Aaron R. Bradley

Programming for Engineers

August 9, 2011

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

To the curious—

May all that you know illuminate,
All that you learn enlighten,
And all that you discover fulfill.

Preface

To the Student

I have learned the hard way that, when it comes to study habits, nothing is too obvious to state explicitly and repeatedly. Let me take this opportunity, at the start of a new voyage of discovery, to make a few suggestions.

First, reading passively is essentially useless. When reading this or any text, read with pencil in hand. Draw figures to help your understanding. **After reading through an example, close the text and try to reproduce the example.** If you cannot reproduce it, identify where you went wrong, study the text, and try again. Stop only when you can comfortably solve the example problem.

Second, incorporate lectures organically into the study process. Study the relevant reading before each lecture. Engage actively in lectures: take notes, ask questions, make observations. Laugh at the instructor's jokes. **The evening after each lecture, resolve the problems that were presented that day.** You will find that actively reviewing each lecture will solidify material beyond what you might now think is possible. Over the course of the semester, you will probably save time—and you will learn the material better than you would otherwise.

Third, solve exercises in the text even when they are not assigned. Use them to gauge your understanding of the material. If you are not confident that you solved a problem correctly, ask your peers for help or go to office hours. I have provided many exercises with solutions and explanations to facilitate an active approach to learning. Therefore, be active.

Finally, **address confusions immediately**. If you procrastinate on clearing up a point of confusion, it is likely to bite you again and again.

This book introduces a subject that is wide in scope. It focuses on concepts and techniques rather than listing how to use libraries and functions. Therefore, use Internet search engines to locate references on C libraries, particularly starting with Chapter 5; the `man` Unix utility to read about Unix programs; Internet search engines to learn how to use editors like `emacs` and

`vim`; the `help` command in `gdb`; and the `help` and `doc` commands in Matlab. Engineers must learn new powerful tools throughout their careers, so use this opportunity to learn *how to learn*.

To learn to program is to be initiated into an entirely new way of thinking about engineering, mathematics, and the world in general. Computation is integral to all modern engineering disciplines. The better you are at programming, the better you will be in your chosen field. Make the most of this opportunity. I promise that you will not regret the effort.

To the Instructor

This book departs radically from the typical presentation of programming: it presents pointers in the very first chapter—and thus in the first or second lecture of a course—as part of the development of a computational model. This model facilitates an ab initio presentation of otherwise mysterious subjects: function calls, call-by-reference, arrays, the stack, and the heap. Furthermore, it allows students to practice the essential skill of memory manipulation throughout the entire course rather than just at the end. Consequently, it is natural to go further in this text than is typical for a one-semester course: abstract data types and linked lists are covered in depth in Chapters 7 and 8. The computational model will also serve students in their adventures with programming beyond the course: instead of falling back on rules, they can think through the model to decide how a new programming concept fits with what they already know.

Another departure from the norm is the emphasis on programming from scratch. Most exercises do not provide starter code; the use of `gcc` and `make` are covered when appropriate. I expect students to leave the course knowing how to open a text editor, write one or multiple program files, compile the code, and execute and debug the resulting program. Many engineering students will not take an additional course on programming; hence, it is essential for them to know how to program from scratch after this course.

This book covers two programming languages: C and Matlab. The computational model and concepts of modularity are developed in the context of C. Matlab provides an engineering context in which students can transfer, and thus solidify, their mastery of programming from C. Matlab also provides an environment in which students, having learned how to create libraries in Chapters 6–8, can be critical users of libraries. They can think through how complex built-in functions and libraries might be implemented and thus learn techniques and patterns "on the job."

There are strong dependencies among chapters, except that Chapters 8 and 10 may be skipped. Furthermore, Chapter 4 is best left as a reading assignment. Of course, chapters may also be eliminated starting from the ending if time is in short supply.

Your results with my approach may vary. Certainly part of my success with this presentation of the material is a result of my aggressive teaching style and the way that I organize my classes. Two studies in particular influence the way I approach teaching. The first investigates our ability, as students, to self-assess:

> Justin Kruger and David Dunning, *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments*, J. of Personality and Social Psychology, v. 77, pp. 1121-1134, 1999.

The second addresses cause-and-effect in cheating and performance:

> David J. Palazzo, Young-Jin Lee, Rasil Warnakulasooriya, and David E. Pritchard, *Patterns, Correlates, and Reduction of Homework Copying*, Phys. Rev. ST Phys. Educ. Res., v. 6, n. 1, 2010.

My experience in the classroom having confirmed these studies, I administer hour-long quizzes every two to three weeks that test the material that students ought to have learned from the text, from lectures and labs, and from homework. Additionally, I give little weight to homework in the final grade. Therefore, students have essentially no incentive to cheat (themselves out of learning opportunities) on homework—and all the possible incentive to use homework to learn the material. Students have responded well to this structure. They appreciate the frequent feedback, and a significant subset attends office hours regularly. Fewer students fall behind. Consequently, I am able to fit all of the material in this book into one semester. In order to motivate students who start poorly, I announce mid-semester that the final exam grade can trump all quiz grades. Many students seem to learn what they need to know from the quizzes, and so many are better prepared for the final exam.

As side benefits, since enacting this teaching strategy in this and another course, I have never had to deal with an honor code violation—which is rare for introductory programming courses—and have not received a single complaint about a final grade, which is rarer still.

Acknowledgments

I developed the material for this book in preparation for and while teaching a first-year course on programming for engineering students at the University of Colorado, Boulder, partly with the support of an NSF CAREER award.[1] The course was offered in the Department of Electrical, Computer & Energy Engineering (ECEE) and also had students from the Department of Aerospace Engineering Sciences (AES). Thanks to Michael Lightner, the chair of ECEE, for allowing me to teach the course my way. I am grateful to the 77 students

[1] This material is based upon work supported by the National Science Foundation under grand No. 0952617. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

of the Spring 2011 offering for their patience with the new material—and for going along with the experiment and producing the best results of any class that I had taught up to that point. I also thank the teaching assistants—Arlen Cox, Justin Simmons, and Cary Goltermann—for their feedback on the material and on how the students were doing. Peter Mathys, a professor in ECEE, took the course and also provided excellent feedback.

Beyond the people already mentioned, thanks to those outside of the course who volunteered to read parts or all of the manuscript: Andrew Bradley, Caryn Sedloff, Sarah Solter, and Fabio Somenzi. Remaining errors, omissions, awkward phrasing, etc., are of course entirely my fault.

I am grateful to Zohar Manna, my PhD advisor and co-author of my first book, also published by Springer. Besides guiding my first foray into the world of crafting technical books, he showed me what work that stands the test of time looks like.

Sarah Solter, my wife and an accomplished professional software engineer, contributed in multiple ways. She acted as a sounding board for my ideas on how to present programming. As always, she supported me in my quest to do the right things well.

Finally, I thank Ronan Nugent and the other folks at Springer for once again being a supportive and friendly publisher.

ARB
Boulder, CO
June 2011

Contents

1

Memory: The Stack

Computation is mathematics projected onto reality: at one level an interplay of time, space, and procedure; at another, energy. The study of computation has yielded deep insights into the universe of the mind—revealing startling consequences of the mathematics that humans have developed since the beginning of recorded history, like the undecidability of certain questions and the hardness of answering others. It also offers a powerful and practical tool for creating and analyzing complex systems, which is why programming has become a fundamental subject of study for engineers.

In the first three chapters, we embark on a practical study of computation. Our goal is to develop and understand a simple but expressive model of computation that will underlie the material in the remainder of this book—and on which you can subsequently draw when learning more advanced programming skills and concepts. In the first chapter, we introduce memory; in the second, procedure. In the third, we combine memory and procedure to study two basic data structures.

