



CONTENTS

1	Tackling Difficult Problems: Positive Semidefinite Matrices	5
1.1	NP Problems	5
1.2	A Past Approach: Minimizing Errors in Neural Networks	6
1.3	Positive Definite and Semidefinite Matrices	6
1.4	Identifying and Constructing a Positive Semidefinite Matrix	8
1.5	The Max Cut Problem	9
1.6	The Max Cut Problem Solved in Python	11
2	Data Representation	13
2.1	Introduction to Text	13
2.1.1	Newlines and Carriage Returns	13
2.1.2	ASCII	13
2.1.3	UTF-8	13
2.2	Alphabets and Accents	14
2.3	Representing Natural Numbers	15
2.3.1	Base-10 (Decimal)	15
2.3.2	Base-2 (Binary)	17
2.3.3	Base-16 (Hexadecimal)	19
2.3.4	Base-8 (Octal)	20
3	Floating Point and Data Compression	23
3.1	Floating Point Arithmetic	23
3.1.1	Why Computers Cannot Count Perfectly	23
3.1.2	Binary and the Floating-Point System	25

3.1.3	The IEEE 754 Standard	29
3.2	Data Compression and Decompression	32
3.2.1	Entropy	32
3.2.2	Entropy and Compression	32
4	Introduction to C	35
4.1	Compiled vs. Interpreted	35
4.2	Your First C Program	36
4.3	Types and Variables	36
4.4	Printing with printf	37
4.5	Conditionals and Loops	38
4.5.1	if / else	38
4.5.2	for and while loops	38
4.6	Functions	39
4.7	The Stack: Blackboards for Functions	40
4.7.1	Parameters are copies	41
4.8	Addresses and Pointers	41
4.8.1	Reading and writing through a pointer	42
4.8.2	NULL pointers	42
4.9	Pass-by-Reference	43
4.10	Structs: Grouping Related Data	44
4.11	The Heap	46
4.11.1	The arrow operator	46
4.11.2	Releasing heap memory	46
4.11.3	A complete heap example	47
4.11.4	Heap arrays	48
4.12	What Python Objects Actually Are	48
4.13	Exercises	48
5	Linked Lists and C++ transition	53
5.1	From C to C++	53
5.2	Linked Lists	53
5.3	Doubly Linked Lists	55
5.4	Exercises	58
6	Bitmaps	61
6.1	Bitmap Structure	62
6.1.1	Bitmap Headers	62
6.1.2	Padding	63

	3
6.2	Creating a Bitmap with C++ 69
6.3	Connection to transformations 74
6.4	Summary 76
7	Stacks and Queues 77
7.1	Stacks 77
7.1.1	Array-Based Stack 78
7.1.2	Linked-List Stack 79
7.1.3	Array vs. Linked: Which One? 80
7.2	Queues 81
7.2.1	Array-Based Queue 82
7.2.2	Linked-List Queue 83
A	Answers to Exercises 85
Index	97

Tackling Difficult Problems: Positive Semidefinite Matrices

With all the computing power available today, you would think no problem would be too difficult to tackle. However, this is not the case. Many important problems have solutions that are easy to check but, as far as anyone knows, extremely hard to find. Brute force would take longer than the age of the universe. For these problems, the practical goal is an algorithm that finds a *good* answer, guaranteed to be close to the best, in a reasonable amount of time.

As you learn more about optimization techniques, you will come across many efficiency algorithms that have been used throughout the years. In the 1990s, mathematicians discovered that many such problems could be approximated remarkably well using positive semidefinite matrices. Today, *semidefinite programming* is a whole field built on them. In this chapter, you will meet the hard problems, the matrices, and one famous example, Max Cut, solved in Python.

1.1 NP Problems

Computer scientists call problems that can be solved quickly (in *polynomial time*) **P** problems: sorting a list, multiplying matrices, finding the shortest route on a map. **NP** problems are those whose solutions can be *checked* quickly. Every P problem is in NP, but some NP problems, called **NP-complete**, have no known fast algorithm at all. Sudoku, generalized to giant $n^2 \times n^2$ grids, is one: checking a filled-in grid is easy, but no one knows a fast way to fill one in. Whether every NP problem is secretly a P problem is the famous P versus NP question.

NP problems show up in many other situations, such as:

- Designing robust communication networks
- Scheduling tasks without conflicts
- Managing supply chains
- Detecting patterns in biological networks
- Figuring out subgroups in social networks
- Predicting the structure of proteins

For each of these, think of large scale problems for which there are many variables. A cloud computing company that provides AI services to thousands of clients must be able to schedule tasks efficiently and in a timely manner, as well as manage the power needed for the computers and cooling the data center. Supply chain management is crucial to figuring out how to pick up, transport, and distribute goods to help provide disaster relief. Understanding protein structure is important for designing drugs that can tackle specific conditions. For large versions of these problems, we settle for solutions that are provably close to the best.

1.2 A Past Approach: Minimizing Errors in Neural Networks

When neural networks were first being developed to recognize things (faces, letters, music, and so on), the goal was to calculate weights between network nodes that would minimize the recognition error. The error space could be visualized as a surface of valleys and hills. The lowest point would have the least error. The idea behind the iterative weight calculations for training the network was to descend down the gradient until reaching the low point. Without getting into the mathematics, you can see by looking at this figure that there are two valleys. Some neural network training ended at a low point, but not the lowest point. Would it not be great if you could set up an optimization problem whose surface has only one valley, so you were guaranteed to land at the bottom? Semidefinite programming does exactly that for the problems it can describe.

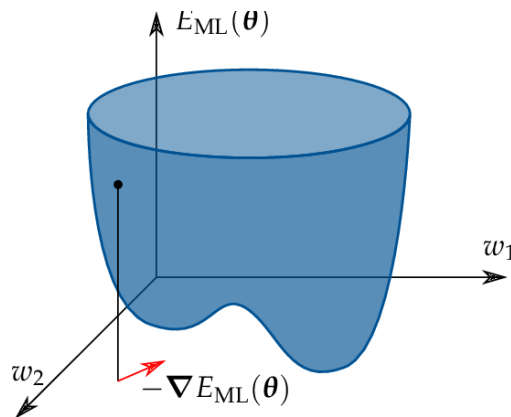


Figure 1.1: A neural network error surface with two valleys. Gradient descent can converge to a local minimum rather than the global minimum.

1.3 Positive Definite and Semidefinite Matrices

Unlike matrices that are defined by their content (such as the identity, zero, and diagonal matrices), positive definite and semidefinite matrices are defined by how they behave. Start with numbers. For a number a , the expression $x \cdot a \cdot x = ax^2$ is positive for every nonzero x exactly when $a > 0$. If $a = 0$, it is never negative, but it can be 0.

Positive definite matrices generalize the first case: $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ for every nonzero vector $\mathbf{x}$. Positive semidefinite matrices generalize the second: $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$ for every $\mathbf{x}$.

Why care? Think of $\mathbf{x}^T \mathbf{A} \mathbf{x}$ as the height of a surface over the point $\mathbf{x}$. If $\mathbf{A}$ is positive definite, the surface is a single bowl, like $x^2 + y^2$: it has one valley and nothing else. Optimization problems built from positive semidefinite matrices are *convex*: every valley is the lowest valley, so a computer that heads downhill cannot get stuck in the wrong one.

Let us look at the formal definition:

A symmetric matrix $\mathbf{A}$ is **positive definite** if

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0 \text{ for every } \mathbf{x} \neq \mathbf{0}$$

and **positive semidefinite** if

$$\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0 \text{ for every } \mathbf{x}$$

There is an easy test: a symmetric matrix is positive semidefinite exactly when all its eigenvalues are ≥ 0 , and positive definite exactly when they are all > 0 .

Take the triplet (a, b, c) and the symmetric matrix:

$$\begin{bmatrix} 1 & a & b \\ a & 1 & c \\ b & c & 1 \end{bmatrix}$$

For what values of a, b, c is this matrix positive semidefinite? If you iterate through all possible combinations of a, b, c , then compute the eigenvalues for each matrix, you will find that some (like $0, 0, 0$) result in a positive semidefinite matrix and some (like $2, 2, 2$) are not positive semidefinite. If you plot the set of triplets that result in a semidefinite matrix, you will see an ellipsope. This shape describes exactly the choices that give a valid semidefinite matrix, so optimizing over it is easier.

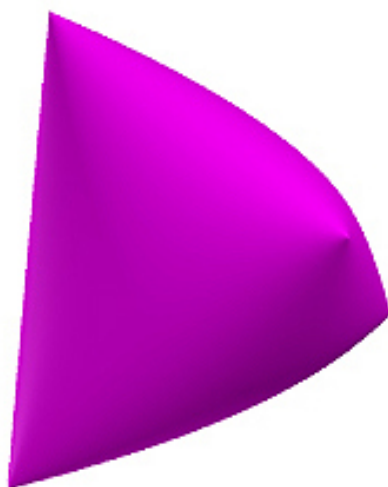


Figure 1.2: Elliptope visualized.

1.4 Identifying and Constructing a Positive Semidefinite Matrix

In the last section, you saw that being symmetric does not guarantee a positive semidefinite matrix. You also saw that a matrix of positive values does not guarantee a positive semidefinite matrix. One way to check for positive semidefinite is to calculate the eigenvalues, which you learned in the previous chapters.

A surefire way to construct a positive semidefinite matrix is:

$$AA^T$$

Worked example: Is $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ positive definite?

Solution: The characteristic equation is $(2 - \lambda)^2 - 1 = 0$, which is $\lambda^2 - 4\lambda + 3 = 0$. So $\lambda = 1$ or $\lambda = 3$. Both eigenvalues are positive, so the matrix is positive definite.

Exercise 1 Figuring out if a matrix is positive semidefinite

Is this matrix positive definite? Show your work.

$$\begin{bmatrix} 2 & 2 \\ 2 & 0 \end{bmatrix}$$

Working Space

Answer on Page 85

Exercise 2 Creating a positive semidefinite matrix

Using any 3 by 3 matrix, create a positive semidefinite matrix. Then show it is positive semidefinite by calculating its eigenvalues. You can either compute this by hand or using Python. In either case, show your work.

Working Space

Answer on Page 85

1.5 The Max Cut Problem

A famous NP-complete problem is Max Cut. Given a graph of interconnected nodes, cut the graph to create two sets of nodes, such that the cut goes through as many edges as possible. (You cannot cut an edge more than once.) Max Cut is important for binary classification in machine learning, circuit design, statistical physics, and more. No fast algorithm is known that always gives an exact solution. (Finding such an algorithm would settle the P versus NP question, which carries a prize of one million dollars from the Clay Mathematics Institute.) Instead, you will see how to approximate a solution to this problem using a positive semidefinite matrix and a technique developed by mathematicians Michel Goemans and David Williamson.

You will not see all the complete details here, as this section is meant to be a quick introduction to how you can apply positive semidefinite matrices.

Take this simple graph of five nodes and six edges. Each node in the graph will take on one of two values (1 or -1) to show which set they fall into after a cut is made. For any two connected nodes, $x_i \cdot x_j = 1$ if $x_i = x_j$ and -1 otherwise.

If you randomly assign 1 and -1 to the nodes, on average half of the edges are cut. By using semidefinite programming, you can guarantee a cut that is at least about 0.878 of the best possible cut, on average.

The Goemans-Williamson technique can be used for any optimization problem where the variables take on the values of 1 and -1 .

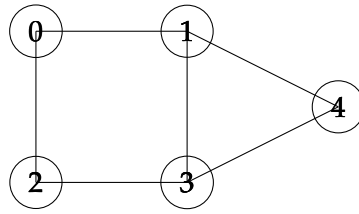


Figure 1.3: MaxCut algorithm visualized.

As a list of edges the graph is:

$$\text{edges} = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 3), (3, 4)]$$

The optimization problem can be formulated as:

$$\text{Max} \sum_{\text{edges}(i,j)} \frac{1 - x_i x_j}{2}$$

for

$$x_i \in \{-1, 1\}$$

However, instead of allowing x_i to be scalar, Goemans-Williamson defines x_i as unit vectors.

$$x_i \in \mathbb{R}^n, \quad \|x_i\| = 1$$

and that makes the optimization equation:

$$\text{Max} \sum_{\text{edges}(i,j)} \frac{1 - x_i^T x_j}{2}$$

It is this “relaxation” that gets us to a semidefinite matrix, because we can now rewrite the problem as a positive semidefinite matrix:

$$X = [x_i^T x_j]_{i,j}$$

Python has a module for solving optimization problems. Using it, you will get the best matrix X . To recover the unit vectors, factor X as $X = VV^T$; the rows of V are the vectors $x_1, \dots, x_n$. (The matrix square root of X is one such V .) Next we need to go from unit vectors to scalars, using a process called rounding.

$$x_i \in \mathbb{R}^n \rightarrow x_i \in \{-1, 1\}$$

Goemans-Williamson leveraged the fact that the end point of a unit vector is on a sphere.

They generated a random plane to bisect the sphere. A vector on one side of the plane is assigned the value of 1 and a vector on the other side a value of -1 .

You can then assign the scalar values to the nodes and make the cut accordingly. The best possible cut for this graph is 5, because the triangle of nodes 1, 3, and 4 can contribute at most two cut edges. The rounding step can find a cut of this size, as shown.

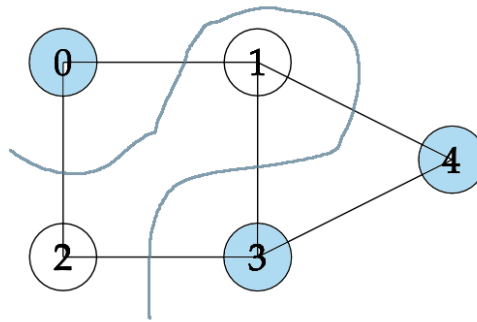


Figure 1.4: The result of the MaxCut.

1.6 The Max Cut Problem Solved in Python

This code uses the `cvxpy` and `scipy` modules, which you can install with `pip install cvxpy scipy`.

```

1 import numpy as np
2 from scipy.linalg import sqrtm
3 import cvxpy as cp # a module for solving optimization problems
4
5 edges = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 3), (3, 4)]
6 n = 5
7
8 # X is the matrix of dot products x_i . x_j; it must be positive semidefinite
9 X = cp.Variable((n, n), symmetric=True)
10 constraints = [X >= 0]
11 # Unit vectors: each x_i . x_i = 1
12 constraints += [X[i, i] == 1 for i in range(n)]
13
14 objective = sum(0.5 * (1 - X[i, j]) for (i, j) in edges)
15 prob = cp.Problem(cp.Maximize(objective), constraints)
16 print("Relaxed value:", round(prob.solve(), 3))
17
18 # Recover the vectors: the rows of the square root of X
19 vectors = np.real(sqrtm(X.value))
20
21 # Round: slice the sphere with a random plane through the origin
22 rng = np.random.default_rng(1)
23 normal = rng.standard_normal(n)

```

```
24 side = np.sign(vectors @ normal)
25 print("Sides:", side)
26
27 cut = sum(1 for (i, j) in edges if side[i] != side[j])
28 print("Edges cut:", cut)
```

You should see something like:

```
1 Relaxed value: 5.185
2 Sides: [-1.  1.  1. -1.  1.]
3 Edges cut: 5
```

The relaxed value, 5.185, is more than any real cut can achieve, because unit vectors have more freedom than ± 1 . That makes it a useful ceiling: no cut can beat 5.185, and the rounding found a cut of 5, the best possible. (A different random plane can give a different, sometimes smaller, cut. In practice you try several planes and keep the best.)

Data Representation

2.1 Introduction to Text

Everything on your computer, from your essays and photos to the songs on your playlist, is stored as a long list of 0s and 1s. This chapter shows how text and whole numbers become bits.

In computer systems, text is represented in files as a sequence of characters, each of which corresponds to a specific number, known as a character code. These character codes are then stored in the file as binary data.

2.1.1 Newlines and Carriage Returns

Two of the character codes that have special meanings are the newline (often represented as `'\n'`) and the carriage return (often represented as `'\r'`).

The newline character signifies the end of a line of text and the beginning of a new one. If you parse a text file with a Python script, it will see your 'enter' key press as a new line symbol. The carriage return character moves the cursor to the beginning of the line. The use of these characters can vary between operating systems. Unix-based systems (like Linux and MacOS) use the newline character to indicate the end of a line, while Windows systems use a combination of a carriage return and a newline (`'\r\n'`).

