints save enormous time

Debugging the 4,000th iteration of a loop by pressing continue 4,000 times is not a plan. `break process if i == 4000` stops exactly once. The condition is any C expression valid at that point.

17.4 Stepping

next (n)	execute one line, stepping over function calls
step (s)	execute one line, stepping into function calls
finish	run until the current function returns, and print its return value
until <i>N</i>	run until line <i>N</i> — useful to escape a loop
continue (c)	resume until the next breakpoint
advance sum_to	run until sum_to is reached
return	abandon the current function immediately
Enter	repeat the previous command — press it to keep stepping

Key idea

next and step differ only at a function call. next runs the whole call and stops on the following line; step descends into the function. Use next by default, step when you suspect the callee.

stepping through sum_to

```
Breakpoint 1, sum_to (n=10) at factorial.c:11
11^^I   int total = 0;
(gdb) next
12^^I   for (int i = 1; i <= n; i++)
(gdb) next
13^^I       total += i;
(gdb) print total
$1 = 0
(gdb) print i
$2 = 1
```

17.5 Examining data

print <i>expr</i> (p)	evaluate and print any C expression
print/x <i>expr</i>	print in hex; also /d /o /t (binary) /c /f /u
print *ptr	dereference
print arr[3]@5	print 5 elements starting at arr[3]
print \$rsp, \$rip	registers
ptype <i>expr</i>	show the <i>type</i> , expanding structs
info locals	every local variable in the current frame
info args	the current function's arguments
info registers	all CPU registers
display <i>expr</i>	print <i>expr</i> automatically after every step
undisplay 1	stop doing that
set var x = 5	change a variable and carry on

print results are numbered (\$1, \$2, ...) and can be reused: print \$1 + 10.

17.5.1 Examining raw memory: x

The x command

`x/NFU address`

where N is a repeat count, F a format letter and U a unit size.

Format F	x hex, d decimal, u unsigned, c char, s string, i instruction, t binary, f float
Unit U	b 1 byte, h 2, w 4, g 8

```
x/16xb &arr      /* 16 bytes of arr in hex      */
x/5dw arr        /* 5 words as signed decimal      */
x/s name         /* as a null-terminated string    */
x/4xg head       /* 4 giant (8-byte) words in hex  */
x/10i $rip       /* the next 10 machine instructions */
```

17.6 The call stack

backtrace (bt)	the chain of calls that led here
bt full	the same, plus every frame's local variables
frame n (f)	select frame n and look at its variables
up / down	move one frame towards main / away from it
info frame	addresses, saved registers, return address of this frame

Key idea

backtrace is the single most useful command in gdb. When a program crashes, it answers “where was I, and how did I get here?” in one line.

A recursive call chain makes the stack visible in a way no diagram can:

```
gdb -q ./factorial
```

```
(gdb) break factorial if n == 1
Breakpoint 1 at 0x1158: file factorial.c, line 5.
(gdb) run
Breakpoint 1, factorial (n=1) at factorial.c:5
5^^I    if (n <= 1) return 1;
(gdb) backtrace
#0 factorial (n=1) at factorial.c:5
#1 0x000055555555172 in factorial (n=2) at factorial.c:6
#2 0x000055555555172 in factorial (n=3) at factorial.c:6
#3 0x000055555555172 in factorial (n=4) at factorial.c:6
#4 0x000055555555172 in factorial (n=5) at factorial.c:6
#5 0x0000555555551c0 in main () at factorial.c:19
(gdb) info frame
Stack level 0, frame at 0x7fffffffe510:
 rip = 0x55555555158 in factorial (factorial.c:5); saved rip = 0x55555555172
 called by frame at 0x7fffffffe530
 Arglist at 0x7fffffffe500, args: n=1
 Locals at 0x7fffffffe500, Previous frame's sp is 0x7fffffffe510
 Saved registers:
  rbp at 0x7fffffffe500, rip at 0x7fffffffe508
(gdb) finish
```

```
Run till exit from #0  factorial (n=1) at factorial.c:5
factorial (n=2) at factorial.c:6
6^I    return n * factorial(n - 1);
Value returned is $1 = 1
```

Five stack frames, one per recursive call, each with its own *n* — this is Figure 12.2 made concrete. `info frame` even shows the saved return address and saved frame pointer described in Chapter 12.

17.7 Watchpoints

A **watchpoint** stops the program whenever a variable’s *value* changes, wherever that happens. It is the tool for “something is corrupting this variable and I have no idea what”.

<code>watch total</code>	break when <code>total</code> is written
<code>rwatch total</code>	break when it is <i>read</i>
<code>awatch total</code>	break on read or write
<code>watch *(int *)0x7fff1234</code>	watch a raw address — survives scope exit
<code>info watchpoints</code>	list them

watching a running total

```
(gdb) break sum_to
(gdb) run
Breakpoint 1, sum_to (n=10) at factorial.c:11
(gdb) watch total
Hardware watchpoint 2: total
(gdb) continue
Hardware watchpoint 2: total
Old value = 529267711
New value = 0
sum_to (n=10) at factorial.c:12
(gdb) continue
Hardware watchpoint 2: total
Old value = 0
New value = 1
(gdb) continue
Hardware watchpoint 2: total
Old value = 1
New value = 3
```

Note the first report: the “old value” of `total` was 529267711 — stack garbage — until line 11 initialised it. That is the uninitialised-memory problem of Chapter 8, caught in the act.

Note

Modern CPUs implement watchpoints in hardware, so they cost almost nothing. A watchpoint on a local variable is deleted automatically when its scope exits.

17.8 Worked session 1: a segmentation fault

crash.c --- two deliberate bugs

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct Student { int id; char name[20]; };
6
7  void print_student(struct Student *s)
8  {
9      printf("id=%d name=%s\n", s->id, s->name);
10 }
11
12 int main(void)
13 {
14     struct Student *list = malloc(3 * sizeof *list);
15     for (int i = 0; i < 3; i++) {
16         list[i].id = 100 + i;
17         strcpy(list[i].name, "student");
18     }
19
20     for (int i = 0; i <= 3; i++)      /* BUG 1: off by one */
21         print_student(&list[i]);
22
23     struct Student *bad = NULL;
24     print_student(bad);              /* BUG 2: null dereference */
25
26     free(list);
27     return 0;
28 }

```

Run it without the debugger and you learn almost nothing:

```

$ ./crash
id=100 name=student
id=101 name=student
id=102 name=student
id=0 name=
Segmentation fault (core dumped)
$ echo $?
139

```

Where did it crash? Which of the two bugs fired? Now with gdb:

```

gdb -q ./crash

(gdb) run
id=100 name=student
id=101 name=student
id=102 name=student
id=0 name=

Program received signal SIGSEGV, Segmentation fault.
0x00005555555551a5 in print_student (s=0x0) at crash.c:9
9^I printf("id=%d name=%s\n", s->id, s->name);
(gdb) backtrace
#0  0x00005555555551a5 in print_student (s=0x0) at crash.c:9
#1  0x000055555555528c in main () at crash.c:24
(gdb) print s

```

```

$1 = (struct Student *) 0x0
(gdb) frame 1
#1 0x000055555555528c in main () at crash.c:24
24^~I    print_student(bad);                /* BUG 2: null dereference */
(gdb) print bad
$2 = (struct Student *) 0x0
(gdb) print list
$3 = (struct Student *) 0x5555555592a0

```

The whole diagnosis is in the first three lines: the crash is at `crash.c:9` in `print_student`, and `gdb` has already printed the culprit in the frame header — `s=0x0`. The backtrace shows it was called from line 24.

The three-command crash recipe

1. run — let it crash. `gdb` stops exactly at the faulting instruction.
2. backtrace — see the chain of calls.
3. print the pointers in the failing frame. In practice the answer is almost always `0x0` (a null pointer), a small number like `0x4` (a null pointer plus a field offset), or a wild value.

Notice the first bug did *not* crash. `list[3]` is one element past the end of the allocation, so `print_student` read heap memory it did not own and printed `id=0 name=` — garbage that happened to look plausible. This is why out-of-bounds access is so dangerous: it usually does not crash, it quietly returns nonsense. Valgrind (Chapter 18) catches this one.

17.9 Worked session 2: inspecting a linked list

This is the payoff for Chapter 23: you can walk the chain of pointers by hand and see the structure.

```

list.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node { int data; struct Node *next; };
5
6  struct Node *push_front(struct Node *head, int value)
7  {
8      struct Node *n = malloc(sizeof *n);
9      n->data = value;
10     n->next = head;
11     return n;
12 }
13
14 int length(struct Node *head)
15 {
16     int c = 0;
17     for (struct Node *p = head; p != NULL; p = p->next) c++;
18     return c;
19 }
20
21 int main(void)
22 {
23     struct Node *head = NULL;
24     for (int i = 1; i <= 4; i++) head = push_front(head, i * 10);
25     printf("length = %d\n", length(head));
26     for (struct Node *p = head; p; p = p->next) printf("%d ", p->data);
27     printf("\n");

```

```

28     return 0;
29 }

```

```
gdb -q ./list
```

```

(gdb) break length
Breakpoint 1 at 0x11d1: file list.c, line 16.
(gdb) run
Breakpoint 1, length (head=0x55555559300) at list.c:16
16^I    int c = 0;
(gdb) print *head
$1 = {data = 40, next = 0x555555592e0}
(gdb) print head->data
$2 = 40
(gdb) print head->next->data
$3 = 30
(gdb) print head->next->next->data
$4 = 20
(gdb) print *head->next
$5 = {data = 30, next = 0x555555592c0}
(gdb) ptype struct Node
type = struct Node {
    int data;
    struct Node *next;
}
(gdb) x/4xg head
0x55555559300: ^I0x0000000000000028^I0x0000555555592e0
0x55555559310: ^I0x0000000000000000^I0x000000000020cf1

```

Several things are worth pausing on:

- `print *head` prints the whole struct with member names. You do not have to print fields one at a time.
- `head->next->next->data` follows the chain as far as you like. If one of those pointers were corrupt, gdb would say `Cannot access memory at address 0x...` and you would know exactly which link broke.
- The `x/4xg` dump shows the raw bytes: `0x28` is 40 in hex — the data field — followed by the next pointer. You are looking directly at Figure 23.1.

Debugging a broken linked list

When a list operation misbehaves, set a breakpoint after the operation and print the chain: `p *head`, `p *head->next`, and so on. The bug is almost always at the point where a next pointer is `0x0` too early, points back at an earlier node (a cycle), or holds a freed address.

17.10 TUI mode

`gdb` has a built-in split-screen text interface showing your source alongside the command prompt.

<code>gdb -tui ./prog</code>	start in TUI mode
<code>Ctrl-x a</code>	toggle TUI on and off from inside gdb
<code>layout src</code>	source window
<code>layout split</code>	source and assembly
<code>layout regs</code>	add a register window
<code>Ctrl-x o</code>	change which window has focus
<code>Ctrl-l</code>	redraw if the display becomes corrupted

```

+--factorial.c-----+
|  9  int sum_to(int n)      |
| 10  {                      |
| 11      int total = 0;      |
B+>| 12      for (int i = 1; i <= n; i++) |
| 13          total += i;      |
| 14      return total;      |
| 15  }                      |
+-----+
native process 4242 In: sum_to      L12  PC: 0x55555555192
(gdb)

```

The `B+>` marker shows a breakpoint and the current line together. For step-by-step work TUI mode is much easier than repeatedly typing `list`.

17.11 Other useful commands

<code>list (l)</code>	show source around the current line; <code>list 20,40</code> for a range
<code>info source</code>	which file is being debugged
<code>info functions <i>regex</i></code>	list matching functions
<code>info line 13</code>	the address range of a source line
<code>disassemble</code>	the assembly for the current function
<code>set listsize 20</code>	how many lines <code>list</code> shows
<code>shell <i>cmd</i></code>	run a shell command without leaving gdb
<code>help <i>cmd</i></code>	built-in documentation
<code>apropos <i>word</i></code>	search the help

17.11.1 Batch mode

`gdb` can be scripted, which is how every transcript in this chapter was produced:

```

$ gdb -q -batch -ex "break sum_to" -ex "run" -ex "info args" \
    -ex "bt" -ex "continue" ./factorial

```

This is useful in a Makefile target or a test script — for instance, to get a backtrace automatically whenever a program crashes.

17.11.2 Core dumps

If the shell allows it, a crashing program writes a **core dump**: a snapshot of its memory at the moment of death. You can debug it after the fact.

```
$ ulimit -c unlimited      # enable core dumps for this shell
$ ./crash
Segmentation fault (core dumped)
$ gdb ./crash core
(gdb) backtrace             # exactly where it died, no need to reproduce
```

17.11.3 .gdbinit

Commands placed in ~/.gdbinit or ./gdbinit run at start-up:

```
set history save on
set print pretty on      /* structs printed one member per line */
set confirm off
set disassembly-flavor intel
```

set print pretty on in particular is worth having permanently.

17.12 Command reference card

Command	Short	Action
<i>Running</i>		
run	r	start the program
continue	c	resume to the next breakpoint
kill		stop the program
quit	q	exit gdb
<i>Breakpoints</i>		
break <i>loc</i>	b	breakpoint at function or line
break <i>loc</i> if <i>cond</i>		conditional breakpoint
tbreak <i>loc</i>		one-shot breakpoint
watch <i>var</i>		break when the value changes
info breakpoints	i b	list
delete / disable	d	remove / switch off
<i>Stepping</i>		
next	n	one line, over calls
step	s	one line, into calls
finish		run until this function returns
until <i>N</i>	u	run until line <i>N</i>
<i>Inspecting</i>		
print <i>expr</i>	p	evaluate an expression
print/x <i>expr</i>		in hexadecimal
display <i>expr</i>		show after every step
x/NFU <i>addr</i>		examine raw memory
ptype <i>expr</i>		show the type
info locals		all locals in this frame
info args		this function's arguments
set var <i>x</i> = <i>v</i>		modify a variable
<i>Stack</i>		
backtrace	bt	the call chain
bt full		call chain with locals
frame <i>n</i>	f	select a frame
up / down		move through frames

17.13 A debugging method

Tools are not a method. When something is wrong:

1. **Reproduce it reliably.** A bug you cannot trigger on demand cannot be verified as fixed. Find the smallest input that fails.
2. **Read the message.** Compiler warnings, sanitizer reports and Valgrind output frequently name the exact line. Do not skip past them to get to the “real” debugging.
3. **Form one hypothesis.** “The loop runs one iteration too many.” Not “something is wrong with the array.”
4. **Test it with one observation.** A breakpoint and a print, or a watchpoint. Change one thing

at a time.

5. **Narrow by bisection.** If the state is good at line 40 and bad at line 80, look at line 60. Four or five halvings localise a bug in a thousand lines.
6. **Fix the cause, not the symptom.** Adding `if (p != NULL)` around a crash hides it; ask why `p` was null.
7. **Verify, then look for siblings.** The same mistake is usually repeated elsewhere in the file.

✓ Exam focus

For coursework, the most valuable habits are: compile with `-Wall -Wextra -g -O0`, run under `valgrind` before submitting, and when something segfaults, run it under `gdb` and type `bt`. Those three steps catch the overwhelming majority of C bugs at this level.

17.14 Chapter summary

- Compile with `-g -O0`; optimised builds mislead the debugger.
- `break` sets breakpoints — by function, by line, or conditionally with `if`.
- `next` steps over calls, `step` steps into them, `finish` runs to the end of the current function and reports the return value.
- `print` evaluates any C expression, including `p->next->next->data`; `p`type shows types; `x` examines raw memory.
- `backtrace` answers “how did I get here?” — the first command to type after any crash.
- `watch` stops the program whenever a variable changes, which is how you find silent corruption.
- The crash recipe: `run`, `bt`, `print` the pointers. `0x0` means null; a wild value means an uninitialised or freed pointer.

Valgrind, Sanitizers and make

18.1 Why memory tools exist

C will not tell you when you write past the end of an array, read freed memory, or forget to call free. Chapter 13 listed those faults; this chapter is about finding them automatically.

Memory leaks *can* be detected manually — during a code review where each allocation and deallocation is checked — but that involves significant effort and time and does not scale. Automated tools make detection far easier in large code bases.

18.2 Valgrind

Valgrind provides tools to profile and debug programs. Its default tool, memcheck, runs your program on a simulated CPU and checks every memory access.

```
$ sudo apt install valgrind
$ gcc -g -O0 -o prog prog.c
$ valgrind --leak-check=full ./prog
```

Compile with `-g` first, or the reports will name addresses instead of line numbers.

Option	Effect
<code>--leak-check=full</code>	report each leak individually with a stack trace
<code>--show-leak-kinds=all</code>	include “still reachable” and “indirect” leaks
<code>--track-origins=yes</code>	say where an uninitialised value came from (slower)
<code>-s</code>	show a summary of suppressed errors
<code>--log-file=out.txt</code>	write the report to a file

18.2.1 Reading a leak report

```
==12345== HEAP SUMMARY:
==12345==    in use at exit: 4,000 bytes in 1 blocks
==12345== total heap usage: 2 allocs, 1 frees, 4,020 bytes allocated
==12345==
==12345== 4,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4848899: malloc (vg_replace_malloc.c:381)
==12345==    by 0x109151: leaky (leaky.c:3)
==12345==    by 0x10916E: main (leaky.c:4)
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 4,000 bytes in 1 blocks
==12345==    indirectly lost: 0 bytes in 0 blocks
==12345==    possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 0 bytes in 0 blocks
```

The stack trace reads bottom-up: main called leaky at line 4, which called malloc at line 3, and that block was never freed.

Leak kind	Meaning
definitely lost	no pointer to the block exists anywhere. A real bug; fix these first.
indirectly lost	reachable only through a definitely-lost block — for example the rest of a linked list whose head leaked. Fixing the head fixes these.
possibly lost	a pointer exists but points into the middle of the block, not its start. Often a sign of pointer arithmetic on an allocation.
still reachable	a pointer still exists at exit — typically a global cache. Not strictly a bug, but untidy.

18.2.2 Errors Valgrind finds besides leaks

Message	Meaning
Invalid write of size 4	writing outside an allocated block — a buffer overflow
Invalid read of size 1	reading outside a block, or reading freed memory
Conditional jump depends on uninitialised value(s)	an if on a variable that was never assigned
Invalid free() / delete	freeing a non-heap pointer, or freeing twice
Mismatched free() / delete	C/C++ allocator mismatch
Source and destination overlap in memcpy	use memmove instead

Key idea

The most useful line in an “Invalid write” report is usually the second block: Address 0x... is 0 bytes after a block of size 20 alloc'd. That phrase — *after a block* — tells you immediately that it is an off-by-one at the end of an array.

18.3 AddressSanitizer

gcc has memory checking built in. AddressSanitizer instruments the code at compile time rather than simulating the CPU, so it runs roughly 2× slower instead of Valgrind’s 10–50×, and its reports are at least as good.

```
$ gcc -g -O0 -fsanitize=address -o prog prog.c
$ ./prog
```

The following reports are real output.

18.3.1 A leak

leak2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void leaky(void) { int *p = malloc(1000 * sizeof(int)); p[0] = 1; }
5
6 int main(void) { leaky(); printf("done\n"); return 0; }
```

```
done
=====
==12==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4000 byte(s) in 1 object(s) allocated from:
    #0 0x71eac82b4887 in __interceptor_malloc
    #1 0x5cac8550321e in leaky leak2.c:3
    #2 0x5cac85503273 in main leak2.c:4

SUMMARY: AddressSanitizer: 4000 byte(s) leaked in 1 allocation(s).
```

18.3.2 Use after free

```
uaf.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int *p = malloc(4 * sizeof(int));
7      p[0] = 1;
8      free(p);
9      printf("%d\n", p[0]);      /* reading freed memory */
10     return 0;
11 }
```

```
==21==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010
READ of size 4 at 0x502000000010 thread T0
    #0 0x5b3d34db52e6 in main uaf.c:8

0x502000000010 is located 0 bytes inside of 16-byte region
freed by thread T0 here:
    #0 0x7595bcab4537 in __interceptor_free
    #1 0x5b3d34db52af in main uaf.c:7

previously allocated by thread T0 here:
    #0 0x7595bcab4887 in __interceptor_malloc
    #1 0x5b3d34db525e in main uaf.c:5

SUMMARY: AddressSanitizer: heap-use-after-free uaf.c:8 in main
```

Three lines, three answers: where it was read (line 8), where it was freed (line 7), where it was allocated (line 5). Without a tool this is a bug that can take an afternoon.