Whereas a traditional programming course reserves "pointers" for late in the semester and may not even mention the stack, let alone how function calling works, this chapter covers both—for two reasons. First, manipulating memory is fundamental to practical programming, yet many students, through lack of practice, leave their first programming course unable to do so effectively. By introducing memory manipulation in the first week, students have a full semester to master the topic. Second, the correct usage of call-by-value, call-by-reference, pointers, and arrays is crucial for writing anything but the simplest of programs. Rather than taking an abstract and rule-based perspective, this chapter covers the program stack and the function call protocol, which naturally give rise to these concepts. A mechanistic understanding of computation lays the foundations for the powerful abstraction methodologies that come later.

1.1 Playing with Memory

1.1.1 A First Foray into Programming

Consider the following code snippet:

```
1 {
2   int a, b, c, d;
3   a = 1;
4   b = 1;
5   c = a + b;
6   d = c + b;
7 }
```

Line 2 declares four **variables** of **type** `int`, short for "integer." This declaration tells the computer to set aside four cells of memory that we shall call `a`, `b`, `c`, and `d`, respectively. Each memory cell can be **read from** and **written to**, and each should be interpreted as holding integer values ($\{\dots, -2, -1, 0, 1, 2, \dots\}$). A memory cell must have a location, which we reference via its **address**. Finally, there is no reason why four variables declared together in the program text should not be neighbors in memory and many reasons why they should be. We visualize the memory using a **stack diagram**:

`int d`	$\otimes$	1012
`int c`	$\otimes$	1008
`int b`	$\otimes$	1004
`int a`	$\otimes$	1000

As a first approximation, a program's memory can be viewed as a contiguous array of memory cells. We visualize memory vertically. In this case, the bottom cell, which we refer to as `a` in our program, is at memory address 1000. Just as we have declared in the program text, the memory for `b` is next to `a` (and at a higher address). Next comes `c`, then `d`. We will discuss why the addresses are the particular values that they are later. Each cell is annotated with its associated variable and the type of that variable. The type indicates how to interpret the data.

The symbol $\otimes$ indicates that a memory cell currently holds **garbage**—that is, a meaningless value left over from the last time this particular memory was used. Since line 2 does not specify **initial values** for the program variables, there is nothing with which to replace the garbage until execution continues.

Line 3 writes the (integer) value 1 to `a`, resulting in a new memory configuration:

`int d`	$\otimes$	1012
`int c`	$\otimes$	1008
`int b`	$\otimes$	1004
`int a`	1	1000

Then line 4 writes the value 1 to `b`, resulting in a similar update to memory.

Line 5 becomes interesting. The instruction `c = a + b` tells the computer to retrieve the values for `a` and `b` from memory, sum them, and then write the sum to the memory cell associated with `c`. After this instruction is executed, memory is configured as follows:

`int d`	⊗	1012
`int c`	2	1008
`int b`	1	1004
`int a`	1	1000

Line 6 describes a similar update, yielding the following configuration:

`int d`	3	1012
`int c`	2	1008
`int b`	1	1004
`int a`	1	1000

Fundamentally, all programs execute in the same manner as this simple program. The reason is simple. Computers operate on a clock. At the beginning of each clock cycle, input values are read from memory; during the cycle, arithmetic occurs over the input values; at the end of the cycle, computed values are written to memory. (I massively oversimplify.) Read, compute, write; read, compute, write; read, compute, write—billions of times per second. This chapter is concerned with reading and writing memory.

Exercise 1.1. Consider this code snippet:

```
1 {
2   int a, b, c;
3   a = 1;
4   a = a + a;
5   a = a + a;
6   b = a;
7   c = a + b;
8 }
```

Fill in the data corresponding to the final memory configuration:

`int c`		1008
`int b`		1004
`int a`		1000

Solution. In this code snippet, `a` is assigned a value multiple times: first 1 at line 3, then 2 at line 4, then 4 at line 5:

`int c`	8	1008
`int b`	4	1004
`int a`	4	1000

□

Exercise 1.2. Consider this code snippet:

```
{
  int a, b, c;
  a = 1;
  b = a + 1;
  c = b + 1;
  a = c + 1;
}
```

Fill in the data corresponding to the final memory configuration:

`int c`		1008
`int b`		1004
`int a`		1000

□

1.1.2 Introduction to Pointers

Memory addresses are nothing more than integers, so we quickly come to the realization that we can manipulate memory using arithmetic. From this insight comes all of programming.

Consider this snippet of code:

```
1 {
2   int a, b;
3   int * x;
4   x = &a;
5   *x = 2;
6   b = *x;
7 }
```

Line 2 is easy enough: it declares two integer variables, `a` and `b`. The next line uses a new symbol that looks like the computer text version of $\times$ (multiplication) but is not. The value of a variable, like `x`, declared with type `int *` is interpreted as a memory address. Furthermore, if the memory cell at the address that `x` holds is accessed, its data is interpreted as being of type `int`, that is, as an integer. As of line 3, memory is configured as follows:

`int * x`	⊗	1008
`int   b`	⊗	1004
`int   a`	⊗	1000

All memory cells hold garbage. Therefore, it would be unwise to use the garbage in `x`'s memory cell as an actual address.

Line 4 uses another new symbol, `&`. Just as `*` is sometimes used for multiplication but has nothing to do with multiplication in our current discussion of memory, `&` has several meanings. In its usage here, `&` is an operator being applied to variable `a`. It computes the address of the memory cell associated with `a`. If we examine the visualization of memory above, we see that `a`'s address is 1000. Therefore, `&a` simply evaluates to 1000, and `x = &a` writes the value 1000 to `x`. After line 4 executes, memory looks as follows:

`int *`	`x`	1000	1008
`int`	`b`	⊗	1004
`int`	`a`	⊗	1000

Now `x` **points to** or **references** `a`: `x` holds the address of `a`'s memory cell. Their types match: `x`, as an `int *`, references an `int` variable; and `a` is indeed an `int` variable. The type `int *` can be read as "pointer to an integer."

Line 5 uses `*` differently than in line 3. In line 3, `*` is part of the variable declaration: it is not being used as a verb (that is, as an operator) but as an adjective. It describes `x` in line 3. In line 5, it is a verb: `*x = 2` tells the computer to write the value 2 to the memory cell whose address `x` currently holds. Since `x` currently holds the value 1000, the computer writes 2 to the memory cell located at address 1000, resulting in the following configuration:

`int *`	`x`	1000	1008
`int`	`b`	⊗	1004
`int`	`a`	2	1000

Finally, line 6 uses `*` in a manner similar but subtly different from its usage in line 5. Here, `*x` is a request to read the datum at the memory cell whose address `x` currently holds. This value is then written to `b`. Since `x` references the memory cell at address 1000, the following memory configuration is obtained:

`int *`	`x`	1000	1008
`int`	`b`	2	1004
`int`	`a`	2	1000

Variables declared with a `*`, as in `int * x`, are traditionally called **pointers** because they "point" to a place in memory. Presentations of pointers often draw arrows coming from a pointer variable's memory cell to the memory cell to which it is pointing. For example, in the memory configuration above, one could draw an arrow from the memory cell associated with `x` to the memory cell associated with `a`. If seeing such arrows would aid your understanding of the memory configurations, then draw them in when convenient. I have elected to emphasize that pointer variables hold data just like other variables by using explicit addresses in illustrations.

It is worth your time to go through this section as many times as necessary until you fully understand the code and the resulting computation. Draw your own memory diagrams rather than relying on the provided ones.

