Programming in C/C++

Notes

Abhishek Bohare

a_bohare@icloud.com

typeset with LaTeX

Contents

1 C Language **1**

1.1 Overview of C . . . 1

1.1.1 History . . . 1

1.2 Importance of C . . . 1

1.3 How a programme is turned into output . . . 2

1.4 A first C programme . . . 2

1.5 Application development using C . . . 2

1.5.1 Why a standalone application depends on the operating system 3

1.5.2 A web application: the language-choice example . . . 4

1.6 Platform dependence . . . 4

1.7 Programming style in C . . . 4

1.7.1 Compiling for each platform . . . 5

1.8 Variables . . . 5

1.8.1 Rules for variable names . . . 5

1.9 Programmes, functions and the remote controller . . . 6

1.10 Building blocks of a programme . . . 6

1.11 Tokens in C . . . 7

1.11.1 Identifiers . . . 7

1.11.2 Keywords . . . 7

2 Data Types, Operators and Characters **8**

2.1 Data types . . . 8

2.1.1 Integer types (`%d`) . . . 8

2.1.2 Floating-point types (`%f`) . . . 8

2.1.3 Two useful symbols . . . 8

2.2 Operators . . . 8

2.2.1 Types of operator . . . 9

2.3 Classification of data types . . . 9

2.4 Storage size of integer types . . . 9

2.4.1 Format specifiers 10
2.5 What happens when a value does not fit 10
2.6 The `char` type 10
2.7 Characters are stored as numbers 11
2.7.1 ASCII 11
2.7.2 Printing a character and its code 12
2.8 Exercise: display the ASCII value of a character 12
2.9 The conditional operator 13

3 Files, Loops and Patterns **14**
3.1 Files in C 14
3.1.1 Copying a file 14
3.1.2 Opening a file: `fopen()` 14
3.1.3 Reading a character from a file 15
3.2 Printing patterns with nested loops 15
3.2.1 A square of stars 15
3.2.2 An increasing triangle 15
3.2.3 Counting the rows downwards 16
3.2.4 Number triangles 16
3.2.5 Inverted triangles 16
3.2.6 A centred pyramid 17

4 Memory, Storage and Computer Hardware **18**
4.1 Where C is used 18
4.2 What are 0 and 1? 18
4.3 Elements of a C programme 18
4.4 Structures and unions 19
4.4.1 Comparing the memory used 19
4.4.2 A union shares its memory 20
4.5 Storage classes 20
4.5.1 Automatic 20
4.5.2 Register variables 21
4.5.3 Where a programme and its variables live 21

4.6 Static and dynamic memory . . . 21
4.7 Arrays in structures . . . 22
4.8 Strings in C . . . 22
4.8.1 The format specifier `%s` . . . 23
4.9 `sizeof()` and `limits.h` . . . 23
4.9.1 `sizeof()` . . . 23
4.9.2 Limits of the data types . . . 23
4.10 Binary code . . . 23
4.10.1 Representing data . . . 24
4.11 Computer hardware . . . 24
4.11.1 Hardware description language . . . 24
4.11.2 Hard drive . . . 24
4.11.3 RAM . . . 25

5 C++ Language **27**
5.1 Origin . . . 27
5.2 Object-oriented programming . . . 27
5.2.1 Class . . . 27
5.2.2 Object . . . 28
5.2.3 Encapsulation . . . 28
5.2.4 Abstraction . . . 28
5.2.5 Polymorphism . . . 28
5.2.6 Inheritance . . . 29
5.2.7 Constructor . . . 29
5.2.8 Destructor . . . 29
5.3 C++ syntax: input and output . . . 30
5.3.1 Examples of loops . . . 30
5.4 Assigning values to the members of an object . . . 30
5.4.1 An example with private, protected and public members . . . 31
5.5 Single-level inheritance: syntax . . . 32
5.6 Special characters . . . 33
5.6.1 Escape characters . . . 33
5.7 Delimiters . . . 33

5.8 Hierarchical inheritance: a worked programme 33
5.9 Hybrid inheritance in C++ 35
5.10 Constructors in object-oriented programming 36
5.11 Destructors in object-oriented programming 36
5.12 Hybrid inheritance: area and volume 37
5.13 Polymorphism 38
5.13.1 Overloading 38
5.13.2 Overriding 39
5.13.3 Overloading is compile-time polymorphism 40
5.13.4 Another example of overloading: areas 40
5.13.5 Another example of overriding: square and circle 41

A Introduction to Computer Science (lecture notes) 42

A.1 Programming paradigms 42
A.1.1 Why concurrency matters: two ATMs 42
A.2 Numbers inside the computer 42

Todo list

The language was designed by Dennis Ritchie. Kernighan co-authored the 1978 book *The C Programming Language*. 1

ANSI ratified the standard in 1989 (C89). ISO adopted it in 1990 (C90). 1

The notes write "International Standards Organisation". 2

The handwritten version has no semicolon after `c=a+b` and ends with `getc;`, which looks like `getch();`. Also, `void main()` and `conio.h` are Turbo C conventions. Standard C uses `int main(void)` and does not include `conio.h`. 2

The notes say an application that is always dependent on the OS is called "operating system" (probably "operating-system dependent"). 4

Android applications also run only on the Android platform. The notes class Android with Java and .NET as platform independent. 4

C89 guarantees at least 31 significant characters for internal identifiers; C99 raises this to 63. 5

C99 allows declarations anywhere in a block. 5

The notes refer to a statement that consists of five tokens, but the example statement is not written on the page. 7

The notes also list `_Packed` and write `Long` and `Register` with capitals. Standard C (C89) has the 32 keywords above. `_Packed` is a compiler-specific extension (IBM), and keywords are written in lower case. . . . 7

In the scan the long ranges are broken across several lines, so the digits were reconstructed from the standard table. 8

Sizes are implementation defined. The table is correct for 16/32-bit systems. On 64-bit Linux and macOS, `long` is 8 bytes, and on Windows it is 4 bytes. Use `sizeof` or `<limits.h>` to find the values on your machine. 8

The page reads "A&&B is false if both are non zero", which looks like a mix of the definition (true if both are non-zero) and the example with $A = 1$, $B = 0$. 9

After `printf("%u", x);` the page shows a box `x = 2`. It does not follow from the code above. With a 16-bit `unsigned short` holding -4, both `%d` and `%u` print 65532. 10

Assigning an out-of-range value to a signed type is implementation defined in C. Wrap-around is what most compilers do. For *unsigned* types the wrap-around is guaranteed by the standard. 10

Standard ASCII has 128 codes (7 bits). The 256 codes of one byte come from extended 8-bit character sets. The figure 150 for the other symbols is approximate. 12

`fputc` returns an `int` in standard C (the character written, or `EOF`). The notes mark `void fget` as wrong and write `ffput`, which is `fputc`. . . 14

The second file name is hard to read. It may be `inputoutput.txt`. . . . 14

The notes do not close the files. Add `fclose(fp1);` and `fclose(fp2);`, and check that `fopen` did not return `NULL`. 14

In the second loop the scan reads `i >= 5`, which would run only once. It is read here as `i >= 1`. 17

The bounds of the `j` and `k` loops are left blank on the page. They were filled in to give the spacing in the console drawing (39, 38, 37, 36 spaces). 17

The notes write `printf(i=1, i<=n, i++)` for the outer loop; this is the `for` statement. 17

A processor contains the control unit, the ALU and its registers. The memory unit ("MU") is the computer's memory, not part of the processor. 18

Compilers usually add padding to a structure, so `sizeof(struct st)` is typically 8, not 7. Unions are still widely used, for example for hardware registers, tagged variants and network protocols. 19

The notes write `var a` and `var b`. The member access is `var.a` and `var.b`. 20

In the handwritten version the `for` line ends with a semicolon, which would make the loop body empty, and `sum = sum+i` has no semicolon. 21

The handwritten version names the first member `eno` and writes `Student` with a capital in the declaration. The table headings in the drawing are `sno`, `sname`, `smarks`, and the structure tag must be written the same way each time. 22

In C, `sizeof` is an *operator*, not a function, and it is evaluated at compile time (except for variable-length arrays). 23

The notes call them "predefined variables" and "global variables". In `limits.h` they are macros that expand to constants, for example `INT_MAX` and `CHAR_BIT`. 23

With ASCII, `W` is `01010111` (87), `k` is `01101011` (107) and `i` is `01101001` (105). The second pattern on the page is the code of `k`. 23

The diagram of the magnetic regions is hard to read, so the arrows and the placement of the reversals are an interpretation of the drawing. . . 24

The circuit drawing on the page is only partly legible. This redrawing keeps the four labelled parts: bit line, word line, FET and capacitor. . 25

The 6T SRAM circuit is not redrawn here. Only the labels M1–M6, V_{DD}, Q, $\overline{Q}$, BL and $\overline{BL}$ could be read from the scan. 25

The notes write “Object Oriented Programming Structure”. The usual expression is “object-oriented programming”. 27