2.1.2 ASCII

The American Standard Code for Information Interchange (ASCII) is one of the earliest character encodings. It uses 7 bits to represent each character, allowing it to define up to $2^7 = 128$ different characters. These include the English alphabet (in both lower and upper cases), digits, punctuation symbols, control characters (like newline and carriage return), and some other symbols.

2.1.3 UTF-8

UTF-8 (8-bit Unicode Transformation Format) is a variable-width character encoding that can represent every character in the Unicode standard, yet remains backward-compatible

with ASCII. For the ASCII range (0-127), UTF-8 is identical to ASCII. However, it can use additional bytes (up to 4 bytes in total) to represent characters that are not included in ASCII, such as characters from other languages, emojis, and many other symbols. This has made UTF-8 a widely used encoding in many modern systems.

2.2 Alphabets and Accents

Software developers often encounter text from diverse languages and cultures. As a developer, it is crucial to have a solid understanding of alphabets and accents to effectively handle and process multilingual text. Alphabets, the building blocks of written language, vary widely across different nations and regions. Meanwhile, accents, diacritical marks, and other phonetic notations play a crucial role in conveying the correct pronunciation and meaning of words.

Alphabets are sets of letters or symbols used to represent the sounds of a language. While the Latin alphabet is widely used in many Western languages, numerous other alphabets exist, such as Cyrillic, Greek, Arabic, Devanagari, and Chinese characters. Each alphabet has its own unique set of letters, often organized in a specific order, and may include uppercase and lowercase variations.

Accents and diacritical marks are additional symbols added to letters to modify their pronunciation or provide additional phonetic information. Accents can appear above, below, or beside a letter, and they can change the sound, stress, or intonation of a word. For example, in French, the acute accent in *é* marks the sound /e/, as in *café*, while the grave accent in *è* marks the sound /è/, as in *père*.

When working with multilingual text, it is essential to consider various factors:

1. **Character encoding:** Different alphabets require specific character encodings to represent their letters digitally. Commonly used character encodings include ASCII, Unicode, and UTF-8. Understanding the appropriate encoding for each language is crucial to ensure proper text rendering and avoid data corruption.
2. **Text input and validation:** Building applications that handle user input requires robust text validation. Account for the diverse set of characters and possible accents that may appear in user-generated content. Implement proper validation and sanitization mechanisms to handle text input securely.
3. **Sorting and collation:** Sorting text from different languages involves considering the specific rules and conventions of each alphabet. Some languages may have unique sorting orders, while others ignore accents or diacritics when determining the order of words. Take into account the appropriate sorting and collation algorithms to ensure consistent and accurate results.
4. **Search and indexing:** Efficient search and indexing systems must be capable of handling

multilingual text. Consider appropriate text normalization techniques to account for different character representations (e.g., case-insensitive matching, ignoring accents), enabling users to find relevant content across languages and variations in spelling or diacritics.

Exercise 3 Counting Bytes

In UTF-8, ASCII characters take 1 byte, accented Latin letters like é take 2 bytes, most Chinese characters take 3 bytes, and emoji take 4 bytes. How many bytes does the text *café* take? What about a message of 10 emoji?

Working Space

Answer on Page 85

2.3 Representing Natural Numbers

Natural numbers are positive whole numbers, such as 1, 2, 3, and so on. -5 is not a natural number. π is not a natural number. $\frac{1}{2}$ is not a natural number.

2.3.1 Base-10 (Decimal)

You are used to seeing the natural numbers represented in a **base-10** *Hindu-Arabic* or the *Decimal* numeral system. That is, when you see 2531 you think “2 thousands, 5 hundreds, 3 tens, and 1 one.” Rewritten, this is:

$$2 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0$$

The *radix* is the number of different digits a base uses before it needs another place. For example, values 0, 1, 2, $\dots$, 8, 9 are all the values for base-10. Once 1 is added to 9, we need to increase the amount of digits in a number.

In any Hindu-Arabic system, the location of the digits is meaningful: 101 is different from 110; the position of a number indicates its magnitude. Here are those numbers in Roman numerals: CI and CX. Roman numerals did not have a symbol for zero at all.

The Hindu-Arabic system gave us really straightforward algorithms for addition and multiplication. For addition, you memorized the following table:

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

When you multiplied two number together, you simply multiplied each pair of digits. 254×26 might look like this:

	2	5	4	
$\times$	2	6		
		2	4	6×4
	3	0		6×5
1	2			6×2
		8		2×4
	1	0		2×5
+	4			2×2
	6	6	0	4

For multiplication, you memorized this table:

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

2.3.2 Base-2 (Binary)

Binary, on the other hand, has only 2 digits it can have: 0 or 1. This kind of number system is often used for electrical and computer systems because the values can only be **on** or **off**.

Each digit in a binary number represents a power of 2, starting from the rightmost digit (the least significant bit, or LSB). For example:

$$\dots + 2^3 + 2^2 + 2^1 + 2^0$$

For example,

$$1011_2 = (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 8 + 0 + 2 + 1 = 11_{10}$$

¹

Because the radix is 2, the numbers increase differently:

0 : 0

1 : 1

2 : 10

3 : 11

4 : 100 $\rightarrow (2^2 \cdot 1) + 0 + 0 = 4$

5 : 101

6 : 110

7 : 111

8 : 1000

...

15 : 1111 ...

¹The subscripts represent the radix or base.

Conversion from Decimal to Binary

To convert 13_{10} to binary, repeatedly divide by 2 and record the remainders until the divisor reaches 0 or 1:

$$\begin{aligned}13 \div 2 &= 6 \text{ remainder } 1 \\6 \div 2 &= 3 \text{ remainder } 0 \\3 \div 2 &= 1 \text{ remainder } 1 \\1 \div 2 &= 0 \text{ remainder } 1\end{aligned}$$

Reading from bottom to top, $13_{10} = 1101_2$.

Bits and Bytes

A singular binary digit (0 or 1) is called a *bit*. A lightbulb, switch, or any sort of *Boolean* value (true or false) can be represented as a bit.

Eight bits form a *byte*, which is a common unit of storage in computer systems:

$$1 \text{ byte} = 8 \text{ bits}$$

For example, the binary sequence 01001000 represents one byte of data.

We will not go into the depths of it here, but here are the basics of computer architecture. Memory is a long list of bytes. Programs group these bytes into data types:

Common Data Types in Memory

Data Type	Size	Bit Length	Typical Use
Character	1 byte	8 bits	ASCII characters, small integers
Integer	4 bytes	32 bits	Whole numbers
Double	8 bytes	64 bits	Decimal (floating-point) values

Strings are formed by listing characters in consecutive memory addresses such that they are marked by a starting memory address and ending memory address.

8-bit binary numbers can be used to signify positive or negative numbers. The most significant bit (MSB) will alter sign, causing the range to be $[-2^7, 2^7 - 1]$. This scheme is called *two's complement*. To negate a number, invert every bit and add 1. To read a byte whose MSB is 1, count the MSB as -128 instead of $+128$ and add the rest as usual:

$$-85_{10} = -1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 10101011$$

Binary can be easily converted to *Hexadecimal*.

2.3.3 Base-16 (Hexadecimal)

Hexadecimal (from the Greek *hexa*, meaning six, and the Latin *decem*, meaning ten, which add up to 16) is a representation of number with a radix of 16. Digits are represented through numeric numbers 0-9 and A-F.

Counting to 16 in hex goes as follows:

1
2
3
...
9
A (decimal value 10)
B (decimal value 11)
C (decimal value 12)
D (decimal value 13)
E (decimal value 14)
F (decimal value 15)
10 (decimal value 16)

Converting between binary and hexadecimal is easy because $16 = 2^4$, so each hex digit matches exactly four binary digits.

It takes 4 binary digits to represent 1 hex digit: $15 = F$

For example,

$$0x47 = 0100\ 0111_2$$

$$0xBD = 1011\ 1101_2$$

Hex Color Codes

Hex colors are represented by 6 hexadecimal digits, grouped as two each for red, green, and blue (RGB). Each pair ranges from 00 (0 in decimal) to FF (255 in decimal), allowing 256 shades per color channel. For example:

#FF0000 = Red, #00FF00 = Green, #0000FF = Blue

By mixing values, you can create millions of unique colors, e.g.:

#FFA500 = Orange, #800080 = Purple, #FFFFFF = White, #000000 = Black

There are $256^3 = 16,777,216$ possible color combinations in the RGB hex system. The higher the red, green, or blue values are (closer to 255), the brighter the component becomes and the overall color shifts closer to white. Conversely, the lower the values are (closer to 0), the darker the component becomes and the overall color shifts closer to black.

Included in your digital resources are multiple hex videos, as well as an interactive hex color guessing game.

2.3.4 Base-8 (Octal)

Octal is another representation of radix 8. Digits range from 0 to 7, and a new digit is added when the count passes 7.

Just as in the above systems, octal increases by the radix — powers of 8. For example,

$$135_8 = (1 \times 8^2) + (3 \times 8^1) + (5 \times 8^0) = 64 + 24 + 5 = 93_{10}$$

Why Octal Matters

Octal was historically important in computing because early computers often worked with word sizes that were multiples of 3 bits (such as 12, 24, or 36). Since each octal digit corresponds exactly to 3 binary digits, it was a convenient shorthand for binary values before hexadecimal became more widespread. For example:

$$110101_2 = 65_8$$

Even though modern systems mostly use hexadecimal, octal is still used in some contexts, such as Unix file permissions (e.g., `chmod 755`).

Whole numbers and text are only part of the story. The next chapter, Floating Point and Data Compression, shows how computers store numbers with decimal points, and how files can be compressed.

Floating Point and Data Compression

Type `0.1 + 0.2` into Python, and it answers `0.30000000000000004`. Is Python broken? No: it is doing exactly what it was designed to do with a finite number of bits. In the Data Representation chapter, you saw how text and whole numbers become 0s and 1s. This chapter tackles the harder cases: numbers with decimal points, which can never all be stored exactly, and files that can be squeezed into fewer bits than they started with.

3.1 Floating Point Arithmetic

3.1.1 Why Computers Cannot Count Perfectly

Computers are known to be very precise. Modern processors can perform billions of calculations per second, so it is not intuitive to realize they have this persistent limitation: they cannot store most numbers exactly.

Let us think about what it would actually take to store a number like $\frac{1}{3}$. You probably know its decimal expansion:

$$\frac{1}{3} = 0.3333333 \dots$$

The digits go on forever. Now imagine you are a computer with a fixed amount of memory. You cannot store infinitely many digits. So you store 0.333 or 0.3333333, depending on how much space you have, and you accept that the result is not exactly right. That gap between the number you wanted and the number you actually stored is called *approximation error*, and it never fully goes away.

This is not just a problem for fractions with repeating decimals. Irrational numbers like $\sqrt{2} = 1.41421356 \dots$ and transcendental numbers like $\pi = 3.14159265 \dots$ also have infinite, non-repeating decimal expansions. Every one of them must be approximated when stored in a computer.

To handle this systematically, mathematicians and engineers developed the *floating-point system*. Before we look at how it works, it helps to understand the two core reasons why any finite system will always fall short of the real number line.

The Number Line is Continuous and Unbounded

Between any two real numbers, no matter how close together, there are infinitely many others. Pick any two decimals you like. For example, if you picked 1.00 and 1.01, you can always find a number between them: 1.005, 1.0051, 1.00500001, and so on without end. This is referred to as *continuity*. A computer's number system cannot be continuous. It stores numbers using a fixed string of digits, which means it can only represent finitely many values. Real numbers have to be rounded to the nearest representable value.

The real numbers stretch infinitely in both directions. There is no largest real number and no smallest positive real number. A computer's memory is finite, so its number system must have both a largest and a smallest representable value. Numbers too large to store cause a problem called *overflow*. Numbers too small to distinguish from zero cause *underflow*.

Exercise 4 Overflow and Underflow

Suppose a floating-point system can only represent numbers between 10^{-4} and 10^8 .

Working Space

1. Planck's constant is approximately 6.63×10^{-34} J·s. Does it cause underflow, overflow, or neither in this system?
2. The speed of light is approximately 3.00×10^8 m/s. Does it cause underflow, overflow, or neither in this system?
3. Give one example of a number that would cause overflow.

Answer on Page 85

So what can we do?

We cannot fix these limitations, because they are baked into the nature of finite computation. What we can do is design the floating-point system carefully so that the errors it introduces are small, predictable, and manageable. That is exactly what engineers and mathematicians have spent decades working out, and it is what the rest of this chapter is

about!

The key insight is that a floating-point number is not just a string of digits. A floating-point number has three parts: a *sign*, a *significand* (sometimes called the mantissa), and an *exponent*.

3.1.2 Binary and the Floating-Point System

In the previous section you saw that computers cannot store most numbers exactly. Now the natural question is: how do they store numbers at all? The answer starts at the hardware level, with the smallest possible piece of information a computer can hold.

Binary System

Every process inside a CPU comes down to the flow of electrical current through tiny switches called *transistors*. A transistor is either on (carrying current) or off (not carrying current). That gives us exactly two states, which we write as 0 and 1. This is why computers use *binary*: it is the only number system that hardware can physically implement with a simple on/off switch.

A single binary digit is called a *bit*. Eight bits grouped together form a *byte*. You have probably seen units like megabyte (MB) or gigabyte (GB) before. One megabyte is 10^6 bytes, and one gigabyte is 10^9 bytes. A modern smartphone holds tens of billions of bits.

Exercise 5 Bits and bytes

Working Space

1. A file is 4 megabytes. How many bytes is that? How many bits?
2. A hard drive stores 500 gigabytes. How many bits can it hold? Write your answer using scientific notation.
3. A single bit can be 0 or 1. Two bits together can be 00, 01, 10, or 11, which are four possible values. How many distinct values can 8 bits (one byte)

represent? How about 64 bits?

Answer on Page 85

Floating-Point Number

Recall that a floating-point number has three parts: a *sign*, a *significand* (sometimes called the mantissa), and an *exponent*.

Before we write any formulas, let us see why all three parts are necessary with a base-10 analogy.

Consider the number -0.001234 . Written in scientific notation it is -1.234×10^{-3} . Three pieces of information do all the work:

- The **sign** ($-$) tells us the number is negative.
- The **significand** (1.234) holds the meaningful digits.
- The **exponent** (-3) tells us how large or small the number is by shifting the decimal point.

Now watch what the exponent does when we fix the significand at 0.1234 and vary it in base 10:

Float	Value
0.1234×10^{-2}	0.001234
0.1234×10^1	1.234
0.1234×10^4	1234
0.1234×10^8	12,340,000

The decimal point *floats*, which means it can sit anywhere, controlled entirely by the exponent. That is why these are called floating-point numbers. The significand always carries the same digits; only the scale changes.

Exercise 6 The floating point in action

The significand is fixed at 0.5050 and the base is 10. Fill in the value represented

Working Space

by each float.

Float	Value
0.5050×10^0	
0.5050×10^2	
0.5050×10^{-1}	
0.5050×10^5	

Now answer: if you want to represent 505,000,000 using the same significand, what exponent do you need?

[Answer on Page 86](#)

Floating-Point System

A floating-point system is defined by four numbers, written (β, p, m, M) . Every float in that system looks like this:

$$\pm (0.d_1 d_2 \cdots d_p)_\beta \times \beta^e$$

This looks intimidating at first, but each symbol has a simple job. Let us unpack them one at a time.

β represents the *base*. This is the number that the exponent is a power of. In everyday life we use $\beta = 10$. Computers use $\beta = 2$ (binary).

p represents the *precision*. This is how many digits the significand gets to use. The significand $0.d_1 d_2 \cdots d_p$ has exactly p digits, each chosen from $\{0, 1, \dots, \beta - 1\}$. More digits means more precision, that is, you can represent numbers more accurately. Fewer digits means faster arithmetic but more rounding error.

e represents the *exponent*. This shifts the significand left or right. The exponent is not free to be any integer; it is capped between a minimum m and a maximum M . Those bounds set the range of the system, i.e., how large and how small the representable numbers can be.

m and M represents the *exponent bounds*. The exponent must satisfy $m \leq e \leq M$. Together

they control overflow and underflow. If a result needs an exponent below m , underflow occurs. If it needs an exponent above M , overflow occurs.

We say the system is *parametrized* by (β, p, m, M) . Changing any one of the four gives a new floating-point system with different capabilities and different limitations.

Exercise 7 Reading the parameters

A toy floating-point system has parameters $(\beta, p, m, M) = (10, 4, -3, 4)$.