	Valgrind	AddressSanitizer
Setup	none — run any binary	must recompile with <code>-fsanitize=address</code>
Slowdown	10–50×	about 2×
Memory	moderate	about 3×
Finds uninitialised reads	✓ (memcheck)	✗ (use <code>-fsanitize=memory</code> in clang)
Finds stack overflows	✗	✓
Course requirement	✓ named in the lectures	useful addition

✓ Exam focus

The syllabus names Valgrind specifically. Know the two commands: `sudo apt install valgrind` and `valgrind --leak-check=full ./executable`, and be able to state that it detects memory leaks, invalid reads and writes, and use of uninitialised values.

18.4 make

Typing the `gcc` command by hand becomes untenable as soon as a project has more than two files. `make` reads a Makefile describing what depends on what, and rebuilds only what is out of date.

18.4.1 Rules

A make rule

```
target: prerequisites
<TAB>recipe
```

“To build *target*, you need *prerequisites*; if any of them is newer than the target, run *recipe*.”

✗ The recipe must begin with a real tab

Not spaces. A space-indented recipe produces `Makefile:4: *** missing separator. Stop.` — the single most common make error, and invisible in most editors. In vim, `:set list` displays tabs as `^I`.

18.4.2 A complete project Makefile

Makefile

```
1 CC      = gcc
2 CFLAGS  = -Wall -Wextra -std=c18 -g -O0
3 LDFLAGS =
4 TARGET  = prog
5 OBJS    = main.o stack.o
6
7 all: $(TARGET)
8
9 $(TARGET): $(OBJS)
10 ^^I$(CC) $(OBJS) -o $@ $(LDFLAGS)
11
12 %.o: %.c
13 ^^I$(CC) $(CFLAGS) -c $< -o $@
14
15 main.o: stack.h
16 stack.o: stack.h
17
18 run: $(TARGET)
19 ^^I./$(TARGET)
20
21 clean:
22 ^^Irm -f $(OBJS) $(TARGET)
23
24 .PHONY: all run clean
```

```

$ make
gcc -Wall -Wextra -std=c18 -g -O0 -c main.c -o main.o
gcc -Wall -Wextra -std=c18 -g -O0 -c stack.c -o stack.o
gcc main.o stack.o -o prog

$ ./prog
25 16 9 4 1

$ touch stack.c && make
gcc -Wall -Wextra -std=c18 -g -O0 -c stack.c -o stack.o
gcc main.o stack.o -o prog

$ make
make: Nothing to be done for 'all'.

```

That is the whole point: after touching `stack.c`, only `stack.c` was recompiled; `main.o` was left alone. With nothing changed, `make` does nothing at all.

18.4.3 The pieces explained

<code>CC = gcc</code>	a variable; used as <code>\$(CC)</code>
<code>\$@</code>	the target of the current rule
<code>\$<</code>	the <i>first</i> prerequisite
<code>\$^</code>	<i>all</i> prerequisites
<code>%.o: %.c</code>	a pattern rule : how to make any <code>.o</code> from the matching <code>.c</code>
<code>main.o: stack.h</code>	an extra prerequisite — if the header changes, <code>main.o</code> is rebuilt. This is why editing a header correctly triggers recompilation of everything that includes it.
<code>.PHONY</code>	declares targets that are not files, so <code>make clean</code> works even if a file named <code>clean</code> exists

18.4.4 Useful invocations

<code>make</code>	build the first target (all here)
<code>make clean</code>	run the <code>clean</code> recipe
<code>make run</code>	build if needed, then run
<code>make -n</code>	print the commands without executing them
<code>make -j4</code>	build up to four targets in parallel
<code>make -B</code>	rebuild everything unconditionally

Key idea

Add targets for the tools as well, so that checking your work is one word:

```

memcheck: $(TARGET)
^^Ivalgrind --leak-check=full ./${TARGET}

asan: clean
^^I$(MAKE) CFLAGS="$(CFLAGS) -fsanitize=address,undefined" $(TARGET)

```

Then `make memcheck` before every submission.

18.5 Chapter summary

- Compile with `-g` before using any memory tool.

- `valgrind --leak-check=full ./prog` reports leaks with the stack trace of the allocation, and also finds invalid reads and writes and uninitialised values.
- “Definitely lost” is the category to fix first; “indirectly lost” usually disappears with it.
- `gcc -fsanitize=address,undefined` is faster and reports where a block was allocated, freed and misused.
- `make` rebuilds only what is out of date. Recipes must be indented with a tab. `$@` is the target, `$<` the first prerequisite, `%.o: %.c` a pattern rule, and header dependencies must be listed explicitly.

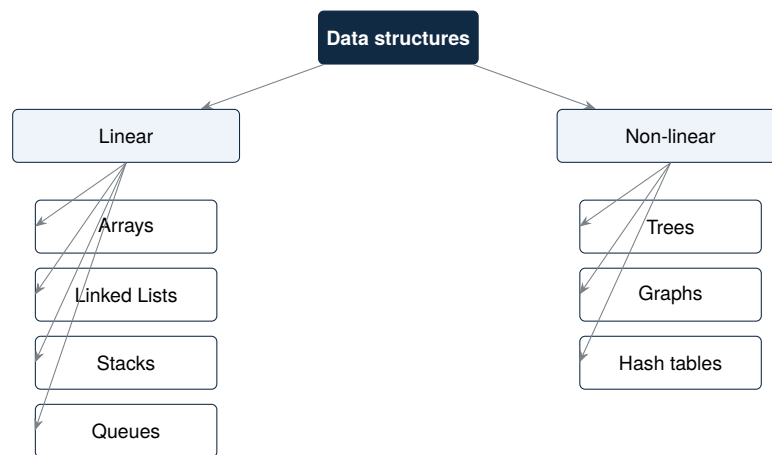
PART IV

Data Structures

Abstract Data Types and the Classification of Data Structures

19.1 Classification

Data structures fall into two main categories.



Linear data structures

have their elements arranged sequentially, so that a given element is attached to the previous and the next adjacent elements. Arrays, linked lists, stacks and queues.

Non-linear data structures

have elements arranged in a non-sequential manner — an element may have many successors. Trees and graphs.

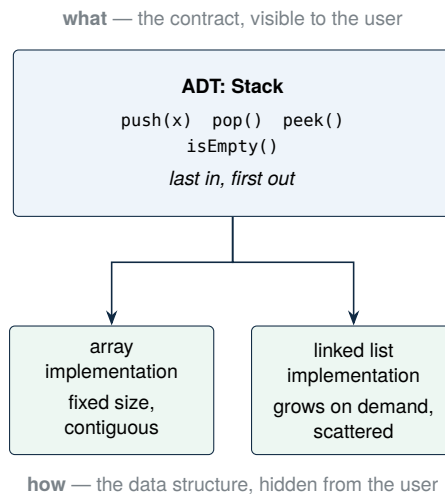
This course concentrates on the linear structures, which is why it is subtitled *linear data structures*; Chapter 24 introduces the non-linear ones for completeness.

19.2 Abstract data types

Key idea

An **abstract data type** (ADT) is a class of objects whose behaviour is defined by a set of values and a set of operations, *as seen by a user*. It represents a high-level concept, independent of the underlying implementation.

A **data structure**, by contrast, provides *how* the behaviour specified by an ADT is implemented.



The separation matters because it lets the implementation change without the user's code changing. A program that only calls push and pop works identically whether the stack is an array or a linked list — but its performance and memory behaviour may differ completely.

✓ **Exam focus**

The distinction is standard examination material. *ADT = specification = what*. *Data structure = implementation = how*. A stack is an ADT; an array-based stack is a data structure.

19.3 The three fundamental operations

With any data structure, three operations must be learned:

1. **How to insert** data into the structure.
2. **How to delete** data from the structure.
3. **How to search** for data.

In addition there will be structure-specific operations — peek for a stack, traverse for a tree, and so on. When you meet a new data structure, these three questions are the ones to ask first, along with a fourth: what does each cost?

19.4 Choosing a structure

Structure	Access	Search	Insert/Delete	Characteristic
Array	$O(1)$	$O(n)$	$O(n)$	contiguous, fixed size, cache friendly
Sorted array	$O(1)$	$O(\log n)$	$O(n)$	binary search possible
Linked list	$O(n)$	$O(n)$	$O(1)^{\dagger}$	grows on demand, no shifting
Stack	$O(1)$ top only	—	$O(1)$	LIFO
Queue	$O(1)$ ends only	—	$O(1)$	FIFO
Binary search tree	—	$O(\log n)^{\ddagger}$	$O(\log n)^{\ddagger}$	ordered traversal
Hash table	—	$O(1)$ average	$O(1)$ average	unordered

[†] once you hold a pointer to the position; finding it is $O(n)$. [‡] if balanced; $O(n)$ in the worst case.

 Key idea

There is no best data structure. There is only the structure whose costs match the operations your program performs most. Chapter [26](#) gives the vocabulary for saying that precisely.

Recursion

20.1 The idea

Key idea

Recursion is defining a problem in terms of itself — in programming terms, a function calling another instance of itself.

Every recursion has two properties:

Base case (anchor point)

the smallest input, solved directly without further recursion. Without it the recursion never stops.

Recursive step

reduces the problem to a smaller instance of the same problem and calls itself.

Recursion is implemented using the **stack memory**: each call gets its own stack frame with its own parameters and locals, which is exactly why the calls do not interfere with one another (Chapter 12).

20.2 Factorial

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \quad (\text{base case}) \\ n \times (n - 1)! & \text{otherwise} \quad (\text{recursive step}) \end{cases}$$

factorial.c

```

1  #include <stdio.h>
2
3  long factorial(int n)
4  {
5      if (n <= 1) return 1;           /* base case */
6      return n * factorial(n - 1);    /* recursive step */
7  }
8
9  int main(void)
10 {
11     for (int i = 0; i <= 10; i++)
12         printf("%2d! = %ld\n", i, factorial(i));
13     return 0;
14 }
```

```

0! = 1
1! = 1
2! = 2
3! = 6
```

```

4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

20.2.1 What the stack does

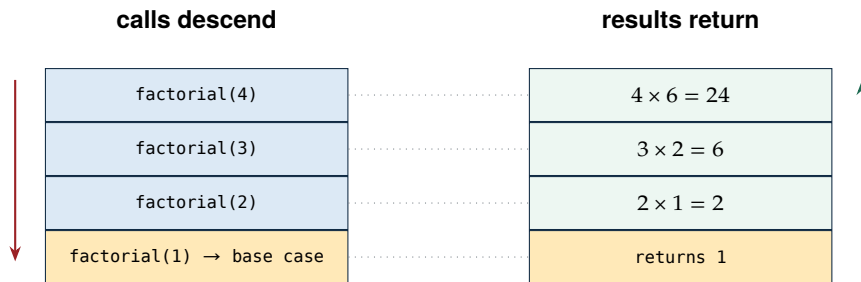


Figure 20.1. factorial(4): four frames are pushed until the base case, then the results multiply back up as the frames pop.

The gdb session in Section 17.6 shows this stack for real, one frame per call, each with its own n .

20.3 Fibonacci — and why naive recursion can be terrible

$$F(n) = \begin{cases} n & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

fib.c

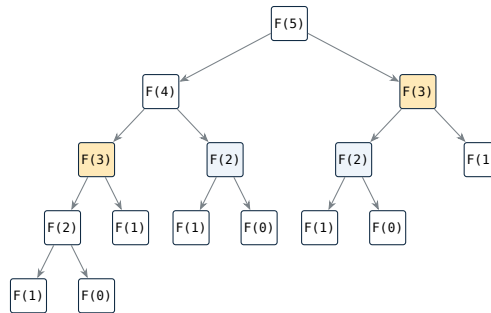
```

1  #include <stdio.h>
2
3  long fib_rec(int n)
4  {
5      if (n <= 1) return n;
6      return fib_rec(n - 1) + fib_rec(n - 2);
7  }
8
9  long fib_iter(int n)
10 {
11     long a = 0, b = 1;
12     for (int i = 0; i < n; i++) { long t = a + b; a = b; b = t; }
13     return a;
14 }
15
16 int main(void)
17 {
18     for (int i = 0; i <= 10; i++) printf("%ld ", fib_rec(i));
19     printf("\n");
20     for (int i = 0; i <= 10; i++) printf("%ld ", fib_iter(i));
21     printf("\n");
22     return 0;
23 }

```

```
0 1 1 2 3 5 8 13 21 34 55
0 1 1 2 3 5 8 13 21 34 55
```

Both are correct. They are not remotely equivalent in cost.



The shaded nodes show the duplication: $F(3)$ is computed twice, $F(2)$ three times. The recursive version performs roughly 2^n calls — $O(2^n)$ — while the loop performs n — $O(n)$. For $n = 50$ that is the difference between microseconds and years.

Key idea

Recursion is not automatically elegant or automatically slow. Naive Fibonacci is exponential because it recomputes overlapping subproblems. Factorial is linear because each call spawns exactly one more.

20.4 Recursion versus iteration

	Recursion	Iteration
Expresses	naturally recursive problems: trees, divide and conquer, grammars	repetition over a sequence
Memory	$O(\text{depth})$ stack frames	$O(1)$
Speed	function-call overhead per step	no overhead
Risk	stack overflow if the depth is large or the base case is wrong	infinite loop if the counter is wrong
Readability	often much clearer for tree-shaped problems	clearer for linear scans

✖ Stack overflow

```
int bad(int n) { return n * bad(n - 1); } /* no base case */
int also_bad(int n) { if (n == 0) return 1; return n * also_bad(n - 2); }
/* odd n never reaches 0 */
```

Both recurse forever and exhaust the 8 MB stack, producing a segmentation fault. The second is nastier: it works for even inputs. **Check that every recursive call moves strictly towards the base case.**

20.5 Tail recursion

A call is **tail recursive** when the recursive call is the very last thing the function does — nothing remains to be computed with its result.

```

/* NOT tail recursive: the multiplication happens after the call returns */
long fact(int n) { return (n <= 1) ? 1 : n * fact(n - 1); }

/* tail recursive: the call is the last action */
long fact_tail(int n, long acc) { return (n <= 1) ? acc : fact_tail(n - 1, n * acc); }

```

A compiler can turn a tail call into a jump, reusing the same stack frame and making the recursion effectively a loop. `gcc -O2` does this. At `-O0`, which is what you should debug with, it does not — so do not rely on it for correctness.

20.6 Classical recursive problems

20.6.1 Greatest common divisor

```
int gcd(int a, int b) { return (b == 0) ? a : gcd(b, a % b); }
```

Euclid's algorithm, and one of the oldest algorithms known. It runs in $O(\log \min(a, b))$.

20.6.2 Towers of Hanoi

```

hanoi.c
1  #include <stdio.h>
2
3  void hanoi(int n, char from, char to, char via)
4  {
5      if (n == 1) { printf("move disk 1: %c -> %c\n", from, to); return; }
6      hanoi(n - 1, from, via, to);
7      printf("move disk %d: %c -> %c\n", n, from, to);
8      hanoi(n - 1, via, to, from);
9  }
10
11 int main(void) { hanoi(3, 'A', 'C', 'B'); return 0; }

```

```

move disk 1: A -> C
move disk 2: A -> B
move disk 1: C -> B
move disk 3: A -> C
move disk 1: B -> A
move disk 2: B -> C
move disk 1: A -> C

```

Seven moves for three disks; in general $2^n - 1$, so the algorithm is $O(2^n)$. The iterative solution is far harder to write and no faster — this is a problem where recursion genuinely is the right tool.

20.6.3 Binary search

```

int bsearch_rec(const int a[], int lo, int hi, int target)
{
    if (lo > hi) return -1;                                /* base case: not found */
    int mid = lo + (hi - lo) / 2;                          /* avoids overflow */
    if (a[mid] == target) return mid;
    if (a[mid] < target) return bsearch_rec(a, mid + 1, hi, target);
    return bsearch_rec(a, lo, mid - 1, target);
}

```

Each call halves the search space, giving $O(\log n)$.

20.7 Recursion in this course

Recursion appears in three later places, and it is worth knowing where it is heading:

- **Linked lists** — traversal, reversal and printing can all be written recursively (Chapter 23).
- **Trees** — every traversal is naturally recursive, because a tree is defined recursively (Chapter 24).
- **Sorting** — merge sort and quicksort are divide-and-conquer algorithms (Chapter 25).

20.8 Chapter summary

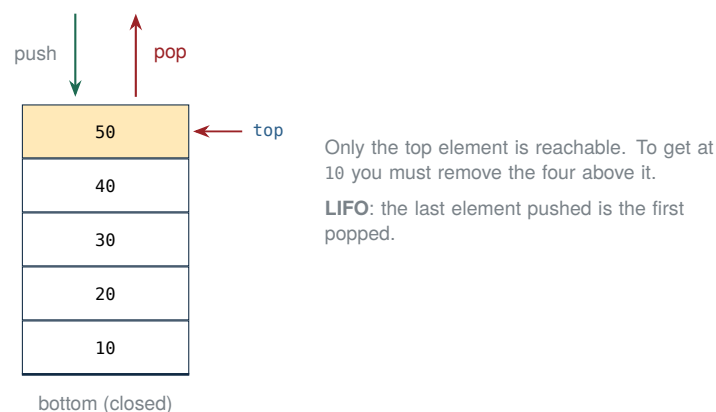
- Recursion defines a problem in terms of itself; every recursion needs a base case and a recursive step that moves towards it.
- Each call gets its own stack frame, which is why locals do not interfere — and why depth costs memory.
- Naive Fibonacci is $O(2^n)$ because subproblems overlap; the iterative version is $O(n)$.
- A missing or unreachable base case causes stack overflow, reported as a segmentation fault.
- Tail recursion can be optimised into a loop, but not at -00.

Stacks

21.1 The stack ADT

Key idea

A **stack** is a linear data structure that follows a **last in, first out** (LIFO) policy. There is only a single entry point, so elements are inserted and deleted through the same point. The pointer representing that entry point is the **top of stack** (TOS).



21.1.1 Operations

Operation	Meaning	Cost
<code>push(element)</code>	insert an element with a specified value into the stack	$O(1)$
<code>pop()</code>	delete the element at the top of the stack and return its value	$O(1)$
<code>peek()</code>	retrieve the element at the top <i>without</i> deleting it	$O(1)$
<code>isEmpty()</code>	is the stack empty?	$O(1)$
<code>isFull()</code>	is the stack full? (array implementation only)	$O(1)$

Every operation is constant time. That is the attraction of a stack: it does very little, very fast.

21.1.2 Applications

Stacks implement a surprising amount of computing:

- **Function calls** — the call stack itself (Chapter 12).
- **Backtracking** — maze solving, puzzle search, undo in an editor.
- **Syntax checking** — matching brackets and delimiters.
- **Expression evaluation** — infix to postfix conversion, and evaluating the result.
- **Memory management** — the stack segment.

- Depth-first search in graphs.

Stacks are implemented using either an array or a linked list. This chapter uses an array; Section 23.8 gives the linked version.

21.2 An array implementation, in separate files

Exercise VII requires the stack to be implemented so that *types and functionality are in separate code files*. That is exactly the header/source split of Chapter 11.

stack.h --- the interface

```

1  #ifndef STACK_H
2  #define STACK_H
3
4  #define STACK_CAPACITY 100
5
6  typedef struct {
7      int data[STACK_CAPACITY];
8      int top;           /* index of the top element; -1 when empty */
9  } Stack;
10
11 void init(Stack *s);
12 int is_empty(const Stack *s);
13 int is_full(const Stack *s);
14 int push(Stack *s, int value);    /* returns 1 on success, 0 if full */
15 int pop(Stack *s, int *out);      /* returns 1 on success, 0 if empty */
16 int peek(const Stack *s, int *out);
17 int size(const Stack *s);
18
19 #endif /* STACK_H */

```

stack.c --- the implementation