The notes add that objects are also known as classes. A *class* is the blueprint (the type) and an *object* is an instance of a class. 27

The notes say “a collection of objects”. A class groups data members and member functions; objects are created from it. 27

More precisely, an object is an *instance* of a class: it is a piece of memory that has the data members of the class. 28

The notes say “defining objects inside the class”. Encapsulation means bundling the data and the functions that use it into one unit (the class) and controlling access to them. 28

The notes label the second diagram “multiple level”. Two base classes with one derived class is called *multiple inheritance*. Multilevel inheritance is the chain A, B, C. 29

The notes say it is executed “at the time of execution”; a destructor runs when the object is destroyed (goes out of scope or is deleted). 29

Standard C++ requires `int main()` and, for `string`, `#include <string>`. 30

In the scan the braces around the `if` statement are misplaced. The code was written so that only even numbers are printed. 30

The handwritten version puts quotation marks around the numbers (`"21"`, `"15"`), which would assign a string to an `int`. It also ends `main` with `};`. The numbers have been written without quotes, and the stray semicolon is omitted. 31

The name of the first function is overwritten in the scan (`takeinput` written over `input`); it is read as `input`. 32

The notes declare `double calc()` but do not return a value. Here it is `void calc()`. 32

This listing is a corrected version of the handwritten programme, which would not compile as written. Changes: `char` members for names became `string`; `cin » "gender"` became `cin » gender`; the missing `cin` statements for the institute and the company were added; `void::getdata` became `void getdata`; `Int` became `int`; missing semicolons were added; and `conio.h` and `getch()` were removed as they are not standard C++. 35

The handwritten version uses `void main()`, `clrscr()`, `getch()` and `conio.h` (Turbo C) and leaves out `using namespace std;`. These were replaced by standard C++. 36

The notes write `student(a=0, b=0)` without types and without a body. The version above is a working default constructor with two default arguments. 36

The handwritten programme derives `calcvol` from a class `volume` that is not written on the page, and uses the names `area` and `volume` both for classes and for data members, which C++ does not allow. The base classes are named `dimensions` and `sphere` here, and `sphere` was added. The volume of a sphere is $\frac{4}{3}\pi r^3$; the notes have `radius*radius` (two factors), which is a mistake. 38

In the handwritten code the derived function is written `Display()` with a capital D. That is a different function, so it does not override `display()`. With a lower-case `d`, `bs.display()` prints the derived-class message. . 40

This page of the notes is almost empty: only the comment, the header `bits/stdc++.h` and `using namespace std;` are written, and the rest of the programme is not on the page. 40

For the cuboid, $2(ab + bc + ca)$ is its *surface* area. 41

The brace on the page groups C with both "object oriented" and "procedure oriented", and a Hindi word next to "paradigms" is not legible. The table is an interpretation of the list; the Scheme and Assembly descriptions are my additions. 42

1 C Language

1.1 Overview of C

1.1.1 History

The origin of C is closely tied to the development of the Unix operating system, originally implemented in assembly language on a PDP-7 by Dennis Ritchie and Ken Thompson, incorporating several ideas from colleagues. The developers were considering rewriting the system using the B language, which was the early version of the C language. The name of C was chosen simply as the next letter after B.

The development of C started in 1972 on the PDP-11 Unix system. A compiler for the Honeywell 6000 was written within the first year of C's history, while an IBM System/370 port followed soon. In 1978, Brian Kernighan and Dennis Ritchie published the first edition of *The C Programming Language*; the language it described became known as K&R C.

Features of K&R C

- Standard I/O library
- `long int` data types
- `unsigned int` data types
- Compound assignment

C is a computer programming language.

- Developed by Brian Kernighan and Dennis Ritchie of AT&T Bell Labs in 1972.
- In 1983, ANSI began the standardisation process.
- In 1989, ISO continued the same process as ANSI.
- In 1990, it became known as Standard C.
- Known as K&R C.

The language was designed by Dennis Ritchie. Kernighan co-authored the 1978 book *The C Programming Language*.

ANSI ratified the standard in 1989 (C89). ISO adopted it in 1990 (C90).

1.2 Importance of C

Language is a set of instructions for communication. A **computer language** is a set of programs, which become applications and then software.

People share information and communicate with each other, and also with a computer. To communicate with a computer they write a program (the *source code*) using a language such as C, C++, Java or .NET.

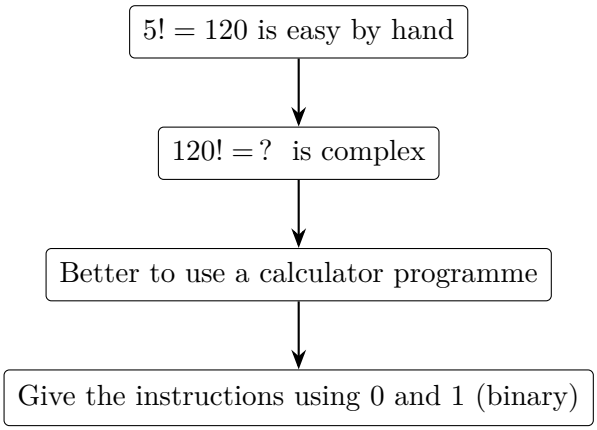


Figure 1.1: Why we write programmes: some calculations are too complex to do by hand.

A computer cannot understand the English language. Compilers are used for communication with the computer.

1.3 How a programme is turned into output

- **ANSI**: American National Standards Institute.
- **ISO**: International Organization for Standardization.

The notes write "International Standards Organisation".

Unix is written in C.