Working Space

1. What base does this system use?
2. How many significant digits does it carry?
3. What are the smallest and largest allowed exponents?
4. Can this system represent 12,345? Try writing 12,345 in the form $0.d_1d_2d_3d_4 \times 10^e$ and check whether the exponent is within bounds.
5. Can it represent 0.00001? Show your reasoning.
6. Write two numbers that *can* be represented exactly, and one that causes overflow.

Answer on Page 86

Connecting to Binary Floating-Point

All of the above applies to a computer's binary floating-point system, with $\beta = 2$ instead of $\beta = 10$. The significand digits $d_1, d_2, \dots, d_p$ are now bits: each one is either 0 or 1. The exponent is a power of 2.

So a binary float looks like:

$$\pm (0.d_1 d_2 \cdots d_p)_2 \times 2^e, \quad d_i \in \{0, 1\}, \quad m \leq e \leq M$$

The specific choices of p , m , and M define how much memory is used and how accurately numbers are stored. The international standard that fixes these choices for every modern computer is called *IEEE 754*, and it is the subject of the next section.

3.1.3 The IEEE 754 Standard

In 1985, after years of debate, the Institute of Electrical and Electronics Engineers published the *IEEE 754 standard*. It was accepted internationally in 1989, and it is still the standard used in virtually every computer, phone, and calculator you use today.

Floats

If we are going to store a float in a fixed number of bits, how do we split those bits among the three parts: sign, significand, and exponent?

For the sign, there are only two options, positive and negative, so a single bit handles it: 0 for positive, 1 for negative.

That leaves $N - 1$ bits to split between the significand and the exponent. IEEE 754 defines two standard ways to do this split, called *single precision* and *double precision*.

	Total bits	Stored significand bits	Exponent bits
Single precision	32	23	8
Double precision	64	52	11

In both cases the layout is the same (Figure 3.1). The sign occupies the first bit, the exponent fills the next bits, and the significand takes the remaining p bits. Double precision uses twice as many bits in total, which gives it both greater precision and a wider range of representable exponents.

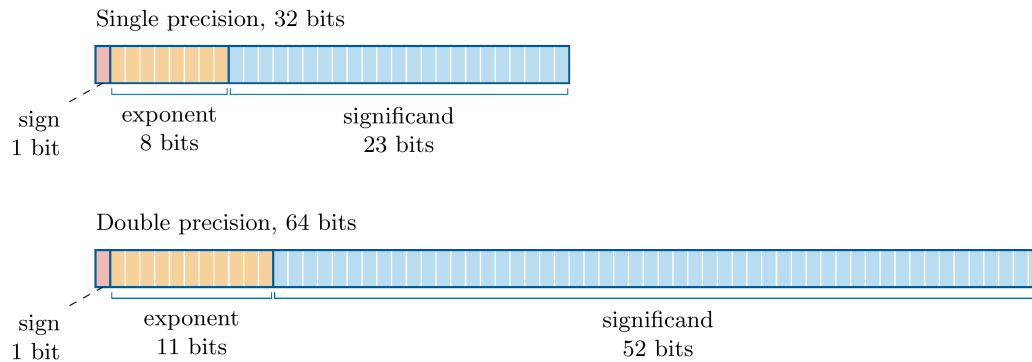


Figure 3.1: IEEE 754 bit layouts. Each cell is one bit, and the bits are stored in the order sign, exponent, significand.

For single precision, the 8 exponent bits can represent integers up to $M = 127$, so the largest float in single precision is roughly 10^{38} . Double precision is the standard used in modern computers.

Exercise 8 Bit Allocation

A floating-point system uses $N = 16$ bits total. The sign takes 1 bit and the significand takes 10 bits.

1. How many bits are left for the exponent?
2. Five bits give $2^5 = 32$ different exponents. If they represent the integers from -15 to 16 , what is the largest exponent M ?
3. Would you expect this system to be more or less precise than single precision? Why?

Working Space

Answer on Page 86

Machine Epsilon

The *machine epsilon* is commonly written as u . It measures the largest relative gap between a real number and its nearest float. In other words, it tells you the worst-case precision of your system.

For a system that rounds to the nearest float, machine epsilon is

$$u = \frac{1}{2}\beta^{1-p},$$

where β is the base and p is the precision. In IEEE 754, every nonzero binary significand starts with a 1, so that bit is not stored: double precision keeps 52 bits but has precision $p = 53$ (single precision: 23 stored, $p = 24$). For double precision, $\beta = 2$ and $p = 53$, giving $u = 2^{-53} \approx 1.11 \times 10^{-16}$. That means any real number is stored with a relative error no worse than about 1.1×10^{-16} : roughly 16 significant decimal digits of accuracy.

If you have access to Python, you can check your machine epsilon directly (`()`):

```
1 import sys
2 print(sys.float_info.epsilon) # double precision machine epsilon
```

Every machine running IEEE 754 double precision should return the same value: `2.220446049250313e-16`. This is twice the value of u above: Python reports the gap between 1 and the next float, 2^{-52} , while u is half a gap, because rounding to the nearest float is never off by more than half the gap.

Exercise 9 Machine Epsilon

A base-10 floating-point system has precision $p = 6$ and uses rounding.

1. Compute machine epsilon u for this system.
2. A real number x is stored as a float $\tilde{x}$. Using your answer above, what is the worst-case relative error $\frac{|\tilde{x}-x|}{|x|}$?
3. If p were doubled to $p = 12$, how would u change?

Working Space

Answer on Page 87

Good to know: IEEE 754 has been updated since 1985. The current standard includes *quadruple precision* (128 bits) for applications that need the highest accuracy. This includes physics simulations and cryptographic computations. Whole numbers can also be stored in integer types, such as `int32` and `int64`. Use them when you only need whole numbers, since integer arithmetic avoids the rounding error of floating-point arithmetic.

The existence of multiple precisions is a reminder that there is always a trade-off. More bits means more accuracy and a wider range, but also more memory and slower computation. Choosing the right precision for a given problem is the real engineering decision.

3.2 Data Compression and Decompression

Data compression and decompression are fundamental techniques used in modern computing, enabling efficient storage and transmission of data. The concept of entropy, borrowed from the field of information theory, plays a crucial role in determining the compression rate.

Data compression is the process of reducing the amount of data needed to represent a particular set of information. The two main types of data compression are lossless and lossy. Lossless compression ensures that the original data can be perfectly reconstructed from the compressed data, whereas lossy compression allows some loss of data for more significant compression rates. Decompression is the reverse process of compression, reconstructing the original data from the compressed format.

3.2.1 Entropy

In information theory, entropy measures the unpredictability or randomness of information content. More specifically, it quantifies the expected value of the information contained in a message. Lower entropy implies less randomness and more repetitiveness, which in turn means the data can be compressed more.

3.2.2 Entropy and Compression

The role of entropy in data compression is fundamental. The entropy of a source of data is the minimum number of bits required, on average, to encode symbols drawn from the source. It serves as a lower bound on the best possible lossless compression rate.

For a source X with probability distribution $p(x)$, the entropy $H(X)$ is defined as:

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x) \quad (3.1)$$

If the entropy of the data is high (i.e., the data is random and unpredictable), the potential for compression is low. On the other hand, if the entropy is low (the data is predictable), the data can be compressed to a smaller size.

Worked example. A fair coin flip has $p(\text{heads}) = p(\text{tails}) = \frac{1}{2}$:

$$H = - \left(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{2} \log_2 \frac{1}{2} \right) = 1 \text{ bit}$$

No scheme can store a long run of fair coin flips in less than 1 bit per flip. But a coin that lands heads 90% of the time has

$$H = -(0.9 \log_2 0.9 + 0.1 \log_2 0.1) \approx 0.47 \text{ bits}$$

so a long record of its flips can be squeezed to less than half a bit per flip. Predictable data compresses well.

Exercise 10 Entropy of a Message

A source sends the letter A half the time, and B and C a quarter of the time each. Compute its entropy. Then show that the code $A = 0$, $B = 10$, $C = 11$ achieves it.

Working Space

Answer on Page 87

Introduction to C

You already know how to write programs in Python. In the next few chapters, you are going to learn C. Your Python knowledge will make this easier, because C has the same fundamental ideas: variables, functions, loops, and conditionals. The syntax looks different, and there are a few genuinely new concepts, but you will recognize the shape of everything you write.

Why learn C? Two reasons. First, C is used wherever performance matters: operating systems, game engines, embedded systems, and scientific computing. Second, and more important for this course, C forces you to think about memory explicitly. Python quietly manages memory on your behalf; C makes you do it yourself. Once you understand how C uses memory, you will also understand what Python is doing underneath, and you will be ready to implement data structures from scratch.

4.1 Compiled vs. Interpreted

When you run a Python program, the Python interpreter reads your `.py` file and executes it line by line. There is no separate step between writing and running.

C works differently. Before a C program can run, a *compiler* must translate it into machine code, the raw instructions that your CPU executes directly. The compiler reads your `.c` file, checks it for errors, and produces an *executable* file. You then run that executable.

`hello.c` $\xrightarrow{\text{compiler}}$ `hello (executable)` $\xrightarrow{\text{run}}$ `output`

Figure 4.1: The compile-then-run cycle

Because the compiler sees your entire program before anything runs, it can catch many mistakes, like using a variable before declaring it or passing the wrong type to a function, that Python would only discover at runtime.

To compile a C file called `hello.c`, you run:

```
1 gcc hello.c -o hello
```

This `gcc` is the compiler that turns your C code into an executable. The `hello.c` is the input file, and `-o hello` tells the compiler to name the output executable `hello`. On macOS and

Linux systems, this may come pre installed. On Windows, however, this likely is not installed. You can find the installation instructions [here](#).

Then you run it:

```
1 ./hello
```

4.2 Your First C Program

Create a file called `hello.c` and type the following:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, world!\n");
6     return 0;
7 }
```

Compile and run it. You should see:

```
1 Hello, world!
```

Let us look at each piece.

`#include <stdio.h>` tells the compiler to include the standard input/output library, which provides `printf` for printing. Think of it like Python's `import` statement.

`int main()` is the function where your program starts. Every C program must have exactly one main function. The `int` means it returns an integer; `return 0` signals to the operating system that the program finished successfully.

`printf("Hello, world!\n");` prints a line of text. The `\n` is the newline character.

Notice the semicolons at the end of statements, and the curly braces `{ }` around the body of `main`. In Python, indentation defines blocks; in C, curly braces do.

4.3 Types and Variables

In Python, you write:

```

1 x = 5
2 y = 3.14

```

Python figures out the type of each variable automatically.

In C, you must declare the type of every variable when you create it:

```

1 int x = 5;
2 double y = 3.14;

```

This is called *static typing*. Once you declare a variable as `int`, it can only hold integers. The compiler enforces this and will refuse to compile code that mixes types incorrectly.

The common C types are:

int A whole number: 0, 42, -7. Typically 4 bytes.

double A floating-point number: 3.14, -0.001. Typically 8 bytes. (There is also `float`, which uses 4 bytes and has less precision.)

char A single character: 'A', 'z', '7'. Use single quotes.

Boolean In C, booleans are just integers: 0 means false, and any non-zero value means true. You can include `<stdbool.h>` to get the names `true` and `false`.

You can also declare a variable without immediately giving it a value:

```

1 int count;      /* declared but not initialized */
2 count = 10;     /* now it has a value */

```

Warning: an uninitialized variable in C contains whatever random bytes happen to be in that memory location. Unlike Python, C will not raise an error if you read an uninitialized variable; it will just give you garbage. Always initialize your variables.

Note that C uses `/* ... */` for comments, not `#`. You can also use `//` for single-line comments in modern C.

4.4 Printing with `printf`

`printf` uses format specifiers to print different types:

```
1 int age = 17;
2 double height = 1.75;
3 printf("Age: %d, Height: %f\n", age, height);
```

Output:

```
1 Age: 17, Height: 1.750000
```

Common format specifiers:

Specifier	Type
%d	int
%f	double or float
%c	char
%s	string (char pointer)

4.5 Conditionals and Loops

The logic is the same as Python; only the syntax changes.

4.5.1 if / else

```
1 int score = 85;
2
3 if (score >= 90) {
4     printf("A\n");
5 } else if (score >= 80) {
6     printf("B\n");
7 } else {
8     printf("C or below\n");
9 }
```

The condition must be in parentheses (), and each branch is surrounded by curly braces { }.

4.5.2 for and while loops

```
1 for (int i = 0; i < 5; i++) {
```

```

2     printf("%d\n", i);
3 }

```

A C for loop has three parts separated by semicolons: the initializer (`int i = 0`), the condition (`i < 5`), and the update (`i++`). The `++` increments by 1, the same as `i = i + 1`.

```

1  int i = 0;
2  while (i < 5) {
3      printf("%d\n", i);
4      i++;
5  }

```

4.6 Functions

In C, you must declare the return type and the type of each parameter:

```

1  #include <stdio.h>
2
3  double square(double x)
4  {
5      return x * x;
6  }
7
8  int main()
9  {
10     printf("%f\n", square(4.0)); /* prints 16.000000 */
11     printf("%f\n", square(2.5)); /* prints 6.250000 */
12     return 0;
13 }

```

If a function returns nothing, its return type is `void`:

```

1  void printGreeting(char *name)
2  {
3      printf("Hello, %s!\n", name);
4  }

```

Functions must be defined *before* they are called, or you must provide a *forward declaration* (just the signature, ending in a semicolon) above `main`:

```

1  #include <stdio.h>
2

```

```
3 double square(double x);    /* forward declaration */
4
5 int main()
6 {
7     printf("%f\n", square(3.0));
8     return 0;
9 }
10
11 double square(double x)    /* full definition can come later */
12 {
13     return x * x;
14 }
```

4.7 The Stack: Blackboards for Functions

Every function, while it is running, needs somewhere to store its local variables. C gives each function a *frame*: a reserved chunk of memory that holds all of that function's local variables and parameters.

Think of a frame as a **blackboard**. When a function starts running, it gets a fresh blackboard. It can write anything it wants on that blackboard, create variables, change their values. When the function finishes, the blackboard is *erased and discarded*. Local variables do not survive after their function returns.

Where do these frames live? In a region of memory called the *stack*. The stack works exactly like a physical stack of objects: the most recently added item is always on top. When `main` calls a function, that function's frame is pushed onto the top of the stack. When the function returns, its frame is popped off, and `main`'s frame is on top again.

Here is a concrete example:

```
1 #include <stdio.h>
2
3 int cookingMinutes(int weightPounds)
4 {
5     int minutes = 15 + 15 * weightPounds;
6     return minutes;
7 }
8
9 int main()
10 {
11     int totalWeight = 10;
12     int gibletsWeight = 1;
13     int turkeyWeight = totalWeight - gibletsWeight;
14     int result = cookingMinutes(turkeyWeight);
15     printf("Cook for %d minutes.\n", result);
16     return 0;
17 }
```

17 }

When `main` calls `cookingMinutes(9)`, the stack looks like this:

cookingMinutes()	<code>weightPounds = 9</code> <code>minutes = 150</code>
main()	<code>totalWeight = 10</code> <code>gibletsWeight = 1</code> <code>turkeyWeight = 9</code> <code>result = ?</code>

`main`'s blackboard sits below; `cookingMinutes`'s blackboard sits on top of it. The variable `minutes` inside `cookingMinutes` is completely separate from anything in `main`. Each function only sees its own blackboard.

When `cookingMinutes` returns 150, its frame is discarded. `main`'s frame resumes, and `result` is filled in with 150.

4.7.1 Parameters are copies

When you call `cookingMinutes(turkeyWeight)`, C copies the value of `turkeyWeight` into the new parameter `weightPounds` on `cookingMinutes`'s blackboard. Changing `weightPounds` inside the function would not change `turkeyWeight` in `main`. They are separate variables. This is called *pass-by-value*.

4.8 Addresses and Pointers

Every variable in your program lives at some address in memory. You can find out the address of a variable using the `&` operator, called the *address-of* operator:

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int i = 17;
6      printf("i stores its value at %p\n", (void *)&i);
7      return 0;
8  }
```

Run this and you will see something like:

```
1 i stores its value at 0x7ffeea3c4738
```

The address is printed in hexadecimal. Your address will differ, because the operating system decides where in memory your variables live.

A *pointer* is a variable that holds an address. If you want to store the address of an `int`, you declare a variable of type `int *` (“pointer to `int`”):

```
1 int i = 17;
2 int *addressOfI = &i;
3 printf("i is at %p\n", (void *)addressOfI);
```

The `*` in the declaration is part of the type; it means “this variable holds an address, and the thing at that address is an `int`.”