```

1  #include "stack.h"
2
3  void init(Stack *s)    { s->top = -1; }
4  int is_empty(const Stack *s) { return s->top == -1; }
5  int is_full(const Stack *s) { return s->top == STACK_CAPACITY - 1; }
6  int size(const Stack *s) { return s->top + 1; }
7
8  int push(Stack *s, int value)
9  {
10     if (is_full(s)) return 0;
11     s->data[++s->top] = value;    /* pre-increment: move up, then store */
12     return 1;
13 }
14
15 int pop(Stack *s, int *out)
16 {
17     if (is_empty(s)) return 0;
18     *out = s->data[s->top--];    /* post-decrement: read, then move down */
19     return 1;
20 }
21
22 int peek(const Stack *s, int *out)
23 {
24     if (is_empty(s)) return 0;
25     *out = s->data[s->top];
26     return 1;
27 }

```

main.c --- using it

```

1  #include <stdio.h>
2  #include "stack.h"
3
4  int main(void)
5  {
6      Stack s;
7      init(&s);
8      printf("empty? %d size=%d\n", is_empty(&s), size(&s));
9
10     for (int i = 1; i <= 5; i++) {
11         push(&s, i * 10);
12         printf("push %d -> size %d\n", i * 10, size(&s));
13     }
14
15     int v;
16     peek(&s, &v);
17     printf("peek = %d (size still %d)\n", v, size(&s));
18
19     while (pop(&s, &v)) printf("pop %d ", v);
20     printf("\nempty? %d\n", is_empty(&s));
21     printf("pop on empty returns %d\n", pop(&s, &v));
22     return 0;
23 }

```

```

$ gcc -Wall -Wextra -std=c18 -o prog main.c stack.c
$ ./prog
empty? 1 size=0
push 10 -> size 1
push 20 -> size 2
push 30 -> size 3
push 40 -> size 4
push 50 -> size 5
peek = 50 (size still 5)
pop 50 pop 40 pop 30 pop 20 pop 10
empty? 1
pop on empty returns 0

```

 Why pop returns a status, not the value

```

int pop(Stack *s);           /* what should this return on an empty stack? */
int pop(Stack *s, int *out); /* status returned; value written through out */

```

The first design has no way to report failure — every `int` is a legitimate stack value, so no sentinel works. Returning a status and delivering the value through a pointer parameter is the standard C solution, and it is why Chapter 6 spent time on pointer parameters.

✗ ++s->top versus s->top++

`s->data[++s->top] = value` increments `top` first, *then* stores — correct, because `top` is the index of the current top element and the new element goes above it. Writing `s->data[s->top++] = value` would store at the old `top`, overwriting the existing element. The two-line form is clearer if you are unsure:

```

s->top = s->top + 1;
s->data[s->top] = value;

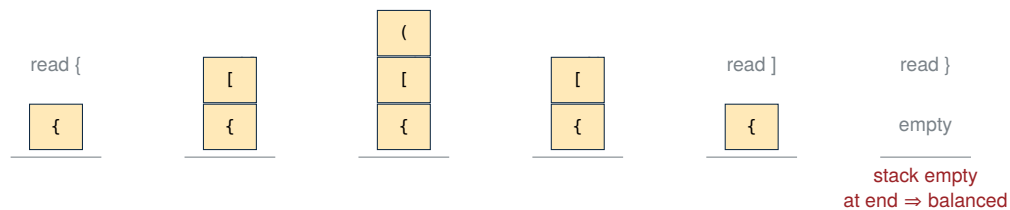
```

21.3 Application: bracket matching

Exercise VII question 1(d) is the classic stack application. The algorithm:

1. Read the input string character by character. If a character is not one of () [] { }, ignore it and read the next.
2. If the character is an **opening** symbol, push it.
3. If it is a **closing** symbol, pop the stack and compare the popped character with the matching opening symbol. If they match, continue; if not, report the mismatch and stop.
4. When the entire string has been processed, the stack must be empty. If so, report that all brackets are in order.

{ a [b (c) d] e }



brackets.c

```

1  #include <stdio.h>
2
3  #define CAP 256
4
5  typedef struct { char d[CAP]; int top; } CStack;
6
7  static void cinit(CStack *s) { s->top = -1; }
8  static int cempty(const CStack *s) { return s->top == -1; }
9  static int cpush(CStack *s, char c) { if (s->top >= CAP-1) return 0;
10                                     s->d[++s->top] = c; return 1; }
11 static int cpop(CStack *s, char *o) { if (cempty(s)) return 0;
12                                     *o = s->d[s->top--]; return 1; }
13
14 /* the opening symbol that matches a given closing symbol */
15 static char match(char c) { return c=='?' ? '(' : c==']' ? '[' : c=='}' ? '{' : 0; }
16
17 int check(const char *str)
18 {
19     CStack s; cinit(&s);
20     char popped;
21
22     for (int i = 0; str[i] != '\0'; i++) {
23         char c = str[i];
24         if (c=='(' || c=='[' || c=='{') {
25             cpush(&s, c);
26         } else if (c==')' || c==']' || c=='}') {
27             if (!cpop(&s, &popped)) {
28                 printf(" error: '%c' at %d has no opening symbol\n", c, i);
29                 return 0;
30             }
31             if (popped != match(c)) {
32                 printf(" error: '%c' at %d does not match '%c'\n", c, i, popped);
33                 return 0;
34             }
35         }
36         /* any other character is ignored */
37     }

```

```

38     if (!empty(&s)) { printf(" error: unclosed '%c'\n", s.d[s.top]); return 0; }
39     return 1;
40 }
41
42 int main(void)
43 {
44     const char *tests[] = {
45         "{a[b(c)d]e}",
46         "int f(int a[]) { return a[0]; }",
47         "()",
48         "((a)",
49         "a)b"
50     };
51     for (int i = 0; i < 5; i++) {
52         printf("%-34s ", tests[i]);
53         if (check(tests[i])) printf("all brackets are in order\n");
54     }
55     return 0;
56 }

```

{a[b(c)d]e}	all brackets are in order
int f(int a[]) { return a[0]; }	all brackets are in order
()	error: ']' at 1 does not match '('
((a)	error: unclosed '('
a)b	error: ')' at 1 has no opening symbol

There are three distinct failure modes, and a correct solution must detect all three: a mismatched pair, a closing symbol with nothing to match, and an opening symbol never closed. The third is caught only by the final emptiness test — a common omission.

21.4 Application: addition using stacks

Exercise VII question 1(b) adds two numbers digit by digit using three stacks. The algorithm:

1. Initialise three stacks: two for the operands and one for the result.
2. Push each operand into its stack one digit at a time — so the *least* significant digit ends up on top.
3. Pop both stacks, add the digits and any carry. Push the units digit of the sum onto the result stack and carry the tens forward.
4. Repeat until both operand stacks are empty.
5. Pop the result stack to read the final answer, most significant digit first.

The stacks do the digit reversal for free — that is the whole trick.

```

addstack.c
1  #include <stdio.h>
2  #include "stack.h"
3
4  static void push_digits(Stack *s, const char *num)
5  {
6      for (int i = 0; num[i] != '\0'; i++)
7          push(s, num[i] - '0');    /* character to digit */
8  }
9
10 int main(void)
11 {

```

```

12  const char *A = "4728", *B = "9615";
13  Stack a, b, result;
14  init(&a); init(&b); init(&result);
15
16  push_digits(&a, A);
17  push_digits(&b, B);
18
19  int carry = 0;
20  while (!is_empty(&a) || !is_empty(&b)) {
21      int x = 0, y = 0;
22      pop(&a, &x);           /* leaves x as 0 if that stack is empty */
23      pop(&b, &y);
24      int sum = x + y + carry;
25      push(&result, sum % 10);
26      carry = sum / 10;
27  }
28  if (carry) push(&result, carry);
29
30  printf(" %s\n+ %s\n= ", A, B);
31  int digit;
32  while (pop(&result, &digit)) printf("%d", digit);
33  printf("\n");
34  return 0;
35 }

```

```

4728
+ 9615
= 14343

```

This is not a silly exercise. It is how arbitrary-precision arithmetic works: the numbers can be thousands of digits long, far beyond any integer type, because only one digit is handled at a time.

21.5 Application: infix to postfix

Postfix (reverse Polish) notation needs no parentheses and no precedence rules: $3\ 4\ +$ means $3+4$. Converting infix to postfix is the classic shunting-yard algorithm, and it is a stack application.

```

/* precedence: higher binds tighter */
int prec(char op) { return (op=='+'||op=='-') ? 1 : (op=='*'||op=='/') ? 2 : 0; }

/* for each token:
    operand      -> output it
    '('          -> push
    ')'          -> pop to output until '(' is popped
    operator     -> pop to output while the stack top has >= precedence,
                  then push this operator
    at the end, pop everything remaining to output */

```

Infix	Postfix
$3 + 4 * 2$	$3\ 4\ 2\ * +$
$(3 + 4) * 2$	$3\ 4\ +\ 2\ *$
$a + b * c - d$	$a\ b\ c\ * + d -$

21.6 Chapter summary

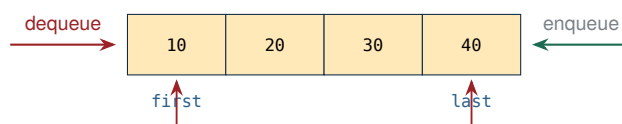
- A stack is LIFO with a single entry point, the top of stack.
- push, pop and peek are all $O(1)$.
- peek reads the top without removing it; pop removes it.
- Always check `isEmpty` before popping and `isFull` before pushing in an array implementation.
- Return a status and deliver the value through a pointer, so that failure is distinguishable from a legitimate value.
- Applications: function calls, backtracking, bracket matching, expression conversion, undo, depth-first search.

Queues

22.1 The queue ADT

Key idea

A **queue** is a linear data structure that follows a **first in, first out (FIFO)** policy. Because it is FIFO, *two* pointers are required: one for the first element and one for the last. These are called *first* and *last* (also *front* and *rear*).



Insert at one end, remove from the other. The element waiting longest is the one served next.

22.1.1 Operations

Operation	Meaning	Cost
enqueue(element)	insert an element with a specified value at the end of the queue, using the last pointer	$O(1)$
dequeue()	delete the element at the first pointer and return its value	$O(1)$
front() / peek()	read the first element without removing it	$O(1)$
isEmpty(), isFull()	state tests	$O(1)$

22.1.2 Applications

Queues appear wherever things must be served in arrival order:

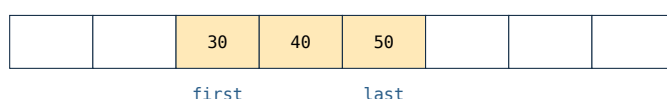
- **Process scheduling** — the OS ready queue (often a priority queue).
- **Keyboard input buffer** — keystrokes are processed in the order typed.
- **Print spoolers**, network packet buffers, message queues.
- **Breadth-first search** in graphs and trees.

Like stacks, queues are implemented using arrays or linked lists.

22.2 The problem with a naive array queue

The obvious implementation — first starts at 0, last advances on each enqueue — fails quickly.

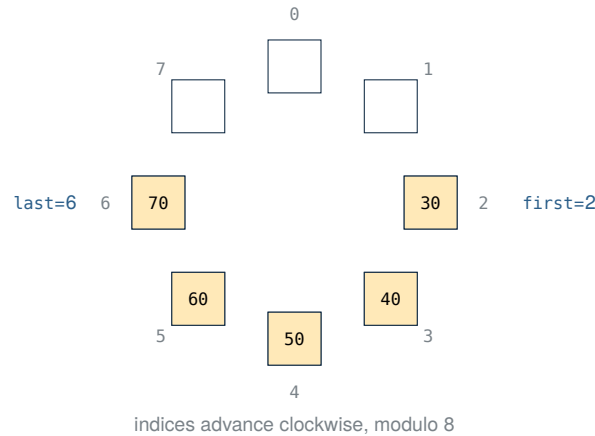
after 5 enqueues then 2 dequeues:



Cells 0 and 1 are free but unreachable — last can only move right. The queue reports “full” with room to spare.

The circular queue

Wrap both indices around using modular arithmetic: $\text{last} = (\text{last} + 1) \% \text{CAPACITY}$. The array is then treated as a ring, and no cell is ever wasted.



22.3 A circular array implementation

queue.h

```

1  #ifndef QUEUE_H
2  #define QUEUE_H
3
4  #define QUEUE_CAPACITY 8
5
6  typedef struct {
7      int data[QUEUE_CAPACITY];
8      int first;    /* index of the front element */
9      int last;     /* index of the rear element */
10     int count;     /* how many elements are stored */
11 } Queue;
12
13 void q_init(Queue *q);
14 int q_is_empty(const Queue *q);
15 int q_is_full(const Queue *q);
16 int enqueue(Queue *q, int value);
17 int dequeue(Queue *q, int *out);
18 int q_front(const Queue *q, int *out);
19 int q_size(const Queue *q);
20
21 #endif /* QUEUE_H */

```

queue.c

```

1  #include "queue.h"
2
3  void q_init(Queue *q)    { q->first = 0; q->last = -1; q->count = 0; }
4  int q_is_empty(const Queue *q){ return q->count == 0; }
5  int q_is_full(const Queue *q) { return q->count == QUEUE_CAPACITY; }
6  int q_size(const Queue *q)   { return q->count; }
7
8  int enqueue(Queue *q, int value)
9  {
10     if (q_is_full(q)) return 0;
11     q->last = (q->last + 1) % QUEUE_CAPACITY;    /* wrap around */

```

```

12     q->data[q->last] = value;
13     q->count++;
14     return 1;
15 }
16
17 int dequeue(Queue *q, int *out)
18 {
19     if (q_is_empty(q)) return 0;
20     *out = q->data[q->first];
21     q->first = (q->first + 1) % QUEUE_CAPACITY; /* wrap around */
22     q->count--;
23     return 1;
24 }
25
26 int q_front(const Queue *q, int *out)
27 {
28     if (q_is_empty(q)) return 0;
29     *out = q->data[q->first];
30     return 1;
31 }

```

Why keep a count?

Without it, `first == last + 1` is ambiguous: it means both “empty” and “full”. The alternatives are to keep a count (simplest, used here) or to waste one cell so that full and empty are distinguishable. Keeping the count also makes `q_size` $O(1)$.

main.c

```

1  #include <stdio.h>
2  #include "queue.h"
3
4  static void show(const Queue *q)
5  {
6      printf("  first=%d last=%d count=%d  cells:[", q->first, q->last, q->count);
7      for (int i = 0; i < QUEUE_CAPACITY; i++) printf("%s%d", i ? " " : "", q->data[i]);
8      printf("]\n");
9  }
10
11 int main(void)
12 {
13     Queue q;  q_init(&q);
14
15     for (int i = 1; i <= 5; i++) enqueue(&q, i * 10);
16     printf("after 5 enqueues:\n");  show(&q);
17
18     int v;
19     dequeue(&q, &v);  dequeue(&q, &v);
20     printf("after 2 dequeues (last removed was %d):\n", v);  show(&q);
21
22     for (int i = 6; i <= 10; i++)
23         if (!enqueue(&q, i * 10)) printf("  queue full, cannot add %d\n", i * 10);
24     printf("after wrapping around:\n");  show(&q);
25
26     printf("drain: ");
27     while (dequeue(&q, &v)) printf("%d ", v);
28     printf("\n");
29     return 0;
30 }

```

```

after 5 enqueues:
    first=0 last=4 count=5  cells:[10 20 30 40 50 0 529267711 0]
after 2 dequeues (last removed was 20):
    first=2 last=4 count=3  cells:[10 20 30 40 50 0 529267711 0]
after wrapping around:
    first=2 last=1 count=8  cells:[90 100 30 40 50 60 70 80]
drain: 30 40 50 60 70 80 90 100

```

Read the third line carefully: `last=1` is *less* than `first=2`. The queue has wrapped — 90 and 100 occupy cells 0 and 1, which the two dequeues freed. Nothing was wasted, and the drain still comes out in the correct FIFO order. Note also the garbage in cells 5–7 in the first two dumps: those cells were never written, so they hold whatever was on the stack, exactly as in Chapter 8.

✘ The dequeued values still sit in the array

dequeue only advances `first`; it does not erase the cell. The values 10 and 20 remain visible in the dump. They are logically gone — nothing can reach them — and they will be overwritten when the queue wraps. If a queue holds pointers or sensitive data, overwrite the slot explicitly.

22.4 Priority queues

Exercise VII question 2(a) asks for a **priority queue**: tasks arrive in arbitrary order but are dequeued by priority — High, then Medium, then Low — and in FIFO order within each priority. The exercise proposes two designs.

22.4.1 Design (i): three queues, one per priority

pq3.c

```

1  #include <stdio.h>
2  #include "queue.h"
3
4  typedef enum { HIGH = 0, MEDIUM = 1, LOW = 2 } Priority;
5  static const char *pname[] = { "High", "Medium", "Low" };
6
7  typedef struct { Queue q[3]; } PQ;
8
9  void pq_init(PQ *p) { for (int i = 0; i < 3; i++) q_init(&p->q[i]); }
10
11 int pq_enqueue(PQ *p, Priority pr, int task) { return enqueue(&p->q[pr], task); }
12
13 /* dequeue scans High, then Medium, then Low */
14 int pq_dequeue(PQ *p, int *task, Priority *pr)
15 {
16     for (int i = 0; i < 3; i++)
17         if (dequeue(&p->q[i], task)) { *pr = (Priority) i; return 1; }
18     return 0;
19 }
20
21 int main(void)
22 {
23     PQ p; pq_init(&p);
24     pq_enqueue(&p, LOW, 101);
25     pq_enqueue(&p, HIGH, 102);
26     pq_enqueue(&p, MEDIUM, 103);
27     pq_enqueue(&p, HIGH, 104);
28     pq_enqueue(&p, LOW, 105);

```

```

29
30     int task; Priority pr;
31     while (pq_dequeue(&p, &task, &pr))
32         printf("serving task %d (%s)\n", task, pname[pr]);
33     return 0;
34 }

```

```

serving task 102 (High)
serving task 104 (High)
serving task 103 (Medium)
serving task 101 (Low)
serving task 105 (Low)

```

Both high-priority tasks are served first, and within High, 102 precedes 104 because it arrived first. Enqueue is $O(1)$; dequeue is $O(k)$ where k is the number of priority levels — here 3, so effectively constant.

22.4.2 Design (ii): a single queue holding all priorities

Store the priority alongside the value and search for the highest-priority element on each dequeue.