1.4 A first C programme

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    int a, b, c;
    printf("enter the value of a and b");
    scanf("%d %d", &a, &b);
    c = a + b;
    printf("%d", c);
    getch();
}
```

The handwritten version has no semicolon after `c=a+b` and ends with `getc;`, which looks like `getch();`. Also, `void main()` and `conio.h` are Turbo C conventions. Standard C uses `int main(void)` and does not include `conio.h`.

Save the programme with the `.c` extension.

1.5 Application development using C

Applications are of two kinds: standalone applications and web applications.

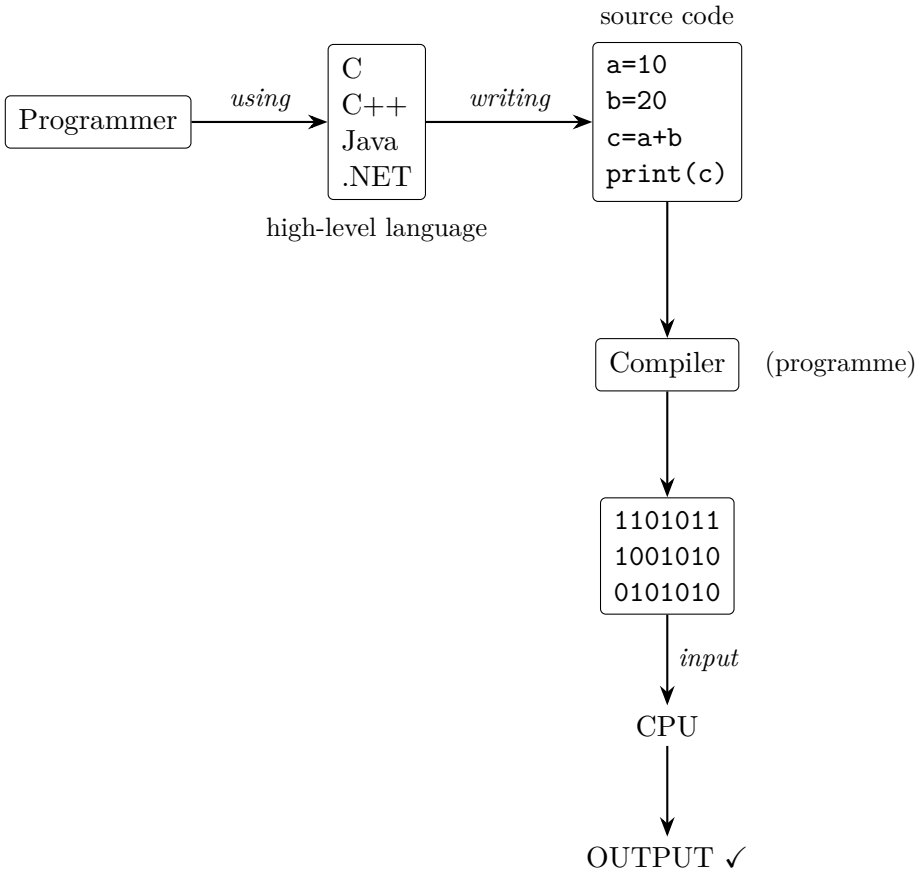


Figure 1.2: From source code to output: the compiler translates the source code into binary machine code that the CPU executes.

	Standalone applications	**Web applications**
Installation	must be installed, e.g. VLC, MS Office, Chrome, Origin	used through the internet without installing software, e.g. facebook.com, web.whatsapp.com, gmail.com
Operating system	compatible with a single operating system; the application depends on the OS	independent of the operating system
Necessary to install?	yes	no

1.5.1 Why a standalone application depends on the operating system

Each kind of file is opened by a matching application: an `abc.txt` file is not opened by VLC and an `xyz.mp4` file is not opened by Notepad. In the same way, an installable application has a different form on each operating system:

Operating system	File extensions
Windows	`.exe`
macOS	`.dmg`
Linux	`.rpm`, `.tar`

The notes say an application that is always dependent on the OS is called "operating system" (probably "operating-system dependent").

Programming languages for standalone applications: C, C++, Java, .NET, Android.

1.5.2 A web application: the language-choice example

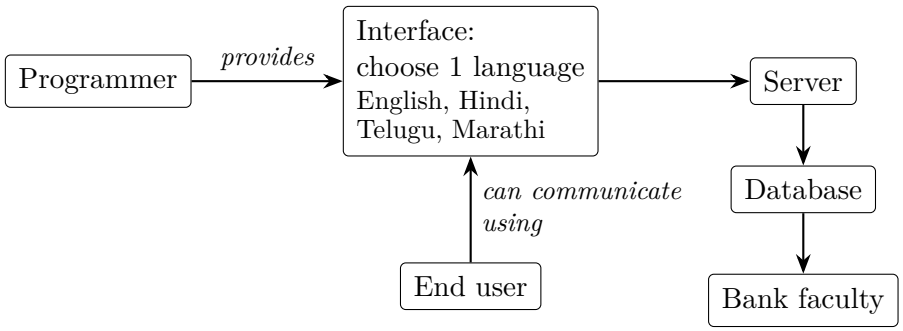


Figure 1.3: A programmer provides an interface; the end user communicates through it with a server and a database.

1.6 Platform dependence

Language	Property	Kind of application
C, C++	drawback: platform dependent (on the OS)	standalone applications
Java, .NET, Android	platform independent	standalone applications and web applica

Android applications also run only on the Android platform. The notes class Android with Java and .NET as platform independent.

- The best games are developed in C++.
- Embedded programming is done in C.

1.7 Programming style in C

A C programme begins with preprocessor lines that bring in header files.

`#include<stdio.h>` `#` is the *preprocessor*; `include` means *add*; `stdio` is *standard input output*; `.h` marks a
`#include<conio.h>` `conio` is *console input output*

1.7.1 Compiling for each platform

The compiler and the library depend on the operating system. Windows produces a `.exe` file and macOS a `.dmg` file, each with the compiler and the library for that system.

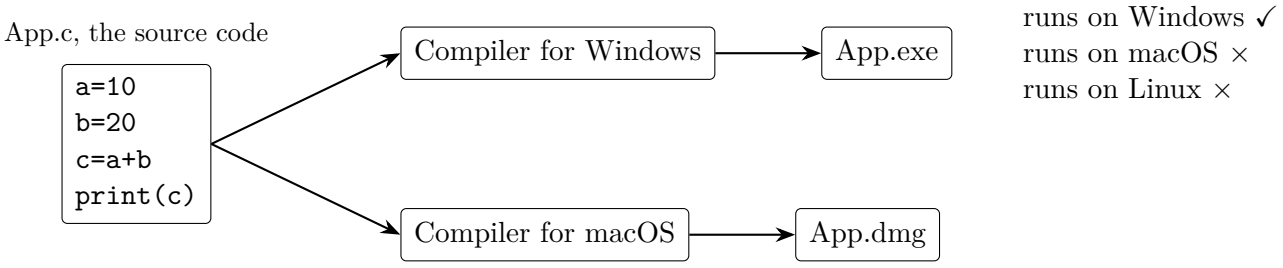


Figure 1.4: The same source code gives a different executable for each platform, and an executable runs only on the platform it was built for.

1.8 Variables

A variable is a named memory location. In memory, each location has an address (a positive integer) and holds a value:

	10	a
128		5098

memory locations with their addresses

Syntax:

```
data_type identifier;
```

For example

```
int a;
a = 10;
printf("%d", a);       /* prints 10 */
```

1.8.1 Rules for variable names

- Valid characters are letters, digits and the underscore `_`.
- The first character cannot be a digit.
- 31 characters are recognised for local variables.
- Variables must be declared before use, at the start of the block, immediately after `{`.

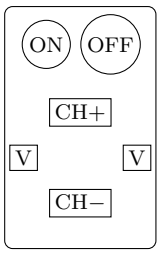
C89 guarantees at least 31 significant characters for internal identifiers; C99 raises this to 63.

C99 allows declarations anywhere in a block.

```
int add(int x, int y)
{
    int z = x + y;
    return z;
}
```

1.9 Programmes, functions and the remote controller

A *programme* is made of function definitions and function calls. A television remote controller is a good picture: each button calls a function, and the signal is carried by infrared rays (programming plus technology).



1. `on()`
2. `off()`
3. `ch()`
4. `vol()`

Figure 1.5: Each button of a remote controller corresponds to a function.

```
on()  { ..... }
off() { ..... }
ch()  { ..... }
vol() { ..... }
```

1.10 Building blocks of a programme

Variables
change their value: `a, b, c, d, e, f, g, h, ...`

Keywords
have a fixed meaning: `if`, `if-else`, `while`.

Constants
have a fixed value, for example $\pi = 22/7$ or $\pi = 3.14$. The value chosen for a constant can differ from one programme to another, but it does not change while the programme runs.

Identifiers
are names chosen by the programmer, as in `int SUM`.

Operators
act on operands, as in $a \oplus b$.

1.11 Tokens in C

A C programme consists of various tokens. A token is either a keyword, an identifier, a string literal, or a symbol.

The notes refer to a statement that consists of five tokens, but the example statement is not written on the page.

1.11.1 Identifiers

A C identifier is a name used to identify variables, functions or any other user-defined item. An identifier starts with a letter `A` to `Z` or `a` to `z`, or with an underscore `_`, followed by zero or more letters, underscores and digits (`0` to `9`).

1.11.2 Keywords

Keywords are the reserved words in C.

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
const     float   short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

The notes also list `_Packed` and write `Long` and `Register` with capitals. Standard C (C89) has the 32 keywords above. `_Packed` is a compiler-specific extension (IBM), and keywords are written in lower case.

2 Data Types, Operators and Characters

2.1 Data types

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored there is interpreted.

2.1.1 Integer types (%d)

The following table gives the standard integer types with their storage size and value range.

Type	Storage size	Value range
`char`	1 byte	−128 to 127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	−128 to 127
`int`	2 or 4 bytes	−32 768 to 32 767 or −2 147 483 648 to 2 147 483 647
`unsigned int`	2 or 4 bytes	0 to 65 535 or 0 to 4 294 967 295
`short`	2 bytes	−32 768 to 32 767
`unsigned short`	2 bytes	0 to 65 535
`long`	4 bytes	−2 147 483 648 to 2 147 483 647
`unsigned long`	4 bytes	0 to 4 294 967 295

In the scan the long ranges are broken across several lines, so the digits were reconstructed from the standard table.

2.1.2 Floating-point types (%f)

Type	Storage size	Value range	Precision
`float`	4 bytes	1.2×10^{-38} to 3.4×10^{38}	6 decimal places
`double`	8 bytes	2.3×10^{-308} to 1.7×10^{308}	15 decimal places
`long double`	10 bytes	3.4×10^{-4932} to 1.1×10^{4932}	19 decimal places

Sizes are implementation defined. The table is correct for 16/32-bit systems. On 64-bit Linux and macOS, `long` is 8 bytes, and on Windows it is 4 bytes. Use `sizeof` or `<limits.h>` to find the values on your machine.

2.1.3 Two useful symbols

`\n`	used to print on a new line
`/* ... */`	a comment line: it is not printed in the result

2.2 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

2.2.1 Types of operator

1. **Arithmetic:** `+`, `-`, `*`, `/`, `%`, `++`, `--`
2. **Relational:** `==`, `!=`, `>`, `<`, `>=`, `<=`
3. **Logical:** `&&` (AND), `||` (OR), `!` (NOT)
4. **Bitwise:** a bitwise operator works on bits and performs a bit-by-bit operation: `&`, `|`, `^`

Logical operators

`A && B`	AND: true if both operands are non-zero
`A || B`	OR: true if any one of the two operands is non-zero
`!(A)`	NOT: reverses the logical state of its operand

For example, if `A = 1` and `B = 0`, then `A && B` is false and `!(A && B)` is true.

The page reads "A&&B is false if both are non zero", which looks like a mix of the definition (true if both are non-zero) and the example with $A = 1$, $B = 0$.

2.3 Classification of data types

Primitive	Derived	User-derived
`int`	array	structure
`char`	string	union
`float`	pointer	`typedef`
`void`	functions	

These types are discussed below. This is just a classification of the data types.

2.4 Storage size of integer types

A small number such as `10` needs only 1 byte. A phone number or an account number is much larger and needs 4 bytes.

A `short` uses 2 bytes, that is 16 bits, so it can hold $2^{16} = 65\,536$ different bit patterns.

- **signed short** (2 bytes): half of the patterns are negative, so the range is $-32\,768, \ldots, -1, 0, 1, \ldots, 32\,767$ (note $65\,536/2 = 32\,768$).
- **unsigned short** (2 bytes): all bits zero is 0 and all bits one is 65 535, so the range of unsigned data types is 0 to 65 535.

Signed declaration: `short a;` The variable is of type `short`, so this is a declaration.

2.4.1 Format specifiers

`%d`	`int` (signed)
`%u`	unsigned data types

2.5 What happens when a value does not fit

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    short a = 10;
    clrscr();
    printf("%d", a);
}
```

A value outside the range of the type wraps around, like a number on a circle (Figure 2.1).