4.8.1 Reading and writing through a pointer

Once you have a pointer, you can use the `*` operator to read or write the value *at* the address the pointer holds. This is called *dereferencing* the pointer:

```
1 int i = 17;
2 int *p = &i;
3
4 printf("%d\n", *p); /* prints 17 -- reads through p */
5
6 *p = 89;           /* writes 89 to the address p holds */
7 printf("%d\n", i); /* prints 89 -- i was changed! */
```

To summarize:

- `&i` gives you the address of `i`.
- `int *p` declares a variable that can hold such an address.
- `*p` gives you the value stored at the address `p` holds.

4.8.2 NULL pointers

Sometimes you want a pointer that does not yet point to anything. Use `NULL`:

```

1  int *p = NULL;    /* p points to nothing */
2
3  if (p) {
4      printf("%d\n", *p);    /* only runs if p is non-null */
5  } else {
6      printf("p is null\n");
7  }

```

A null pointer is simply the address zero. **Never dereference a null pointer**, your program will crash. Always check before dereferencing a pointer that might be null.

4.9 Pass-by-Reference

Recall that when you call a function in C, arguments are *copied* into the function's frame. But what if you want a function to modify a variable in the calling function? You pass the *address* of that variable instead of its value. The function can then write to that address.

There is a standard C function called `modf()` that splits a floating-point number into its integer part and its fractional part, and needs to return *both*. Since a function can only return one value, it returns the fractional part and writes the integer part to an address you supply:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6      double pi = 3.14;
7      double integerPart;
8      double fractionPart;
9
10     /* Pass the address of integerPart so modf can write there */
11     fractionPart = modf(pi, &integerPart);
12
13     printf("Integer part: %f\n", integerPart);
14     printf("Fraction part: %f\n", fractionPart);
15     return 0;
16 }

```

Output:

```

1  Integer part:  3.000000
2  Fraction part: 0.140000

```

Here is another way to think about it. Imagine you need to give a spy an assignment. You say: “I have left a length of steel pipe at the foot of the angel statue in the park. When you have the information, roll it up and leave it in the pipe. I will pick it up Tuesday.” In the spy business, this is called a *dead drop*.

Pass-by-reference works the same way. You are not giving the function the data directly, you are giving it a *location* where it can leave the result.

Here is our own pass-by-reference function that converts meters to feet and inches:

```

1  #include <stdio.h>
2  #include <math.h>
3
4  void metersToFeetAndInches(double meters,
5                             int *ftPtr,
6                             double *inPtr)
7  {
8      double rawFeet = meters * 3.281;
9      int feet = (int)floor(rawFeet);
10     double inches = (rawFeet - feet) * 12.0;
11
12     if (ftPtr) *ftPtr = feet;
13     if (inPtr) *inPtr = inches;
14 }
15
16 int main()
17 {
18     int feet;
19     double inches;
20     metersToFeetAndInches(1.8, &feet, &inches);
21     printf("%d ft %f in\n", feet, inches);
22     return 0;
23 }
```

The caller declares the variables it wants filled in (feet, inches), passes their addresses to the function, and the function writes the results there.

4.10 Structs: Grouping Related Data

A *struct* lets you bundle several related values into a single named type. In Python you might use a dictionary for this; in C, a struct is the fundamental grouping mechanism.

```

1  struct Person {
2      double heightInMeters;
3      int weightInKilos;
```

```
4 };
```

This defines a new type called `struct Person`. You can then create variables of that type:

```
1 #include <stdio.h>
2
3 struct Person {
4     double heightInMeters;
5     int weightInKilos;
6 };
7
8 double bodyMassIndex(struct Person p)
9 {
10     return p.weightInKilos / (p.heightInMeters * p.heightInMeters);
11 }
12
13 int main()
14 {
15     struct Person mikey;
16     mikey.heightInMeters = 1.70;
17     mikey.weightInKilos = 96;
18
19     struct Person aaron;
20     aaron.heightInMeters = 1.97;
21     aaron.weightInKilos = 84;
22
23     printf("mikey BMI: %f\n", bodyMassIndex(mikey));
24     printf("aaron BMI: %f\n", bodyMassIndex(aaron));
25     return 0;
26 }
```

You access the members of a struct using a dot: `mikey.heightInMeters`.

When `main` is running, the stack looks like this:

main()	mikey.heightInMeters = 1.70 mikey.weightInKilos = 96 aaron.heightInMeters = 1.97 aaron.weightInKilos = 84
---------------	--

The two `Person` structs live right inside `main`'s frame on the stack.

4.11 The Heap

The stack is fast and automatic, but it has two constraints: stack frames are destroyed when functions return, and the sizes of all stack-allocated variables must be known at compile time.

The *heap* is a separate region of memory that has neither constraint. Memory on the heap persists until you explicitly release it, and can be any size determined at runtime.

In C, you claim heap memory with `malloc` and release it with `free`.

```
1 #include <stdlib.h>
2
3 struct Person *mikey = malloc(sizeof(struct Person));
```

`malloc` asks the system for enough heap memory to hold a `Person` struct and returns the address of that memory. The variable `mikey` is a *pointer* to a `Person`, it lives on the stack, but the struct it points to lives on the heap.

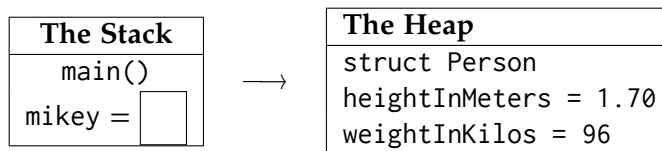


Figure 4.2: A pointer on the stack pointing to a struct on the heap

4.11.1 The arrow operator

When you have a pointer to a struct, you access its members using `->` instead of a dot:

```
1 mikey->heightInMeters = 1.70;
2 mikey->weightInKilos = 96;
```

The code `mikey->weightInKilos` means: “dereference the pointer `mikey` to get the struct, then access its `weightInKilos` member.” It is shorthand for `(*mikey).weightInKilos`.

4.11.2 Releasing heap memory

When you are done with heap-allocated memory, you must release it with `free`:

```

1 free(mikey);
2 mikey = NULL;    /* good habit: prevents accidental reuse */

```

After `free`, the memory is returned to the heap for reuse. Setting the pointer to `NULL` prevents you from accidentally using it again.

If you forget to call `free`, the memory is never returned. This is called a *memory leak*. In a short-lived program it may not matter, but in a long-running program it can consume all available memory.

4.11.3 A complete heap example

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Person {
5      double heightInMeters;
6      int weightInKilos;
7  };
8
9  double bodyMassIndex(struct Person *p)
10 {
11     return p->weightInKilos / (p->heightInMeters * p->heightInMeters);
12 }
13
14 int main()
15 {
16     struct Person *mikey = malloc(sizeof(struct Person));
17     mikey->heightInMeters = 1.70;
18     mikey->weightInKilos = 96;
19
20     printf("mikey BMI: %f\n", bodyMassIndex(mikey));
21
22     free(mikey);
23     mikey = NULL;
24
25     return 0;
26 }

```

`bodyMassIndex` now takes a `struct Person *` instead of a `struct Person` by value. This avoids copying the struct. Functions that work on heap-allocated data almost always take pointers.

4.11.4 Heap arrays

You can also allocate a contiguous block of memory for multiple values. Use `malloc` with a count:

```
1  /* Allocate space for 1000 doubles on the heap */
2  double *buffer = malloc(1000 * sizeof(double));
3
4  buffer[0] = 3.14;
5  buffer[1] = 2.72;
6
7  free(buffer);
8  buffer = NULL;
```

A pointer to the first element is all you need to work with the whole array. The heap memory is contiguous, so `buffer[0]`, `buffer[1]`, and so on are laid out one after another.

4.12 What Python Objects Actually Are

You now know everything you need to understand what a Python object is, under the hood.

When Python creates an object, it:

1. Allocates a struct on the heap large enough to hold the object's data (its instance variables).
2. Stores in that struct a pointer to the class's method table, so that method calls can find the right code.
3. Returns a pointer to that struct.

Your Python variable `obj` is, at the machine level, a pointer to a heap-allocated struct. When Python's garbage collector determines that nothing points to an object anymore, it calls the equivalent of `free` to release that memory.

The next chapter moves to C++ and, with this memory model in hand, builds linked lists from scratch.

4.13 Exercises

Exercise 11 Temperature Conversion

Write a C function `celsiusToFahrenheit` that takes a double temperature in Celsius and returns the equivalent in Fahrenheit. Call it from `main` for the values 0, 20, 37, and 100, and print each result.

*Working Space**Answer on Page 87***Exercise 12 Swap**

Write a function `swap` that takes two `int` pointers and swaps the values at the two addresses. Test it so that after calling `swap(&a, &b)`, the values of `a` and `b` have been exchanged.

*Working Space**Answer on Page 87***Exercise 13 Distance Between Points**

Write a struct `Point` with double members `x` and `y`. Write a function `distance` that takes two `struct Point *` pointers and returns the Euclidean distance between them. Allocate two points on the heap, fill in their coordinates, print the distance, and then free the memory.

*Working Space**Answer on Page 88*

Exercise 14 Memory Leak Hunt

The following program has a memory leak. Identify and fix it.

Working Space

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5      int value;
6  };
7
8  void printValue(int x)
9  {
10     struct Node *n = malloc(sizeof(struct
11                               ↪ Node));
12     n->value = x;
13     printf("%d\n", n->value);
14 }
15
16 int main()
17 {
18     for (int i = 0; i < 5; i++) {
19         printValue(i);
20     }
21     return 0;
22 }
```

Answer on Page 89

Exercise 15 What Time Is It?

A struct `tm` is defined in `<time.h>` and holds a broken-down calendar time. The function `time(NULL)` returns the number of seconds since the Unix epoch, midnight UTC on January 1, 1970. The function `localtime_r(&seconds, &now)` fills a struct `tm` with the corresponding date and time. Here is a snippet to get you started:

Working Space

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main()
5 {
6     time_t secondsSince1970 = time(NULL);
7     struct tm now;
8     localtime_r(&secondsSince1970, &now);
9     printf("The time is %d:%d:%d\n",
10           now.tm_hour, now.tm_min,
11           ↪ now.tm_sec);
12     return 0;
13 }
```

Look up the other fields of struct tm and modify the program to print today's full date. Then modify it to print the date that falls exactly 4,000,000 seconds from now. (Hints: tm_mon is 0-indexed, so January is 0, and tm_year counts years since 1900.)

Answer on Page 89

Linked Lists and C++ transition

5.1 From C to C++

Your music app’s playlist knows which song comes next, and the next button just follows that link. That chain of “this one, then that one” is a *linked list*, one of the most important data structures in computing. We will build it in C++, a language that extends C with object-oriented features such as *classes*, which you met in the Classes chapter. In C++, you can define a class with the `struct` keyword, which works much like `class`.

One keyword difference matters right away: the keyword *new*, which does not exist in C. In C, we have `malloc()`, which allocates a given number of bytes on the heap and returns a pointer to them, and `sizeof`, which returns the size in bytes of a data structure.

```
1 // C : what you already know
2 struct Car *n = malloc(sizeof(struct Car));
```

C++ combines these two commands into one, called *new*. Memory you get with *new* is released with *delete*, the C++ version of *free*.

```
1 // C++ : same thing, no sizeof, no cast
2 Car* n = new Car();
```

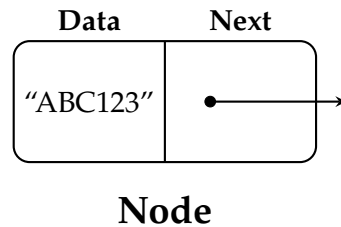
Here is a helpful video on some of the deeper differences that we will not be covering: https://www.youtube.com/watch?v=B_4x7WlQr7M

5.2 Linked Lists

A *linked list* is a linear data structure where each element is a separate object, called a *node*. Each node holds its own data and the address of the next node, called a *successor*, thus forming a chain-like structure. The first node in the list is called the *head*. The head is the only node you need to keep a direct reference to, since every other node can be reached by following successors from it. The last node, the one whose successor pointer is empty, is called the *tail*.

A simple node in a linked list can be represented in C++ as follows:

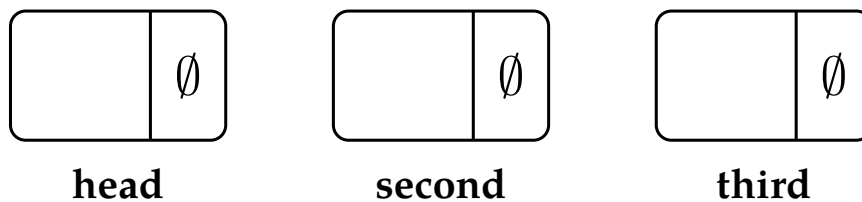
```
1 struct Node {  
2     int data;  
3     Node* next;  
4 };
```



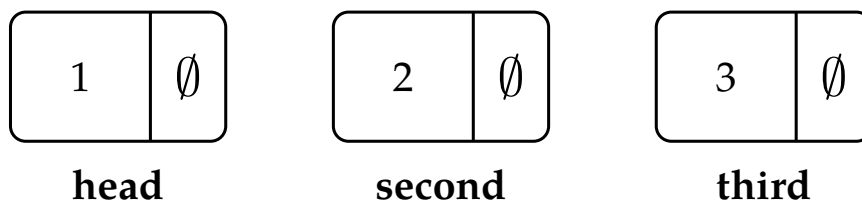
In this structure, 'data' is used to store the data and 'next' is a pointer that holds the address of the next Node in the list. When a pointer has nothing to point to, we mark its field with the empty-set symbol $\emptyset$ — the mathematical symbol for “nothing here.”

Here is a simple example of creating and linking nodes in a linked list:

```
1 // Create nodes  
2 Node* head = new Node();  
3 Node* second = new Node();  
4 Node* third = new Node();
```



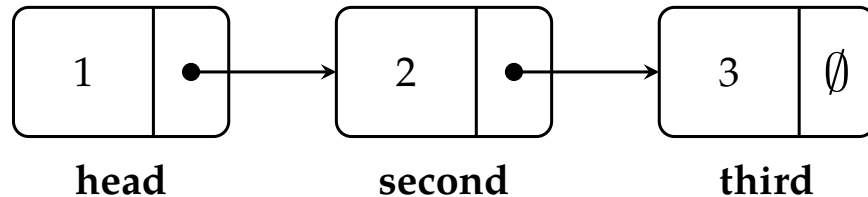
```
1  
2 // Assign data  
3 head->data = 1;  
4 second->data = 2;  
5 third->data = 3;
```



```

1 // Link nodes
2 head->next = second;
3 second->next = third;
4 third->next = nullptr; // The last node points to null, this is the TAIL

```



In this example, we first create three nodes using the ‘new’ keyword, which dynamically allocates memory. We then assign data to the nodes and link them using the ‘next’ pointer.



 **Rock Song**
The Geologists

 **Nuclear Fusion**
Helium Bros

 **Wavelength**
Ampli-Tude

 **Null Pointer**
The Exceptions

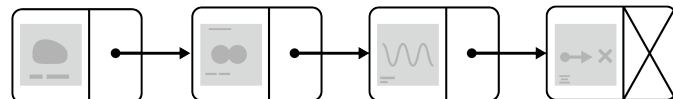


Figure 5.1: A playlist as a linked list. Each song points to the next one; the last song’s pointer is empty ($\emptyset$), so the player knows to stop.

5.3 Doubly Linked Lists

Now that we have established the basics of linked lists, we can move on to a more complex structure: the *doubly linked list*. Doubly linked lists are useful because each node now has two pointers: the same next pointer, and now, the *predecessor*, which we will call *prev*.

```

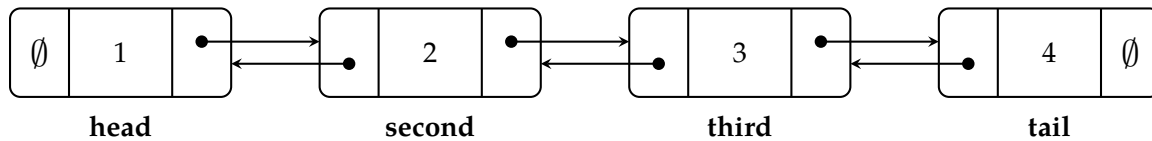
1 struct DoublyNode {

```

```

2   int data;
3   DoublyNode* next;
4   DoublyNode* prev;
5 };

```



Given this structure, we can traverse the entire list starting at the tail to the head or from the head to the tail.