Exercise 1.3. Consider this code snippet:

```
1 {
2   int a;
3   int * x;
4   x = &a;
5   *x = 1;
6   a = *x + a;
7 }
```


Notice that the `*` operator is "stickier," or has higher **precedence**, than the `+` operator, so that `*x + a` is executed as "add the value stored in `a` to the value in the memory cell pointed to by `x`." Fill in the data corresponding to the final memory configuration:

`int *`	`x`		1004
`int`	`a`		1000

Solution. After line 5, the stack is configured as follows:

`int *`	`x`	1000	1004
`int`	`a`	1	1000

Then line 6 modifies `a` again:

`int *`	`x`	1000	1004
`int`	`a`	2	1000

□

Exercise 1.4. Consider this code snippet:

```
{
  int a, b;
  int * x;
  x = &b;
  b = 1;
  a = *x + 1;
}
```

Complete the stack diagram corresponding to the final memory configuration:

`int *`	`x`		1008
`int`	`b`		1004
`int`	`a`		1000

□

1.1.3 Pointers to Pointers

What may now seem like an interesting diversion will be crucial in implementing the sophisticated data structures of Chapter 8 and, of course, those that you encounter subsequently. Therefore, we might as well take the full plunge into pointers. Consider this snippet of code:

```
1 {
2   int a;
3   int * x;
4   int ** y;
5   y = &x;
6   *y = &a;
7   **y = 1;
8 }
```

The initial memory configuration is as follows:

`int **`	`y`	⊗	1008
`int *`	`x`	⊗	1004
`int`	`a`	⊗	1000

All memory cells initially contain garbage, that is, whatever data are left over from the last time the cells were used. Variables `a` and `x` have types that should be familiar, but variable `y`'s type is new: `y` is a *pointer to a pointer to an integer memory cell*. In other words, `y` is intended to reference a memory cell of type `int *` whose own value references a memory cell of type `int`.

Line 5 is where the action begins: `y` is assigned the address of `x`. According to the initial memory configuration, `x`'s address is 1004; hence, the memory configuration after execution of line 5 is the following:

`int **`	`y`	1004	1008
`int *`	`x`	⊗	1004
`int`	`a`	⊗	1000

(You might draw an arrow from `y`'s memory cell to `x`'s memory cell.) Rather than holding garbage, `y` now points to an integer pointer.

At this point, speculate as to what lines 6 and 7 accomplish; draw your own final memory configuration. Check if it matches the remainder of the exposition on this snippet of code. If it doesn't, understand where and why you went awry.

Line 6 assigns the address of `a`, computed with the expression `&a`, to the memory cell at which `y` points. According to the last memory configuration, `y` holds address 1004. Hence, the value of the expression `&a`, which is 1000, is written to the memory cell at address 1004, yielding:

`int **`	`y`	1004	1008
`int *`	`x`	1000	1004
`int`	`a`	⊗	1000

Now `y` points to `x`, and `x` points to `a`. Both are pointing to variables according to their types: `x`, an `int *`, points to an `int`; and `y`, an `int **`, points to an `int *`. Notice how the types can be read in reverse: `int *` is read as "pointer to an integer," while `int **` is read as "pointer to a pointer to an integer."

Line 7, the coda of the code as it were, brings resolution to the flurry of pointer assignments. Whereas `*y = 1` would write a 1 into the memory cell

pointed to by `y`, `**y = 1` writes a 1 into the memory cell pointed to by the memory cell pointed to by `y`. Following the addresses in the previous memory diagram, we see that `y` holds address 1004. At address 1004, we find the value 1000, which is interpreted according to its `int *` type and thus as a pointer to an integer. The 1 is thus written into the memory cell at address 1000, which corresponds to `a`, yielding the final configuration:

`int **`	`y`	1004	1008
`int *`	`x`	1000	1004
`int`	`a`	1	1000

Trace through this code and its execution until you fully understand each line.

A pointer variable, or simply a "pointer," is sometimes called a **reference**, because it refers to a memory location. Applying the `*` operator to a pointer, as in `*x`, is sometimes referred to as **dereferencing** it.

Exercise 1.5. Consider this code snippet:

```
1 {
2   int a;
3   int * x;
4   int ** y;
5   y = &x;
6   x = &a;
7   **y = 1;
8   *x = a + **y;
9   a = *x + **y;
10 }
```

Fill in the data corresponding to the final memory configuration:

`int **`	`y`		1008
`int *`	`x`		1004
`int`	`a`		1000

Solution. After line 7, the stack is configured as follows:

`int **`	`y`	1004	1008
`int *`	`x`	1000	1004
`int`	`a`	1	1000

Then line 8 reads twice from the cell at 1000, adds the two (same) values together, and writes to the same cell:

`int **`	`y`	1004	1008
`int *`	`x`	1000	1004
`int`	`a`	2	1000

Line 9 behaves similarly, except that the value read from the cell is different:

`int **`	`y`	1004	1008
`int *`	`x`	1000	1004
`int`	`a`	4	1000

Hence, `a`, `*x`, and `**y` are all ways of referring to the memory cell at 1000. □

When writing pointer-rich code, one useful trick is to make sure that the number of `*`'s for the type of the expressions on the left and right sides of an assignment agree. (In general, types for the two sides of an assignment should always agree.) For example, in the code snippet of the previous exercise, the type of both expressions `y` and `&x` at line 5 is `int **`; in particular, since `x` is an `int *`, the type of the expression `&x` is `int **`, because it evaluates to the address of a pointer to an integer. Similarly, the type of the expressions at line 6 is `int *`, of those at line 7 is `int` (since dereferencing an `int **` twice yields an integer), and of those at lines 8 and 9 is `int`.

Exercise 1.6. Consider this code snippet:

```
1 {
2   int a, b, * x, * y, ** z;
3   a = 1;
4   x = &a;
5   z = &y;
6   *z = x;
7   b = *y;
8 }
```

Line 2 compactly declares two `int`, `a` and `b`; two `int *`'s, `x` and `y`; and one `int **`, `z`. Fill in the data corresponding to the final memory configuration:

`int **`	z		1016
`int *`	y		1012
`int *`	x		1008
`int`	b		1004
`int`	a		1000

What are the types of the expressions on lines 3–7? □

Exercise 1.7. Consider this code snippet:

```
1 {
2   int a, b, * x, * y, ** z;
3   x = &a;
4   z = &y;
5   *z = &b;
6   *x = 1;
7   *y = 1;
8   **z = a + b;
9 }
```

Fill in the data corresponding to the final memory configuration:

`int **`	z		1016
`int *`	y		1012
`int *`	x		1008
`int`	b		1004
`int`	a		1000

What are the types of the expressions on lines 3–8?

Solution. After line 7, the stack is configured as follows:

`int **`	z	1012	1016
`int *`	y	1004	1012
`int *`	x	1000	1008
`int`	b	1	1004
`int`	a	1	1000

Then line 8 modifies the cell at 1004:

`int **`	z	1012	1016
`int *`	y	1004	1012
`int *`	x	1000	1008
`int`	b	2	1004
`int`	a	1	1000

The types by line are `int *` (line 3), `int **` (line 4), `int *` (line 5), and `int` (lines 6–8). □

Exercise 1.8. Consider this code snippet:

```
1 {
2   int * x, * y, ** z, a, b;
3   z = &y;
4   x = &a;
5   *z = x;
6   *y = 1;
7   **z = 2;
8   *x = 3;
9   b = a;
10 }
```

Fill in the data corresponding to the final memory configuration:

`int`	b		1016
`int`	a		1012
`int **`	z		1008
`int *`	y		1004
`int *`	x		1000

Notice that the memory cells corresponding to variables are ordered according to the order of their declaration. What are the types of the expressions on lines 3–9? □

Exercise 1.9. Write your own pointer-rich code snippet and draw the final memory configuration. Trade puzzles with a few of your colleagues; check each other's work. □

1.1.4 How to Crash Your Program

There is no faster way to crash a program than to make a mistake with memory. (Actually, this statement overstates the case: a program need not crash immediately after an erroneous memory access but can hum merrily and insanely along for a while instead. Fortunately, we have tools, which we discuss in later chapters, to assist us in such situations.) In this section, we take our first look at **bugs**.