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    short x = 32769;
    clrscr();
    printf("%d", x);    /* x = -32767, a "garbage" value */
}
```

```c
#include<stdio.h>
#include<conio.h>
void main()
{
    unsigned short x = -4;
    clrscr();
    printf("%d", x);    /* x = 65532 */
    printf("%u", x);
}
```

After `printf("%u", x);` the page shows a box `x = 2`. It does not follow from the code above. With a 16-bit `unsigned short` holding -4, both `%d` and `%u` print 65532.

Assigning an out-of-range value to a signed type is implementation defined in C. Wrap-around is what most compilers do. For *unsigned* types the wrap-around is guaranteed by the standard.

2.6 The `char` type

Kind	Size	Number of values	Range
signed `char`	1 byte	$2^8 = 256$ (half negative: 256/2)	-128 to 127
unsigned `char`	1 byte	$2^8 = 256$	0 to 255

A signed `char` also uses the + and − signs. Like `short`, it wraps around when a value does not fit:

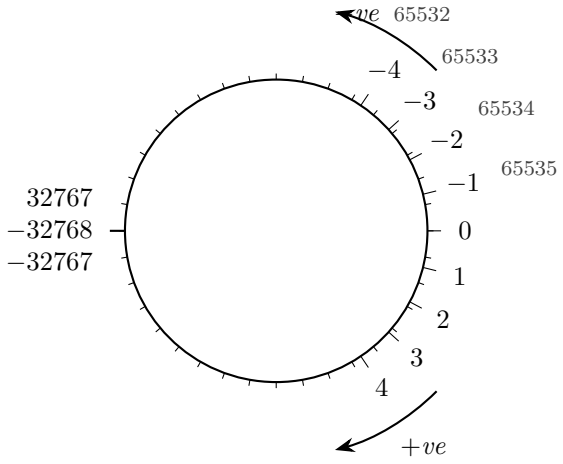


Figure 2.1: Two's-complement numbers form a circle: counting up past 32 767 gives −32 768. Outside the circle: the same bit patterns read as unsigned numbers.

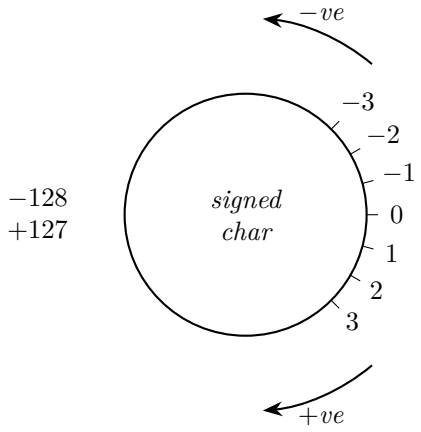


Figure 2.2: A signed `char` runs from −128 to +127 and then wraps around.

2.7 Characters are stored as numbers

A computer stores a character as a number, not as a symbol. The compiler turns the source code into binary patterns such as `1111`, `1011`, `0110`, `0011`. A symbol cannot be stored directly; a number system is used for characters.

2.7.1 ASCII

ASCII stands for American Standard Code for Information Interchange.

A–Z	a–z	0–9	#	space
A = 65	a = 97	0 = 48	# = 35	' ' = 32
B = 66	b = 98	1 = 49		
C = 67	c = 99	2 = 50		
⋮	⋮	⋮		
Z = 90	z = 122	9 = 57		

Counting the characters: $26 + 26 + 10 + 150 \leq 256 = 2^8$, which is why one byte is enough for a `char`.

Standard ASCII has 128 codes (7 bits). The 256 codes of one byte come from extended 8-bit character sets. The figure 150 for the other symbols is approximate.

2.7.2 Printing a character and its code

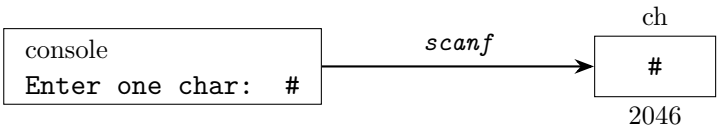
```
#include<stdio.h>
void main()
{
    char ch = 'A';
    printf("%c", ch);      /* prints A */
    printf("%d", ch);      /* prints 65 */
}
```

A value that does not fit in a `char` wraps around. Here 258 mod 256 = 2, which is an unprintable control character, so "unknown character data will print":

```
#include<stdio.h>
void main()
{
    char ch = 258;       /* ch = 2 */
    printf("%c", ch);
}
```

2.8 Exercise: display the ASCII value of a character

```
#include<stdio.h>
void main()
{
    char ch;                          /* not char ch = 'x'; */
    printf("Enter one char: ");
    scanf("%c", &ch);
    printf("ASCII value is: %d", ch);
}
```



`scanf("%c", &ch)` stores the typed character in the variable `ch`, at address 2046. The `&` is the address operator. The next line prints the ASCII value, 35 for `#`.

> Information is very important. How good your logic is does not matter as much as how clearly you provide an interface to the user: that is very important for a developer.

2.9 The conditional operator

The ternary operator `?:` has the syntax

```
condition ? expression1 : expression2;
```

If the condition is true, `expression1` is evaluated; if it is false, `expression2` is evaluated.

```
void main()
{
    int a, b;
    printf("Enter 2 integers");
    scanf("%d %d", &a, &b);
    a > b ? printf("a is big") : printf("b is big");
}
```

For example, `10 > 20` is false, so `b is big` is printed.

3 Files, Loops and Patterns

3.1 Files in C

Syntax:

```
FILE *<identity>;
FILE *ptr;
```

3.1.1 Copying a file

The functions used are

```
int  fgetc(FILE *fp);
void fputc(int ch, FILE *fp);
```

`fputc` returns an `int` in standard C (the character written, or `EOF`). The notes mark `void fget` as wrong and write `ffput`, which is `fputc`.

```
void main()
{
    FILE *fp1, *fp2;
    int ch;
    fp1 = fopen("d:/input.txt", "r");
    fp2 = fopen("d:/output.txt", "w");
    while ((ch = fgetc(fp1)) != EOF)
    {
        fputc(ch, fp2);
        printf("copied");
    }
}
```

The second file name is hard to read. It may be `inputoutput.txt`.

The notes do not close the files. Add `fclose(fp1);` and `fclose(fp2);`, and check that `fopen` did not return `NULL`.

3.1.2 Opening a file: `fopen()`

`fopen()` is used to open a file in different modes.

Prototype:

```
FILE *fopen("path", "mode");
```

- on success it returns the address of the file (a `FILE` pointer);
- on failure it returns a `NULL` pointer.

```
#include<stdio.h>
void main()
{
    FILE *fp;
```

```
    fp = fopen("g:/example/test.txt", "r");   /* "r" is read mode */
    if (fp == NULL)
        printf("file not present\n");
    else
        printf("file opened in read mode\n");
}
```

3.1.3 Reading a character from a file

```
int fgetc(FILE *stream);
```

- on success it returns the ASCII value of the character;
- on failure it returns `EOF`, which is -1. `EOF` means *End of File* and it is a macro.

3.2 Printing patterns with nested loops

The outer loop controls the rows (`i`) and the inner loop controls the columns (`j`). `printf("\n")` after the inner loop starts a new row.

3.2.1 A square of stars

```
for (i = 1; i <= 5; i++)
{
    for (j = 1; j <= 5; j++)
    {
        printf("*");
    }
    printf("\n");
}
```

For each value of `i` the inner loop runs for `j` $= 1, 2, 3, 4, 5$ and stops when `j` becomes 6, the first value that fails `j <= 5`. The output is five rows of five stars.

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

3.2.2 An increasing triangle

Let the inner loop run up to `i`:

```
for (i = 1; i <= 5; i++)
```

```
{
    for (j = 1; j <= i; j++)
    {
        printf("*");
    }
    printf("\n");
}
```

Row `i` contains `i` stars.

3.2.3 Counting the rows downwards

```
for (i = 5; i >= 1; i--)
{
    for (j = 5; j >= i; j--)
    {
        printf("*");        /* or printf("%d", j); */
    }
    printf("\n");
}
```

Row $i = 5$ has one star, row $i = 4$ has two, and so on up to five stars for $i = 1$.

3.2.4 Number triangles

Counting the columns downwards from `i`:

```
for (i = 1; i <= 5; i++)
{
    for (j = i; j >= 1; j--)
    {
        printf("%d", j);     /* or printf("%d", i); */
    }
    printf("\n");
}
```

Printing `j` gives the first triangle below, and printing `i` gives the second:

```
1                   1
2 1                 2 2
3 2 1               3 3 3
4 3 2 1             4 4 4 4
5 4 3 2 1           5 5 5 5 5
```

3.2.5 Inverted triangles

All four of these nested loops print the same inverted triangle. The loops differ in how they count:

Outer loop	Inner loop	Stars in row `i`
`for (i = 1; i <= 5; i++)`	`for (j = i; j <= 5; j++)`	$6 - i$
`for (i = 5; i >= 1; i-)`	`for (j = i; j >= 1; j-)`	i
`for (i = 1; i <= 5; i++)`	`for (j = 5; j >= i; j-)`	$6 - i$
`for (i = 5; i >= 1; i-)`	`for (j = 1; j <= i; j++)`	i

In the second loop the scan reads `i >= 5`, which would run only once. It is read here as `i >= 1`.