Let us construct some code to analyze where the nodes would be in memory. We will create a doubly linked list with four nodes, and then we will print out the addresses of each node and their respective next and previous pointers.

Write the following code, placing it after the `DoublyNode` definition above.

```

1  #include <iostream>
2
3  int main() {
4      // Create nodes
5      DoublyNode* head = new DoublyNode();
6      DoublyNode* second = new DoublyNode();
7      DoublyNode* third = new DoublyNode();
8      DoublyNode* tail = new DoublyNode();
9
10     // Assign data
11     head->data = 1;
12     second->data = 2;
13     third->data = 3;
14     tail->data = 4;
15
16     // Link nodes forward
17     head->next = second;
18     second->next = third;
19     third->next = tail;
20     tail->next = nullptr; // tail has no successor
21
22     // Link nodes backward
23     head->prev = nullptr; // head has no predecessor
24     second->prev = head;
25     third->prev = second;
26     tail->prev = third;
27
28     // Print the address of each node, and the addresses stored in its
29     // prev/next fields, so you can see both directions of the chain.

```

```

30     std::cout << "head is at: " << head << " prev: " << head->prev << "
    ↪ (nullptr)" << " next: " << head->next << std::endl;
31     std::cout << "second is at: " << second << " prev: " << second->prev << "
    ↪ next: " << second->next << std::endl;
32     std::cout << "third is at: " << third << " prev: " << third->prev << "
    ↪ next: " << third->next << std::endl;
33     std::cout << "tail is at: " << tail << " prev: " << tail->prev << "
    ↪ next: " << tail->next << " (nullptr)" << std::endl;
34
35     // Each DoublyNode takes up a fixed number of bytes in memory.
36     std::cout << "\nsizeof(DoublyNode) = " << sizeof(DoublyNode) << " bytes" <<
    ↪ std::endl;
37
38     // Clean up the memory
39     delete head;
40     delete second;
41     delete third;
42     delete tail;
43
44     return 0;
45 }

```

When you run this, you will get real addresses like these (your addresses will differ):

```

1  head is at: 0x100c9dca0 prev: 0 (nullptr) next: 0x100c9dbc0
2  second is at: 0x100c9dbc0 prev: 0x100c9dca0 next: 0x100c9dbe0
3  third is at: 0x100c9dbe0 prev: 0x100c9dbc0 next: 0x100c9dc00
4  tail is at: 0x100c9dc00 prev: 0x100c9dbe0 next: 0 (nullptr)
5
6  sizeof(DoublyNode) = 24 bytes

```

Computer memory is comprised of a very long string of bytes, and each byte has its own address. This line is a large, contiguous stream of ones and zeros when you get to the core of it. Often, programmers separate it into chunks of certain sizes so that we can make them look similar to grids of memory, like matrices.

Look closely at second and third: $0x100c9dbe0 - 0x100c9dbc0 = 0x20$, which is **32** in decimal, not 24 bytes like `sizeof(DoublyNode)` had provided. The same is true between third and tail. Why, then, is there a difference between the 32 stored bytes and 24 bytes we would expect our memory to take up?

`DoublyNode` holds an `int` (4 bytes), 4 bytes of padding so that the pointers are aligned to 8 bytes, and two pointers (8 bytes each on a 64-bit machine), for 24 bytes total. `malloc` does not hand out memory in exactly the size you ask for: the allocator *rounds each allocation up*, and in this run it gives 32 bytes. Your struct only uses 24 of those bytes; the other 8 are allocator *padding* that `sizeof()` never shows you. The command `new` combines `malloc` and `sizeof` so that you do not have to compute the size yourself.

Compare this to the Node struct from the previous section, which has one pointer and one integer. The pointer and integer take 12 bytes, but the compiler adds 4 bytes of padding so that the struct's size is a multiple of its 8-byte alignment, and `sizeof(Node)` is 16. This is why the addresses of the nodes in a linked list are not always exactly the size of the struct apart from each other. The addresses, too, may not always be consecutive.

Alternatively, comparing this to data structures in Python like Lists, we can see that Python's lists are more like arrays than linked lists. Python's lists are implemented as dynamic arrays, which means that they are contiguous blocks of memory that can grow and shrink in size. This allows for fast access to elements by index, but can be less efficient for inserting or deleting elements in the middle of the list, as it may require shifting elements around in memory. In contrast, once you have a pointer to a spot in a linked list, inserting or deleting there takes just a couple of pointer changes, with nothing to shift. The costs: finding the 500th element means following 500 pointers, and every node carries extra memory for its pointers.

5.4 Exercises

Exercise 16 Walking the List

Write a function `void printList(Node* head)` that prints every value in a singly linked list, and a function `int length(Node* head)` that counts its nodes.

Working Space

Answer on Page 89

Exercise 17 Adding to the Front

Write `Node* pushFront(Node* head, int value)`, which creates a new node holding `value`, puts it at the front of the list, and returns the new head. Use it to build the list 1, 2, 3. Why is adding at the front of a linked list fast, while adding at the front of a Python list is

Working Space

slow?

Answer on Page 90

Exercise 18 **Cleaning Up**

Write `void deleteList(Node* head)` that frees every node. Why must you save `head->next` before deleting `head`?

Working Space

Answer on Page 90

Bitmaps

Zoom way in on any photo and it dissolves into tiny colored squares. A bitmap file is the most direct way to store those squares: just a header and then the color of every pixel, one after another. Because it is so simple, it is the perfect format for writing your own image-editing code. The Bitmap, denoted with file type `.bmp`, is an early file format for storing images on computers. Rows of different colors are stored as a grid of pixels. Each pixel is represented by a specific number of bits, which determines the color depth of the image. For example, a 24-bit bitmap can represent over 16 million colors, while an 8-bit bitmap can only represent 256 colors.

Bitmaps are uncompressed, meaning they can take up a lot of storage space compared to other image formats like JPEG or PNG. However, they are simple to read and write in code: no decompression is needed, so a short program can edit them pixel by pixel.

To understand how bitmaps work in practice, it helps to think of an image as nothing more than numbers stored in memory. Each pixel in a bitmap corresponds to a color value, and these values are usually written in hexadecimal (base-16) form, where the digits run 0–9 and then A–F, so FF is $15 \cdot 16 + 15 = 255$. (Chapter 2.3 covers hexadecimal in depth.)

In a common 24-bit bitmap, each pixel is made up of three color components:

- Red
- Green
- Blue

The intensity of each color component is represented by 8 bits. In decimal, this means each component can have a value from 0 to 255, for a total of 256 possible values. In hexadecimal (base 16, which you learned in the Data Representation chapter), this range is represented from 00 to FF.

Each pixel's color is then represented by a 6-digit hexadecimal number, where the first two digits represent the red component, the next two represent green, and the last two represent blue. A total of 16,777,216 different colors can be represented ($256 \times 256 \times 256$).

Together, the intensity of each component determines the final color of the pixel. For example:

- 0xFF0000 represents pure red

- 0x00FF00 represents pure green
- 0x0000FF represents pure blue
- 0xFFFFFFFF represents white
- 0x000000 represents black

Recall that the prefix 0x indicates that the number is in hexadecimal format. When creating or manipulating bitmap images in code, you can directly set the color values of individual pixels using their hexadecimal representations. This allows for precise control over the image's appearance.

6.1 Bitmap Structure

When traversing a bitmap, it is important to understand its structure. A bitmap file typically consists of a header followed by the pixel data. The header contains metadata about the image, such as its width, height, and color depth. The pixel data is stored in a grid format, with each pixel represented by its color value.

In a 24-bit bitmap, each pixel is represented by 3 bytes (one byte for each color component). The pixel data is usually stored in a bottom-up order, meaning the first row of pixel data corresponds to the bottom row of the image.

6.1.1 Bitmap Headers

The bitmap header is typically constructed with a file section and info section. Following those comes the pixel data, represented as bytes.

```
1 | File Header (14 bytes) | Info Header (usually 40 bytes) | Pixel Data
  ↳ (rowSize*height) |
```

```
1 struct BITMAPFILEHEADER
2 {
3     WORD bfType; //specifies the file type
4     DWORD bfSize; //specifies the size in bytes of the bitmap file
5     WORD bfReserved1; //reserved; must be 0
6     WORD bfReserved2; //reserved; must be 0
7     DWORD bfOffBits; //specifies the offset in bytes from the bitmapfileheader to
8         ↳ the bitmap bits
9 };
10 struct BITMAPINFOHEADER
11 {
```

```

12     DWORD biSize; //specifies the number of bytes required by the struct
13     LONG biWidth; //specifies width in pixels
14     LONG biHeight; //specifies height in pixels
15     WORD biPlanes; //specifies the number of color planes, must be 1
16     WORD biBitCount; //specifies the number of bit per pixel
17     DWORD biCompression; //specifies the type of compression
18     DWORD biSizeImage; //size of image in bytes
19     LONG biXPelsPerMeter; //number of pixels per meter in x axis
20     LONG biYPelsPerMeter; //number of pixels per meter in y axis
21     DWORD biClrUsed; //number of colors used by the bitmap
22     DWORD biClrImportant; //number of colors that are important
23 };

```

You can refer to [the BMP file format specification](#) and [the bitmap article](#) for more information. Let us create variables to hold important fields:

- **int** `w` = `bih.biWidth`; - width of the image in pixels
- **int** `h` = `bih.biHeight`; - height of the image in pixels
- **int** `size` = `bih.biSizeImage`; - the total size of the pixel data in bytes
- Note that we create `bytesPerPixel = 3` - the number of bytes per pixel (3 for 24-bit RGB)

Let us also preemptively build a pixel structure that can contain the information of an individual value at coordinate (x, y):

```

1  struct PIXEL
2  {
3      // Each value is in range 0 to 255 represented as a byte
4      BYTE b; // 1 byte
5      BYTE g; // 1 byte
6      BYTE r; // 1 byte
7  };

```

6.1.2 Padding

Remember that bytes are stored in a contiguous block of memory, as one long string of bytes. The computer does not inherently know where one row ends and the next begins (a concept called **row-major order**)¹. We need to calculate the starting index of each pixel based on its row and column position. But, there is an issue: each row of pixel data in a bitmap must be aligned to a 4-byte boundary.

¹In row-major order, 2D arrays like bitmaps are stored sequentially in memory, row by row. To review this concept, refer back to the Vectors and Matrices Chapter, which explains how matrices can be analyzed in row-major order.

Padding exists in bitmap images to ensure that each row of pixel data is aligned to a 4-byte boundary in memory, which was a design choice made to improve performance and simplicity on early computer systems. Processors and hardware are more efficient when reading data that begins at predictable, aligned memory addresses, and forcing each row to occupy a size divisible by four bytes guarantees this alignment.

Because bitmap pixels do not always naturally fill a multiple of four bytes—especially in formats like 24-bit images where each pixel uses three bytes—extra, non-image bytes are added to the end of each row to reach the required alignment. These padding bytes do not represent color information and are ignored when displaying the image, but they ensure that each row starts at a consistent location in memory, making bitmap files easier and faster for software and hardware to process.

This means that if the width of the image (in bytes) is not a multiple of 4, we need to add padding bytes at the end of each row to ensure proper alignment. Padding, then, is extra bytes added to the end of each row of pixels so that the row size is a multiple of 4 bytes.

Let us look at an example. If we create a 4 pixel bitmap, it takes up

$$4 \text{ pixels} \times 3 \text{ bytes} = 12 \text{ bytes}$$

The memory looks something like

1

...		??		??		[B G R]	[B G R]	[B G R]	[B G R]		??		??		...
-----	--	----	--	----	--	---------	---------	---------	---------	--	----	--	----	--	-----

Since memory is contiguous, the question marks represent unrelated memory that does not belong to the bitmap's pixel data. Attempting to access memory outside the bounds of the image means the program is reading or writing data it does not own. Modern operating systems protect memory by dividing it into regions assigned to each program. If a program tries to access memory outside of its permitted region, the operating system immediately stops the program and reports an error known as a Segmentation Fault. This mechanism prevents programs from corrupting other data and helps ensure overall system stability.

12 bytes is evenly divisible by 4, so *no padding is needed*.

However, if we create a 3 pixel bitmap, it would take up:

$$3 \text{ pixels} \times 3 \text{ bytes} = 9 \text{ bytes}$$

9 is not evenly divisible by 4, so 3 bytes of memory must be added as padding for row-alignment.

1

...		??		??		[B G R]	[B G R]	[B G R]	[P]	[P]	[P]		??		??		...
-----	--	----	--	----	--	---------	---------	---------	-------	-------	-------	--	----	--	----	--	-----

Note that we read the pixels in B G R order in memory, but hex colors are read usually to humans via R G B order. This gets into a concept called *little endian*. Put simply in a [stack overflow post](#):

RGB is a byte-order. But a deliberate implementation choice of most vanilla Graphics libraries is that they treat colours as unsigned 32-bit integers internally, with the three (or four, as alpha is typically included) components packed into the integer.

On a little-endian machine (such as x86) the integer `0x01020304` will actually be stored in memory as `0x04030201`. And thus `0x00BBGGRR` will be stored as `0xRRGGBB00`!

So how can we create an equation for finding a pixel at the coordinates (x, y) if memory is in a contiguous line?

The calculation `rawRowSize = width * bytesPerPixel` provides a raw row estimate, but if padding is included, we need to round up by 4. The multiples of 4 look like 0, 4, 8, 12, 16, 20, 24, 28, 32...

- If `rawRowSize` is already one of these, we want to keep it.
- If not, we want to round upwards to the next multiple.

One way to accomplish this rounding is to take advantage of integer division. By adding a small offset before dividing, we can ensure that any value which is not already divisible by 4 is pushed into the next group of four bytes. Specifically, adding 3 to the raw row size guarantees that the result will round upward when divided by 4 using integer arithmetic.

This gives us the following expression for the true number of bytes in a single row, including padding:

$$\text{rowSize} = \left\lceil \frac{\text{rawRowSize} + 3}{4} \right\rceil \times 4$$

In C++, this is commonly written as:

```
1 int rowSize = ((width * bytesPerPixel + 3) / 4) * 4;
```

This value represents the number of bytes that must be traversed in memory to move from the beginning of one row of pixels to the beginning of the next. Recall that our `bytesPerPixel` is 3, but can also be calculated by `int bytesPerPixel = bih.biBitCount / 8;`

Once the padded row size is known, we can compute the location of any pixel at coordinates (x, y). Because bitmap pixel data is stored row by row in a contiguous block of

memory, the offset to the start of row y is given by

$$y \times \text{rowSize}$$

Within that row, each pixel occupies `bytesPerPixel` bytes, so the offset to column x is:

$$x \times \text{bytesPerPixel}$$

Together, these two offsets combined yields the final memory index for pixel (x, y) :

$$\text{index}(x, y) = y \times \text{rowSize} + x \times \text{bytesPerPixel}$$

In code, this calculation appears as:

```
1  int offset = y * rowSize + x * bytesPerPixel;
```

This formula will come in handy later in our code!