Consider this code snippet:

```
1 {
2   int a, b;
3   a = b;
4 }
```

What is the value of `a` at the end of execution? Of `b`? Both variables' memory cells start with garbage, so line 3 merely assigns `b`'s garbage to `a`. At the end of execution, the two memory cells hold equal (and equally meaningless) values. This code snippet illustrates the possibility of unintentionally using **uninitialized** memory, but it won't crash the program.

One method to avoid using uninitialized memory is to initialize variables at declaration:

```
1 {
2   int a = 0, b = 0;
3   a = b;
4 }
```

In practice it is not always possible to find reasonable values to which to initialize variables, and one can still unintentionally use the initializing value when another value was intended. But initializing variables at least avoids the introduction of truly garbage data, data that can be any arbitrary value.

Here is a far more dangerous use of uninitialized variables:

```
1 {
2   int * x;
3   *x = 1;
4 }
```

What happens in line 3? The value 1 is written to somewhere in memory, but to where exactly? The value `NULL`, which is simply a standard way of writing address 0, can be used to initialize pointers:

```
1 {
2   int * x = NULL;
3   *x = 1;
4 }
```


Line 3 will now definitely cause a **segmentation fault**. A segmentation fault occurs when a program reads from or writes to memory outside of the address range allotted to the program by the operating system. Address `NULL` (address 0) is never in a program's memory range. While a segmentation fault is annoying, it is not nearly so annoying as when `*x = 1` silently corrupts a program's data by writing a 1 somewhere (but where?) in memory. Initializing pointers to `NULL` thus causes a buggy program to crash as soon as possible rather than later—or, worse, never—in its execution.

Exercise 1.10. Find the memory error in the following code snippet:

```
1 {
2   int a = 0;
3   int * x;
4   *x = 1;
5 }
```

Would it necessarily crash the program? (Hint: Find an initial value for the pointer that would allow execution to complete but in an unintended way.) At what point would the following variation cause a segmentation fault?

```
1 {
2   int a = 0;
3   int * x = NULL;
4   *x = 1;
5 }
```

Solution. At line 4 of the first code snippet, `x` is uninitialized; hence, its associated memory cell has garbage data. If this garbage happened to form the address corresponding to `a`'s memory cell, then the program would not crash, although `a` would unexpectedly have the value 1 instead of 0.

In the second version, dereferencing `x`, which holds address `NULL`, at line 4 would immediately cause a segmentation fault. □

Exercise 1.11. Find the memory error in the following code snippet:

```
1 {
2   int a, b, * x, * y, ** z;
3   a = 1;
4   z = &y;
5   *z = x;
6   b = *y;
7 }
```

Would it necessarily crash the program? (Hint: Find initial values for the pointers that would allow execution to complete but in an unintended way.) At what point would the following variation cause a segmentation fault?

```
1 {
2   int a = 0, b = 0;
```

```
  int * x = NULL, * y = NULL;
  int ** z = NULL;
  a = 1;
  z = &y;
  *z = x;
  b = *y;
}
```

□

Exercise 1.12. Write your own memory bug puzzle and swap with colleagues. Check each other's work. □

1.2 Functions and the Stack

So far we have only seen examples of static memory usage. However, the memory requirements of a program typically change throughout its execution. The use of the **stack** to facilitate **function calls** is the most fundamental dynamic memory mechanism.

1.2.1 Introduction to Functions

A **function** is a modular unit of computation. It accepts input in the form of variables called **parameters** and possibly produces output in the form of a **return value**. Here is a simple arithmetic function for computing the sum of three integers:

```
1 int sum3(int a, int b, int c) {
2   int sum = 0;
3   sum = a + b + c;
4   return sum;
5 }
```

The function is called `sum3`—a reasonably descriptive name, although any name would do. The function's parameters, or input, are the integer variables `a`, `b`, and `c`. Its output type is given by the leftmost `int` declaration on line 1, and the `return` statement at line 4 indeed returns an integer value, in particular the contents of the `int` variable `sum`. Hence, `sum3` is a function mapping three integers to an integer.[1] This code snippet illustrates how to call `sum3`:

[1] In mathematical notation, one can describe the input–output characteristics of `sum3` as `sum3` : $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, or more compactly, `sum3` : $\mathbb{Z}^3 \to \mathbb{Z}$, where $\mathbb{Z}^3$ is the **domain** of the function and $\mathbb{Z}$ is its **range**. Of course, the actual computer implementation of `sum3` is over integers of fixed maximum magnitude, as we discuss in Section 1.3.


```
1 {
2   int x = 1;
3   x = sum3(x, x, x);
4   x = sum3(x, x, x);
5 }
```

What is the final value of `x`? (At line 2, it is assigned 1; at line 3, it is assigned 3 since `sum3(1, 1, 1)` returns 3; and at line 4, it is assigned 9 since `sum3(3, 3, 3)` returns 9.)

Of all possible function names, one name is reserved for special usage: the `main` function, which is where execution begins when a program is run. The following code forms a full program:

```
 1 int sum3(int a, int b, int c) {
 2   int sum = 0;
 3   sum = a + b + c;
 4   return sum;
 5 }
 6
 7 int main(int argc, char ** argv) {
 8   int x = 1;
 9   x = sum3(x, x, x);
10   x = sum3(x, x, x);
11   return 0;
12 }
```

Line 7 is currently beyond your understanding, but we can use it as a "magic incantation" for now. Briefly, `main`'s input is an array of strings from the command line, represented as the number of elements (`argc`) and the actual array (`argv`). We introduce arrays and strings in Chapter 3 and use them extensively in practice.

Saving this code in file `sum.c` and compiling it with the command `gcc -Wall -Wextra sum.c` yields the executable `a.out`. Execution of `a.out` effectively begins at line 7, not at line 1. It is traditional on Unix variants—e.g., Linux, BSD, AIX, etc.—for `main` to return 0 to indicate successful execution; non-0 values are typically returned to indicate that the program encountered an error or an otherwise exceptional situation during execution.

1.2.2 A Protocol for Calling Functions

Let's examine how functions and memory work together. The C compiler constructs a **stack frame** for every function of the program. A function's stack frame is a template of the function's memory requirements, including space for parameters and its return value as well as declared variables. Consider again the function `sum3`. The stack frame for the function is the following:

int	sum		20
void *	pc		16
int	rv		12
int	c		8
int	b		4
int	a		0

Because this visualized stack frame is a template and not part of the stack, addresses show the offsets of the memory cells relative to the frame. When an instance of a stack frame is placed on the stack, the addresses are instantiated to start at the current top of the stack, as we'll see by example shortly.

The bottom three memory cells correspond to the parameters. Next comes the memory cell reserved for the return value of the function. Since `sum3` returns `int` data, the return value, `rv`, has type `int`.