```
* * * * *
* * * *
* * *
* *
*
```

The inner statements are the same in every case: `printf("*");` inside the inner loop and `printf("\n");` after it.

3.2.6 A centred pyramid

To centre a pyramid on a console that is 80 columns wide, each row first prints spaces and then the stars. The centre column is 40, so row i starts with $40 - i$ spaces: 39 spaces for one star, 38 for two, 37 for three, 36 for four.

```
void main()
{
    int n, i, j, k;
    scanf("%d", &n);                    /* number of rows */
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= 40 - i; j++)    /* leading spaces */
        {
            printf(" ");
        }
        for (k = 1; k <= i; k++)         /* the stars */
        {
            printf("* ");
        }
        printf("\n");
    }
}
```

The bounds of the `j` and `k` loops are left blank on the page. They were filled in to give the spacing in the console drawing (39, 38, 37, 36 spaces).

The notes write `printf(i=1, i<=n, i++)` for the outer loop; this is the `for` statement.

console, 80 columns

```
39 spaces *
38 spaces * *
37 spaces * * *
36 spaces * * * *
```

4 Memory, Storage and Computer Hardware

4.1 Where C is used

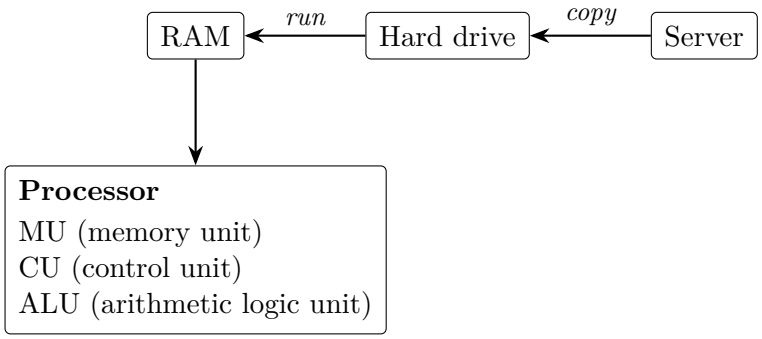
C is important because much important software is written in it:

- Oracle;
- the Android core library;
- MySQL, and all device drivers.

C is popular for hardware interaction.

4.2 What are 0 and 1?

A programme is stored on a server, copied to the hard drive, and run from the RAM by the processor. The processor contains a control unit and an arithmetic logic unit; it is a circuit that reads an instruction and decodes it.



a circuit that reads an instruction and decodes it

Figure 4.1: The path of a programme from the server to the processor.

A processor contains the control unit, the ALU and its registers. The memory unit ("MU") is the computer's memory, not part of the processor.

4.3 Elements of a C programme

- `#include <stdio.h>`: the `#include` line is handled by the *preprocessor*.
- **Identifiers:** constants (primitive or non-primitive), variables and keywords.
- **Variables** are the names of the locations where we store information.

The instructions of a programme are of four kinds:

- data type declaration;
- input/output instructions;
- arithmetic instructions;
- control instructions.

4.4 Structures and unions

Structures and unions are user-defined data types.

```
struct Identity
{
    data_type e1;
    data_type e2;
    ...
    data_type en;
};
```

```
union Identity
{
    data_type e1;
    data_type e2;
    ...
    data_type en;
};
```

The members of a union cannot be processed at the same time.

4.4.1 Comparing the memory used

```
struct st
{
    char c;
    short s;
    float f;
};
```

```
union un
{
    char c;
    short s;
    float f;
};
```



Figure 4.2: A structure stores all its members one after the other. A union stores them in the same place, so its size is that of its largest member.

A structure makes it easy to process all the data at a time. A union is more efficient in memory, because the structure stores 7 bytes and the union takes the minimum. Some programmers say that no one uses unions now because we no longer have a memory problem.

Compilers usually add padding to a structure, so `sizeof(struct st)` is typically 8, not 7. Unions are still widely used, for example for hardware registers, tagged variants and network protocols.

4.4.2 A union shares its memory

```
union um
{
    short a;
    short b;
};

main()
{
    union um var;
    var.a = 10;
    printf("b val: %d\n", var.b);      /* prints 10 */
    var.b = 20;
    printf("a val: %d\n", var.a);      /* prints 20 */
}
```

The members `a` and `b` occupy the same 2 bytes (at address 2046). Writing `a` changes what is read from `b`, and the other way round.

The notes write `var a` and `var b`. The member access is `var.a` and `var.b`.

4.5 Storage classes

There are four storage classes:

- automatic;
- register;
- static;
- external.

A storage class describes four things about a variable: its **memory location**, its **default value**, its **scope** and its **lifetime**.

4.5.1 Automatic

Keyword	`auto`
Memory	RAM
Default value	garbage value
Lifetime and scope	declared either inside a block or in `main`

```
void main()
{
    auto int a = 10;                   /* method scope */
    {
        auto int a;                    /* block scope: a garbage value
    */
        printf("a: %d\n", a);
    }
```

```
    printf("a: %d\n", a);              /* prints 10 */
}
```

The a declared inside the inner block hides the outer a until the block ends.

4.5.2 Register variables

Keyword	`register`
Memory	CPU registers
Default value	garbage value

4.5.3 Where a programme and its variables live

Consider a programme that adds the numbers from 1 to n:

```
void main()
{
    int n, sum = 0, i;
    printf("Enter n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        sum = sum + i;
    }
    printf("sum %d\n", sum);
}
```

In the handwritten version the `for` line ends with a semicolon, which would make the loop body empty, and `sum = sum+i` has no semicolon.

The programme is stored on the hard drive (secondary memory). When it runs, the complete programme is fetched into RAM, where the variables n, i and sum get their values. The processor takes the instructions from RAM and keeps the values it is using in its registers.

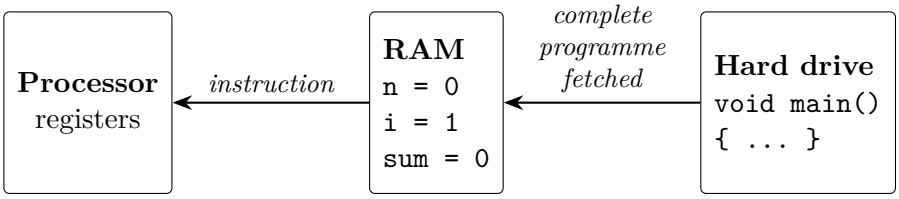


Figure 4.3: The programme is fetched from the hard drive into RAM, and the processor takes the instructions from RAM.

4.6 Static and dynamic memory

Memory can vary from time to time.

- **Static memory** is allocated at compile time and is *fixed*.

- **Dynamic memory** is allocated at run time. It starts with some initial memory: inserting data makes it *grow*, and deleting data makes it *shrink*.

4.7 Arrays in structures

```
struct emp
{
    int   eno;
    char  ename[20];
    float esal;
};
struct emp e1, e2, e3;
```

An array can be a member of a structure (`sname`, `smarks`), and a structure variable can itself be an array:

```
struct student
{
    int  sno;
    char sname[20];
    int  smarks[4];
};
struct student s[3];
```

The handwritten version names the first member `eno` and writes `Student` with a capital in the declaration. The table headings in the drawing are `sno`, `sname`, `smarks`, and the structure tag must be written the same way each time.

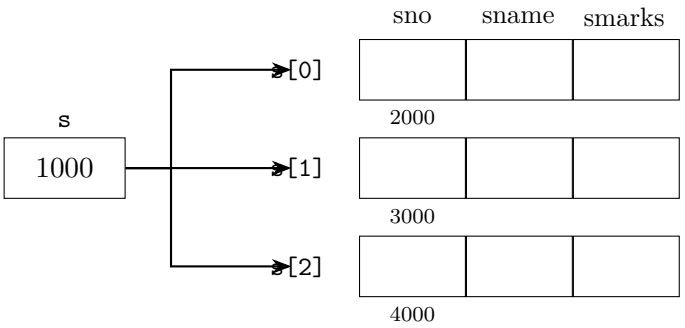


Figure 4.4: An array of three `student` structures: each element holds `sno`, `sname` and `smarks`. The variable `s` holds the address (here 1000) of the first element.

4.8 Strings in C

A string is a one-dimensional character array, `char[ ]`.

Syntax:

```
char identity[size];
```

```
char s1[5] = "HARI";
char s2[4] = {'O', 'n', 'e'};
```

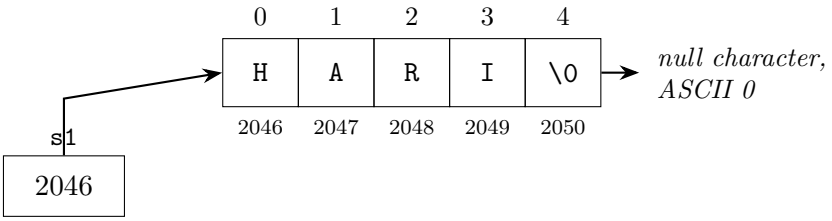


Figure 4.5: `"HARI"` takes five bytes: the four letters and the null character `'\0'`. The variable `s1` holds the address of the first character.

The size of 5 counts the null character that ends every string. In `s2`, the fourth element is set to `'\0'` automatically.

4.8.1 The format specifier %s

`"%s"` is the format specifier that reads and writes a whole string, so there is no need to use an iterator (a loop) over its elements.

4.9 `sizeof()` and `limits.h`

4.9.1 `sizeof()`

`sizeof()` is a predefined function that returns the size of different variables, data types, expressions, pointers and so on.

In C, `sizeof` is an *operator*, not a function, and it is evaluated at compile time (except for variable-length arrays).

4.9.2 Limits of the data types

The limits of the data types are given by predefined values in the header file `limits.h`. They are global: they can be accessed from any C application. They are constants, so they cannot be modified.

The notes call them "predefined variables" and "global variables". In `limits.h` they are macros that expand to constants, for example `INT_MAX` and `CHAR_BIT`.

4.10 Binary code

A binary code represents text, computer processor instructions, or any other data using a two-symbol system. The two-symbol system used is often the binary system, 0 and 1. The binary code assigns a pattern of binary digits, also known as bits, to each character, instruction and so on. In computing and telecommunications, binary codes are used for various methods of encoding data.

```
01010111  -- W
01101011  -- i
```

With ASCII, `W` is `01010111` (87), `k` is `01101011` (107) and `i` is `01101001` (105). The second pattern on the page is the code of `k`.

4.10.1 Representing data

We have all seen computers do seemingly miraculous things with all kinds of sounds, pictures, graphics, numbers and text. It seems that we can build a replica of parts of our world inside the computer. You might think that this amazing machine is also amazingly complicated; it really is not. All of the wonderful multimedia that we see on modern computers is constructed from simple ON/OFF switches.

$$\begin{aligned} \text{OFF} &= 0 \\ \text{ON} &= 1 \end{aligned}$$

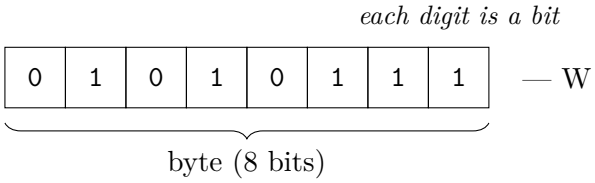


Figure 4.6: A byte is eight bits. This pattern is the ASCII code of the letter W.

A single byte can represent many different kinds of data. What data it actually represents depends on how the computer uses the byte.

4.11 Computer hardware



4.11.1 Hardware description language

In computer engineering, a hardware description language (HDL) is a specialised computer language used to describe the structure and behaviour of electronic circuits and digital logic circuits.

4.11.2 Hard drive

A hard disk drive (HDD) is an electromechanical data-storage device that uses magnetic storage to store and retrieve digital information using one or more rapidly rotating disks (platters) coated with magnetic material.

Here **R** means *reverse* and **N** means *no reverse*. Two reversals in a row encode a 1 and no reversal encodes a 0.

The diagram of the magnetic regions is hard to read, so the arrows and the placement of the reversals are an interpretation of the drawing.

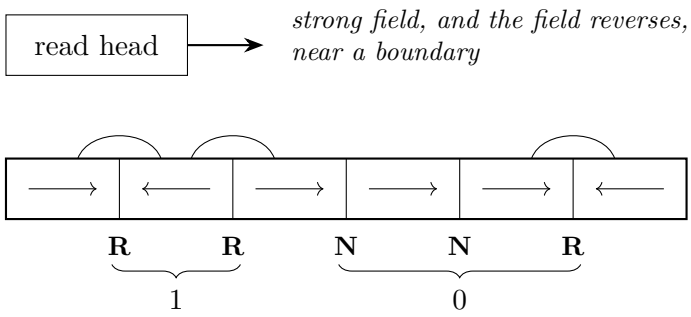


Figure 4.7: Bits on a disk: a reversal of the magnetisation (R) between neighbouring regions gives a strong field near the boundary that the read head detects. State: binary value encoded.

4.11.3 RAM

Random-access memory (RAM) is a form of computer data storage that stores data and machine code currently being used. The number of data lines is 8 bit or 16 bit. Dynamic RAM was introduced in October 1970.

DRAM: one transistor and one capacitor

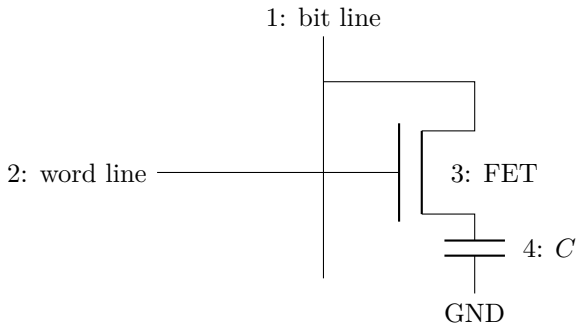


Figure 4.8: A DRAM cell (1T1C): a field-effect transistor (FET) connects the capacitor to the bit line when the word line is high.

The circuit drawing on the page is only partly legible. This redrawing keeps the four labelled parts: bit line, word line, FET and capacitor.

DRAM is based around a capacitor. Charging and discharging this capacitor can store a “1” or a “0” in the cell.

SRAM: six transistors

An SRAM cell (6T) uses six transistors, M1 to M6. Four of them (M1 to M4) form two cross-coupled inverters between the supply voltage V_{DD} and ground; they store the bit as the states Q and $\overline{Q}$. The two access transistors (M5 and M6) connect the cell to the bit lines BL and $\overline{BL}$ when the word line is high.

The 6T SRAM circuit is not redrawn here. Only the labels M1–M6, V_{DD}, Q, $\overline{Q}$, BL and $\overline{BL}$ could be read from the scan.

The SRAM memory cell

In SRAM, the memory cell is a type of flip-flop circuit, usually implemented using FETs. The memory cell is the fundamental building block of computer memory: an electronic circuit that stores one bit of binary information. It must be *set* to store a logic 1 (high voltage level) and *reset* to store a logic 0 (low voltage level). Its value is maintained until it is changed by the set/reset process.

Types of RAM

- static RAM (SRAM);
- dynamic RAM (DRAM);
- synchronous dynamic RAM (SDRAM);
- single data rate synchronous dynamic RAM (SDR SDRAM);
- double data rate synchronous dynamic RAM (DDR, DDR2, DDR3, DDR4).

5 C++ Language

5.1 Origin

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features. It was designed by Bjarne Stroustrup. By 1983 several modifications and extensions had already been made to C, under the name *C with classes*, and in that year the name C++ was coined. Ever since then, the `++` has become a synonym for object orientation.

First appeared	1985
Standardised by	ISO, the International Organization for Standardization

5.2 Object-oriented programming

OOP stands for object-oriented programming. It is a programming pattern in which we divide our programme into small units called *objects*. OOP is a collection of individual concepts, as follows:

The notes write "Object Oriented Programming Structure". The usual expression is "object-oriented programming".

The notes add that objects are also known as classes. A *class* is the blueprint (the type) and an *object* is an instance of a class.

1. class;
2. object;
3. encapsulation;
4. abstraction;
5. polymorphism;
6. inheritance.

5.2.1 Class

A class is a collection of members, which may be variables, methods, a constructor, a destructor and so on.

The notes say "a collection of objects". A class groups data members and member functions; objects are created from it.