```

typedef struct { int task; Priority pr; } Item;

/* dequeue: scan the whole queue for the best priority, remove that element
   and shift the rest down */

```

	(i) Three queues	(ii) One queue
Enqueue	$O(1)$	$O(1)$
Dequeue	$O(k)$, k = number of levels	$O(n)$ scan, plus $O(n)$ shift
Memory	k arrays, some possibly empty	one array
Adding a level	new queue needed	no change
Verdict	better when levels are few and fixed	better when priorities are arbitrary integers

Note

For arbitrary numeric priorities and large n , neither design is used in practice: a **binary heap** gives $O(\log n)$ for both insert and extract-min. That is the structure behind OS schedulers and Dijkstra's algorithm.

22.5 Deques

A **double-ended queue** allows insertion and removal at both ends — `push_front`, `push_back`, `pop_front`, `pop_back`. It generalises both the stack and the queue: use one end only and it is a stack; use opposite ends and it is a queue.

22.6 Stack versus queue

	Stack	Queue
Policy	LIFO — last in, first out	FIFO — first in, first out
Entry points	one (the top)	two (first and last)
Pointers	top	first and last
Insert	push at the top	enqueue at the last
Delete	pop from the top	dequeue from the first
Typical use	function calls, backtracking, undo, DFS	scheduling, buffering, BFS

✓ Exam focus

Two facts are near-certain examination material: (i) a stack is LIFO with one pointer, a queue is FIFO with two; and (ii) an array queue must be *circular* — using modular arithmetic on both indices — otherwise space at the front is wasted and the queue falsely reports itself full.

22.7 Chapter summary

- A queue is FIFO, with first and last pointers.
- enqueue adds at the last, dequeue removes at the first; both are $O(1)$.
- A linear array queue wastes the space freed by dequeues; the circular queue fixes this with $\text{index} = (\text{index} + 1) \% \text{CAPACITY}$.
- Keep an explicit count to distinguish full from empty.
- A priority queue dequeues by priority, FIFO within a priority. Three separate queues give $O(1)$ enqueue and $O(k)$ dequeue.

Linked Lists

23.1 Why arrays are not enough

When considering arrays, the size has to be known in advance so that the relevant amount of memory can be allocated. Such static allocation results in wasted memory in some cases, and inefficiency when the array size must be increased to hold a large set of elements — growing an array means allocating a new one and copying everything across.

To address these shortcomings, a dynamic structure is required that allocates memory when needed and can resize itself.

Key idea

To accommodate dynamic resizing, a basic block is required with a **data** component and **links** to other elements, so that all blocks can be linked to one another to form a data structure. That basic block is a **node**. By varying how the pointers behave and how many there are, different data structures result.

23.2 The node

A **node** is a basic block allocated from heap storage — via `malloc` — where and when required to store data. Nodes are usually built with structs.

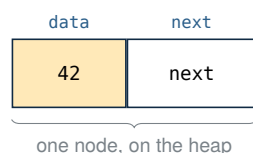
Node

```
struct Node {
    [<data_type> <member_name>]++;
    [struct Node* <pointer_name>]++;
};
```

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;

Node *temp = malloc(sizeof(struct Node)); /* create dynamically */
temp->data = 42;                          /* pointer-to-struct notation */
temp->next = NULL;
```

Recall from Chapter 10 that a struct may not contain an instance of itself, but may contain a *pointer* to itself. That is the whole basis of every linked structure.



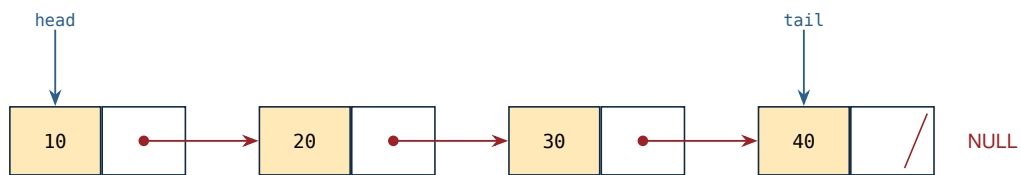
data holds the payload.
next holds the address of another node,
or NULL to mark the end.

23.3 The linked list

When nodes are chained sequentially, a **linked list** is produced. The behaviour of the pointers, and how many a node has, give three main variants:

- **Singly linked list** — one pointer per node, forward only.
- **Doubly linked list** — two pointers, forward and backward.
- **Circular linked list** — single or double, with the tail connected back to the head.

Nodes are created in heap memory and a pointer is provided to access each. But when a sequential linked structure is created it is important to keep track of its beginning and its end, since — unlike an array — it has no natural first and last elements at fixed positions. Two pointers named **head** and **tail** are therefore kept, referring to the first and last elements.



Nodes are scattered across the heap; the next pointers impose the order. Nothing about their addresses is sequential.

Figure 23.1. A singly linked list of four elements.

23.4 Array versus linked list

	Array	Linked list
Size	fixed at compile time	grows and shrinks at run time
Memory	one contiguous block	scattered nodes, one malloc each
Access element i	$O(1)$ — computed address	$O(n)$ — walk from the head
Insert at front	$O(n)$ — shift everything	$O(1)$
Insert at back	$O(1)$ if space	$O(1)$ with a tail pointer
Delete	$O(n)$ — shift to close the gap	$O(1)$ once positioned
Memory overhead	none	one pointer (8 bytes) per element
Cache behaviour	excellent — contiguous	poor — scattered

Key idea

Arrays trade insertion cost for access speed; linked lists trade the reverse. Use an array when you index a lot and resize rarely; use a linked list when you insert and delete a lot, especially at the ends, and rarely need element i .

23.5 Singly linked list operations

There are two variants each of insertion and deletion.

23.5.1 AddToHead(X)

1. Create a new node (Temp) and insert the value x .

2. Set Temp's next reference to the current head node.
3. Update the head reference to Temp.

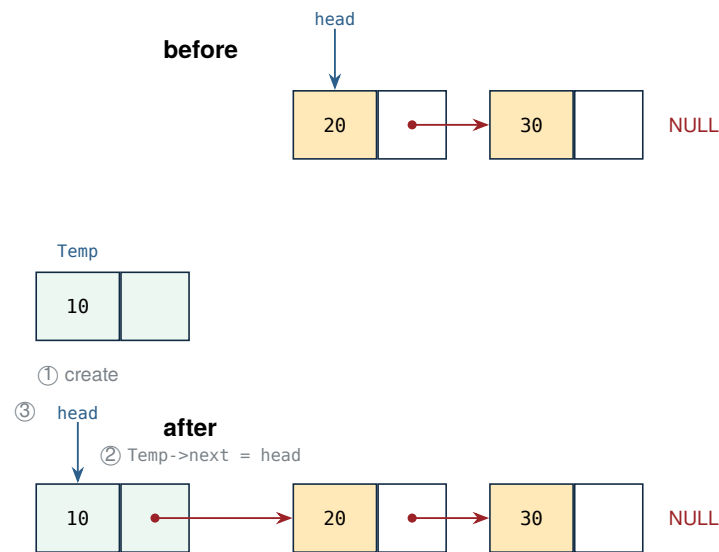


Figure 23.2. AddToHead: the order of steps 2 and 3 is critical.

✗ Order matters

```
head = temp;           /* WRONG ORDER */
temp->next = head;     /* temp now points at ITSELF -- a one-node cycle,
                       and the rest of the list is leaked */
```

Always attach the new node to the list *before* moving head.

23.5.2 AddToTail(X)

1. Create a new node (Temp) and insert the value X.
2. Set the tail node's next reference to Temp.
3. Update the tail reference to Temp.

23.5.3 DeleteFromHead()

1. Record the data value of the head in a temporary variable.
2. Update the head reference to the next element of the head.
3. Free the old node and return the data.

23.5.4 DeleteFromTail()

1. Record the data value of the tail in a temporary variable.
2. **Find the node before the tail.**
3. Update the tail reference to that node, and set its next to NULL.

📌 Key idea

Step 2 is why DeleteFromTail is $O(n)$ in a singly linked list: there is no backward link, so the only way to find the predecessor is to walk from the head. This single asymmetry is the

main argument for a doubly linked list.

23.5.5 The special cases

Lecture 9 asks: *what are the special cases for insertion and deletion?* There are three, and every one of them is a source of bugs.

Case	What must be handled
Empty list	Inserting into an empty list must set <i>both</i> head and tail to the new node. tail->next cannot be dereferenced when tail is NULL.
Single-element list	Deleting the only node must set <i>both</i> head and tail back to NULL.
Operating at the head	Deleting the first node has no predecessor whose next can be updated — head itself must move.

✓ Exam focus

When asked to write a linked-list operation in an examination, handle the empty list and the single-element list explicitly. Marks are lost for code that works only on lists of three or more elements.

23.6 A complete singly linked list

sll.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Node { int data; struct Node *next; } Node;
5  typedef struct { Node *head; Node *tail; int count; } List;
6
7  void list_init(List *l) { l->head = l->tail = NULL; l->count = 0; }
8
9  int add_to_head(List *l, int x)
10 {
11     Node *temp = malloc(sizeof *temp);
12     if (!temp) return 0;
13     temp->data = x;
14     temp->next = l->head;          /* (2) attach BEFORE moving head */
15     l->head = temp;                /* (3) */
16     if (l->tail == NULL) l->tail = temp; /* special case: was empty */
17     l->count++;
18     return 1;
19 }
20
21 int add_to_tail(List *l, int x)
22 {
23     Node *temp = malloc(sizeof *temp);
24     if (!temp) return 0;
25     temp->data = x;
26     temp->next = NULL;
27     if (l->tail == NULL) l->head = l->tail = temp; /* special case: empty */
28     else { l->tail->next = temp; l->tail = temp; }
29     l->count++;
30     return 1;
31 }
32

```

```

33 int delete_from_head(List *l, int *out)
34 {
35     if (l->head == NULL) return 0;           /* special case: empty */
36     Node *temp = l->head;
37     *out = temp->data;
38     l->head = temp->next;
39     if (l->head == NULL) l->tail = NULL;      /* special case: was one */
40     free(temp);
41     l->count--;
42     return 1;
43 }
44
45 int delete_from_tail(List *l, int *out)
46 {
47     if (l->head == NULL) return 0;
48     *out = l->tail->data;
49     if (l->head == l->tail) {                 /* special case: one node */
50         free(l->head);
51         l->head = l->tail = NULL;
52         l->count--;
53         return 1;
54     }
55     Node *p = l->head;
56     while (p->next != l->tail) p = p->next;   /* O(n) walk */
57     free(l->tail);
58     l->tail = p;
59     p->next = NULL;
60     l->count--;
61     return 1;
62 }
63
64 Node *search(const List *l, int x)
65 {
66     for (Node *p = l->head; p != NULL; p = p->next)
67         if (p->data == x) return p;
68     return NULL;
69 }
70
71 int delete_value(List *l, int x)
72 {
73     Node *p = l->head, *prev = NULL;
74     while (p && p->data != x) { prev = p; p = p->next; }
75     if (!p) return 0;                         /* not found */
76     if (prev == NULL) l->head = p->next;       /* deleting the head */
77     else prev->next = p->next;
78     if (p == l->tail) l->tail = prev;          /* deleting the tail */
79     free(p);
80     l->count--;
81     return 1;
82 }
83
84 void print_list(const List *l)
85 {
86     printf("head -> ");
87     for (Node *p = l->head; p; p = p->next) printf("[%d] -> ", p->data);
88     printf("NULL (count=%d)\n", l->count);
89 }
90
91 void free_list(List *l)
92 {
93     Node *p = l->head;
94     while (p) { Node *n = p->next; free(p); p = n; } /* save next FIRST */

```

```

95     list_init(l);
96 }
97
98 int main(void)
99 {
100     List l; list_init(&l);
101     printf("empty:      "); print_list(&l);
102
103     add_to_head(&l, 30); add_to_head(&l, 20); add_to_head(&l, 10);
104     printf("3 x AddToHead:  "); print_list(&l);
105
106     add_to_tail(&l, 40); add_to_tail(&l, 50);
107     printf("2 x AddToTail:  "); print_list(&l);
108
109     int v;
110     delete_from_head(&l, &v); printf("DeleteFromHead=%d: ", v); print_list(&l);
111     delete_from_tail(&l, &v); printf("DeleteFromTail=%d: ", v); print_list(&l);
112
113     printf("search(30) -> %s\n", search(&l, 30) ? "found" : "not found");
114     printf("search(99) -> %s\n", search(&l, 99) ? "found" : "not found");
115
116     delete_value(&l, 30); printf("Delete(30):      "); print_list(&l);
117
118     free_list(&l);
119     printf("after free_list:  "); print_list(&l);
120     return 0;
121 }

```

```

empty:      head -> NULL    (count=0)
3 x AddToHead:  head -> [10] -> [20] -> [30] -> NULL    (count=3)
2 x AddToTail:  head -> [10] -> [20] -> [30] -> [40] -> [50] -> NULL    (count=5)
DeleteFromHead=10: head -> [20] -> [30] -> [40] -> [50] -> NULL    (count=4)
DeleteFromTail=50: head -> [20] -> [30] -> [40] -> NULL    (count=3)
search(30) -> found
search(99) -> not found
Delete(30):    head -> [20] -> [40] -> NULL    (count=2)
after free_list: head -> NULL    (count=0)

```

```

$ gcc -g -O0 -fsanitize=address -o sll sll.c && ./sll > /dev/null
$ # no output from the sanitizer: no leaks, no invalid accesses

```

Note the ordering in `add_to_head`: three `AddToHead` calls with 30, 20, 10 produce the list 10, 20, 30 — each new element goes in *front*. That reversal is exactly what makes a head-inserting linked list behave as a stack.

✗ Freeing a list

```

for (Node *p = l->head; p != NULL; p = p->next)
    free(p);          /* WRONG: p->next is read AFTER p is freed */

```

The loop's update expression dereferences memory that has just been released — a use-after-free that AddressSanitizer reports immediately. Save the next pointer first:

```

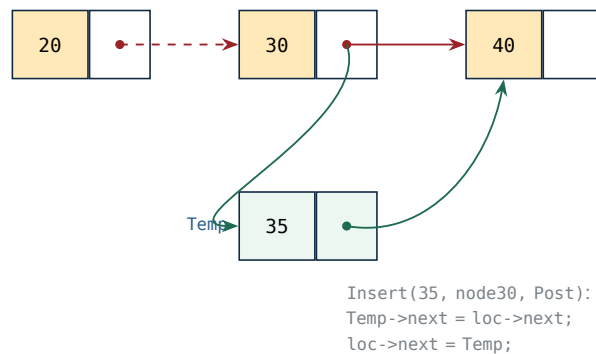
Node *p = l->head;
while (p) { Node *n = p->next; free(p); p = n; }

```

23.7 Generic insertion and deletion

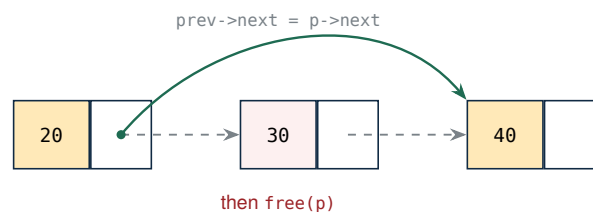
23.7.1 Insert(*X*, *Location*, *Pre/Post*)

1. Find the node at the location.
2. For *Post*: add the new Temp node with value *X* as the next of the node at the location, and set Temp's next to that node's former next.
3. For *Pre*: add Temp as the next of the node *previous* to the location, and set Temp's next to the location node.



23.7.2 Delete(*X*)

1. Search for the node with data value *X*.
2. Set the node's *previous* node's next element to the node's next element.
3. Free the node. (*X* may also be given as a position rather than a value.)



23.8 Stacks and queues from a singly linked list

ADT	Mapping	Cost
Stack push(<i>x</i>)	AddToHead(<i>x</i>)	$O(1)$
Stack pop()	DeleteFromHead()	$O(1)$
Queue enqueue(<i>x</i>)	AddToTail(<i>x</i>)	$O(1)$ with a tail pointer
Queue dequeue()	DeleteFromHead()	$O(1)$

Key idea

Both mappings deliberately avoid DeleteFromTail, which is the only $O(n)$ operation. A stack could equally use tail-insert and tail-delete, and a queue could enqueue at the head and dequeue at the tail — but both would then be $O(n)$ per operation. Choosing which end to operate on is the entire design decision.

```

/* stack, using the list above */
void push(List *l, int x)      { add_to_head(l, x); }
int pop(List *l, int *out)    { return delete_from_head(l, out); }

/* queue */
void enqueue(List *l, int x)   { add_to_tail(l, x); }
int dequeue(List *l, int *out){ return delete_from_head(l, out); }

```

Compared with the array versions of Chapters 21 and 22, the linked versions have no capacity limit and no `isFull`, but they pay 8 bytes per element for the pointer and a `malloc` per insertion.

23.9 Doubly linked lists

The doubly linked list maintains a link to the next node *and* to the previous node, through `next` and `prev` pointers.

```

typedef struct DNode {
    int      data;
    struct DNode *next;
    struct DNode *prev;
} DNode;

```

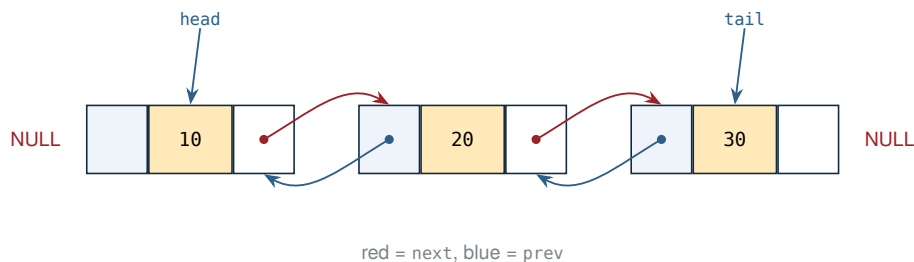


Figure 23.3. A doubly linked list. Every link is now bidirectional.

How do the operations change when a pointer to the previous node is present?

Operation	Singly	Doubly
DeleteFromTail	$O(n)$ — must walk to find the predecessor	$O(1)$ — <code>tail->prev</code> is the predecessor
Delete(node)	needs <code>prev</code> tracked during the search, or a second walk	$O(1)$ given the node: <code>p->prev->next = p->next</code>
Traverse backwards	impossible	natural
Insert before a node	needs the predecessor	$O(1)$
Memory	one pointer per node	two pointers per node
Bookkeeping	one link to update per change	two links, easy to forget one

```

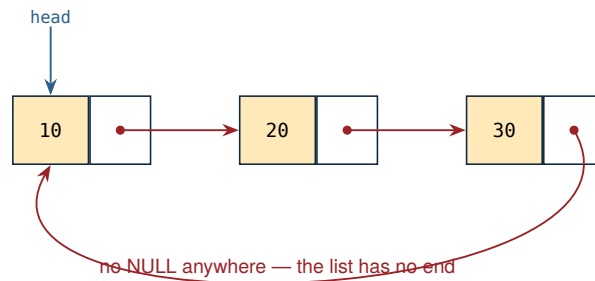
/* deleting node p from a doubly linked list -- no search needed */
if (p->prev) p->prev->next = p->next; else head = p->next;
if (p->next) p->next->prev = p->prev; else tail = p->prev;
free(p);

```

The four lines above contain all four special cases: `p` is the head, `p` is the tail, `p` is both (single-element list), `p` is in the middle.

23.10 Circular linked lists

A circular linked list may be singly or doubly linked. The difference from the other two is that **the tail is connected to the head**, and in the doubly linked case the head's prev points back to the tail.



Circular lists suit round-robin problems: process scheduling, turn-taking games, a repeating playlist. The traversal condition changes from `p != NULL` to `do { ... } while (p != head)`.

✖ Common pitfall

A circular list has no NULL to stop at. A loop written `while (p != NULL)` spins forever. Every traversal must instead compare against the starting node.

23.11 Debugging linked lists

Section 17.8 gives a full gdb session on this data structure. The essentials:

Symptom	Likely cause
Segfault on traversal	a next pointer is wild — a node was used before next was assigned, or after being freed
Infinite loop	a cycle: some next points back to an earlier node. Print p addresses and watch for a repeat
Last element missing	tail not updated after an insert or delete
Crash on the second delete	the empty/single-element special case is missing
Leak reported by ASan	free_list missing, or free skipped in an early-return path

```

(gdb) p *head          /* {data = 40, next = 0x555555592e0} */
(gdb) p *head->next     /* follow the chain by hand          */
(gdb) p head->next->next->data

```

23.12 Chapter summary

- A node is a heap-allocated struct holding data and one or more pointers to other nodes; a struct may contain a pointer to its own type.
- head and tail track the ends, because a linked list has no fixed positions.
- AddToHead, AddToTail, DeleteFromHead are $O(1)$; DeleteFromTail is $O(n)$ in a singly linked list because the predecessor must be found.
- Always attach the new node before moving head, and always save next before calling free.

- Three special cases must be handled every time: empty list, single-element list, operating at the head.
- Stack = AddToHead + DeleteFromHead. Queue = AddToTail + DeleteFromHead.
- A doubly linked list adds a prev pointer, making backward traversal and deletion $O(1)$ at the cost of a second pointer per node.
- A circular list connects tail to head and has no NULL terminator.

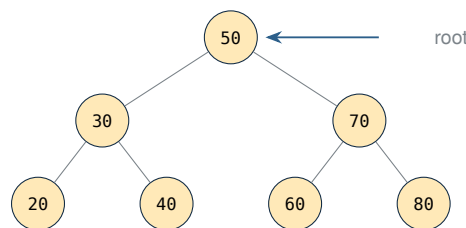
Trees and Graphs

This chapter goes beyond the linear structures the syllabus concentrates on. It is here because the terminology appears in the classification of Chapter 19, because trees are the natural next step after linked lists, and because they show what the node concept becomes when a node has more than one link.

24.1 Trees

Key idea

A **tree** is a non-linear data structure in which each node may have several successors (children) but exactly one predecessor (parent) — except the **root**, which has none. A tree of n nodes has exactly $n - 1$ edges and no cycles.



Term	Meaning
Root	the single node with no parent
Leaf	a node with no children
Internal node	a node with at least one child
Depth of a node	number of edges from the root to it (root has depth 0)
Height of a tree	the greatest depth of any node
Subtree	a node together with all its descendants
Degree	the number of children a node has

24.1.1 Binary trees and binary search trees

A **binary tree** has at most two children per node, called `left` and `right`. A **binary search tree** (BST) adds an ordering invariant:

for every node, all values in its left subtree are $<$ its value,
and all values in its right subtree are $>$ its value.