The next memory cell holds the address of the instruction that should be executed immediately after the return of `sum3`. When a program is compiled, the resulting binary file (called `a.out` by default) is a sequence of machine instructions. Execution proceeds by essentially running the machine instructions in order, except that function calls and control statements (the subject of the next chapter) cause out-of-order execution. The **program counter** is a special **register**, or segment of on-chip memory, in the computer that holds the address of the currently executing machine instruction. When a function call occurs, the address of the subsequent instruction is saved so that, at the end of execution of the function, the computer can recall where to resume. We illustrate this process in detail shortly.

The final memory cell is a result of the **local variable** `sum` of the function `sum3`. Local variables are variables that are declared inside a function; they are only visible within the context of the function in which they are declared, hence their characterization as "local."

Consider the following invocation of `sum3`. To simplify execution, we have omitted the standard parameters of `main`; the resulting code still compiles.

```
1 int main() {
2   int x = 1;
3   x = sum3(x, x, x);
4   return 0;
5 }
```

At the beginning of execution of line 3 of `main`, memory is configured as follows:

int	x	1	1000
void *	pc	"system"	996
int	rv	⊗	992

The addresses are arbitrary, and so we choose ones that are convenient. In particular, 1000 is used throughout the text as the first interesting address. This configuration of memory is directly related to `main`'s stack frame, which

consists of the return value `rv`, the cell `pc` to hold a reference to where execution should return once `main` has completed, and the local variable `x`. The `rv` cell eventually holds the value 0 because of the `return` statement at line 4 but is uninitialized until then. The location "system" refers to the standard code that is inserted into every binary file during compilation: it interfaces between the system and the program, taking care of such tasks as transferring command-line arguments (see Chapter 5) to `main` and `main`'s return value back to the operating system.

We are finally ready to treat program memory as the **stack** that we have been calling it throughout the chapter. The name "stack" is purposely descriptive: think of a stack of plates in a cafeteria. One can **push** data (plates) onto the stack and **pop** data (plates) off the stack. In both cases, the operations affect only the **top** of the stack. Similarly, stack frames are pushed onto and popped off the stack as their corresponding functions are called and return.

Calling the function `sum3` at line 3 causes the following steps to occur, which form the **function call protocol**:

1. The **arguments** to `sum3` are pushed onto the stack. In this case, the three arguments are all 1 because the expression `x` at line 3 evaluates to 1, as memory cell 1000 indicates. The term "arguments" refers to the data that are the input to a function, while the term "parameters" refers to the variables that hold that input from the called function's perspective. In other words, a parameter is a hole; an argument fills a hole. Pushing the arguments yields the following memory configuration:

int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void *	pc	"system"	996
int	rv	⊗	992

   The double line indicates the separation between `main`'s stack frame and the stack frame for `sum3` that is currently under construction.

2. Next, space is made for the data that `sum3` will return:

int	rv	⊗	1016
int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void *	pc	"system"	996
int	rv	⊗	992

   Notice that `rv`'s memory cell holds garbage at this point, since nothing has yet been computed.

3. The computer needs to remember to return to the calling location after execution of `sum3` finishes, so the address of the subsequent instruction is pushed. We represent this address with "line 3+," which indicates that, when execution of `sum3` completes, control should finish the tasks indicated at line 3, in particular, the assignment of the return value (acquired from `rv`) to `x`:

`void *`	`pc`	"line 3+"	1020
`int`	`rv`	⊗	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The type `void *` describes a pointer to an address corresponding to any type of data, which in this case is a machine instruction.

4. Finally, space for `sum3`'s local variables is allocated. The code for `sum3` initializes `sum` to 0:

`int`	`sum`	0	1024
`void *`	`pc`	"line 3+"	1020
`int`	`rv`	⊗	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

This step yields `sum3`'s full stack frame, just as it was described above except that the parameters and `pc` have context-specific values.

Whew! That's a lot of work! And the execution of `sum3` hasn't even begun.

Execution of `sum3` begins when the program counter is updated to point to the first of the machine instructions that `sum3` compiled into. The execution of the statements `sum = a + b + c` and `return sum` yield the following memory configuration:

`int`	`sum`	3	1024
`void *`	`pc`	"line 3+"	1020
`int`	`rv`	3	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The statement `return sum` writes the value of `sum` into `rv`'s memory cell.

With `sum3` completed, it is time to deconstruct `sum3`'s stack frame and return control to the calling context. The following steps of the **function return protocol** accomplish these tasks:

1. Memory for local variables is popped:

`void *`	`pc`	"line 3+"	1020
`int`	`rv`	3	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

2. The program counter is restored from `pc`, and then memory for `pc` is popped:

`int`	`rv`	3	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

3. Control is now back at line 3 of the calling context: `x = sum3(x, x, x)`. The variable `x`, local to `main`, is updated according to `rv`, and `rv` is popped:

`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	3	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

4. The arguments are popped:

`int`	`x`	3	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

As you perhaps predicted, the final value of `x` is 3.

Exercise 1.13. Walk through the more complicated `main` of the previous section to check your understanding:

```c
int main(int argc, char ** argv) {
  int x = 1;
  x = sum3(x, x, x);
  x = sum3(x, x, x);
  return 0;
}
```

The final value of this `main`'s `x` should be 9. As you step through this exercise, notice how the analogy of a stack of plates is apt: the stack grows with the first call to `sum3`, then shrinks, then grows again with the second call to `sum3`, then shrinks. Each function manipulates the memory near the top of the stack.

Since `main` has parameters, the initial memory configuration is as follows:

`int`	`x`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992
`char **`	`argv`		988
`int`	`argc`		984

For now, we ignore the possible initial values of `argc` and `argv`. Chapter 5 discusses their usage in depth. □

While modern architectures facilitate more efficient function call and return protocols through the use of on-chip memory (registers), the protocols for calling and returning from a function presented here are representative of those employed by typical compilers and architectures. Furthermore, the treatment of memory as a stack is fundamental. These protocols and the stack are important components of our computational model.

Exercise 1.14. Consider the following `main` function:

```c
int main() {
  int a, * x;
  x = &a;
  *x = sum3(1, 2, 3);
  a = sum3(*x, a, *x);
  return 0;
}
```

Trace through the execution of the program, and draw the critical memory configurations. □

Exercise 1.15. Consider the following program, which calls a function that multiplies a given number by 10 using only addition:

```c
 1 /* Computes and returns ``10 * a'' without using
 2  * multiplication.
 3  */
 4 int times10(int a) {  // input: int a, output: an int
```



```c
 5   int x, y;             // local variables
 6   x = a + a;            // 2 * a
 7   y = x + x;            // 4 * a
 8   y = y + y;            // 8 * a
 9   return x + y;         // 2*a + 8*a == 10*a
10 }
11
12 int main() {
13   int n = 42;           // n is local to main
14   n = times10(n);
15   return 0;
16 }
```

While the multiplication operator `*` is available, the sequence of additions at lines 6–9 can be faster than multiplying by 10 on some platforms.

Trace through the execution of this program, and draw the critical memory configurations. Remember that execution begins in the `main` function.