Let us write the full code for bitmap analysis. Starting with store the bitmap that our program reads into a memory allocated array, we can load our bitmap into a file, and read all the file information from it. We also can then utilize our equations `rowSize` and `idx` to get the individual pixel at $(x, y) \in (w, h)$.

```
1  // program arguments: ./bitmap_reader inputname.bmp outputname.bmp
2  if (argc !=3) return 1;
3  string inputname = argv[1];
4  string outputname = argv[2];
5
6
7  FILE *inputfile = fopen(inputname.c_str(), "rb"); // read byte only mode
8
9  BITMAPFILEHEADER bfh;
10 BITMAPINFOHEADER bih;
11
12 fread(&bfh.bfType, 2, 1, inputfile);
13 fread(&bfh.bfSize, 4, 1, inputfile);
14 fread(&bfh.bfReserved1, 2, 1, inputfile);
15 fread(&bfh.bfReserved2, 2, 1, inputfile);
16 fread(&bfh.bfOffBits, 4, 1, inputfile);
17 fread(&bih, sizeof(BITMAPINFOHEADER), 1, inputfile);
18
19 int w = bih.biWidth;
20 int h = bih.biHeight;
21 int bytesPerPixel = bih.biBitCount / 8; // 3 BYTES, stored in BGR order
22 int rowSize = ((w * bytesPerPixel + 3) / 4) * 4;
23 int size = rowSize * abs(h);
24 bih.biSizeImage = size;
```

```

25 |bfh.bfSize = bfh.bfOffBits + bih.biSizeImage;
26 |
27 |fseek(inputfile, bfh.bfOffBits, SEEK_SET); // offset from header
28 |BYTE* data = (BYTE *)malloc(size);
29 |fread(data, size, 1, inputfile);
30 |fclose(inputfile);
31 |
32 |// ----- FORCE STANDARD BMP HEADER HERE -----
33 |bih.biSize = 40;
34 |bih.biSizeImage = size;
35 |bfh.bfOffBits = 54;
36 |bfh.bfSize = 54 + size;

```

From there, we can set up an output file:

```

1 | FILE *outfile = fopen(outputname.c_str(), "wb"); // creates a new file in write
   | ↪ bytes mode
2 | BYTE* out = (BYTE *) malloc(size);
3 |
4 | fwrite(&bfh.bfType, 2, 1, outfile);
5 | fwrite(&bfh.bfSize, 4, 1, outfile);
6 | fwrite(&bfh.bfReserved1, 2, 1, outfile);
7 | fwrite(&bfh.bfReserved2, 2, 1, outfile);
8 | fwrite(&bfh.bfOffBits, 4, 1, outfile);
9 | fwrite(&bih, sizeof(BITMAPINFOHEADER), 1, outfile);

```

And then, we can loop through to the w and h to analyze each individual pixel. What we will do as a first test of pixel alteration is *swap red and blue values* in our output.

```

1 | for (int x = 0; x < w; x++)
2 | {
3 |     for (int y = 0; y < h; y++)
4 |     {
5 |         int idx = y * rowSize + x * 3;
6 |
7 |         PIXEL p;
8 |         BYTE B = data[idx];
9 |         BYTE G = data[idx + 1];
10 |        BYTE R = data[idx + 2];
11 |        p = { B, G, R };
12 |
13 |        PIXEL o;
14 |        o = { R, G, B }; // Swaps red and blue values
15 |        out[idx + 0] = o.b;
16 |        out[idx + 1] = o.g;
17 |        out[idx + 2] = o.r;
18 |    }
19 | }

```

Note that we search our data array at `idx` to locate the blue component, and adding 1 and 2 respectively gets the next two bytes (green and red). This works because

- Memory is a linear array of bytes
- Pixels are stored contiguously
- The order is fixed by the BMP format

Finally, write the swapped pixel data to the output file and release memory:

```
1 fwrite(out, size, 1, outfile);  
2 fclose(outfile);  
3 free(data);  
4 free(out);
```

After compiling, running this using `./bitmap_reader cow.bmp output.bmp` and `./bitmap_reader gradient.bmp output.bmp` produces the following outputs (see Figures 6.1 and 6.2):

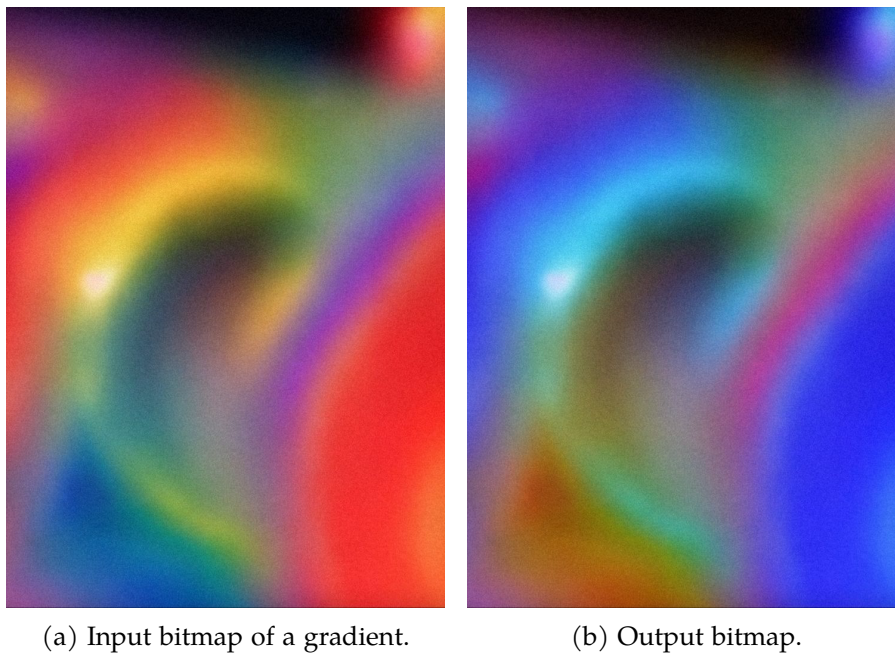


Figure 6.1: Running `bitmap_reader.cpp` on a gradient, swapping red and blue channels.



(a) Input image of a highland cow.

(b) Output bitmap.

Figure 6.2: Running `bitmap_reader.cpp` on the cow, swapping red and blue channels.

The entire base code for swapping the red and blue channels is in your Digital Resources, as well as a basic copy paste input-output bitmap. A lot of this stays the same for other pixel-based modifications, such as the next Exercise.

Exercise 19 Eliminate the Green component

Now try it yourself:

Alter both `cow.bmp` and `gradient.bmp` such that the green channel is eliminated. Keep the red and blue channels the same (do not swap them).

Working Space

Answer on Page 90

6.2 Creating a Bitmap with C++

What if we are not given an input file? Can we create a bitmap?

Of course, let us create a simple 3x3 bitmap of the following colors:

```
1 [ Red ] [ White ] [ White ]
2 [ Black ] [ Blue ] [ White ]
3 [ Black ] [ Black ] [ Green ]
```

Although the image is conceptually two-dimensional, the bitmap file stores its pixel data as a one-dimensional sequence of bytes.

There are, however, a few constraints:

- 14-byte file header
- 40-byte info header
- pixel data with required row padding

```
1 | File Header (14) | Info Header (40) | Pixel Data |
```

Calculations:

- Each pixel = 3 bytes
- Row = 3 pixels $\times$ 3 bytes per pixel = 9 bytes
- 3 padding bytes $\implies$ 12 bytes per row
- 12 bytes $\times$ 3 rows = 36 bytes

Here is the basic bitmap struct with filled in values:

```
1 #pragma pack(push, 1)
2 struct BITMAPFILEHEADER {
3     uint16_t bfType = 0x4D42; // 'BM'
4     uint32_t bfSize;
5     uint16_t bfReserved1 = 0;
6     uint16_t bfReserved2 = 0;
7     uint32_t bfOffBits = 54; // 14 + 40
8 };
9
10 struct BITMAPINFOHEADER {
11     uint32_t biSize = 40;
12     int32_t biWidth = 3;
13     int32_t biHeight = 3;
14     uint16_t biPlanes = 1;
15     uint16_t biBitCount = 24;
16     uint32_t biCompression = 0; // BI_RGB
17     uint32_t biSizeImage;
```

```

18     int32_t biXPelsPerMeter = 0;
19     int32_t biYPelsPerMeter = 0;
20     uint32_t biClrUsed = 0;
21     uint32_t biClrImportant = 0;
22 };
23 #pragma pack(pop)

```

These structures match the on-disk layout of a bitmap header exactly. The `#pragma pack` directive ensures that no padding bytes are inserted by the compiler.

Recall that we can reuse our calculations from the first example to find row size and other variables, but this time, we define them instead of fetching them from the input:

```

1  const int width = 3;
2  const int height = 3;
3  const int bytesPerPixel = 3;
4  const int rowSize = ((width * bytesPerPixel + 3) / 4) * 4; // 12
5  const int imageSize = rowSize * height; // 36

```

Now we can create a new file and write the file headers individually:

```

1  FILE* f = fopen("custom_made.bmp", "wb");
2
3  BITMAPFILEHEADER bfh;
4  BITMAPINFOHEADER bih;
5
6  bih.biSizeImage = imageSize;
7  bfh.bfSize = bfh.bfOffBits + imageSize;
8
9  fwrite(&bfh, sizeof(bfh), 1, f);
10 fwrite(&bih, sizeof(bih), 1, f);

```

At this point, the file contains only metadata. No pixel values have been written yet.

Recall that bitmap pixel data is stored bottom-up, meaning the first row written corresponds to the bottom row of the image.

```

1  Memory order:
2  Row 3 (bottom)
3  Row 2
4  Row 1 (top)

```

Let us write the third row first:

```
1 uint8_t row[rowSize] = {0}; // padding bytes stay 0
2 // ROW 3 (bottom)
3 // Black      Black      Green
4 row[0] = 0;   row[1] = 0;   row[2] = 0;   // Black
5 row[3] = 0;   row[4] = 0;   row[5] = 0;   // Black
6 row[6] = 0;   row[7] = 255; row[8] = 0;   // Green
7 fwrite(row, rowSize, 1, f);
```

Each group of three bytes represents one pixel in BGR order. Any remaining bytes in the row serve as padding and are ignored when the image is displayed.

Row 2:

```
1 // ROW 2
2 // Black      Blue      White
3 row[0] = 0;   row[1] = 0;   row[2] = 0;   // Black
4 row[3] = 255; row[4] = 0;   row[5] = 0;   // Blue
5 row[6] = 255; row[7] = 255; row[8] = 255; // White
6 fwrite(row, rowSize, 1, f);
```

Row 1:

```
1 // ROW 1 (top)
2 // Red      White      White
3 row[0] = 0;   row[1] = 0;   row[2] = 255; // Red
4 row[3] = 255; row[4] = 255; row[5] = 255; // White
5 row[6] = 255; row[7] = 255; row[8] = 255; // White
6 fwrite(row, rowSize, 1, f);
7 fclose(f);
```

And that is it! The compiler automatically converts the 255s to 0xFF. Close the file and run the program, and you get an extremely small output. Enlarge the output and you get:

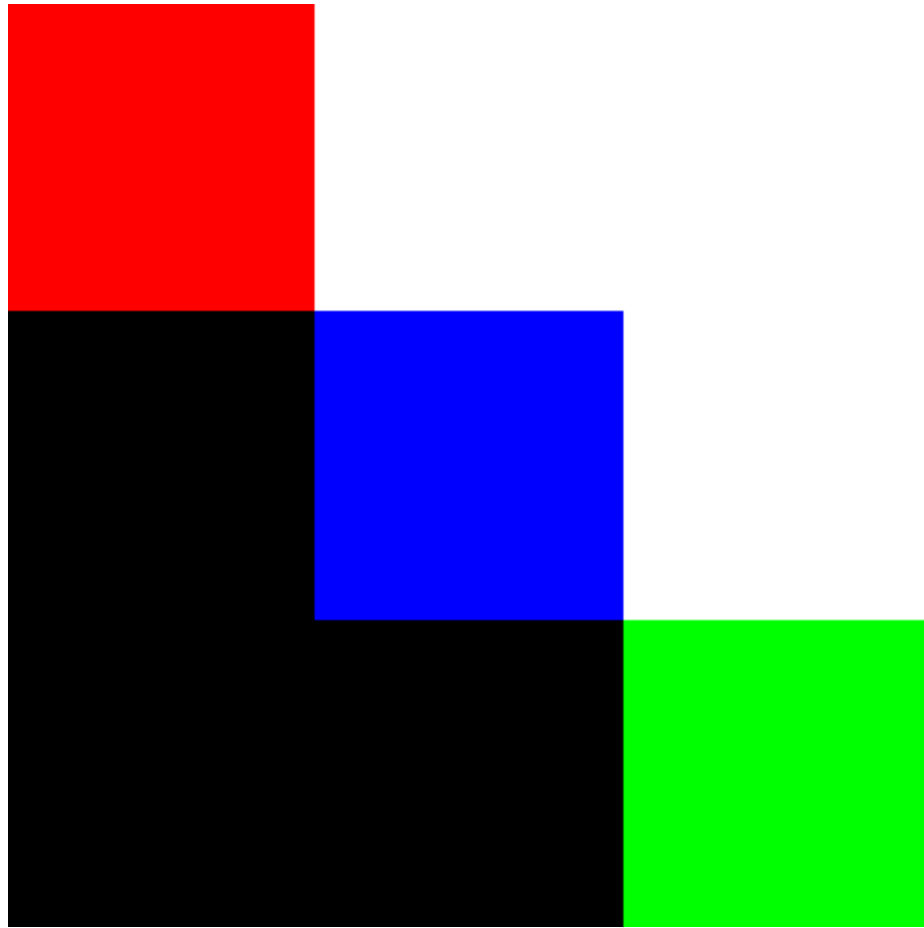


Figure 6.4: Your custom bitmap!

Exercise 20 8 Pixel Colorful Bitmap

Working Space

Create a 4×2 bitmap with the following layout:

1	[Yellow]	[Magenta]	[Cyan]	[White]
2	[Black]	[Red]	[Green]	[Blue]

Bitmap Requirements:

- Width: 4 pixels
- Height: 2 pixels

- Color depth: 24-bit (RGB, stored as BGR)
- Compression: None (BI_RGB)
- Row padding: Rows must be aligned to 4-byte boundaries

Answer the following questions before writing any code:

1. How many bytes does each pixel use?
2. How many bytes are required for one row before padding?
3. How many padding bytes are required per row?
4. What is the total size of the pixel data?
5. What is the total file size?

Answer on Page 92

6.3 Connection to transformations

What if we alter the pixel values in a more complex way? For example, instead of altering the colors themselves, we alter the pixel positions. This is what is called a transformation, and is a fundamental concept in computer graphics. A transformation is a mathematical operation that changes the position, size, or orientation of an object in a graphical space. In the context of bitmaps, transformations can be applied to the pixel data to achieve various effects, such as rotation, scaling, translation, and shearing.

What if you took a selfie, but then wanted to mirror it horizontally, to get a view of your face like the one in a mirror? This is a common transformation called a horizontal flip. To achieve this, you would iterate through each row of pixels in the bitmap and swap the pixels on the left side with the corresponding pixels on the right side. This can be done by calculating the index of each pixel and performing the necessary swaps. Note that nothing on the vertical direction of the image is changed, only the horizontal positions of the pixels are altered.

Since we are storing our pixel data as (x, y) coordinates, we can easily apply transformations by manipulating these coordinates. For an image with width w and height h , the horizontal flip transformation can be expressed mathematically as:

$$x' = w - 1 - x, \quad y' = y$$

This operation does not change the pixel values themselves, but instead sets the new position of each pixel according to the transformation rule. The entire image transformation is therefore a systematic coordinate mapping applied across the matrix.

In the language of linear algebra: If we have a pixel coordinate expressed as a simple vector $\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix}$, we can define a transformation function T that takes this pixel coordinate and maps it to a new coordinate $\mathbf{p}'$:

$$\mathbf{p}' = T\mathbf{p}$$

In this case, T is a matrix, just like the reflections in the Matrices as Transformations chapter. A reflection across the y -axis is:

$$T = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

This matrix, when multiplied with the pixel coordinate vector $\mathbf{p}$, will yield the new coordinate $\mathbf{p}'$ that corresponds to the horizontally flipped position of the original pixel. Specifically, the -1 in the first row of the matrix indicates that the x -coordinate is inverted (flipped), while the 1 in the second row indicates that the y -coordinate remains unchanged. This matrix sends x to $-x$, which puts the image on the wrong side of the origin. To keep the flipped image inside the frame, we must also shift each x -coordinate by $w - 1$. A shift is not a linear transformation, so this matrix alone cannot do the whole job.

Let us do this with code! The only thing that changes in our for loop is the calculation of the output pixel's position. Instead of writing to the same index, we write to the flipped index:

```

1  for (int y = 0; y < h; y++)
2  {
3      for (int x = 0; x < w; x++)
4      {
5          int newX = w - 1 - x; // Calculate the new x-coordinate for the horizontal
           ↪ flip
6          int srcIdx = y * rowSize + x * 3;
7          int dstIdx = y * rowSize + newX * 3;
8
9          // read the pixel values from the source index
10         BYTE B = data[srcIdx];
11         BYTE G = data[srcIdx + 1];

```

```
12     BYTE R = data[srcIdx + 2];
13
14     //output the pixel values to the destination index
15     out[dstIdx + 0] = B;
16     out[dstIdx + 1] = G;
17     out[dstIdx + 2] = R;
18 }
19 }
```

Taking the input bitmap of the cow from 6.2a and running this code produces the horizontally flipped output in 6.6.



Figure 6.6: Output of a horizontally flipped cow bitmap after applying the new pixel position transformation.