```

typedef struct TNode {
    int      data;
    struct TNode *left;
}

```

```

    struct TNode *right;
} TNode;

```

Compare this with `struct Node` from Chapter 23: it is the same idea with two links instead of one. That is the entire structural difference between a linear and a non-linear structure.

bst.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct TNode { int data; struct TNode *left, *right; } TNode;
5
6  TNode *insert(TNode *root, int x)
7  {
8      if (root == NULL) {
9          TNode *n = malloc(sizeof *n);
10         n->data = x; n->left = n->right = NULL;
11         return n;
12     }
13     if (x < root->data) root->left = insert(root->left, x);
14     else if (x > root->data) root->right = insert(root->right, x);
15     return root;          /* duplicates ignored */
16 }
17
18 TNode *search(TNode *root, int x)
19 {
20     if (root == NULL || root->data == x) return root;
21     return (x < root->data) ? search(root->left, x) : search(root->right, x);
22 }
23
24 void inorder(TNode *r) { if (r) { inorder(r->left); printf("%d ", r->data);
25                          inorder(r->right); } }
26 void preorder(TNode *r) { if (r) { printf("%d ", r->data); preorder(r->left);
27                                preorder(r->right); } }
28 void postorder(TNode *r) { if (r) { postorder(r->left); postorder(r->right);
29                                printf("%d ", r->data); } }
30
31 int height(TNode *r)
32 {
33     if (r == NULL) return -1;
34     int l = height(r->left), rr = height(r->right);
35     return 1 + (l > rr ? l : rr);
36 }
37
38 void free_tree(TNode *r) { if (r) { free_tree(r->left); free_tree(r->right); free(r); } }
39
40 int main(void)
41 {
42     TNode *root = NULL;
43     int values[] = {50, 30, 70, 20, 40, 60, 80};
44     for (int i = 0; i < 7; i++) root = insert(root, values[i]);
45
46     printf("in-order   : "); inorder(root); printf("\n");
47     printf("pre-order  : "); preorder(root); printf("\n");
48     printf("post-order : "); postorder(root); printf("\n");
49     printf("height    : %d\n", height(root));
50     printf("search(40) : %s\n", search(root, 40) ? "found" : "not found");
51     printf("search(45) : %s\n", search(root, 45) ? "found" : "not found");
52     free_tree(root);
53     return 0;
54 }

```

```

in-order   : 20 30 40 50 60 70 80
pre-order  : 50 30 20 40 70 60 80
post-order : 20 40 30 60 80 70 50
height     : 2
search(40) : found
search(45) : not found

```

Key idea

In-order traversal of a BST visits the values in sorted order. That one property is what makes the ordering invariant worth maintaining.

24.1.2 The three traversals

Traversal	Order	Used for
Pre-order	node, left, right	copying a tree; writing a prefix expression
In-order	left, node, right	retrieving BST values in sorted order
Post-order	left, right, node	freeing a tree; evaluating a postfix expression

Note that `free_tree` must be post-order: you cannot free a node before its children, because you would need its pointers afterwards.

24.1.3 Complexity

Operation	Balanced	Degenerate (sorted input)
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

✖ Inserting sorted data destroys a BST

Insert 10, 20, 30, 40, 50 into an empty BST and every node becomes the right child of the previous one. The tree degenerates into a linked list and every operation becomes $O(n)$. Self-balancing variants — AVL trees, red-black trees — rotate on insertion to keep the height at $O(\log n)$.

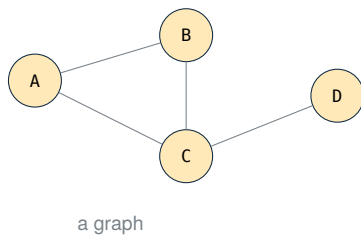
24.2 Graphs

Key idea

A **graph** $G = (V, E)$ is a set of vertices V and a set of edges E connecting them. Unlike a tree, a graph may contain cycles, may be disconnected, and imposes no parent/child hierarchy.

Graphs are *directed* or *undirected*, and *weighted* or *unweighted*. Road networks, social networks, dependency graphs and state machines are all graphs.

24.2.1 Two representations



adjacency matrix

	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

adjacency list

```

A -> B -> C
B -> A -> C
C -> A -> B -> D
D -> C

```

	Adjacency matrix	Adjacency list
Space	$O(V^2)$	$O(V + E)$
Is edge (u, v) present?	$O(1)$	$O(\text{degree}(u))$
List neighbours of u	$O(V)$	$O(\text{degree}(u))$
Best for	dense graphs	sparse graphs (the usual case)

24.2.2 Traversal: BFS and DFS

The two fundamental graph algorithms are also the best illustration of why stacks and queues matter.

	Breadth-first search	Depth-first search
Data structure	queue (FIFO)	stack (LIFO), or recursion
Order	all neighbours, then their neighbours	one path as deep as possible, then backtrack
Finds	shortest path in an unweighted graph	connectivity, cycles, topological order
Complexity	$O(V + E)$	$O(V + E)$

Key idea

BFS and DFS are the *same algorithm*. The only difference is whether the frontier is stored in a queue or a stack. That is the clearest possible demonstration that the choice of data structure determines behaviour.

```

/* the shared skeleton */
mark start as visited and put it in the container
while the container is not empty:
    take a vertex v out          /* dequeue -> BFS ; pop -> DFS */
    visit v
    for each unvisited neighbour w of v:
        mark w visited
        put w in the container

```

24.3 Chapter summary

- Trees and graphs are non-linear: a node may have several successors.
- A binary tree node is a linked-list node with two pointers instead of one.
- A BST keeps left < node < right; in-order traversal therefore yields sorted output, and search is $O(\log n)$ when balanced, $O(n)$ when degenerate.
- Traversals: pre-order (copy), in-order (sorted), post-order (free).

-
- Graphs are represented by an adjacency matrix ($O(V^2)$, dense) or adjacency lists ($O(V + E)$, sparse).
 - BFS uses a queue and finds shortest paths; DFS uses a stack or recursion. They differ only in the container.

Searching and Sorting

Sorting and searching are where the analysis of Chapter 26 earns its keep: the algorithms here all solve the same problem, and differ only in cost.

25.1 Searching

25.1.1 Linear search

Examine each element in turn until the target is found or the array ends.

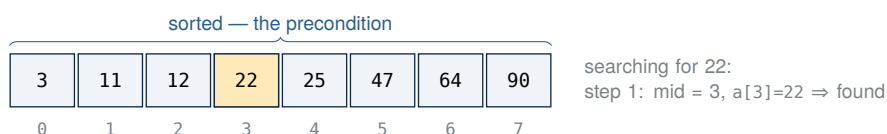
```
int linear_search(const int a[], int n, int target)
{
    for (int i = 0; i < n; i++)
        if (a[i] == target) return i;
    return -1;
}
```

Best case $O(1)$ (first element), worst and average $O(n)$. Requires nothing of the data — it need not be sorted.

25.1.2 Binary search

Requires a **sorted** array. Compare with the middle element and discard half the remaining range each time.

```
int binary_search(const int a[], int n, int target)
{
    int lo = 0, hi = n - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;    /* not (lo+hi)/2 -- avoids overflow */
        if (a[mid] == target) return mid;
        else if (a[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```



Each iteration halves the range, so after k steps the range is $n/2^k$. The search ends when $n/2^k = 1$, i.e. $k = \log_2 n$: $O(\log n)$.

	Requires	Best	Average	Worst
Linear search	nothing	$O(1)$	$O(n)$	$O(n)$
Binary search	sorted data	$O(1)$	$O(\log n)$	$O(\log n)$

Key idea

For $n = 1,000,000$, linear search needs up to a million comparisons; binary search needs at most 20. But sorting first costs $O(n \log n)$ — so sort only if you will search many times.

✗ $(lo + hi) / 2$ can overflow

For large arrays, $lo + hi$ may exceed `INT_MAX` and become negative, producing an out-of-bounds index. $lo + (hi - lo) / 2$ is mathematically identical and cannot overflow. This bug survived in the Java standard library for nine years.

25.2 Sorting

All five algorithms below were run on the same input; the outputs are real.

```
original    64 25 12 22 11 90 3 47
bubble      3 11 12 22 25 47 64 90
selection   3 11 12 22 25 47 64 90
insertion   3 11 12 22 25 47 64 90
merge       3 11 12 22 25 47 64 90
quick       3 11 12 22 25 47 64 90
```

25.2.1 Bubble sort

Repeatedly step through the array, swapping adjacent elements that are out of order. After pass i , the largest i elements have bubbled to the end.

```
void bubble_sort(int a[], int n)
{
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for (int j = 0; j < n - 1 - i; j++)
            if (a[j] > a[j+1]) {
                int t = a[j]; a[j] = a[j+1]; a[j+1] = t;
                swapped = 1;
            }
        if (!swapped) break;          /* already sorted -- early exit */
    }
}
```

The swapped flag gives a best case of $O(n)$ on already-sorted input. Without it, bubble sort is $O(n^2)$ always. It is the slowest of the five and is taught because it is the easiest to trace by hand.

25.2.2 Selection sort

Find the minimum of the unsorted portion and swap it into place.

```
void selection_sort(int a[], int n)
{
```

```

for (int i = 0; i < n - 1; i++) {
    int m = i;
    for (int j = i + 1; j < n; j++)
        if (a[j] < a[m]) m = j;
    if (m != i) { int t = a[i]; a[i] = a[m]; a[m] = t; }
}

```

Always $O(n^2)$ comparisons, but at most $n - 1$ *swaps* — the fewest of any of these algorithms. That matters when moving an element is expensive.

25.2.3 Insertion sort

Take each element and insert it into its correct place among those already sorted — the way most people sort a hand of cards.

```

void insertion_sort(int a[], int n)
{
    for (int i = 1; i < n; i++) {
        int key = a[i], j = i - 1;
        while (j >= 0 && a[j] > key) { a[j+1] = a[j]; j--; }
        a[j+1] = key;
    }
}

```

$O(n)$ on nearly-sorted data and $O(n^2)$ in the worst case. It is the fastest of all algorithms for small n , which is why real library sorts switch to it below about 16 elements.

25.2.4 Merge sort

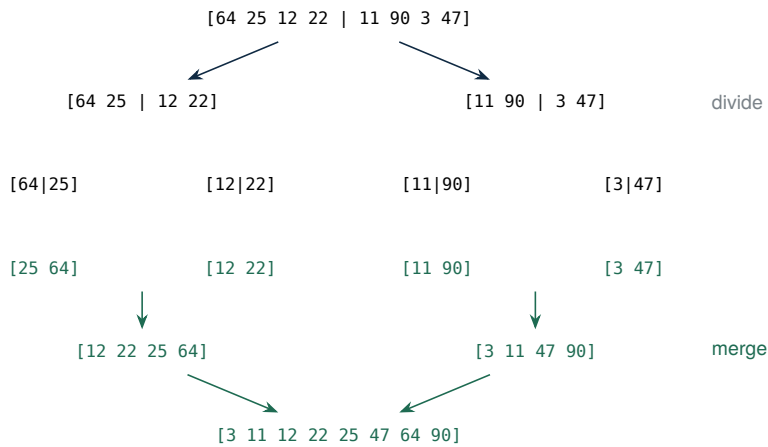
Divide the array in half, sort each half recursively, then merge the two sorted halves — the merge step of Section 14.10.2.

```

static void merge(int a[], int l, int m, int r, int tmp[])
{
    int i = l, j = m + 1, k = l;
    while (i <= m && j <= r) tmp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    while (i <= m) tmp[k++] = a[i++];
    while (j <= r) tmp[k++] = a[j++];
    for (int x = l; x <= r; x++) a[x] = tmp[x];
}

void merge_sort(int a[], int l, int r, int tmp[])
{
    if (l >= r) return; /* base case */
    int m = l + (r - l) / 2;
    merge_sort(a, l, m, tmp);
    merge_sort(a, m + 1, r, tmp);
    merge(a, l, m, r, tmp);
}

```



$\log n$ levels of recursion, $O(n)$ work merging at each level, giving $O(n \log n)$ *always* — best, average and worst. The cost is $O(n)$ extra memory for the temporary array.

25.2.5 Quicksort

Choose a **pivot**, partition the array so that everything smaller precedes it and everything larger follows, then sort the two partitions recursively.

```

static int partition(int a[], int lo, int hi)
{
    int pivot = a[hi], i = lo - 1;
    for (int j = lo; j < hi; j++)
        if (a[j] <= pivot) { i++; int t=a[i]; a[i]=a[j]; a[j]=t; }
    int t = a[i+1]; a[i+1] = a[hi]; a[hi] = t;
    return i + 1;
}

void quick_sort(int a[], int lo, int hi)
{
    if (lo < hi) {
        int p = partition(a, lo, hi);
        quick_sort(a, lo, p - 1);
        quick_sort(a, p + 1, hi);
    }
}

```

$O(n \log n)$ on average and the fastest in practice — it sorts in place with excellent cache behaviour. But if the pivot is consistently the smallest or largest element (which happens with already-sorted input and a last-element pivot) the partitions are maximally unbalanced and it degrades to $O(n^2)$. Real implementations choose the pivot as the median of three, or at random.

25.3 Comparison

Algorithm	Best	Average	Worst	Space	Stable?
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	no
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	yes
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	no

A sort is **stable** if equal elements keep their original relative order — which matters when sorting records by one field after another.

✓ Exam focus

Memorise this table. The three facts most often asked: merge sort is $O(n \log n)$ in *all* cases but needs $O(n)$ extra space; quicksort is $O(n^2)$ in the worst case despite being fastest in practice; insertion sort is $O(n)$ on nearly-sorted data.

25.4 The library sort

`<stdlib.h>` provides a generic sort. It takes a `void *` array, the element count, the element size, and a comparison function pointer — an application of both Section 7.7.4 and Section 7.7.8.

```
#include <stdlib.h>

int cmp_int(const void *a, const void *b)
{
    int x = *(const int *) a, y = *(const int *) b;
    return (x > y) - (x < y);      /* avoids overflow of x - y */
}

qsort(arr, n, sizeof arr[0], cmp_int);
```

✗ Common pitfall

The comparison function must return *negative*, *zero* or *positive* — not a boolean. Returning `x - y` works for small values but overflows for large ones; the `(x > y) - (x < y)` idiom does not.

25.5 Chapter summary

- Linear search is $O(n)$ and needs nothing; binary search is $O(\log n)$ and needs sorted data. Use `lo + (hi - lo) / 2`.
- Bubble, selection and insertion sort are $O(n^2)$ and sort in place.
- Merge sort is $O(n \log n)$ always but uses $O(n)$ extra space.
- Quicksort is $O(n \log n)$ on average, $O(n^2)$ in the worst case, and fastest in practice.
- Insertion sort beats everything for small n and nearly-sorted data.
- `qsort` from `<stdlib.h>` sorts anything, given a comparison function.

PART V

Analysis of Algorithms

Analysis of Algorithms

The slides for this topic are almost entirely images with no extractable text. This chapter reconstructs the material in full, works every code example posed in the lecture, and adds the proofs the slides only gesture at.

26.1 Why study algorithms

To create efficient programs. Which immediately raises two questions.

What is an efficient program? One that uses fewer resources — time and memory, chiefly — to produce the same result. Efficiency is not correctness; a wrong answer delivered quickly is worthless. But among correct programs, efficiency is what separates one that finishes from one that does not.

How does studying data structures and algorithms improve efficiency? Because the improvements available from choosing a better algorithm dwarf anything else. Consider searching one million sorted records: linear search takes up to 10^6 comparisons, binary search at most 20. No amount of compiler optimisation, faster hardware or hand-tuned C will close a gap of that size. The algorithm is the dominant term.

26.2 Units of measurement

We have well-established units for weight, volume, distance. How do we do this with computers? What must be measured to improve a program's efficiency?

26.2.1 Attempt 1: measure the execution time

The obvious answer is to time the program. In C, use `<time.h>`:

```
#include <time.h>
clock_t A = clock();
/* ... the work ... */
clock_t B = clock();
double seconds = (double)(B - A) / CLOCKS_PER_SEC;
```

In Java the equivalent is `System.nanoTime()`.

This is exactly the experiment Exercise II question 6(f) sets: time each program, run `clock()` repeatedly, plot the values against the run number and fit a regression line. Do it, and you will find the measurements scatter — which is the point of the exercise.

26.2.2 What is wrong with measured time

Problem	Consequence
Hardware dependent	the same program takes different times on different machines; results cannot be compared or published
Compiler dependent	-O0 and -O2 can differ by a factor of five
Language dependent	a C loop and a Python loop differ by 50× with no algorithmic difference
Input dependent	timing one input says nothing about behaviour on larger ones
Noisy	OS scheduling, caches, other processes: repeated runs of an identical program give different times
Requires the program to exist	you cannot time an algorithm you have not yet written — but you must choose between algorithms <i>before</i> writing them

Key idea

Measured time answers “how fast was this program, on this machine, today?” The question we need answered is “how does the cost *grow* as the input grows?” That question has a machine-independent answer.

26.2.3 Attempt 2: count operations

Instead of seconds, count the elementary operations — assignments, comparisons, arithmetic — and treat each as costing one unit of time. Consider:

Code	Operations	Cost
<code>total = 1;</code>	one assignment	1
<code>total = 5 * 6;</code>	one multiplication, one assignment	2
<code>for (i=0; i<n; i++) total += i;</code>	n additions, n assignments, n comparisons	$\approx 3n$

This is machine independent and can be done on paper before writing a line of code. It is the basis of everything that follows.

26.3 Order of growth

Key idea

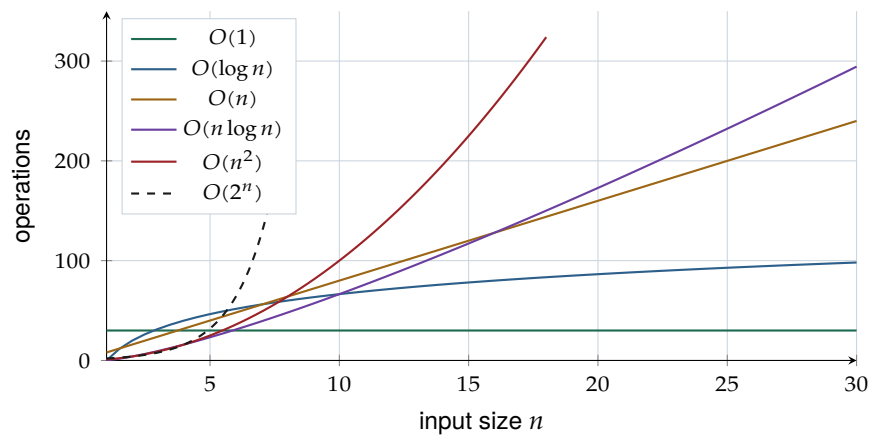
The **order of growth** represents how the computation time increases with the number of inputs. It is a rough estimate relating the size of the input to the time — and the exact order of growth is irrelevant; a rough estimate is good enough.

Consider the lecture’s own example:

$$f(n) = n^2 + 100n + 10^4$$

n	n^2	$100n$	10^4	$f(n)$	(% from n^2)
10	100	1,000	10,000	11,100	(0.9%)
100	10,000	10,000	10,000	30,000	(33%)
1,000	1,000,000	100,000	10,000	1,110,000	(90%)
10,000	10^8	10^6	10^4	$\approx 1.01 \times 10^8$	(99%)
100,000	10^{10}	10^7	10^4	$\approx 1.001 \times 10^{10}$	(99.9%)

For small n the constant 10^4 dominates. By $n = 1000$ the n^2 term contributes 90% of the total, and its share only grows. **As n becomes large, only the fastest-growing term matters.** Everything else — lower-order terms and constant factors — is noise.

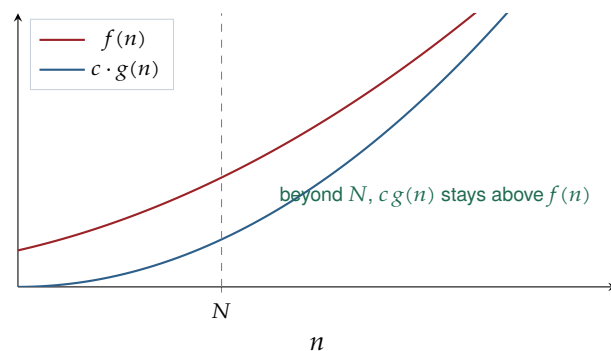


26.4 Big-O notation

Definition 26.1 (Big-O). $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq N.$$

In words: beyond some point N , f never exceeds a constant multiple of g . Big-O describes an **upper bound** on the efficiency of the program.



26.4.1 Properties

Property 26.1 (Smallest bound). Big-O is inherently imprecise — if $f(n)$ is $O(n)$ it is also $O(n^2)$ and $O(2^n)$. By convention the *smallest* possible $g(n)$ is selected.

Property 26.2 (Transitivity). If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Property 26.3 (Sum rule). If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Property 26.4 (Constant factors vanish). If $f(n) = c \cdot g(n)$ then $f(n)$ is $O(g(n))$. Constants are absorbed into the c of the definition.

Property 26.5 (Powers). The function an^k is $O(n^k)$.

Property 26.6 (Logarithm base is irrelevant). $\log_a n$ is $O(\log_b n)$ for any bases $a, b > 1$, because $\log_a n = \log_b n / \log_b a$ and $1/\log_b a$ is just a constant. This is why we write $O(\log n)$ without a base.

26.4.2 Worked proofs — the lecture's three problems

1. $T(n) = a_k n^k + \dots + a_1 n + a_0$ is $O(n^k)$

For $n \geq 1$ we have $n^i \leq n^k$ for every $i \leq k$. Therefore

$$|T(n)| \leq \sum_{i=0}^k |a_i| n^i \leq \sum_{i=0}^k |a_i| n^k = \left(\underbrace{\sum_{i=0}^k |a_i|}_c \right) n^k.$$

Taking $c = \sum |a_i|$ and $N = 1$ satisfies the definition, so $T(n)$ is $O(n^k)$. ■

Consequence: to find the Big-O of any polynomial, discard every term but the highest power, and discard its coefficient.

2. $1000n^2 + 50n$ is $O(n^2)$

We must produce a c and an N . For $n \geq 1$, $50n \leq 50n^2$, so

$$1000n^2 + 50n \leq 1000n^2 + 50n^2 = 1050n^2.$$

Take $c = 1050$ and $N = 1$. The definition is satisfied, so $1000n^2 + 50n$ is $O(n^2)$. ■

Note that the enormous constant 1000 is irrelevant to the classification. It matters in practice — a $1000n^2$ algorithm is genuinely a thousand times slower than an n^2 one — but it does not change the *growth*.

3. $g(n) = 2n^3 + 4n$ is **not** $O(n^2)$

Proof by contradiction. Suppose it were. Then there exist $c > 0$ and N with

$$2n^3 + 4n \leq c n^2 \quad \text{for all } n \geq N.$$

Divide both sides by n^2 (legitimate, since $n \geq N > 0$):

$$2n + \frac{4}{n} \leq c.$$

The left-hand side is at least $2n$, which grows without bound, so for any fixed c we may choose $n > \max(N, c/2)$ and the inequality fails. No such c exists, so $g(n)$ is not $O(n^2)$. ■

26.5 The other notations

Big-O is only an upper bound. Three companions complete the picture.

Notation	Bound	Definition
$O(g)$	upper	$f(n) \leq c g(n)$ for $n \geq N$ — “grows no faster than”
$\Omega(g)$	lower	$f(n) \geq c g(n)$ for $n \geq N$ — “grows at least as fast as”
$\Theta(g)$	tight	both: $c_1 g(n) \leq f(n) \leq c_2 g(n)$ — “grows exactly like”
$o(g)$	strict upper	$f(n)/g(n) \rightarrow 0$ — “grows strictly slower than”

Note

Strictly, “insertion sort is $O(n^2)$ ” says only that it is no worse than quadratic — which is also true of a linear algorithm. Θ is the precise statement. In practice everyone writes O and means Θ , and this course follows that convention.

26.5.1 Best, worst and average case

Case	Meaning	Linear search
Best	the most favourable input	$O(1)$ — first element
Worst	the least favourable input; the guarantee	$O(n)$ — absent or last
Average	expected over all inputs	$O(n)$ — about $n/2$ comparisons

Worst case is usually what is quoted, because it is a guarantee rather than a hope.

26.6 Analysing programs

Key idea

Algorithm analysis estimates the efficiency — the **complexity** — of a program. Interest centres on **time complexity** and, to a lesser extent, **memory complexity**. To compute time complexity, count the number of operations (assignments, comparisons and so on) as a function of the input size.

26.6.1 The counting rules

Construct	Cost
Simple statement	$O(1)$
Sequence of statements	the <i>maximum</i> of their costs (sum rule)
if/else	the cost of the test plus the more expensive branch
Loop	(iterations) $\times$ (cost of the body)
Nested loops	the product of their iteration counts
Function call	the cost of the function body

26.7 The ten lecture examples, analysed

Every example posed in the Analysis of Algorithms slides is worked below. The empirical iteration counts quoted were produced by running the loops.

26.7.1 Example 1 — constant

```
a = 5;
b = 7;
print(a*b);
```

Three statements, none depending on any input. The cost is 3 operations regardless of anything. $O(1)$ — constant time.

26.7.2 Example 2 — a fixed loop, then a variable one

```
for i in range(100):
    a = 5; b = 7; print(a*b);
```

The loop runs exactly 100 times. 100 is a constant, so the total is 300 operations — still $O(1)$. A large constant is still a constant.

```
N = input('Enter integer');
for i in range(int(N)):
    a = 5; b = 7; print(a*b);
```

What changes: the iteration count is now the input $N = n$. The body is $O(1)$ and runs n times, so the cost is $O(n)$ — linear. This is the single most important transition in the whole topic: a bound that is fixed in the source is a constant; a bound that comes from the input is a variable.

26.7.3 Example 3 — two nested loops

```
j = int(N);
for i in range(j):
    for k in range(j):
        a = 5; b = 7; print(a*b);
```

The outer loop runs n times; for each, the inner runs n times. Total $n \times n = n^2$ executions of a constant body. $O(n^2)$ — quadratic.

26.7.4 Example 4 — three nested loops

```
for i in range(j):
    for k in range(j):
        for l in range(j):
            a = 5; b = 7; print(a*b);
```

$n \times n \times n$. $O(n^3)$ — cubic. In general, k nested loops each running n times give $O(n^k)$.

26.7.5 Example 5 — the break that removes a loop

```
for i in range(j):
    for k in range(j):
        a = 5; b = 7; print(a*b);
        break;
```

Key idea

The break executes at the end of the inner loop's *first* iteration, so the inner loop always runs exactly once, whatever n is. Its cost is $O(1)$, and the outer loop runs n times: $O(n)$, not $O(n^2)$.

This example exists to teach that you must count what a loop *does*, not count the nesting depth. Two nested loops do not automatically mean $O(n^2)$.

26.7.6 Example 6 — $i*i < n$

```
for (int i = 0; i*i < n; i++) { total = 5*6; }
```

The loop continues while $i^2 < n$, i.e. while $i < \sqrt{n}$. It therefore runs $\lceil \sqrt{n} \rceil$ times. $O(\sqrt{n})$.

```
n=    100 iterations=10
n=   1000 iterations=32
n=  10000 iterations=100
n= 100000 iterations=317
n=1000000 iterations=1000
```

$\sqrt{10^4} = 100$ and $\sqrt{10^6} = 1000$: the measured counts match exactly.

26.7.7 Example 7 — multiplying the counter

```
for (int i = 1; i < n; i = i*2) { total = 5*6; }
```

i takes the values $1, 2, 4, 8, \dots, 2^k$. The loop stops when $2^k \geq n$, i.e. $k \geq \log_2 n$. $O(\log n)$.

```
n=      8 iterations=3
n=     64 iterations=6
n=    512 iterations=9
n=   4096 iterations=12
n=  32768 iterations=15
n= 262144 iterations=18
```

Multiplying n by 8 adds exactly 3 iterations each time — because $\log_2 8 = 3$. That flat additive growth *is* logarithmic behaviour.

26.7.8 Example 8 — the triangular-number loop

```
int j = 0;
for (int i = 1; j <= n; i++) { j = j + i; }
```

Note the loop condition tests j , not i . After k iterations,

$$j = 1 + 2 + \dots + k = \frac{k(k+1)}{2} \approx \frac{k^2}{2}.$$

The loop stops when $j > n$, i.e. when $k^2/2 \approx n$, giving $k \approx \sqrt{2n}$. $O(\sqrt{n})$.

```
n=    100 iterations=14
n=   1000 iterations=45
n=  10000 iterations=141
n= 100000 iterations=447
```

```
n=1000000 iterations=1414
```

$\sqrt{2 \times 10^6} = 1414.2$ — the measured 1414 confirms the derivation, including the constant $\sqrt{2}$ that Big-O then discards.

26.7.9 Example 9 — linear outer, logarithmic inner

```
for (int i = 0; i < n; i++)
    for (int j = 1; j < n; j = j*2)
        code;
```

The outer loop is $O(n)$ (Example 2); the inner is $O(\log n)$ (Example 7). Nested loops multiply: $O(n \log n)$.

This is the complexity of merge sort and of the best comparison sorts, and it is why they are considered efficient — $O(n \log n)$ is very close to linear for realistic n .

26.7.10 Example 10 — halving the counter

```
for (int i = n; i > 1; i = i/2) { code; }
```

i takes the values $n, n/2, n/4, \dots$, stopping when it reaches 1 after $\log_2 n$ halvings. $O(\log n)$ — the same as Example 7, run in the opposite direction. This is exactly the structure of binary search.

```
n=      8 iterations=3
n=     64 iterations=6
n=    512 iterations=9
n=   4096 iterations=12
n=  32768 iterations=15
n= 262144 iterations=18
```

26.7.11 Summary of the ten

#	Complexity	Reason
1	$O(1)$	no dependence on input
2a	$O(1)$	loop bound is the constant 100
2b	$O(n)$	loop bound comes from the input
3	$O(n^2)$	two nested input-bounded loops
4	$O(n^3)$	three nested input-bounded loops
5	$O(n)$	break makes the inner loop constant
6	$O(\sqrt{n})$	condition $i^2 < n$
7	$O(\log n)$	counter multiplied by 2
8	$O(\sqrt{n})$	counter accumulates triangular numbers
9	$O(n \log n)$	linear outer $\times$ logarithmic inner
10	$O(\log n)$	counter halved

✓ How to analyse a loop on sight

Ask what the loop *counter* does. Adds a constant, bound $n \Rightarrow O(n)$. Multiplied or divided by a constant $\Rightarrow O(\log n)$. Squared in the condition $\Rightarrow O(\sqrt{n})$. Nested loops multiply; sequential

loops take the maximum.

26.8 The common complexity classes

Class	Name	Typical example
$O(1)$	constant	array indexing, stack push/pop, hash lookup
$O(\log n)$	logarithmic	binary search, balanced BST operations
$O(n)$	linear	linear search, traversing a list, finding a max
$O(n \log n)$	linearithmic	merge sort, heapsort, average quicksort
$O(n^2)$	quadratic	bubble/selection/insertion sort, nested loops
$O(n^3)$	cubic	naïve matrix multiplication
$O(2^n)$	exponential	naïve Fibonacci, subset enumeration
$O(n!)$	factorial	brute-force travelling salesman

26.8.1 What the classes mean in practice

Operations required, at one billion operations per second:

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	3	10	33	100	1,024
100	7	100	664	10,000	1.3×10^{30}
1,000	10	1,000	9,966	10^6	—
10^6	20	10^6	2×10^7	10^{12}	—

time at 10^9 ops/s for $n = 10^6$:

20 ns	1 ms	20 ms	17 minutes	beyond the age of the universe
-------	------	-------	-------------------	--------------------------------

Key idea

The gap between $O(n \log n)$ and $O(n^2)$ at $n = 10^6$ is the gap between 20 milliseconds and 17 minutes. That is why sorting algorithms matter, and why no hardware upgrade substitutes for choosing the right one.

26.9 Space complexity

The same analysis applies to memory. Count the *extra* storage an algorithm needs beyond its input.

$O(1)$	bubble, selection, insertion sort — a few variables only
$O(\log n)$	quicksort — the recursion stack
$O(n)$	merge sort — the temporary array; also any recursion n deep

There is frequently a time–space trade-off: a lookup table turns computation into memory, memoisation turns exponential Fibonacci into linear at the cost of an array.

26.10 Amortised analysis

Occasionally an operation is usually cheap and rarely expensive. Averaging the cost over a sequence of operations is **amortised analysis**.

The growable array of Section 13.7 is the standard example. Appending is $O(1)$ — except when the array is full, when it costs $O(n)$ to reallocate and copy. Is appending therefore $O(n)$?

No. Doubling the capacity means that after n appends the copying work totals

$$1 + 2 + 4 + \dots + n < 2n,$$

so the total cost of n appends is $O(n)$ and the *amortised* cost per append is $O(1)$.

✗ Common pitfall

The result depends on *doubling*. If the array grew by a fixed amount — say 10 elements at a time — there would be $n/10$ reallocations each copying $O(n)$ elements, giving $O(n^2)$ total and $O(n)$ amortised per append. Growth strategy is not a detail.

26.11 Complexity of the structures in this course

Structure	Access	Search	Insert	Delete
Array (unsorted)	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Array (sorted)	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$
Stack (array or list)	$O(1)^*$	—	$O(1)$	$O(1)$
Queue (circular)	$O(1)^*$	—	$O(1)$	$O(1)$
Singly linked list	$O(n)$	$O(n)$	$O(1)^+$	$O(1)^+$
Doubly linked list	$O(n)$	$O(n)$	$O(1)^+$	$O(1)^+$
Binary search tree (balanced)	—	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary search tree (degenerate)	—	$O(n)$	$O(n)$	$O(n)$
Hash table (average)	—	$O(1)$	$O(1)$	$O(1)$

* ends only. ⁺ given a pointer to the position; DeleteFromTail on a singly linked list is $O(n)$.

26.12 Chapter summary

- Measured time is hardware, compiler, language and input dependent. Counting operations as a function of input size is machine independent.
- Order of growth describes how cost rises with input. Only the fastest-growing term matters; constants and lower-order terms are discarded.
- $f(n)$ is $O(g(n))$ if $\exists c, N$ such that $f(n) \leq c g(n)$ for all $n \geq N$. Big-O is an upper bound; Ω is a lower bound; Θ is both.
- A polynomial's Big-O is its highest power, coefficient discarded.
- Loop analysis: additive counter $\Rightarrow O(n)$; multiplicative or halving counter $\Rightarrow O(\log n)$; $i^2 < n \Rightarrow O(\sqrt{n})$. Nesting multiplies; sequence takes the maximum; a break can collapse a loop to constant.
- $O(n \log n)$ versus $O(n^2)$ at $n = 10^6$ is 20 milliseconds versus 17 minutes.
- Amortised analysis averages cost over a sequence: doubling gives $O(1)$ amortised append, fixed-increment growth does not.

PART VI

Appendices

Worked Exercise Solutions

This appendix collects the exercise questions whose answers are most useful to have written out — principally the “dry run and predict the output” questions, which are the model for the MCQ paper. Solutions that appear in the body of the text are cross-referenced rather than repeated.

A.1 Exercise I — Introduction

Question	Where solved
1. Valid variable names	Section 1.10, worked box
2. sizeof of the primitive types	Section 2.2.1
3. Intermediate compiler outputs	Section 1.8, all four stages
3(d) How comments are handled	Section 1.8.1 — removed by the preprocessor
3(e) How printf appears in the .s file	Section 1.8.2 — replaced by puts by the optimiser
4. <limits.h> and <float.h>	Section 2.2.2
5. All the printf formatting puzzles	Section 3.3, four worked boxes

A.2 Exercise II — Types, operators and expressions

Questions 1, 2 and 3 are traced in full in Sections 4.8, 4.9 and 4.10.

A.2.1 Question 4 — true or false

	Verdict	Reason
(a) <code>printf</code> always begins printing at a newline	FALSE	<code>printf</code> writes exactly what the format string says. Output begins at the current cursor position; a newline appears only if you write <code>\n</code> .
(b) All variables must be defined before use	TRUE	C requires a declaration to be in scope before a name is used.
(c) It is not mandatory to have a data type for a variable	FALSE	Every variable has a type. Implicit <code>int</code> was removed in C99.
(d) <code>%</code> can only be used with integer operands	TRUE	<code>7.5 % 2</code> is a compile error; use <code>fmod()</code> for floating point.
(e) <code>*</code> , <code>+</code> , <code>-</code> , <code>/</code> have the same precedence	FALSE	<code>*</code> and <code>/</code> (level 3) bind tighter than <code>+</code> and <code>-</code> (level 4).
(f) All operators have left-to-right associativity	FALSE	Unary operators, the ternary operator and all assignment operators associate right to left.
(g) Five <code>printf</code> statements are needed for five lines	FALSE	One <code>printf</code> with four or five <code>\n</code> characters suffices.
(h) “Integer division” names division of integer variables	TRUE	When both operands are integers the fractional part is discarded.
(i) <code>ThisIsAVariable</code> and <code>thisisavariabLe</code> are the same	FALSE	C identifiers are case sensitive.
(j) <code><stdio.h></code> must be included to use <code>printf</code>	TRUE	Without the prototype the call is a constraint violation in C99 and later.

A.2.2 Question 5 — max and min using only operators

No `if` statements are permitted, so the comparison result — which is 1 or 0 — is used as a multiplier.

ex2q5.c

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 17, b = 42, c = 8;
6
7      /* (a) assign 1 to the maximum-valued variable */
8      int max_a = (a >= b) && (a >= c);
9      int max_b = (b > a) && (b >= c);
10     int max_c = (c > a) && (c > b);
11
12     /* (b) assign 1 to the minimum-valued variable */
13     int min_a = (a <= b) && (a <= c);
14     int min_b = (b < a) && (b <= c);
15     int min_c = (c < a) && (c < b);
16
17     printf("max flags: a=%d b=%d c=%d\n", max_a, max_b, max_c);
18     printf("min flags: a=%d b=%d c=%d\n", min_a, min_b, min_c);
19
20     /* the maximum and minimum VALUES, still without if */
21     printf("max=%d min=%d\n", a*max_a + b*max_b + c*max_c,
22           a*min_a + b*min_b + c*min_c);
23     return 0;
24 }
```

```
max flags: a=0 b=1 c=0
min flags: a=0 b=0 c=1
max=42 min=8
```

The strict inequalities on the second and third tests are what break ties, so that exactly one flag is set even when values are equal.

A.2.3 Question 6(a) — Celsius to Fahrenheit

```
double celsius = 37.0;
double fahrenheit = 9.0 / 5.0 * celsius + 32.0;    /* 98.6 */
```

✖ Common pitfall

Writing `9 / 5 * celsius + 32` gives $1 \times 37 + 32 = 69$, because `9 / 5` is integer division yielding 1. Write the constants as `9.0` and `5.0`.