Solution. The stack frame for `main` consists of the return value, the cell `pc` to hold a reference to where execution should return once `main` has completed, and one memory cell for the local variable `n`, which is initialized to 42:

`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Line 14 calls `times10`, so the stack frame for `times10` is pushed:

`int`	`y`	⊗	1020
`int`	`x`	⊗	1016
`void *`	`pc`	"line 14+"	1012
`int`	`rv`	⊗	1008
`int`	`a`	42	1004
`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Notice how the parameter `a` is initialized to the value of the argument `n`. Next, lines 6–7 execute, yielding the following configuration:

`int`	`y`	168	1020
`int`	`x`	84	1016
`void *`	`pc`	"line 14+"	1012
`int`	`rv`	⊗	1008
`int`	`a`	42	1004
`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Line 8 computes the final value for `y`; then the `return` statement at line 9 causes the value 420 to be written to the memory cell corresponding to `times10`'s return value:

`int`	`y`	336	1020
`int`	`x`	84	1016
`void *`	`pc`	"line 14+"	1012
`int`	`rv`	420	1008
`int`	`a`	42	1004
`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

and the local memory to be popped:

`void *`	`pc`	"line 14+"	1012
`int`	`rv`	420	1008
`int`	`a`	42	1004
`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The `pc` cell at the top of the stack allows control to return to line 14, where the task of assigning `n` remains; once there, `pc` can be popped:

`int`	`rv`	420	1008
`int`	`a`	42	1004
`int`	`n`	42	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The assignment to `n` then occurs: the value in the `rv` cell at the top of the stack is transferred to `n`, and the remainder of `times10`'s stack frame is popped:

`int`	`n`	420	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The `return` statement at line 15 assigns 0 to `main`'s return value:

`int`	`n`	420	1000
`void *`	`pc`	"system"	996
`int`	`rv`	0	992

Finally, local memory is popped, and the `pc` and `rv` cells are used to return to the system code, at which point the remainder of the stack is popped. □

Exercise 1.16. Consider replacing the `main` of the program of Exercise 1.15 with the following `main`:


```
int main() {
  int x;
  x = times10(12);
  x = times10(x);
  return 0;
}
```

Trace through the execution of this alternate program, and draw the critical memory configurations. □

1.2.3 Call-by-Value and Call-by-Reference

Suppose that we want to write a function that computes division just as you did in elementary school: given a dividend (the number being divided) and a divisor, it should compute a quotient and a remainder. For example, dividing 7 (the dividend) by 3 (the divisor) yields a quotient of 2 and a remainder of 1. How can we return two values from the function? Consider the following implementation, which uses **call-by-value** semantics for the first two parameters and **call-by-reference** semantics for the latter two:

```
void divide(int dividend, int divisor,
            int * quotient, int * remainder) {
  *quotient = dividend / divisor;
  *remainder = dividend % divisor;
  return;
}
```

The `/` and `%` operators compute **integer division** and **modulo**, respectively.[2] The return type of `void` indicates that `divide` does not return any value through the `return` statement.

The idea of call-by-reference is to use pointer parameters to update data in the caller's stack frame. Let's visualize the following call to `divide`:

```
1 int main() {
2   int q, r;
3   divide(7, 3, &q, &r);
4   return 0;
5 }
```

At the beginning of line 3 of `main`, the stack is as follows:

`int`	`r`	⊗	1004
`int`	`q`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

[2] More precisely, the **modulus operator** computes the remainder when applied to nonnegative integers, but its operation on negative integers is machine dependent.

The function call builds up the stack frame for `divide`:

`void *`	`pc`	"line 3+"	1024
`int *`	`remainder`	1004	1020
`int *`	`quotient`	1000	1016
`int`	`divisor`	3	1012
`int`	`dividend`	7	1008
`int`	`r`	⊗	1004
`int`	`q`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Notice that, since `divide`'s return type is `void`, indicating that it does not return a value, a memory cell for a return value is not pushed. Furthermore, `divide` does not have any local variables. Hence, its stack frame consists of its parameters and `pc`.

Study the memory configuration carefully. What are the arguments to `divide`? Consequently, to which values are its parameters initialized? In particular, where do `quotient` and `remainder` point? Trace through the execution of `divide`. Do you get the following memory configuration at line 5 of `divide`?

`void *`	`pc`	"line 3+"	1024
`int *`	`remainder`	1004	1020
`int *`	`quotient`	1000	1016
`int`	`divisor`	3	1012
`int`	`dividend`	7	1008
`int`	`r`	1	1004
`int`	`q`	2	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

When `divide` returns, memory is configured as follows:

`int`	`r`	1	1004
`int`	`q`	2	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

This configuration—particularly the values of `q` and `r`—is precisely what we hoped to obtain from the call to `divide`.

Exercise 1.17. Trace through the execution of the following program, and draw the critical memory configurations:

```
1  void incr(int * x) {
2    *x = *x + 1;
3  }
4
5  int main() {
6    int a = 0;
7    incr(&a);
8    incr(&a);
9    return 0;
10 }
```


Although `incr` lacks an explicit `return` statement, it behaves as if it has one after line 2.

Solution. The stack frame for `main` consists of the return value, the `pc` cell, and one memory cell for the local variable `a`, which is initialized to 0:

`int`	`a`	0	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Line 7 calls `incr`, so the stack frame for `incr` is pushed:

`void *`	`pc`	"line 8"	1008
`int *`	`x`	1000	1004
`int`	`a`	0	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Notice that, since `incr` has a `void` return type—that is, it does not return anything—the stack frame lacks a cell for a return value. Also, since line 7 does not include an assignment, control returns to line 8 upon `incr`'s completion. Line 2 then executes to increment the value in the cell associated with `a`:

`void *`	`pc`	"line 8"	1008
`int *`	`x`	1000	1004
`int`	`a`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Control then returns to line 8:

`int`	`a`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Another call to `incr` is executed, yielding the following configuration just before `incr` returns:

`void *`	`pc`	"line 9"	1008
`int *`	`x`	1000	1004
`int`	`a`	2	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Upon return, `a` has value 2. The `return` statement sets `main`'s return value to 0:

int	a	2	1000
void *	pc	"system"	996
int	rv	0	992

Finally, local memory is popped, and the `pc` and `rv` cells are used to return to the system code, at which point the remainder of the stack is popped. □

Exercise 1.18. Trace through the execution of the following program, and draw the critical memory configurations:

```
void incrBy(int * x, int a) {
  *x = *x + a;
}

int main() {
  int a = 0;
  incrBy(&a, 3);
  incrBy(&a, a);
  return 0;
}
```

Notice that the parameter `a` of `incrBy` is unrelated, except in name, to the variable `a` of `main`; in particular, they correspond to distinct memory cells. □

1.2.4 Building Fences

Functions are the basic unit of modularity in programs. As such, functions can be executed in contexts that you, as the function writer, may not have predicted. Hence, it's good practice to protect the function that you're writing. What are potential problems that could occur if a naive user calls `divide`? Here are two:

- The `divisor` may be 0, which would lead to a divide-by-zero runtime error. Additionally, we may want to assume that the divisor is always positive, as you probably did in elementary school.
- The `quotient` or the `remainder` parameters may be `NULL`, leading to a segmentation fault.

A standard method of protecting code is to use **assertions**, which are checked at runtime. If the assertion does not hold, the program stops with a message so that the programmer can fix the problem. Here is how we might use assertions for `divide`:


```
1 void divide(int dividend, int divisor,
2             int * quotient, int * remainder) {
3   assert (divisor > 0);
4   assert (quotient != NULL);
5   assert (remainder != NULL);
6   *quotient = dividend / divisor;
7   *remainder = dividend % divisor;
8   return;
9 }
```

These assertions are being used as **function preconditions**. They define the (pre)conditions that must hold for the function to behave correctly. If, say, the user were to pass a divisor of 0, the assertion at line 3 would be triggered: the program would print a message to the console indicating that the assertion failed and then abort.