The same idea works for any of the transformations from that chapter: compute where each pixel goes, and copy its color there. Rotations by angles other than 90° land between pixels, so real image editors blend neighboring colors, a process called *interpolation*.

6.4 Summary

In this chapter, we experimented with bitmaps. Specifically,

- loading a bitmap in C++
- creating and copying bitmap file and info headers
- creating a pixel structure to work with hexadecimal bytes and hexcodes
- swapping the channels of a bitmap
- creating a bitmap from scratch in C++
- mirroring an image by remapping pixel coordinates

Stacks and Queues

7.1 Stacks

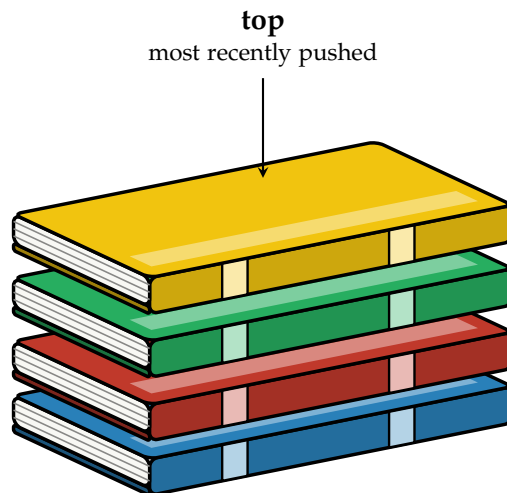
Every time you press Ctrl-Z, your editor undoes the most recent change, then the one before that, and so on backward. The list of changes is a stack.

A *stack* is a linear data structure that only allows access at one end, called the *top*.

Data comes off in the reverse order it went on: the last thing pushed onto the stack is the first thing popped off. This ordering is called *LIFO*, for “last in, first out.”

You have already seen a stack in a previous workbook, even if it was not called one: the call stack from the Introduction to C chapter behaves exactly this way, with each function call pushing a new frame on top and returning popping it back off.

We can think of this like a stack of books: you can only access the book on top, and you can only see the cover of the top book. If you want to get to a book underneath, you have to remove the books above it first.



Every stack, no matter how it is built underneath, supports the same handful of operations:

- `push(value)` put a new value on top of the stack.
- `pop()` remove and return the value on top of the stack.

- `peek()` look at the value on top of the stack, without removing it.
- `isEmpty()` true if the stack has nothing in it.

Notice that we can only use the value on top of the stack, no matter how large it is.

We are going to build a stack two different ways: once backed by a fixed-size array, and once backed by the linked-list nodes from the previous chapter. Both give you `push/pop/peek/isEmpty`, but they make very different tradeoffs underneath.

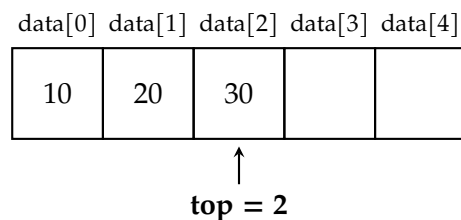
7.1.1 Array-Based Stack

The simplest way to build a stack is on top of a plain array. We keep a fixed-size buffer and a single integer, `top`, that tracks the index of the top element. An empty stack is represented by `top == -1`.

```

1  const int CAPACITY = 5;
2
3  struct ArrayStack {
4      int data[CAPACITY];
5      int top;    // index of the top element; -1 means empty
6  };

```



`push` and `pop` only ever touch the `top` index and whatever cell it points at, so both run in constant time, $O(1)$. The catch is `CAPACITY`: once the array is full, there is nowhere left to push, so a correct implementation has to check for and reject that case rather than writing past the end of the array.

```

1  bool isEmpty(const ArrayStack& s) {
2      return s.top == -1;
3  }
4
5  bool isFull(const ArrayStack& s) {
6      return s.top == CAPACITY - 1;
7  }
8

```

```

9 void push(ArrayStack& s, int value) {
10     // prevent pushing onto a full stack
11     if (isFull(s)) {
12         std::cout << "push(" << value << ") failed: stack overflow" << std::endl;
13         return;
14     }
15     s.top++;
16     s.data[s.top] = value;
17 }
18
19 int pop(ArrayStack& s) {
20     // prevent popping from an empty stack
21     if (isEmpty(s)) {
22         std::cout << "pop() failed: stack underflow" << std::endl;
23         return -1;
24     }
25     int value = s.data[s.top];
26     s.top--;
27     return value;
28 }

```

Notice that `pop` does not actually erase `s.data[s.top]`, it just moves `top` down by one. The old value is still sitting in the array, but nothing can reach it anymore, since every operation only ever looks at indices 0 through `top`.

The full, runnable version of this file, including a `main()` that demonstrates both overflow and underflow, is in your digital resources. That version also prints memory addresses, showing that the array is one contiguous block.

7.1.2 Linked-List Stack

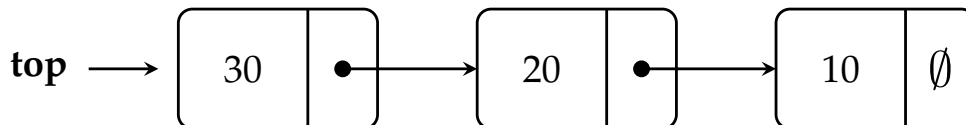
The array-based stack has one real weakness: `CAPACITY` is fixed at compile time. If you do not know in advance how much data you will be pushing, you either waste memory reserving more than you need, or you risk overflow. A stack built on the `Node` type from the Linked Lists chapter has no limit, since it grows and shrinks on the heap as needed. The downside is that each element takes more memory, since it has to store a pointer to the next node.

```

1 struct Node {
2     int data;
3     Node* next;
4 };
5
6 struct LinkedStack {
7     Node* top;    // nullptr means empty
8 };

```

Instead of an array index, `top` is now a pointer to the first node. Pushing means creating a new node whose next points at the old `top`, then moving `top` to the new node; popping is the reverse.



```

1  bool isEmpty(const LinkedStack& s) {
2      return s.top == nullptr;
3  }
4
5  void push(LinkedStack& s, int value) {
6      Node* n = new Node();
7      n->data = value;
8      n->next = s.top;    // the new node points at the old top
9      s.top = n;         // the new node becomes the top
10 }
11
12 int pop(LinkedStack& s) {
13     if (isEmpty(s)) {
14         std::cout << "pop() failed: stack underflow" << std::endl;
15         return -1;
16     }
17     Node* oldTop = s.top;
18     int value = oldTop->data;
19     s.top = oldTop->next;
20     delete oldTop;
21     return value;
22 }
  
```

Both `push` and `pop` still run in constant time, same as the array version, since neither one has to walk the list. But unlike the array version, `pop` here calls `delete` on the old `top` node; if it did not, the node would sit on the heap forever.

The full version of this file is in your digital resource!

7.1.3 Array vs. Linked: Which One?

	Array-Based	Linked-Based
Max size	Fixed at compile time	Limited only by available memory
Memory per element	Just the value	Value + one pointer

If you know a safe upper bound on how much data you will ever hold, the array version is the most efficient choice. If you do not, the linked version is safer, since it will never

overflow.

Exercise 21 **Balanced Brackets**

Code editors use a stack to check that brackets match. Read the string left to right. Push every opening bracket, (, [, or {. At each closing bracket, pop the stack and check that the popped bracket is the matching opener. The string is balanced if every check passes and the stack is empty at the end. Trace the stack for {[()](())} and for ([)]. Which is balanced?

Working Space

Answer on Page 95

7.2 Queues

Imagine you are at a concert. You are arriving at around 1pm, but the line has been growing since about 3am that same day (wow!). Naturally, by way of timing, the people who arrived earliest are going to be able to enter the venue first, while the people who arrive closest to the show time will enter the venue last.

Assuming supplements like VIP packages or fast-track lines are not of concern here, we can summarize this type of line in a similar phrase: **First In, First Out**. The first people to arrive in the line are the first people out of the line, and, subsequently, into the venue. We call this way of organizing data *FIFO*.

Similar to stacks, we only need a handful of operations to build a queue. Every queue, no matter how it is built underneath, supports these operations:

- `enqueue(value)` put a new value at the back of the queue. You may see this as `enq`.
- `dequeue()` remove and return the value at the front of the queue. You may see this as `deq`.
- `peek()` look at the value at the front of the queue, without removing it.
- `isEmpty()` true if the queue has nothing in it.

Similarly to stacks, we can implement a queue two ways: a fixed-size array and a linked list. Both have benefits and tradeoffs.

7.2.1 Array-Based Queue

We can build a queue with an array. As with stacks, we must know the capacity in advance. The queue needs two numbers: `front`, the index of the value that leaves next, and `count`, the number of values stored. The back of the queue is always at index `front + count`, taken modulo the capacity.

```
1  const int CAPACITY = 5;
2
3  struct ArrayQueue {
4      int data[CAPACITY];
5      int front = 0;    // index of the front element
6      int count = 0;    // number of elements in the queue
7  };
8
9  bool isEmpty(const ArrayQueue& q) {
10     return q.count == 0;
11 }
12
13 bool isFull(const ArrayQueue& q) {
14     return q.count == CAPACITY;
15 }
16
17 bool enqueue(ArrayQueue& q, int value) {
18     if (isFull(q)) return false;
19     int back = (q.front + q.count) % CAPACITY; // wraps around
20     q.data[back] = value;
21     q.count++;
22     return true;
23 }
24
25 bool dequeue(ArrayQueue& q, int& out) {
26     if (isEmpty(q)) return false;
27     out = q.data[q.front];
28     q.front = (q.front + 1) % CAPACITY; // wraps around
29     q.count--;
30     return true;
31 }
32
33 bool peek(const ArrayQueue& q, int& out) {
34     if (isEmpty(q)) return false;
35     out = q.data[q.front];
36     return true;
37 }
```

The % operator makes the array circular. When front or the back index passes the last slot, it wraps around to index 0. Without wrapping, the queue would seem full at the end of the array, even when the slots at the start are empty after several dequeues. Every operation here takes constant time, $O(1)$.

Exercise 22 Tracing a Circular Queue

An ArrayQueue has CAPACITY equal to 4 and starts empty. Perform these operations in order: enqueue(1), enqueue(2), enqueue(3), dequeue(), dequeue(), enqueue(4), enqueue(5). What are front and count at the end, and which values are in the queue, in order?

Working Space

Answer on Page 95

7.2.2 Linked-List Queue

A linked queue needs two pointers. front points to the first node, where values leave. back points to the last node, where new values arrive. The back pointer lets enqueue add to the end without walking the list.

```

1  struct LinkedQueue {
2      Node* front = nullptr; // nullptr means empty
3      Node* back = nullptr;
4  };
5
6  bool isEmpty(const LinkedQueue& q) {
7      return q.front == nullptr;
8  }
9
10 void enqueue(LinkedQueue& q, int value) {
11     Node* n = new Node();
12     n->data = value;
13     n->next = nullptr;
14     if (q.back == nullptr) { // empty queue: the new node is both front and back
15         q.front = n;
16     } else {
17         q.back->next = n; // link after the old back
18     }
19     q.back = n;

```

```
20 }
21
22 bool dequeue(LinkedQueue& q, int& out) {
23     if (isEmpty(q)) return false;
24     Node* oldFront = q.front;
25     out = oldFront->data;
26     q.front = oldFront->next;
27     if (q.front == nullptr) q.back = nullptr;    // the queue is now empty
28     delete oldFront;
29     return true;
30 }
```

Both enqueue and dequeue run in constant time, like the array version. Unlike the array version, the linked queue has no capacity limit. It grows and shrinks on the heap, and dequeue calls delete on the old front node so that the memory is released.

Answers to Exercises

Answer to Exercise 1 (on page 8)

No. The characteristic equation is $\lambda^2 - 2\lambda - 4 = 0$, so $\lambda = 1 \pm \sqrt{5}$. The eigenvalues are about 3.24 and -1.24 . A negative eigenvalue means the matrix is not positive semidefinite either.

Answer to Exercise 2 (on page 9)

One answer: take $A = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$. Then $AA^T = \begin{bmatrix} 5 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 9 \end{bmatrix}$. Its eigenvalues are 9 and $3 \pm 2\sqrt{2}$ (about 5.83 and 0.17). All are nonnegative, so AA^T is positive semidefinite.

Answer to Exercise 3 (on page 15)

c, a, and f take 1 byte each, and é takes 2: 5 bytes for 4 characters. Ten emoji take 40 bytes. (In Python, `len("café".encode("utf-8"))` returns 5.)

Answer to Exercise 4 (on page 24)

1. Underflow. 6.63×10^{-34} is far below 10^{-4} .
2. Overflow. 3.00×10^8 is larger than 10^8 .
3. Any number greater than 10^8 .

Answer to Exercise 5 (on page 25)

1. 4×10^6 bytes; $4 \times 10^6 \times 8 = 3.2 \times 10^7$ bits.

2. $500 \times 10^9 \times 8 = 4 \times 10^{12}$ bits.
3. $2^8 = 256$ values for one byte; $2^{64} \approx 1.8 \times 10^{19}$ values for 64 bits.

Answer to Exercise 6 (on page 26)

- $0.5050 \times 10^0 = 0.5050$
- $0.5050 \times 10^2 = 50.50$
- $0.5050 \times 10^{-1} = 0.05050$
- $0.5050 \times 10^5 = 50,500$

For $505,000,000 = 0.5050 \times 10^9$, the exponent needed is 9.

Answer to Exercise 7 (on page 28)

1. Base 10.
2. 4 significant digits.
3. Smallest: $m = -3$; largest: $M = 4$.
4. $12,345 = 0.12345 \times 10^5$. The exponent 5 exceeds $M = 4$, so this causes **overflow**. The system would need to round to $0.1235 \times 10^5 = 12,350$ and even then the exponent is out of range.
5. $0.00001 = 0.1000 \times 10^{-4}$. The exponent -4 is below $m = -3$, so this causes **underflow**.
6. Any number in the range 10^{-4} to $(1 - 10^{-4}) \times 10^4$ works, for example $0.5000 \times 10^2 = 50.00$ or $0.9999 \times 10^4 = 9999$. Overflow example: any number with magnitude above 9999, such as 100,000.

Answer to Exercise 8 (on page 30)

1. $16 - 1 - 10 = 5$ bits for the exponent.
2. $M = 16$.
3. Less precise than single precision. Single precision has $p = 23$ significand bits; this system has only $p = 10$, so it can store fewer digits and will produce larger roundoff errors.

Answer to Exercise 9 (on page 31)

1. $u = \frac{1}{2} \times 10^{1-6} = \frac{1}{2} \times 10^{-5} = 5 \times 10^{-6}$.
2. The worst-case relative error is $u = 5 \times 10^{-6}$, meaning about 6 significant digits of accuracy.
3. $u = \frac{1}{2} \times 10^{1-12} = 5 \times 10^{-12}$. Doubling the precision shrinks machine epsilon by a factor of 10^6 .

Answer to Exercise 10 (on page 33)

$$H = -\left(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{4} \log_2 \frac{1}{4} + \frac{1}{4} \log_2 \frac{1}{4}\right) = \frac{1}{2}(1) + \frac{1}{4}(2) + \frac{1}{4}(2) = 1.5 \text{ bits}$$

The code uses 1 bit half the time and 2 bits the other half, an average of $\frac{1}{2}(1) + \frac{1}{2}(2) = 1.5$ bits per letter, which matches the entropy. (Because no code word is the start of another, the bits can be decoded without spaces: 01011 is A, B, C.)

Answer to Exercise 11 (on page 49)

```

1  #include <stdio.h>
2
3  double celsiusToFahrenheit(double c)
4  {
5      return c * 9.0 / 5.0 + 32.0;
6  }
7
8  int main()
9  {
10     double temps[] = {0, 20, 37, 100};
11     for (int i = 0; i < 4; i++) {
12         printf("%.1f C = %.1f F\n", temps[i], celsiusToFahrenheit(temps[i]));
13     }
14     return 0;
15 }
```

The output ends with 100.0 C = 212.0 F. (Note 9.0 / 5.0, not 9 / 5: in C, dividing two ints throws away the remainder, so 9 / 5 is 1.)