A.2.4 Question 6(b) — target heart rate

```
heartrate.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int age = 20;
6     int max_rate = 220 - age;
7     double lower = 0.50 * max_rate;
8     double upper = 0.85 * max_rate;
9
10    printf("Age           : %d\n", age);
11    printf("Maximum heart rate : %d bpm\n", max_rate);
12    printf("Target range (50-85%): %.1f - %.1f bpm\n", lower, upper);
13    return 0;
14 }
```

```
Age           : 20
Maximum heart rate : 200 bpm
Target range (50-85%): 100.0 - 170.0 bpm
```

A.2.5 Question 6(g) — income tax without conditionals

The constraint is that no flow control or conditional statements may be used. The technique is the same as question 5: a comparison yields 1 or 0, and multiplying by it switches a term on or off. Combined with clamping each bracket to its width, the whole progressive calculation becomes one arithmetic expression per bracket.

```
tax.c
1 #include <stdio.h>
2
3 /* amount of 'income' that falls inside the band [lo, hi) */
4 #define BAND(income, lo, hi) \
5     ( ((income) > (lo)) * ( ((income) < (hi)) * ((income) - (lo)) \
6       + ((income) >= (hi)) * ((hi) - (lo)) ) )
7
8 int main(void)
```

```

9 {
10     double salary = 275000.0;    /* monthly salary income */
11     double extra  = 180000.0;    /* monthly additional income */
12
13     /* salary: first 100k free, then 6% more per 50k block, max 36% */
14     double stax = BAND(salary, 100000, 150000) * 0.06
15             + BAND(salary, 150000, 200000) * 0.12
16             + BAND(salary, 200000, 250000) * 0.18
17             + BAND(salary, 250000, 300000) * 0.24
18             + BAND(salary, 300000, 350000) * 0.30
19             + (salary > 350000) * (salary - 350000) * 0.36;
20
21     /* additional: first 100k at 10%, then +10% per 50k block, max 40% */
22     double etax = BAND(extra, 0, 100000) * 0.10
23             + BAND(extra, 100000, 150000) * 0.20
24             + BAND(extra, 150000, 200000) * 0.30
25             + (extra > 200000) * (extra - 200000) * 0.40;
26
27     printf("Monthly salary tax      : %12.2f\n", stax);
28     printf("Monthly additional income tax: %12.2f\n", etax);
29     printf("Monthly take-home salary   : %12.2f\n", salary - stax);
30     printf("Monthly take-home total    : %12.2f\n",
31            (salary - stax) + (extra - etax));
32     printf("Total annual income       : %12.2f\n", (salary + extra) * 12);
33     printf("Annual salary tax         : %12.2f\n", stax * 12);
34     printf("Annual additional income tax : %12.2f\n", etax * 12);
35     printf("Total annual income tax    : %12.2f\n", (stax + etax) * 12);
36     return 0;
37 }

```

Monthly salary tax	:	24000.00
Monthly additional income tax:	:	29000.00
Monthly take-home salary	:	251000.00
Monthly take-home total	:	402000.00
Total annual income	:	5460000.00
Annual salary tax	:	288000.00
Annual additional income tax	:	348000.00
Total annual income tax	:	636000.00

Checking the salary tax by hand: of 275,000, the first 100,000 is free; three full 50,000 bands are taxed at 6%, 12% and 18% (3000 + 6000 + 9000); and the remaining 25,000 falls in the 24% band (6000). Total 24,000.

A.2.6 Question 8 — self-study answers

(a) and (b) puts

Covered in Section 3.5. `puts(s)` writes the string and appends a newline; it does no format processing. Both are declared in `<stdio.h>`, both write to `stdout` and both return an `int`. `puts` is safer for fixed messages because a `%` in the text is harmless.

(c) Constants in `<math.h>`

The C standard itself defines only `HUGE_VAL`, `HUGE_VALF`, `HUGE_VALL`, `INFINITY`, `NAN` and the classification macros. The familiar mathematical constants — `M_PI`, `M_E`, `M_SQRT2`, `M_LN2`, `M_LOG2E` and the rest — are a POSIX extension, exposed by `glibc` but hidden under `-std=c18` unless a feature-test macro is defined. Define your own `PI` if portability matters.

A.3 Exercise IV — pointers, functions and arrays

Question	Where solved
1(a) Formatted input of five types	Section 3.7.3
1(b) Validating integer input	Section 3.7.1
2(a) Adding via pointers	Section 7.6
2(b) Size of a pointer	Section 7.4
2(c) Incrementing three pointer types	Section 7.5, with explanation
3(a) Power	Section 6.6.1
3(b) Odd or even	Section 6.6.2
3(c) GCD three ways	Section 6.6.3
3(g) Opposite case	Section 9.6.3
4(b) Binary to decimal	Section 8.9.4
4(c) Palindrome	Section 8.9.3
4(d) Array via pointers	Section 8.5
4(e) Average, min, max, standard deviation	Section 8.6
4(f), 4(g) Dry-run traces	Sections 8.9.1 and 8.9.2

A.3.1 Question 3(h) — point on a circle

```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define PI 3.14159265358979323846
5
6  int on_circle(double x, double y, double ox, double oy, double r)
7  {
8      double dx = x - ox, dy = y - oy;
9      return fabs(dx*dx + dy*dy - r*r) < 1e-9;    /* tolerance, not == */
10 }
11
12 int main(void)
13 {
14     double ox = 0, oy = 0, r = 5;
15
16     printf("(3,4)   on circle? %s\n", on_circle(3, 4, ox, oy, r) ? "True" : "False");
17     printf("(5,0)   on circle? %s\n", on_circle(5, 0, ox, oy, r) ? "True" : "False");
18     printf("(1,1)   on circle? %s\n", on_circle(1, 1, ox, oy, r) ? "True" : "False");
19
20     printf("circumference = %.4f\n", 2 * PI * r);
21     printf("area          = %.4f\n", PI * r * r);
22     printf("sphere volume = %.4f\n", 4.0 / 3.0 * PI * r * r * r);
23     return 0;
24 }

```

```

(3,4)   on circle? True
(5,0)   on circle? True
(1,1)   on circle? False
circumference = 31.4159
area          = 78.5398
sphere volume = 523.5988

```

The tolerance test is essential: $dx*dx + dy*dy == r*r$ fails for values that are mathematically equal but not exactly representable (Section 2.5).

A.3.2 Question 3(i) and 3(j) — angles and intersections

lines.c

```

1  #include <stdio.h>
2  #include <math.h>
3
4  #define PI 3.14159265358979323846
5
6  double angle_with_x_axis(double x1, double y1, double x2, double y2)
7  {
8      return atan2(y2 - y1, x2 - x1);    /* atan2 gets the quadrant right */
9  }
10
11 /* returns 1 and sets (*px,*py) if lines AB and CD intersect */
12 int intersect(double x1,double y1,double x2,double y2,
13              double x3,double y3,double x4,double y4,
14              double *px, double *py)
15 {
16     double d = (x1-x2)*(y3-y4) - (y1-y2)*(x3-x4);
17     if (fabs(d) < 1e-12) return 0;      /* parallel or coincident */
18     double t = ((x1-x3)*(y3-y4) - (y1-y3)*(x3-x4)) / d;
19     *px = x1 + t*(x2-x1);
20     *py = y1 + t*(y2-y1);
21     return 1;
22 }
23
24 int main(void)
25 {
26     double a = angle_with_x_axis(0, 0, 1, 1);
27     printf("angle of (0,0)-(1,1): %.6f rad = %.2f deg\n", a, a * 180.0 / PI);
28
29     double px, py;
30     if (intersect(0,0, 4,4, 0,4, 4,0, &px, &py))
31         printf("lines intersect at (%.2f, %.2f)\n", px, py);
32     else
33         printf("the lines do not intersect\n");
34
35     if (!intersect(0,0, 4,4, 1,1, 5,5, &px, &py))
36         printf("collinear/parallel lines: no unique intersection point\n");
37     return 0;
38 }

```

```

angle of (0,0)-(1,1): 0.785398 rad = 45.00 deg
lines intersect at (2.00, 2.00)
collinear/parallel lines: no unique intersection point

```

`atan2(dy, dx)` is used rather than `atan(dy/dx)` because it returns the correct quadrant and does not divide by zero on a vertical line. Compile with `-lm`.

A.4 Exercise VI — the memory simulator

Question 2 asks for a simulation of `malloc` and `free` over a 200-element integer array treated as memory. This is an excellent exercise because it makes the allocator's job concrete.

memsim.c

```

1  #include <stdio.h>
2  #include <string.h>
3
4  #define MEMSIZE 200

```

```

5
6 static int memory[MEMSIZE];      /* the simulated memory */
7 static int used[MEMSIZE];        /* 0 = free, else block id */
8 static int next_id = 1;
9
10 /* find the first run of 'blocks' free cells (first-fit) */
11 void *NewMalloc(int dataType)
12 {
13     int blocks = dataType;      /* size in "cells" */
14     int run = 0;
15     for (int i = 0; i < MEMSIZE; i++) {
16         run = used[i] ? 0 : run + 1;
17         if (run == blocks) {
18             int start = i - blocks + 1;
19             for (int k = start; k <= i; k++) used[k] = next_id;
20             next_id++;
21             return &memory[start];
22         }
23     }
24     return NULL;                /* memory full */
25 }
26
27 void NewFree(void *ptr)
28 {
29     if (!ptr) return;
30     int start = (int)((int *) ptr - memory);
31     int id = used[start];
32     if (id == 0) return;        /* not an allocated block */
33     for (int i = 0; i < MEMSIZE; i++) if (used[i] == id) used[i] = 0;
34 }
35
36 /* repack all allocations so that they are contiguous from cell 0 */
37 void Defragment(void)
38 {
39     int write = 0;
40     for (int read = 0; read < MEMSIZE; read++) {
41         if (used[read]) {
42             memory[write] = memory[read];
43             used[write] = used[read];
44             if (write != read) used[read] = 0;
45             write++;
46         }
47     }
48     while (write < MEMSIZE) used[write++] = 0;
49 }
50
51 static void show(const char *label)
52 {
53     printf("%-14s", label);
54     for (int i = 0; i < 40; i++) printf("%c", used[i] ? '#' : '.');
55     int free_cells = 0;
56     for (int i = 0; i < MEMSIZE; i++) if (!used[i]) free_cells++;
57     printf(" free=%d\n", free_cells);
58 }
59
60 int main(void)
61 {
62     memset(used, 0, sizeof used);
63
64     void *a = NewMalloc(4);      /* an int */
65     void *b = NewMalloc(8);      /* a double */
66     void *c = NewMalloc(1);      /* a char */

```

```

67     void *d = NewMalloc(4);
68     show("4,8,1,4:");
69
70     NewFree(b);   show("free b:");
71     NewFree(c);   show("free c:");
72
73     Defragment(); show("defragmented:");
74
75     (void) a; (void) d;
76     return 0;
77 }

```

```

4,8,1,4:      #####.....      free=183
free b:       #####.....      free=191
free c:       #####.....      free=192
defragmented: #####.....      free=192

```

The middle two lines show **external fragmentation**: 191 cells are free but they are not contiguous, so a request for 12 consecutive cells could fail even though there is plenty of space. Defragmentation repacks the live blocks and restores one large free region — which is exactly what a real garbage collector with a compacting phase does, and exactly why C's `malloc`, which cannot move blocks (pointers would break), instead uses free lists and size classes.

A.5 Exercise VII — stacks and queues

Question	Where solved
1(a) Stack of integers, separate files	Section 21.2
1(b) Addition using three stacks	Section 21.4
1(c) Stack of characters	Section 21.3, CStack
1(d) Bracket and delimiter matcher	Section 21.3
2(a)(i) Priority queue with three queues	Section 22.4.1
2(a)(ii) Priority queue with one queue	Section 22.4.2

Linux Command Reference

The course requires a Linux environment, `vim` and the command line — strictly no IDEs. This appendix covers the commands from Lab Sheets 1 and 2 and answers the dialogue exercises.

B.1 Setting up the environment

Lab Sheet 1 walks through installing VirtualBox, creating an Ubuntu virtual machine, and installing the tools. The essential commands once Ubuntu is running:

```
$ sudo apt update
$ sudo apt remove --assume-yes vim-tiny      # remove the cut-down vim
$ sudo apt-get -y install vim                # install full vim
$ vi --version                               # check
$ sudo apt-get -y install build-essential    # gcc, make, headers
$ gcc --version
$ sudo apt install valgrind gdb
```

Open a terminal with `Ctrl+Alt+T`, or right-click a folder and choose *Open in Terminal*.

Note

Linux is case sensitive: `Desktop` and `desktop` are different directories. Use the **Up arrow** to recall previous commands and **Tab** to complete file names — both save a great deal of typing and prevent typos.

B.2 Navigation and files

<code>pwd</code>	print working directory — where am I?
<code>ls</code>	list the contents of the current directory
<code>ls -l</code>	long listing: permissions, owner, size, date
<code>ls -a</code>	include hidden files (those starting with <code>.</code>)
<code>ls -la</code>	both
<code>cd <i>dir</i></code>	change directory
<code>cd ~</code>	go to the home directory
<code>cd ..</code>	go to the parent directory
<code>cd -</code>	go back to the previous directory
<code>mkdir <i>name</i></code>	make a directory
<code>mkdir -p a/b/c</code>	make nested directories
<code>rmdir <i>name</i></code>	remove an <i>empty</i> directory
<code>rm <i>file</i></code>	remove a file
<code>rm -r <i>dir</i></code>	remove a directory and its contents
<code>touch <i>file</i></code>	create an empty file, or update its timestamp
<code>cp <i>src dst</i></code>	copy
<code>cp -r <i>src dst</i></code>	copy a directory
<code>mv <i>src dst</i></code>	move or rename
<code>cat <i>file</i></code>	display a file's contents
<code>less <i>file</i></code>	page through a file (q to quit)
<code>head -n 20 <i>file</i></code>	first 20 lines
<code>tail -n 20 <i>file</i></code>	last 20 lines
<code>find . -name "<i>pattern</i>"</code>	search for files by name

✖ Common pitfall

`rm` is permanent — there is no recycle bin. `rm -rf` on the wrong directory is unrecoverable. Check with `ls` before deleting, and never run `rm -rf` with a variable you have not verified.

B.3 Text processing and redirection

<code>echo "text"</code>	print text
<code>echo "text" > f</code>	write to a file, overwriting it
<code>echo "text" >> f</code>	append to a file
<code>cmd1 cmd2</code>	pipe: cmd1's output becomes cmd2's input
<code>grep pat file</code>	print lines containing the pattern
<code>grep -i pat file</code>	case-insensitive
<code>grep -v pat file</code>	lines <i>not</i> containing the pattern
<code>grep -c pat file</code>	count matching lines
<code>grep -n pat file</code>	show line numbers
<code>wc file</code>	count lines, words, characters
<code>wc -l file</code>	count lines only
<code>sort file</code>	sort lines alphabetically
<code>sort -r file</code>	reverse order
<code>sort -n file</code>	numeric sort
<code>uniq file</code>	remove <i>adjacent</i> duplicate lines
<code>sort f uniq</code>	remove all duplicates
<code>diff f1 f2</code>	show differences between two files

B.4 Permissions and system information

<code>chmod -w file</code>	remove write permission (make read-only)
<code>chmod +x file</code>	add execute permission
<code>chmod 644 file</code>	rw for owner, r for group and others
<code>chmod 755 file</code>	rxw for owner, rx for group and others
<code>whoami</code>	the logged-in username
<code>date</code>	system date and time
<code>cal</code>	this month's calendar
<code>history</code>	previously used commands
<code>clear</code>	clear the terminal (or Ctrl+L)
<code>man cmd</code>	the manual page — the first place to look
<code>which cmd</code>	where a command lives

B.5 Lab Sheet 2 — Part B dialogue answers

#	Command	Expected output
1	<code>pwd</code>	the absolute path of the current directory, e.g. <code>/home/ravmax</code>
2	<code>ls</code>	the names of the files and directories here
3	<code>cd Desktop</code>	no output; the prompt changes
4	<code>mkdir project_work</code>	no output; verify with <code>ls</code>
5	<code>cd project_work</code>	no output
6	<code>touch notes.txt</code>	no output; <code>ls</code> shows the file
7	<code>echo "Linux is powerful" > notes.txt</code>	no output
8	<code>cat notes.txt</code>	Linux is powerful
9	<code>cp notes.txt notes_copy.txt</code>	no output
10	<code>rm notes_copy.txt</code>	no output

Note

On Unix, silence means success. Most commands print nothing when they work and print a message only when they fail. Verify with `ls` or `cat`.

B.6 Lab Sheet 2 — Part C dialogue answers

#	Command	Explanation
1	<code>ls -a</code>	<code>-a</code> includes entries beginning with <code>.</code>
2	<code>mkdir -p ~/Desktop/project && cd ~/Desktop/project</code>	<code>&&</code> runs the second command only if the first succeeded
3	<code>touch a.txt b.txt c.txt</code>	<code>touch</code> accepts several names at once
4	<code>ls -l</code>	long listing with permissions, size and date
5	<code>grep -i apple file</code>	<code>-i</code> ignores case
6	<code>wc -l file</code>	count lines
7	<code>chmod -w file</code>	remove write permission
8	<code>chmod +x file</code>	add execute permission
9	<code>grep -c banana file</code>	count matching lines; equivalently <code>grep banana file wc -l</code>
10	<code>sort file uniq</code>	<code>uniq</code> only removes <i>adjacent</i> duplicates, so it must be sorted first
11	<code>grep -v apple file</code>	<code>-v</code> inverts the match
12	<code>sort -r file</code>	reverse alphabetical order

Why sort before uniq

`uniq` compares only *consecutive* lines, so `apple`, `banana`, `apple` keeps both `apple`s. Sorting first brings duplicates together. `sort -u` does both in one step, and `uniq -c` prefixes each line with its count — the standard idiom for a frequency tally:

```
$ sort words.txt | uniq -c | sort -rn | head
```

B.7 The C workflow on the command line

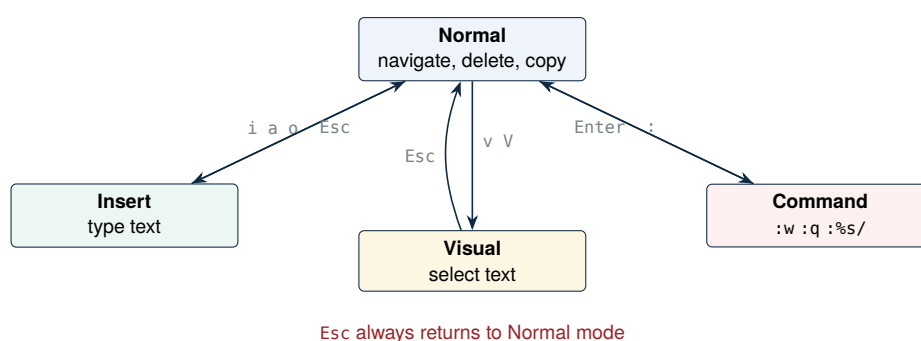
```
$ mkdir -p ~/Desktop/Day_1 && cd ~/Desktop/Day_1
$ vim main.c                                # write the program
$ gcc -Wall -Wextra -std=c18 -g -O0 -o main main.c
$ ./main                                    # run it
$ echo $?                                  # check the exit status
$ valgrind --leak-check=full ./main         # check for leaks
$ gdb ./main                               # debug if needed
```

Compiling with no `-o` produces `a.out`, which is run with `./a.out`. The `./` is required — see Section 1.7.1.

The vim Editor

The course requires vim and forbids IDEs. vim is modal: what a key does depends on which mode you are in, which is the source of every beginner's confusion and, eventually, of its speed.

C.1 Modes



Key idea

If you are ever lost, press Esc twice. You are then in Normal mode, from which everything else is reachable.