Another worthwhile discipline is to use assertions to state expectations, although this task can be much more difficult than stating preconditions. In `divide`, we expect a certain arithmetic property to hold upon completion of execution, namely the following:

```
1 void divide(int dividend, int divisor,
2             int * quotient, int * remainder) {
3   assert (divisor > 0);
4   assert (quotient != NULL);
5   assert (remainder != NULL);
6   *quotient = dividend / divisor;
7   *remainder = dividend % divisor;
8   assert (divisor * (*quotient) + (*remainder) == dividend);
9   return;
10 }
```

In typical C fashion, the character `*` means different things in different contexts. At line 8, it is being used once to indicate multiplication and twice to dereference pointers. The assertion at line 8 states the key property of division: the sum of the remainder and the product of the divisor and the quotient yields the dividend. This assertion is being used as a **function postcondition**. It states the condition that is expected to hold after execution of the function, that is, just before it returns.

The value of assertions is that they identify the effect of a bug near the buggy lines. Furthermore, they can be deactivated during compilation when performance is desired.

Assertions are defined in the **standard library** `assert.h`, so we need to **include** the `assert` library in the source file. In fact, we also need to include `stdlib.h`, which defines `NULL`:

```
#include <assert.h>
#include <stdlib.h>

void divide(int dividend, int divisor,
            int * quotient, int * remainder) {
  assert (divisor > 0);
  assert (quotient != NULL);
```

```
  assert (remainder != NULL);
  *quotient = dividend / divisor;
  *remainder = dividend % divisor;
  assert (divisor * (*quotient) + (*remainder) == dividend);
}

int main() {
  int q, r;
  divide(7, 3, &q, &r);
  return 0;
}
```

We omitted the `return` statement of `divide` in this version because it is not necessary when the function does not have a return value.

Assertions are not always desirable for functions that should work in any environment. In the next chapter, we add a return value to `divide` that indicates whether there is an input error.

Exercise 1.19. Write a function that swaps the values of two `int` variables. It should have the following **prototype**, or interface:

```
void swap(int * a, int * b);
```

For example, calling `swap(&x, &y)` should result in `y`'s having `x`'s original value and `x`'s having `y`'s original value. Write a `main` function that calls `swap`. Using assertions, write function preconditions and postconditions. Illustrate various interesting memory configurations during its execution. Write the code in a file called `swap.c`, compile it, and run it.

Solution. The following program tests the `swap` function. The use of the entry function `main` is as a **unit test** of the function `swap`: it tests `swap` in a specific environment. Writing unit tests—that is, tests of modules such as functions or, in Chapter 7, abstract data types—is good engineering practice. In large programming efforts, it is desirable to catch as many bugs as possible before attempting to integrate many units. Writing unit tests in `main` functions is one method of unit testing.[3]

```
#include <assert.h>
#include <stdlib.h>

void swap(int * x, int * y) {
  assert (x != NULL);
  assert (y != NULL);
  int t = *x;
  *x = *y;
  *y = t;
}

int main() {
  int a = 0, b = 1;
  swap(&a, &b);
  assert (a == 1);
  assert (b == 0);
  return 0;
}
```

[3] In Chapter 5, we will write general `main` functions in order to make general-purpose programs. Then unit tests can take the form of external scripts that call the program with various command-line arguments.

To compile and run the resulting executable, we run the following on the terminal:

```
$ gcc -Wall -Wextra -o swap swap.c
$ ./swap
```

Nothing is printed to the terminal; however, an assertion failure would be obvious, so we conclude that `swap` passed the test.

Out of curiosity, let's implement `swap` incorrectly to see an assertion failure. We modify `swap` as follows:

```
void swap(int * x, int * y) {
  assert (x != NULL);
  assert (y != NULL);
  // Wrong!
  *x = *y;
  *y = *x;
}
```

Again, we compile and run the program:

```
$ gcc -Wall -Wextra -o swap swap.c
$ ./swap
swap: swap.c:16: main: Assertion ‘b == 0’ failed.
Aborted
```

The assertion failure points to a mistake in the implementation of `swap`. □

Exercise 1.20. Write a function that swaps the values of three `int` variables. It should have the following prototype:

```
void swap3(int * a, int * b, int * c);
```

For example, calling `swap3(&x, &y, &z)` should result in `z`'s having `y`'s original value, `y`'s having `x`'s original value, and `x`'s having `z`'s original value. Use assertions to protect the function. Write a unit test of `swap3` in a `main` function. Illustrate various interesting memory configurations during its execution. How can `swap3` be called in order to swap the values of two variables rather than three, given that three arguments must be passed? □

1.3 Bits, Bytes, and Words

As long as we are discussing computer memory, it is worth a brief aside to discuss how computers actually represent data. Computers work in **binary**, or **base 2** arithmetic, instead of **decimal**, or **base 10** arithmetic. While knowing how to compute in binary arithmetic is not essential, the basics of base 2 arithmetic explain why addresses so far have been multiples of 4. So let's take a brief tour of binary arithmetic.

0 and 1 are just, well, 0 and 1. But 10 in binary is 2 in decimal, and 100 in binary is 4 in decimal. Here is the general case. To convert the binary number

$$d_k d_{k-1} \ldots d_1 d_0 \ ,$$

where each digit d_i is either 0 or 1, compute

$$\sum_{i=0}^{k} d_i 2^i \ .$$

For example, 1001101 in binary is

$$1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 \ ,$$

which simplifies to $1 + 4 + 8 + 64$, or 77 in decimal.

There is a formal way of converting from decimal to binary, but the easiest on-the-fly method is simply to subtract largest powers of 2 until you are left with 0. For example, 29 in base 10 is computed as 11101 in base 2:

$$\begin{aligned} 29 &= 2^4 + 13 \\ &= 2^4 + 2^3 + 5 \\ &= 2^4 + 2^3 + 2^2 + 1 \\ &= 2^4 + 2^3 + 2^2 + 2^0 \\ &= 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \end{aligned}$$

Just as powers of 10 are important in decimal, powers of 2 are important in binary. For that reason, memory is divided hierarchically into powers of two. A **bit** is one binary digit: either 0 or 1. A **nybble** is four (2^2) bits, and a **byte** is eight (2^3) bits. A **word** is not standard: 32-bit architectures have 32-bit, or 4-byte, words; 64-bit architectures have 64-bit, or 8-byte, words. We assume 32-bit words in this text to keep address arithmetic more manageable.

How many different values can a bit take on? Two, of course: 0 or 1. How many different values can a byte take on? The smallest byte is 00000000, that is, 0 in decimal; the largest byte is 11111111, that is, 255 in decimal. Hence, a byte can take on $2^8 = 256$ different values. A 32-bit word can take on $2^{32} = 4,294,967,296$ different values—a lot but still a long way from infinitely many. Basic computer arithmetic is limited by the finiteness of number representations. For example, in computer arithmetic, adding 1 to an `int` value of $2^{31} - 1 = 2,147,483,647$ yields $-2^{31} = -2,147,483,648$ for reasons that are beyond the scope of this text.[4]

[4] Read about **two's complement** representation if you are curious.

In our computational model, addresses refer to bytes and occupy 32 bits—that is, four bytes, or one word. Therefore, memory cells corresponding to pointer and `int` variables both occupy four bytes, so that memory cells have addresses that are typically multiples of 4. In later chapters, we encounter variable types that require different numbers of bytes.

Exercise 1.21.

(a) Compute the binary representation of 89. Pad it with 0s so that it consumes a byte.
(b) Compute the decimal representation of 01101001.
(c) How can you tell if a binary number is even or odd? If it is a multiple of 4? Of 8? Of 32?
(d) Write a list of random decimal and binary numbers, and convert them back and forth.