```
class class_name
{
    access_specifiers:
        members;
};
```

5.2.2 Object

An object stands for a class and refers to the members of the class. An object is created by writing the class name followed by the object name:

```
class_name object_name;
```

More precisely, an object is an *instance* of a class: it is a piece of memory that has the data members of the class.

5.2.3 Encapsulation

Encapsulation means wrapping of data. In other words, defining the data and the functions that work on it inside the class is known as encapsulation.

The notes say "defining objects inside the class". Encapsulation means bundling the data and the functions that use it into one unit (the class) and controlling access to them.

5.2.4 Abstraction

Abstraction means hiding of data inside the class. It is implemented with the help of access specifiers. There are three basic access specifiers in C++:

1. **Public.** A member with the public access specifier can be accessed anywhere in the programme with the help of objects.
2. **Protected.** A member declared with the protected specifier can be accessed inside the parent class and in the child class of that class.
3. **Private.** A member with the private access specifier is accessible only inside the class where it is defined.

Here the *parent class* is the class that is inherited from, and the *child class* is the class that inherits.

5.2.5 Polymorphism

When we define a function more than one time with the same name and it can produce different output, this condition is known as polymorphism. Polymorphism can be categorised into two parts.

1. **Overloading.** Functions with the same name but different parameters.

```
void calc()
{
    statements and expressions
}

void calc(arg1)
{
    statements and expressions
}
```

2. **Overriding.** Functions with the same name and the same parameters (inheritance is mandatory).

5.2.6 Inheritance

Inheritance is used to extend the properties of a parent class to a child class. A parent class is also known as a *base class*, and a child class is also known as a *derived class*. Inheritance can be categorised into five types.

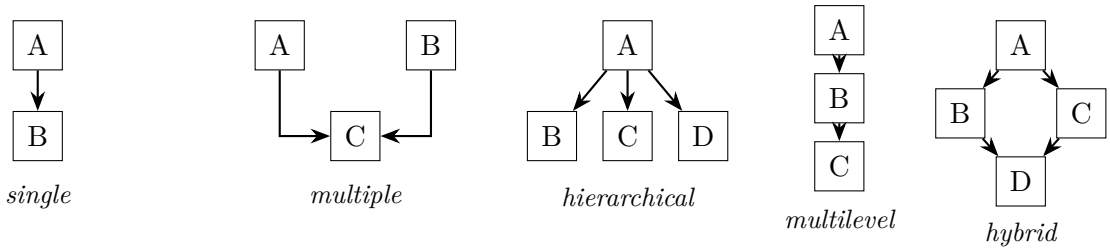


Figure 5.1: The five types of inheritance: an arrow goes from the parent (base) class to the child (derived) class.

The notes label the second diagram "multiple level". Two base classes with one derived class is called *multiple inheritance*. Multilevel inheritance is the chain A, B, C.

5.2.7 Constructor

A constructor is a special type of method, defined with the class name, in C++. Its property is that it is executed automatically at the time of object initialisation.

```
class class_name
{
    class_name()            // constructor
    {
        statements and expressions
    }
    access_specifiers:
        members;
};
```

5.2.8 Destructor

A destructor is just like a constructor, but there is a big difference: it is used to release memory, and it is executed when the object is destroyed. It is also defined with the class name, with a tilde in front.

The notes say it is executed "at the time of execution"; a destructor runs when the object is destroyed (goes out of scope or is deleted).

```
class class_name
{
    ~class_name()           // destructor
    {
        statements and expressions
    }
    access_specifiers:
        members;
};
```

5.3 C++ syntax: input and output

A C++ programme includes `iostream` and uses the standard namespace. `printf` becomes `cout` and `scanf` becomes `cin`.

```
#include<iostream>
using namespace std;
main()
{
    string name;
    int age;
    cout << "enter the value of name";
    cin >> name;
    cout << "Enter your age";
    cin >> age;
    cout << "Name" << name << endl << "age" << age;
}
```

Standard C++ requires `int main()` and, for `string`, `#include <string>`.

5.3.1 Examples of loops

A multiplication table, with a loop inside a loop:

```
int a;
for (a = 1; a <= 10; a++)
{
    int b;
    for (b = 1; b <= 10; b++)
    {
        cout << a * b << "\t";
    }
    cout << endl;
}
```

The even numbers up to 100:

```
int a;
for (a = 1; a <= 100; a++)
{
    if (a % 2 == 0)
    {
        cout << a << endl;
    }
}
```

In the scan the braces around the `if` statement are misplaced. The code was written so that only even numbers are printed.

5.4 Assigning values to the members of an object

```
#include<iostream>
using namespace std;
class employee
{
```

```
    public:
        string name;
        int roll, age;
        void display()
        {
            cout << "Name = " << name << endl;
            cout << "Roll No. = " << roll << endl;
            cout << "Age = " << age << endl;
        }
};
main()
{
    employee obj;
    obj.name = "Abhishek";
    obj.age  = 21;
    obj.roll = 15;
    obj.display();
}
```

The handwritten version puts quotation marks around the numbers (`"21"`, `"15"`), which would assign a string to an `int`. It also ends `main` with `};`. The numbers have been written without quotes, and the stray semicolon is omitted.

5.4.1 An example with private, protected and public members

```
#include<iostream>
using namespace std;
class employee
{
    private:
        int roll;
    protected:
        int age;
    public:
        string name;
        void input(string n, int r, int a)
        {
            name = n;
            roll = r;
            age  = a;
        }
        void display()
        {
            cout << "Name = " << name << endl;
            cout << "Roll No. = " << roll << endl;
            cout << "Age = " << age << endl;
        }
};
main()
{
    employee obj;
    obj.input("Abhishek", 23, 19);
    obj.display();
}
```

The members `roll` and `age` cannot be assigned directly from `main`: `roll` is private and `age` is protected. They are set through the public function `input`, which

belongs to the class.

5.5 Single-level inheritance: syntax

A `bank` base class holds the customer data, and an `account` class inherits from it with the keyword `public`.

```
#include<iostream>
using namespace std;
class bank
{
    protected:
        string name, address;
        double amount, withdraw, deposit, balance;
    public:
        void input(string n, string a, double amt, double w, double d)
        {
            name     = n;
            address  = a;
            amount   = amt;
            withdraw = w;
            deposit  = d;
        }
};
class account : public bank
{
    public:
        void calc()
        {
            balance = amount + deposit - withdraw;
        }
        void output()
        {
            cout << "Customer Name = " << name << endl;
            cout << "Customer Address = " << address << endl;
            cout << "Net balance = " << amount << endl;
            cout << "Withdrawal = " << withdraw << endl;
            cout << "Deposit = " << deposit << endl;
            cout << "Remaining balance = " << balance << endl;
        }
};
```

The members of `bank` are `protected`, so the derived class `account` can use them directly, but code outside the two classes cannot.

The name of the first function is overwritten in the scan (`takeinput` written over `input`); it is read as `input`.

The notes declare `double calc()` but do not return a value. Here it is `void calc()`.

5.6 Special characters

Symbol	Name	Symbol	Name
+	plus	[	left bracket
*	asterisk (multiply)	]	right bracket
-	hyphen (minus)	,	comma
\	back slash	:	colon
/	forward slash	;	semicolon
<	less than	' '	single quotes
>	greater than	" "	double quotes
(	left parenthesis	{	left brace
		}	right brace

5.6.1 Escape characters

Escape	Effect
\b	move the cursor to the previous position of the current line
\a	produce a beep, an alert
\r	move the cursor to the beginning position of the current line
\n	move the cursor to the beginning of the next line
\0	null
\v	move the cursor to the next vertical position
\t	move the cursor to the next horizontal position

5.7 Delimiters

Delimiter	Use
( )	used for expressions
[ ]	arrays
{ }	used for statements (blocks)
;	end of a statement

5.8 Hierarchical inheritance: a worked programme

Task. Write a C++ programme that creates `student` and `employee` classes inheriting from `Person`, using hierarchical inheritance (one base class and two derived classes).

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

class Person
{
```

```
    string name, gender;
    int age;
    public:
        void getdata()
        {
            cout << "Name: ";    cin >> name;
            cout << "Age = ";    cin >> age;
            cout << "Gender = "; cin >> gender;
        }
        void display()
        {
            cout << "Name: "   << name   << endl;
            cout << "Age: "    << age    << endl;
            cout << "Gender: " << gender << endl;
        }
};

class student : public Person
{
    string institute, level;
    public:
        void getdata()
        {
            Person::getdata();
            cout << "Name of college/school: "; cin >> institute;
            cout << "Level: ";                  cin >> level;
        }
        void display()
        {
            Person::display();
            cout << "Name of college: " << institute << endl;
            cout << "Level: "           << level     << endl;
        }
};

class employee : public Person
{
    string company;
    float  salary;
    public:
        void getdata()
        {
            Person::getdata();
            cout << "Name of company: "; cin >> company;
            cout << "Salary: Rs. ";      cin >> salary;
        }
        void display()
        {
            Person::display();
            cout << "Name of company: " << company << endl;
            cout << "Salary: Rs. "      << salary  << endl;
        }
};

int main()
{
```

```
    student  s;
    employee e;
    cout << "Student" << endl;
    s.getdata();
    s.display();
    cout << "Employee" << endl;
    e.getdata();
    e.display();
    return 0;
}
```

This listing is a corrected version of the handwritten programme, which would not compile as written. Changes: `char` members for names became `string`; `cin » "gender"` became `cin » gender`; the missing `cin` statements for the institute and the company were added; `void::getdata` became `void getdata`; `Int` became `int`; missing semicolons were added; and `conio.h` and `getch()` were removed as they are not standard C++.