C.2 Survival commands

vim file.c	open (or create) a file
i	insert before the cursor
Esc	return to Normal mode
:w	write (save)
:q	quit
:wq or ZZ	write and quit
:q!	quit without saving — discard changes
:x	write only if changed, then quit

C.3 Movement (Normal mode)

h j k l	left, down, up, right	0	start of line
w	next word	^	first non-blank of line
b	previous word	\$	end of line
e	end of word	gg	first line of file
{ }	previous/next paragraph	G	last line of file
%	jump to the matching bracket	25G	go to line 25
Ctrl+f	page down	:25	go to line 25
Ctrl+b	page up	``	back to the previous position

% is worth learning immediately: put the cursor on a brace and press it to jump to its partner. It is the fastest way to find an unbalanced brace.

C.4 Editing

i	insert before cursor	x	delete character
a	append after cursor	dd	delete (cut) line
I	insert at line start	5dd	delete 5 lines
A	append at line end	dw	delete word
o	open a line below	D	delete to end of line
O	open a line above	yy	yank (copy) line
r	replace one character	p	paste after
cw	change word	P	paste before
u	undo	Ctrl+r	redo
.	repeat the last change	J	join with the next line

The grammar

vim commands compose as *operator* + *motion*: d (delete) + w (word) = dw. c (change) + \$ (to end of line) = c\$. y (yank) + } (paragraph) = y}. A count multiplies: 3dw deletes three words. Learning the operators and motions separately gives you their product, not their sum.

C.5 Search and replace

/pattern	search forwards; n for next, N for previous
?pattern	search backwards
*	search for the word under the cursor
:%s/old/new/g	replace every occurrence in the file
:%s/old/new/gc	the same, but confirm each one
:s/old/new/g	replace on the current line only
:noh	clear the search highlighting

C.6 Configuration for C

Put this in ~/.vimrc:

```

syntax on          " syntax highlighting
set number         " line numbers -- match them to compiler errors
set tabstop=4      " a tab displays as 4 columns
set shiftwidth=4   " indent by 4
set expandtab       " insert spaces instead of tab characters
set autoindent     " keep the previous line's indent
set smartindent    " add indent after {
set showmatch      " briefly jump to the matching bracket
set hlsearch       " highlight search results
set incsearch      " search as you type
set ruler          " show line,column in the corner
set wildmenu       " tab completion for : commands
filetype plugin indent on

```

✖ Makefiles need real tabs

set expandtab inserts spaces, which breaks Makefile recipes (Section 18.4). Add an exception:

```
autocmd FileType make setlocal noexpandtab
```

:set list displays tabs as ^I so you can see which you have.

C.7 Useful for editing C

gg=G	re-indent the whole file
%	jump between matching (), [], {}
Ctrl+n	complete a word from elsewhere in the file (Insert mode)
:sp file/:vsp file	split the window horizontally / vertically
Ctrl+w then hjkl	move between split windows
:make	run make without leaving vim
:!gcc -Wall %	compile the current file (% is its name)
:!./a.out	run the result
:e!	reload the file, discarding changes

📌 Key idea

The whole edit–compile–run cycle can stay inside vim: :w to save, then :!gcc -Wall -Wextra -std=c18 -o %< % && ./%< to build and run. % is the current file name and %< is that name without its extension.

C.8 Learning more

Run vimtutor from the shell. It is a 30-minute interactive tutorial that ships with vim and is the single most efficient way to become productive in it.

Reference Tables

D.1 ASCII table (printable characters)

Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch	Dec	Hex	Ch
32	20	space	56	38	8	80	50	P	104	68	h
33	21	!	57	39	9	81	51	Q	105	69	i
34	22	"	58	3A	:	82	52	R	106	6A	j
35	23	#	59	3B	;	83	53	S	107	6B	k
36	24	\$	60	3C	<	84	54	T	108	6C	l
37	25	%	61	3D	=	85	55	U	109	6D	m
38	26	&	62	3E	>	86	56	V	110	6E	n
39	27	'	63	3F	?	87	57	W	111	6F	o
40	28	(	64	40	@	88	58	X	112	70	p
41	29	)	65	41	A	89	59	Y	113	71	q
42	2A	*	66	42	B	90	5A	Z	114	72	r
43	2B	+	67	43	C	91	5B	[	115	73	s
44	2C	,	68	44	D	92	5C	\	116	74	t
45	2D	-	69	45	E	93	5D	]	117	75	u
46	2E	.	70	46	F	94	5E	^	118	76	v
47	2F	/	71	47	G	95	5F	_	119	77	w
48	30	0	72	48	H	96	60	`	120	78	x
49	31	1	73	49	I	97	61	a	121	79	y
50	32	2	74	4A	J	98	62	b	122	7A	z
51	33	3	75	4B	K	99	63	c	123	7B	{
52	34	4	76	4C	L	100	64	d	124	7C	
53	35	5	77	4D	M	101	65	e	125	7D	}
54	36	6	78	4E	N	102	66	f	126	7E	~
55	37	7	79	4F	O	103	67	g	127	7F	DEL

Control characters worth knowing: 0 NUL ('`\0`'), 7 BEL ('`\a`'), 8 BS ('`\b`'), 9 TAB ('`\t`'), 10 LF ('`\n`'), 13 CR ('`\r`'), 27 ESC.

Three relationships are worth memorising rather than looking up: '`0`' = 48, '`A`' = 65, '`a`' = 97 = '`A`' + 32.

D.2 printf conversion specifiers

Spec	Argument	Output
%d %i	int	signed decimal
%u	unsigned	unsigned decimal
%o	unsigned	octal
%x %X	unsigned	hexadecimal, lower / upper
%f	double	125.456735 (6 decimals default)
%e %E	double	1.254567e+02
%g %G	double	shorter of %e/%f
%a %A	double	hex float, exact
%c	int	one character
%s	char *	string up to '\0'
%p	void *	pointer value
%zu	size_t	the result of sizeof
%ld %lld	long, long long	
%%	—	a literal %

Flags - left justify, + always sign, *space* sign placeholder, # alternate form (0x, 0, forced point), 0 zero pad.

Width minimum characters; never truncates numbers. **Precision** .n: decimals for %f, minimum digits for %d, maximum characters for %s.

D.3 Escape sequences

\n	newline (10)	\\	backslash
\t	horizontal tab (9)	\'	single quote
\r	carriage return (13)	\"	double quote
\a	alert / bell (7)	\0	null character (0)
\b	backspace (8)	\101	octal escape
\v	vertical tab (11)	\x41	hex escape

D.4 Operator precedence (highest first)

Level	Operators	Associativity
1	() [] . -> ++ -- (postfix)	L → R
2	++ -- (prefix) + - ! ~ * & sizeof (cast)	R → L
3	* / %	L → R
4	+ -	L → R
5	<< >>	L → R
6	< <= > >=	L → R
7	== !=	L → R
8	&	L → R
9	^	L → R
10		L → R
11	&&	L → R
12		L → R
13	?:	R → L
14	= += -= *= /= %= &= ^= = <<= >>=	R → L
15	,	L → R

D.5 Standard library headers

<stdio.h>	printf scanf fopen fclose fgetc fgets fprintf fseek ftell FILE EOF NULL
<stdlib.h>	malloc calloc realloc free exit atoi strtol rand srand qsort abs
<string.h>	strlen strcpy strncpy strcat strcmp strchr strstr memcpy memset memmove
<math.h>	sqrt pow fabs sin cos tan atan2 exp log ceil floor fmod (<i>link with -lm</i>)
<ctype.h>	isalpha isdigit isalnum isspace toupper tolower
<time.h>	clock time difftime clock_t CLOCKS_PER_SEC
<limits.h>	INT_MIN INT_MAX CHAR_BIT LONG_MAX UINT_MAX
<float.h>	FLT_MAX DBL_MAX DBL_DIG DBL_EPSILON
<stddef.h>	size_t ptrdiff_t NULL offsetof
<stdbool.h>	bool true false (C99)
<stdint.h>	int8_t uint32_t int64_t (C99)
<assert.h>	assert
<errno.h>	errno ERANGE EDOM

D.6 The 32 C keywords (C89) plus later additions

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

C99 and later:

inline	restrict	_Bool	_Complex	_Imaginary	
_Alignas	_Alignof	_Atomic	_Generic	_Noreturn	_Static_assert

D.7 Type sizes on 64-bit Linux (x86-64)

Type	Bytes	Range
char	1	−128...127
unsigned char	1	0...255
short	2	−32,768...32,767
int	4	$\pm 2.1 \times 10^9$
unsigned int	4	0... 4.29×10^9
long	8	$\pm 9.22 \times 10^{18}$
long long	8	$\pm 9.22 \times 10^{18}$
float	4	$\pm 3.4 \times 10^{38}$, 6 digits
double	8	$\pm 1.8 \times 10^{308}$, 15 digits
long double	16	extended precision
any pointer	8	—
size_t	8	unsigned

Cheat Sheets

Condensed revision material. Everything here is developed in full in the body of the book.

E.1 The build and debug cycle

<code>gcc -Wall -Wextra -std=c18 -g -O0 -o prog prog.c</code>	the command to use for everything in this course
<code>... prog.c -lm</code>	add if you use <code><math.h></code> — <i>after</i> the sources
<code>gcc -E f.c</code>	stop after preprocessing → <code>.i</code>
<code>gcc -S f.c</code>	stop after compiling → <code>.s</code> (assembly)
<code>gcc -c f.c</code>	stop after assembling → <code>.o</code> (object)
<code>gcc a.o b.o -o prog</code>	link
<code>./prog</code>	run (the <code>./</code> is required)
<code>echo \$?</code>	the exit status
<code>valgrind --leak-check=full ./prog</code>	check for leaks
<code>gcc -fsanitize=address,undefined ...</code>	catch memory errors at run time
<code>gdb ./prog</code> then run, bt	find where it crashed
<code>size prog nm prog.o ldd prog</code>	inspect the binary

E.2 gdb in ten commands

<code>break f</code>	<code>b</code>	stop on entry to <code>f</code> ; <code>break f if n==0</code> for a condition
<code>run</code>	<code>r</code>	start
<code>next</code>	<code>n</code>	one line, over calls
<code>step</code>	<code>s</code>	one line, into calls
<code>finish</code>		run to the end of this function, print the return value
<code>continue</code>	<code>c</code>	resume
<code>print x</code>	<code>p</code>	any expression: <code>p *head, p a[3], p/x n</code>
<code>backtrace</code>	<code>bt</code>	how did I get here?
<code>info locals</code>		all locals in this frame
<code>watch x</code>		stop when <code>x</code> changes

The crash recipe: `run` → `bt` → print the pointers. `0x0` means null.

E.3 Complexity

Pattern in the code	Reason	Complexity
no loops	fixed work	$O(1)$
for (i=0; i<100; i++)	constant bound	$O(1)$
for (i=0; i<n; i++)	bound from input	$O(n)$
two nested $O(n)$ loops	multiply	$O(n^2)$
three nested	multiply	$O(n^3)$
inner loop with break on first pass	inner is constant	$O(n)$
for (i=0; i*i<n; i++)	runs to $\sqrt{n}$	$O(\sqrt{n})$
for (i=1; i<n; i*=2)	doubling counter	$O(\log n)$
for (i=n; i>1; i/=2)	halving counter	$O(\log n)$
j+=i until j>n	triangular numbers	$O(\sqrt{n})$
$O(n)$ outer $\times$ $O(\log n)$ inner	multiply	$O(n \log n)$
two sequential loops	take the maximum	$O(n)$

Algorithm	Best	Avg	Worst	Space	Note
Linear search	$O(1)$	$O(n)$	$O(n)$	$O(1)$	unsorted OK
Binary search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	needs sorted
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	stable
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	fewest swaps
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	best for small n
Merge sort	$O(n \log n)$ always			$O(n)$	stable
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	fastest in practice

Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$
Stack / Queue	$O(1)$ ends	—	$O(1)$	$O(1)$
Singly linked list	$O(n)$	$O(n)$	$O(1)^{\dagger}$	$O(1)^{\dagger}$
BST (balanced)	—	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hash table (avg)	—	$O(1)$	$O(1)$	$O(1)$

[†] given the position. DeleteFromTail on a singly linked list is $O(n)$.

E.4 Pointers

<code>int *p;</code>	<code>p</code> is a pointer to <code>int</code>
<code>p = &x;</code>	<code>p</code> holds the address of <code>x</code>
<code>*p</code>	the object <code>p</code> points at
<code>&p</code>	the address of the pointer itself (type <code>int **</code>)
<code>p + 1</code>	advances by <code>sizeof(*p)</code> bytes
<code>p - q</code>	number of <i>elements</i> between them
<code>*p++</code>	dereference, then advance the pointer
<code>(*p)++</code>	increment the value
<code>const int *p</code>	data is read-only; the pointer may move
<code>int *const p</code>	the pointer is fixed; the data may change
<code>a[i]</code>	is <i>defined as</i> <code>*(a + i)</code>
<code>p->m</code>	is <code>(*p).m</code>

Kind	Meaning
NULL pointer	points nowhere; the standard sentinel
Void pointer	<code>void *</code> , no type; cannot be dereferenced without a cast
Wild pointer	uninitialised — always initialise, at least to NULL
Dangling pointer	points at freed or out-of-scope memory
Function pointer	<code>int (*f)(double)</code> — holds a code address
Double pointer	<code>int **</code> — needed to modify a caller's pointer

E.5 Memory

Segment	Holds
Text	executable instructions; shareable, read-only
Data	initialised globals and statics
BSS	uninitialised or zero globals and statics; no file space
Heap	<code>malloc/calloc/realloc</code> ; grows <i>up</i>
Stack	locals, parameters, return addresses; grows <i>down</i> ; <code>\$SP</code>

<code>malloc(n)</code>	<code>n</code> bytes, uninitialised
<code>calloc(k, n)</code>	<code>k</code> items of <code>n</code> bytes, zeroed
<code>realloc(p, n)</code>	resize; use a temporary , never <code>p = realloc(p, n)</code>
<code>free(p)</code>	release; then set <code>p = NULL</code>

Faults: leak (never freed) • dangling (freed but still used) • double free • wild (uninitialised) • overrun (past the end) • aliasing (two names, one object).

E.6 Files

Mode	Exists	Missing	Access
r	opens, keeps	error (NULL)	read
w	truncates	creates	write
a	keeps, appends	creates	write (end)
r+	keeps	error	read + write
w+	truncates	creates	read + write
a+	keeps, appends	creates	read + write (end)

Read: `fgetc fgets fscanf` Write: `fputc fputs fprintf` Move: `fseek ftell rewind (SEEK_SET/SEEK_CUR/SEEK_END)`
fgetc returns int, not char — 256 byte values plus EOF do not fit in a byte. **Never** write while `(!feof(fp))`.

E.7 The twenty mistakes that cost marks

1. `if (x = 5)` instead of `==`.
2. Missing `break` in a `switch` — silent fall-through.
3. `for (i = 0; i <= n; i++)` over an array of size `n`.
4. `int / int` where real division was intended.
5. Comparing floats with `==`.
6. Comparing strings with `==` instead of `strcmp`.
7. Assigning strings with `=` instead of `strcpy`.
8. Forgetting `&` in `scanf`.
9. Forgetting the `'\0'` when building a string by hand.
10. Array of size `n` for an `n`-character string (needs `n+1`).
11. `sizeof(arr)/sizeof(arr[0])` inside a function.
12. Missing `-lm` when using `<math.h>`.
13. Using an uninitialised (wild) pointer.
14. Not checking `malloc` or `fopen` for `NULL`.
15. Forgetting `free` — a memory leak.
16. Using a pointer after `free` — dangling.
17. `p = realloc(p, n)` — leaks on failure.
18. Returning the address of a local variable.
19. Not handling the empty and single-element cases in a linked list.
20. Storing `fgetc`'s result in a `char`.

E.8 Definitions to be able to state

Compiler vs interpreter	whole program ahead of time to an executable, versus statement by statement at run time; Java is hybrid via bytecode
The four compilation stages	preprocess (-E), compile (-S), assemble (-c), link
ADT vs data structure	specification (what) versus implementation (how)
Linear vs non-linear	elements have one predecessor and successor, versus many
LIFO vs FIFO	stack (one pointer, top) versus queue (two, first and last)
Call by value vs reference	copies the value versus copies the address; C is always by value
Static vs dynamic allocation	fixed at compile time versus obtained at run time from the heap
Memory leak	allocated on the heap and never freed
Dangling pointer	points to freed or out-of-scope memory
Aliasing	the same location accessed through different names
Big-O	$f(n)$ is $O(g(n))$ if $\exists c, N : f(n) \leq c g(n)$ for all $n \geq N$
Order of growth	how computation time increases with input size
Recursion	a function calling another instance of itself; needs a base case and a recursive step
Row-major order	C stores 2-D arrays row by row; $a[i][j]$ at $i \times \text{cols} + j$
Padding	bytes inserted by the compiler so struct members are aligned

Index

- abstract data type, 175
- actual parameters, 61
- address, 69
- address space layout randomisation, 69
- Aliasing, 126
- amortised analysis, 232
- Arguments, 61
- arithmetic shift, 40
- array, 81
- ASCII, 17
- assembler, 4
- assembly language, 9
- assignment operator, 37
- assignment suppression, 35
- Associativity, 40

- binary heap, 197
- binary mode, 133
- binary search tree, 209
- binary tree, 209
- block, 7, 49
- breakpoint, 156
- buffered, 132
- bytecode, 5

- cast, 22
- clockwise, 79
- compiler, 4
- complexity, 227
- constant, 18
- conversion specification, 27
- core dump, 163

- dangling, 123
- dangling pointer, 127
- data alignment, 101
- data structure, 175
- declaration, 62
- definition, 62
- dereference, 70
- double-ended queue, 197
- Dynamic memory allocation, 121

- entry point, 7
- enumeration, 19
- escape sequence, 31
- exit status, 146
- expression, 37

- external fragmentation, 242

- formal parameters, 61
- function, 59
- function pointer, 78
- function prologue, 9

- generic, 76
- graph, 211
- guard clause, 51

- head, 200
- homogeneous, 81
- Horner's method, 90
- hybrid implementation, 5

- include guard, 110
- indirection, 70
- integer division, 37
- internal linkage, 112
- interpreter, 4, 5
- Iterative statements, 53

- lifetime, 115
- line markers, 9
- linkage, 115
- linked list, 200
- little-endian, 104
- local, 66
- logical address space, 69
- logical shift, 40

- macro, 19, 107
- mantissa, 17
- members, 99
- memory leak, 125
- merge, 142

- node, 199
- null pointer, 74

- object file, 10
- operating system, 4
- operator, 37
- order of growth, 224

- padding, 101
- Parameters, 61
- pattern rule, 171

- [persist](#), [131](#)
- [pivot](#), [218](#)
- [pointer](#), [70](#)
- [pointer array](#), [77](#)
- [Postfix](#), [190](#)
- [Precedence](#), [40](#)
- [Precision](#), [28](#)
- [preprocessor directive](#), [7](#)
- [priority queue](#), [196](#)
- [processor time](#), [24](#)
- [program](#), [5](#)
- [Programming](#), [5](#)
- [programming language](#), [3](#)
- [prototype](#), [62](#)
-
- [queue](#), [193](#)
-
- [Recursion](#), [179](#)
- [relational operator](#), [37](#)
- [root](#), [209](#)
-
- [scanset](#), [34](#), [95](#)
- [scope](#), [49](#), [115](#)
- [segmentation fault](#), [69](#)
- [sequence point](#), [41](#)
- [shareable](#), [118](#)
- [spiral](#), [79](#)
- [stable](#), [219](#)
-
- [stack](#), [185](#)
- [stack frame](#), [9](#), [118](#), [119](#)
- [statement](#), [49](#)
- [stored-program](#), [3](#)
- [streams](#), [27](#)
- [string](#), [91](#)
- [structure](#), [99](#)
-
- [tail](#), [200](#)
- [tail recursive](#), [181](#)
- [top of stack](#), [185](#)
- [tree](#), [209](#)
- [two's complement](#), [16](#)
- [type](#), [13](#)
- [type conversion](#), [21](#)
- [typecasting](#), [21](#)
-
- [undefined behaviour](#), [16](#)
- [union](#), [102](#)
- [usual arithmetic conversions](#), [21](#)
-
- [variable](#), [11](#)
- [void pointer](#), [75](#)
-
- [watchpoint](#), [159](#)
- [Width](#), [28](#)
- [wild pointer](#), [75](#)