Answer to Exercise 12 (on page 49)

```
1  #include <stdio.h>
2
3  void swap(int *p, int *q)
4  {
5      int temp = *p;
6      *p = *q;
7      *q = temp;
8  }
9
10 int main()
11 {
12     int a = 3, b = 7;
13     swap(&a, &b);
14     printf("a = %d, b = %d\n", a, b);    /* a = 7, b = 3 */
15     return 0;
16 }
```

Answer to Exercise 13 (on page 49)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  struct Point {
6      double x;
7      double y;
8  };
9
10 double distance(struct Point *p, struct Point *q)
11 {
12     double dx = q->x - p->x;
13     double dy = q->y - p->y;
14     return sqrt(dx * dx + dy * dy);
15 }
16
17 int main()
18 {
19     struct Point *a = malloc(sizeof(struct Point));
20     struct Point *b = malloc(sizeof(struct Point));
21     a->x = 1; a->y = 2;
22     b->x = 4; b->y = 6;
23     printf("distance = %f\n", distance(a, b));    /* 5.000000 */
24     free(a);
25     free(b);
26     return 0;
27 }
```

Compile with `gcc point.c -o point -lm`; the `-lm` links the math library that provides `sqrt`.

Answer to Exercise 14 (on page 50)

Each call to `printValue` allocates a `Node` on the heap, but nothing ever frees it. When the function returns, the pointer `n` disappears with the stack frame, and the heap memory can never be reached again. Five calls leak five nodes. The fix is to add `free(n)`; after the `printf` in `printValue`. (Even better: a single `int` does not need the heap at all.)

Answer to Exercise 15 (on page 50)

```

1  #include <stdio.h>
2  #include <time.h>
3
4  int main()
5  {
6      time_t seconds = time(NULL) + 4000000;
7      struct tm later;
8      localtime_r(&seconds, &later);
9      printf("In 4,000,000 seconds it will be %d-%02d-%02d\n",
10             later.tm_year + 1900, later.tm_mon + 1, later.tm_mday);
11     return 0;
12 }
```

4,000,000 seconds is about 46.3 days.

Answer to Exercise 16 (on page 58)

Start at the head and follow next until you reach `nullptr`:

```

1  void printList(Node* head) {
2      for (Node* n = head; n != nullptr; n = n->next) {
3          std::cout << n->data << " ";
4      }
5      std::cout << std::endl;
6  }
7
8  int length(Node* head) {
9      int count = 0;
10     for (Node* n = head; n != nullptr; n = n->next) {
11         count++;
12     }
```

```

12     }
13     return count;
14 }

```

Answer to Exercise 17 (on page 58)

```

1 Node* pushFront(Node* head, int value) {
2     Node* n = new Node();
3     n->data = value;
4     n->next = head;
5     return n;
6 }

```

Push 3, then 2, then 1. Adding at the front changes just one pointer, no matter how long the list is. A Python list is a contiguous array, so inserting at the front means shifting every element over one slot.

Answer to Exercise 18 (on page 59)

```

1 void deleteList(Node* head) {
2     while (head != nullptr) {
3         Node* nextNode = head->next; // save it before deleting!
4         delete head;
5         head = nextNode;
6     }
7 }

```

After delete head, that memory no longer belongs to you, so reading head->next would be reading freed memory. Save the pointer first.

Answer to Exercise 19 (on page 69)

The only thing that changes in our for loop is the writing of the output pixel.

```

1 // CODE ABOVE STAYS THE SAME
2 for (int x = 0; x < w; x++)
3 {
4     for (int y = 0; y < h; y++)
5     {

```

```
6         int idx = y * rowSize + x * 3;
7
8         PIXEL p;
9         BYTE B = data[idx];
10        BYTE G = data[idx + 1];
11        BYTE R = data[idx + 2];
12        p = { B, G, R };
13
14        out[idx + 0] = p.b;
15        out[idx + 1] = 0; // NOTE THE CHANGE HERE
16        out[idx + 2] = p.r;
17    }
18 }
```

What is happening here?

In an RGB image, each pixel's final color is the combination of red, green, and blue intensities. If the green value is removed, every pixel changes from (R,G,B) to (R,0,B). Colors that relied heavily on green—such as greens, yellows, and many skin tones—lose a major part of their intensity and appear much darker or shifted in hue. Pure green areas become black, yellow areas (red + green) become red, and cyan areas (green + blue) become blue.

Visually, the image often takes on a magenta or purplish tint (see Figures 6.3a and 6.3b), because magenta is the combination of red and blue with no green. Overall brightness decreases, since green contributes significantly to the brightness in human vision. Refer to Figures 6.2a and 6.1a for the original bitmaps.

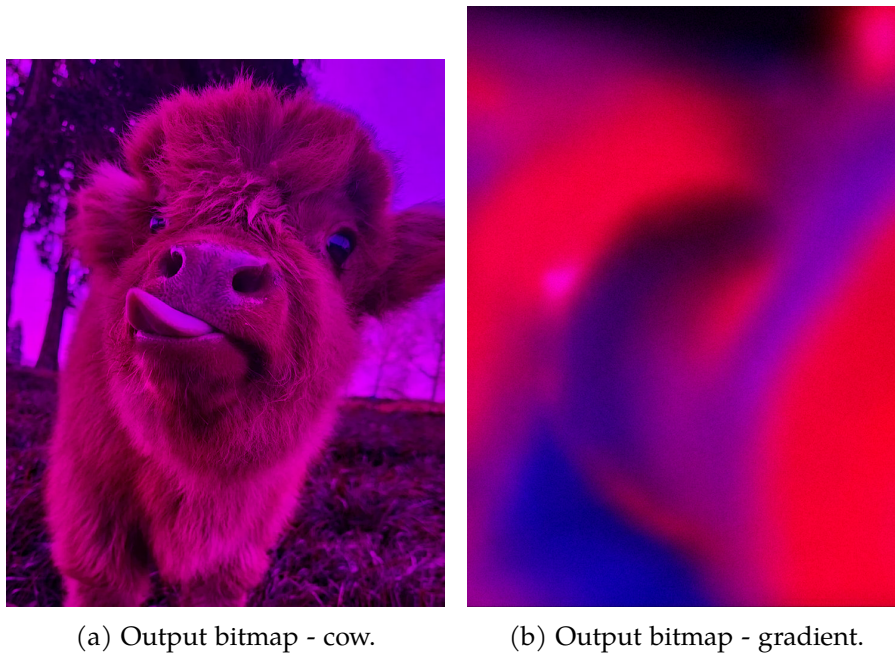


Figure 6.3: Running `bitmap_no_green.cpp` on both provided bitmaps.

Answer to Exercise 20 (on page 73)

1. How many bytes does each pixel use? - 3 bytes per pixel
2. How many bytes are required for one row before padding? $4 \text{ pixels} \times 3 \text{ bytes per pixel} = 12 \text{ bytes}$
3. How many padding bytes are required per row? 0 padding bytes
4. What is the total size of the pixel data? $12 \text{ bytes per row} \times 2 \text{ rows} = 24 \text{ bytes}$
5. What is the total file size, including header structs? 14 bytes for BFH + 40 bytes for BIH = 54 bytes $\Rightarrow 24 + 54 = 78 \text{ bytes}$

Remember that we need to write the rows bottom-up, so the second row is written first followed by the first row.

Let us establish the colors needed. Remember that we need to swap to BGR order:

- Black = (0, 0, 0)
- Red = (0, 0, 255)
- Green = (0, 255, 0)

- Blue = (255, 0, 0)
- Yellow = (0, 255, 255) (R=255,G=255,B=0 $\Rightarrow$ BGR = 0,255,255)
- Magenta = (255, 0, 255) (R=255,B=255 $\Rightarrow$ BGR = 255,0,255)
- Cyan = (255, 255, 0) (G=255,B=255 $\Rightarrow$ BGR = 255,255,0)
- White = (255, 255, 255)

Overall the program looks like this:

```

1  #include <stdio>
2  #include <stdint>
3  #include <string>
4
5
6  #pragma pack(push, 1)
7  struct BITMAPFILEHEADER {
8      uint16_t bfType = 0x4D42; // 'BM'
9      uint32_t bfSize;
10     uint16_t bfReserved1 = 0;
11     uint16_t bfReserved2 = 0;
12     uint32_t bfOffBits = 54;
13 };
14
15 struct BITMAPINFOHEADER {
16     uint32_t biSize = 40;
17     int32_t biWidth = 4;
18     int32_t biHeight = 2;
19     uint16_t biPlanes = 1;
20     uint16_t biBitCount = 24;
21     uint32_t biCompression = 0;
22     uint32_t biSizeImage;
23     int32_t biXPelsPerMeter = 0;
24     int32_t biYPelsPerMeter = 0;
25     uint32_t biClrUsed = 0;
26     uint32_t biClrImportant = 0;
27 };
28 #pragma pack(pop)
29
30 int main(int argc, char const *argv[])
31 {
32     FILE* f = fopen("fourbytwo.bmp", "wb");
33
34     BITMAPFILEHEADER bfh;
35     BITMAPINFOHEADER bih;
36
37     const int width = 4;
38     const int height = 2;
39     const int bytesPerPixel = 3;
40     const int rowSize = ((width * bytesPerPixel + 3) / 4) * 4; // 12
41     const int imageSize = rowSize * height; // 24

```

```

42     bih.biSizeImage = imageSize;
43     bfh.bfSize = bfh.bfOffBits + imageSize;
44
45     fwrite(&bfh, sizeof(bfh), 1, f);
46     fwrite(&bih, sizeof(bih), 1, f);
47     uint8_t row[rowSize];
48
49     // -----
50     // Write bottom row first:
51     // [ Black ] [ Red ] [ Green ] [ Blue ]
52     // -----
53     std::memset(row, 0, rowSize);
54
55     // Black
56     row[0] = 0;   row[1] = 0;   row[2] = 0;
57     // Red (BGR = 0,0,255)
58     row[3] = 0;   row[4] = 0;   row[5] = 255;
59     // Green (BGR = 0,255,0)
60     row[6] = 0;   row[7] = 255; row[8] = 0;
61     // Blue (BGR = 255,0,0)
62     row[9] = 255; row[10] = 0;   row[11] = 0;
63
64     fwrite(row, rowSize, 1, f);
65
66     // -----
67     // Write top row:
68     // [ Yellow ] [ Magenta ] [ Cyan ] [ White ]
69     // -----
70     std::memset(row, 0, rowSize);
71
72     // Yellow (BGR = 0,255,255)
73     row[0] = 0;   row[1] = 255; row[2] = 255;
74     // Magenta (BGR = 255,0,255)
75     row[3] = 255; row[4] = 0;   row[5] = 255;
76     // Cyan (BGR = 255,255,0)
77     row[6] = 255; row[7] = 255; row[8] = 0;
78     // White (BGR = 255,255,255)
79     row[9] = 255; row[10] = 255; row[11] = 255;
80
81     fwrite(row, rowSize, 1, f);
82
83     fclose(f);
84     return 0;
85 }
86

```

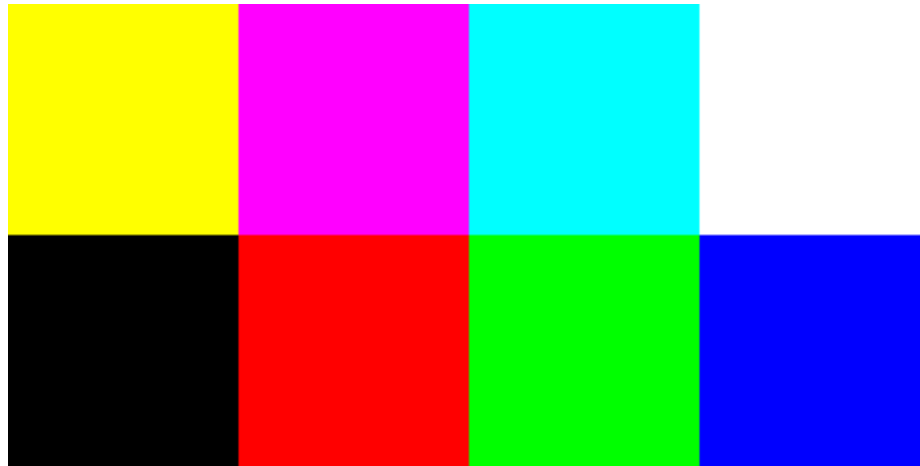


Figure 6.5: Exercise Output of a 4 by 2 bitmap.

Answer to Exercise 21 (on page 81)

{[()][()]}: push {, push [, push (;) pops ((match);] pops [(match); push (;) pops ((match); } pops { (match). The stack ends empty: balanced.

([]): push (, push [; then) pops [, which does not match. Not balanced.

Answer to Exercise 22 (on page 83)

The table below shows front and count after each step. The values that the queue holds are in brackets.

Operation	front	count	Back index
enqueue(1)	0	1	0
enqueue(2)	0	2	1
enqueue(3)	0	3	2
dequeue() returns 1	1	2	–
dequeue() returns 2	2	1	–
enqueue(4)	2	2	$(2 + 1) \bmod 4 = 3$
enqueue(5)	2	3	$(2 + 2) \bmod 4 = 0$

At the end, front is 2 and count is 3. The queue holds 3, 4, and 5, in that order. The value 5 wraps around to index 0.



INDEX

- A-F, 19
- Accents, 14
- acute accent, 14
- address-of operator, 41
- Alphabets, 14
- approximation error, 23
- arrow operator, 46
- ASCII, 13

- base, 27
- base-2, 17
- BGR order, 65
- binary, 17, 25
 - counting in, 17
- bit, 18, 25
- BITMAPINFOHEADER, 62
- bitmaps, 61
 - structure of, 62
- BMP file format, 61
- Boolean, 18
- byte, 18, 25

- C types, 37
- carriage return, 13
- character code, 13
- character encoding, 14
- classes, 53
 - C++, 53
- collation, 14
- compiler, 35
- computer architecture, 18

- continuity, 24
- control characters, 13
- convex, 7
- Cyrillic, 14

- data compression, 32
- data types, 18
- Decimal, 15
- decimal, 15
- decompression, 32
- dequeue, 81
- dereferencing, 42
- Diacritical Marks, 14
- double precision, 29
- doubly linked list, 55

- elliptope, 7
- enqueue, 81
- entropy, 32
 - compression rate, 32
 - formula, 32
- exponent, 25–27
- exponent bounds, 27

- FIFO, 81
- float, 25
- floating-point system, 23
- forward declaration, 39
- free, 46

- gcc, 35
- Goemans-Williamson technique, 9

- head, 53
- head node, 53
- heap, 46
 - arrays, 48
- hello world program, 36
- hex colors, 20
- Hexadecimal, 19
- hexadecimal, 19, 20, 61
- Hindu-Arabic, 15
- Hindu-Arabic numerals, 15
- horizontal flip, 74

- IEEE 754, 29
- IEEE 754 standard, 29
- information theory, 32
- irrational number, 23

- LIFO, 77
- linked list, 53
- little endian, 65
- lossless, 32
- lossy, 32

- machine epsilon, 31
- malloc, 46
- mantissa, 25
- Max Cut, 9
 - python script for , 11
- megabyte, 25
- memory, 18
 - padding, 57
- memory leak, 47
- modf, 43

- natural numbers, 15
- new, 53
- new operator, 53
- newline, 13
- node, 53
- NP problems, 5
- NULL, 42
- null pointer, 54

- Octal, 20
- overflow, 24

- padding, 57, 64
- parametrized, 28

- pass-by-reference, 43
- pass-by-value, 41
- peek, 77, 81
- pointer, 41, 42
- pop, 77
- positive definite matrix, 6
- positive semidefinite matrix, 6
 - constructing, 8
 - non-negative result, 7
- pragma pack, 71
- precision, 27
- predecessor, 55
- predecessor node, 55
- printf, 37
- push, 77
- Python
 - objects under the hood, 48

- quadruple precision, 32
- queue, 81
 - array-based, 82
 - linked-list, 83

- radix, 15
- relaxation
 - semidefinite, 9
- Roman numerals, 15
- rounding, 10, 24
- row-major order, 63

- segmentation fault, 64
- semidefinite programming, 5, 10
- sign, 25, 26
- signed bits, 18
- significand, 25, 26
- single precision, 29
- stack, 40, 77
 - array-based, 78
 - linked-list, 79
 - top, 77
- stack frame, 40
- stack overflow, 79
- static typing, 37
- struct, 44
- struct padding, 57
- successor, 53
- successor node, 53

sys.float_info, [31](#)

tail, [53](#)

tail node, [53](#)

text, [13](#)

text normalization, [14](#)

top, [77](#)

transcendental number, [23](#)

transistors, [25](#)

two's complement, [18](#)

underflow, [24](#)

Unicode, [13](#)

UTF-8, [13](#)

variable

 declaration in C, [36](#)

variable-width encoding, [13](#)