5.9 Hybrid inheritance in C++

In hybrid inheritance, class `B` inherits from `A`, and class `D` inherits from both `B` and `C`. This combines multilevel inheritance ($A \to B \to D$) with multiple inheritance ($B, C \to D$).

```
#include<iostream>
using namespace std;
int a, b, c, d, e;

class A
{
    public:
        void getab()
        {
            cout << "\nEnter a and b: ";
            cin >> a >> b;
        }
};

class B : public A
{
    public:
        void getc()
        {
            cout << "Enter c value: ";
            cin >> c;
        }
};

class C
{
    public:
        void getd()
        {
            cout << "Enter d value: ";
            cin >> d;
        }
};

class D : public B, public C
{
```

```
    public:
        void result()
        {
            getab();
            getc();
            getd();
            e = a + b + c + d;
            cout << "\n Addition is = " << e;
        }
};

int main()
{
    D d1;
    d1.result();
    return 0;
}
```

The handwritten version uses `void main()`, `clrscr()`, `getch()` and `conio.h` (Turbo C) and leaves out `using namespace std;`. These were replaced by standard C++.

The five types of inheritance

single level ✓ multiple ✓ multilevel ✓ hybrid ✓ hierarchical ✓

5.10 Constructors in object-oriented programming

In class-based OOP, a constructor is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set the required member variables.

```
#include<iostream>
class student
{
    public:
        int a, b;
        student(int x = 0, int y = 0)    // default constructor
        {
            a = x;
            b = y;
        }
};
int main()
{
}
```

The notes write `student(a=0, b=0)` without types and without a body. The version above is a working default constructor with two default arguments.

5.11 Destructors in object-oriented programming

In OOP, a destructor is a method that is invoked automatically when the object is destroyed. This can happen

- when its lifetime is bound to a scope and the execution leaves the scope;

- when it is embedded in another object whose lifetime ends; or
- when it was allocated dynamically and is released explicitly.

A destructor is declared with the name of the class prefixed by a tilde (~).

5.12 Hybrid inheritance: area and volume

A second example: `calcarea` and `calcvol` each inherit from a base class, and the class `display` inherits from both of them.

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

class dimensions                        // base class of calcarea
{
    protected:
        double length, width, area;
    public:
        void triangledim(double l, double b)
        {
            length = l;
            width  = b;
        }
};

class calcarea : public dimensions
{
    public:
        void calculate()
        {
            area = 0.5 * length * width;
        }
};

class sphere                            // base class of calcvol
{
    protected:
        double radius, volume;
    public:
        void takeinput(double r)
        {
            radius = r;
        }
};

class calcvol : public sphere
{
    public:
        void calculatevol()
        {
            volume = (4 * 3.14 * radius * radius * radius) / 3;
        }
};
```

```
class display : public calcarea, public calcvol
{
    public:
        void output()
        {
            cout << "Area = "   << area   << endl;
            cout << "Volume = " << volume << endl;
        }
};

int main()
{
    display obj;
    obj.triangledim(10, 5);
    obj.takeinput(15);
    obj.calculate();
    obj.calculatevol();
    obj.output();
    return 0;
}
```

The handwritten programme derives `calcvol` from a class `volume` that is not written on the page, and uses the names `area` and `volume` both for classes and for data members, which C++ does not allow. The base classes are named `dimensions` and `sphere` here, and `sphere` was added. The volume of a sphere is $\frac{4}{3}\pi r^3$; the notes have `radius*radius` (two factors), which is a mistake.

5.13 Polymorphism

Polymorphism has two forms: overloading and overriding.

5.13.1 Overloading

Function overloading means that several functions have the same name but different parameters (a different type or a different number of parameters). The compiler chooses the right one from the arguments of the call.

```
// C++ programme to illustrate function overloading
#include<iostream>
using namespace std;

// overloading functions
void test(int);
void test(float);
void test(int, float);

int main()
{
    int a = 5;
    float b = 5.5;

    // overloading functions with different types and numbers of
    parameters
    test(a);
    test(b);
    test(a, b);
    return 0;
```

```
}

// Method 1
void test(int var)
{
    cout << "Integer number: " << var << endl;
}

// Method 2
void test(float var)
{
    cout << "Float number: " << var << endl;
}

// Method 3
void test(int var1, float var2)
{
    cout << "Integer number: " << var1;
    cout << " and float number: " << var2;
}
```

The three calls print `Integer number: 5`, `Float number: 5.5` and `Integer number: 5 and float number: 5.5`.

5.13.2 Overriding

Function overriding means that a derived class gives a new working to a function that the base class already has. The base-class function must be declared `virtual` so that the version of the object that is really used is called.

```
// C++ programme to illustrate function overriding
#include<iostream>
using namespace std;

class BaseClass
{
    public:
        virtual void display()
        {
            cout << "\nThis is display() method of BaseClass";
        }
        void show()
        {
            cout << "\nThis is show() method of BaseClass";
        }
};

class DerivedClass : public BaseClass
{
    public:
        // overriding method: new working of the base class's display
    method
        void display()
        {
```

```
            cout << "\nThis is display() method of DerivedClass";
        }
};

// driver code
int main()
{
    DerivedClass dr;
    BaseClass &bs = dr;
    bs.display();      // calls the DerivedClass version (virtual)
    dr.show();         // calls the BaseClass version
}
```

In the handwritten code the derived function is written `Display()` with a capital D. That is a different function, so it does not override `display()`. With a lowercase `d`, `bs.display()` prints the derived-class message.

5.13.3 Overloading is compile-time polymorphism

Overloading is also called *compile-time polymorphism*, because the function to call is decided when the programme is compiled. Overriding is decided when the programme runs.

```
// C++ programme for function overloading
#include<bits/stdc++.h>
using namespace std;
```

This page of the notes is almost empty: only the comment, the header `bits/stdc++.h` and `using namespace std;` are written, and the rest of the programme is not on the page.

5.13.4 Another example of overloading: areas

The function `calc` is overloaded three times. One argument gives the area of a circle, two arguments the area of a rectangle, and three arguments the surface area of a cuboid.

```
#include<iostream>
using namespace std;
class calculator
{
    public:
        double result;
        void calc(double a)
        {
            result = 3.14 * a * a;
            cout << "Area of circle = " << result << endl;
        }
        void calc(double a, double b)
        {
            result = a * b;
            cout << "Area of rectangle = " << result << endl;
        }
        void calc(double a, double b, double c)
        {
            result = 2 * ((a * b) + (b * c) + (c * a));
            cout << "Area of cuboid = " << result << endl;
        }
};
int main()
```

```
{
    calculator obj;
    obj.calc(10);          // circle:    314
    obj.calc(17, 5);       // rectangle: 85
    obj.calc(5, 6, 3);     // cuboid:    126
}
```

For the cuboid, $2(ab+bc+ca)$ is its *surface* area.

5.13.5 Another example of overriding: square and circle

The class `circle` overrides the virtual function `calc` of the class `square`. It first calls the version of the base class with `square::calc(a)`, and then does its own calculation.

```
#include<iostream>
using namespace std;
class square
{
    public:
        double result;
        virtual void calc(double a)
        {
            result = a * a;
            cout << "Area of square = " << result << endl;
        }
};
class circle : public square
{
    public:
        void calc(double a)
        {
            square::calc(a);
            result = 3.14 * a * a;
            cout << "Area of circle = " << result << endl;
        }
};
int main()
{
    circle obj;
    obj.calc(15);     // prints 225, then 706.5
}
```

A Introduction to Computer Science (lecture notes)

A.1 Programming paradigms

These notes follow an introductory computer-science lecture (Jerry Cain, Stanford). Different languages support different programming paradigms, which matter in industry and research:

Language	Paradigm
C	procedure oriented
C++	object oriented (classes)
Assembly	machine-level instructions
Concurrent programming	several tasks at the same time
Scheme	functional programming
Python	good library support

The brace on the page groups C with both "object oriented" and "procedure oriented", and a Hindi word next to "paradigms" is not legible. The table is an interpretation of the list; the Scheme and Assembly descriptions are my additions.

A.1.1 Why concurrency matters: two ATMs

Suppose you use ATM 1 and your friend uses ATM 2. If both try to dispatch money from the same account at the same time, the account must still be updated correctly. Dealing with this kind of problem is part of concurrent programming.

A.2 Numbers inside the computer

The letter `A` has the ASCII value 65. In binary, $65 = 64 + 1 = 2^6 + 2^0$, so the pattern is

```
01000001
```

with the bit weights running from 2^6 down to 2^0 (the leftmost bit is 2^7 and is zero).