□

Exercise 1.22. Explain why the function `times10` of Exercise 1.15 works. □

2

Control

Computation rests on two foundations: memory and control. Having developed a memory model in Chapter 1, this chapter extends the computational model with **control** statements. Such statements direct the flow of computation: the `if`/`else` construct enables conditional computation; and the `while` and `for` constructs facilitate iterative computation. Function calls, when combined with conditional statements, can yield even more complex control in the form of recursion.

2.1 Conditionals

The most basic control statement is the **conditional**. Consider this improvement of `divide`, which checks the input and either returns –1, indicating malformed input, or computes the quotient and remainder and returns 0, indicating a successful computation:[1]

```
1  int divide(int dividend, int divisor,
2             int * quotient, int * remainder) {
3    if (divisor <= 0 ||
4        quotient == NULL ||
5        remainder == NULL) {
6      // error: malformed input
7      return -1;
8    }
9    else {
10     *quotient = dividend / divisor;
11     *remainder = dividend % divisor;
```

[1] Returning a negative integer to indicate an error or 0 to indicate success is a custom based on this observation: "There are many ways of messing up, but only one way of getting it right." However, some libraries, including some standard C libraries, use other customs. For example, some functions may return 0 or 1 to indicate an error or successful completion, respectively.


```
12     // successful computation
13     return 0;
14   }
15 }
```

Lines 6 and 12 are **comments**, which are ignored by the compiler but are intended to be useful to the reader. Lines 3–5 check if `divisor <= 0` or `quotient == NULL` or `remainder == NULL`. The operator `<=` is read as "less than or equal to" or "at most," while the operator `||` is read as "or." Because `=` is reserved for assignment, `==` is read as "equals." If any one (or more) of the predicates is true, then the **block** of code after the `if` is executed; otherwise, the block of code after the `else` is executed.

A caller could then check for an indication of an error:

```
1 int main() {
2   int q, r;
3   int errorCode = divide(7, 3, &q, &r);
4   assert(!errorCode);
5   return 0;
6 }
```

In this case, the error checking is minimal. The `!` operator is read as "not": `!0` is 1, while `!`n is 0 for any $n \neq 0$. Since an `assert` is triggered if its argument is false, which in C is 0, the assertion at line 4 is triggered precisely when `divide` returns -1, that is, when its input is malformed. While the overall effect in this particular use of `divide` is the same as in the previous chapter, the idea is that this new version of `divide` allows the caller to recover from an error if appropriate.

We have seen two **logical operators** so far: "or," `||`, and "not," `!`. The operator `&&` is read as "and." Using `&&`, `divide` can be implemented equivalently as follows:

```
int divide(int dividend, int divisor,
           int * quotient, int * remainder) {
  if (divisor > 0 &&
      quotient != NULL &&
      remainder != NULL) {
    *quotient = dividend / divisor;
    *remainder = dividend % divisor;
    // successful computation
    return 0;
  }
  else {
    // error: malformed input
    return -1;
  }
}
```

Logical operators are also called **Boolean operators** after George Boole, whose contribution to mathematics includes the study of Boolean algebras. One particular Boolean algebra is the algebra of logical 0 and 1, also called "false" and "true," respectively. Here are some basic identities written using C syntax:

- `(!0) == 1`, `(!1) == 0`;
- when `x` is 0 or 1, `(!!x) == x`;
- `(x && y) == (y && x)`,
  `(x || y) == (y || x)`;
- `(x && (y && z)) == ((x && y) && z)`,
  `(x || (y || z)) == ((x || y) || z)`;
- `(0 && x) == 0`,
  `(1 && x) == x`;
- `(0 || x) == x`,
  `(1 || x) == 1`;
- `(!(x && y)) == (!x || !y)`,
  `(!(x || y)) == (!x && !y)`.

Developing an intuition for logical arithmetic is useful in programming because conditional statements are sometimes complex.

Exercise 2.1. Apply these identities to solve the following problems:

(a) Manipulate `!(x && (y || !z))` so that `!` is only applied to variables.
Solution. One application of the penultimate identity above, known as De Morgan's law, yields `!x || !(y || !z)`; an application of its dual, the final identity, yields `!x || (!y && !!z)`; and an application of the second identity yields `!x || (!y && z)`.
(b) Write an expression equivalent to `x || y || z` that uses only `!` and `&&`.
(c) Write your own logic manipulations and trade with your colleagues.

□

Conditional statements can extend beyond two options. Consider the following function, which computes the "sign" of an integer: it returns −1, 0, or 1 if the given integer is negative, 0, or positive, respectively:

```
int sign(int x) {
  int s = 0;
  if (x < 0)
    s = -1;
  else if (x == 0)
    s = 0;
  else
    s = 1;
  return s;
}
```


Notice that this code snippet does not use braces (`{` and `}`) for the conditional blocks. Braces are not required when a block consists of only one statement. However, one must be careful to avoid introducing bugs by accidentally omitting braces.

A function can have multiple `return` statements, a freedom that becomes relevant with control. The following is a functionally equivalent but more concise version of `sign`:

```
int sign(int x) {
  if (x < 0)        return -1;
  else if (x == 0)  return 0;
  else              return 1;
}
```

Notice that spacing can be used to clarify (or obscure) code.

Exercise 2.2. Modify the `swap` function of Exercise 1.19 so that it check its input and returns −1 if it is malformed and 0 otherwise.

Solution. Rather than asserting that neither `x` nor `y` is `NULL` as in Exercise 1.19, which causes the program to abort on bad input, we use an `int` return value to indicate whether the function executes successfully. If either is `NULL`, the function returns −1; otherwise, it executes normally and returns 0:

```
#include <assert.h>
#include <stdlib.h>

int swap(int * x, int * y) {
  if (x == NULL || y == NULL) return -1;
  int t = *x;
  *x = *y;
  *y = t;
  return 0;
}

int main() {
  int a = 0, b = 1;
  int rv = swap(&a, &b);
  assert (rv == 0);
  assert (a == 1);
  assert (b == 0);
  rv = swap(&a, NULL);
  assert (rv != 0);
  assert (a == 1);
  return 0;
}
```

The unit test implemented in `main` tests both normal and abnormal situations for `swap`. The second call to `swap` would cause the program to abort with the old version of `swap`. □

Exercise 2.3. Modify the `swap3` function of Exercise 1.20 so that it check its input and returns −1 if it is malformed and 0 otherwise. □

Exercise 2.4. Write a function that returns the absolute value of an integer variable. It should have the following prototype:

```
int abs(int a);
```

Write a unit test of `abs` in `main`.

Solution. We explore various equivalent ways of implementing this function. Given that this function is so simple, the variety in even this example suggests that, as we tackle ever more interesting programming problems, there will be ever greater freedom in the design and implementation choices.

The first implementation is verbose but straightforward:

```
#include <assert.h>

int abs(int a) {
  int x;
  if (a < 0) {
    x = -a;
  }
  else {
    x = a;
  }
  return x;
}

int main() {
  int x = -3;
  int y = abs(x);
  assert (x == y || -x == y);
  assert (y >= 0);
  x = abs(y);
  assert (y == x);
  return 0;
}
```

There are two tests in `main`: `abs` should return a nonnegative number that is equal in magnitude to the original number, and it should leave a positive number unchanged.

In this variation, we realize that we don't need a local variable:

```
int abs(int a) {
  if (a < 0)
    a = -a;
  return a;
}
```

In the final variant, we realize that we don't need to change the value of `a` at all but can instead use multiple `return` statements:


```
int abs(int a) {
  if (a < 0) return -a;
  return a;
}
```

□

Exercise 2.5. Write a function that computes the minimum and the maximum of two integer variables and returns them through call-by-reference parameters. It should have the following prototype:

```
int minmax(int a, int b, int * min, int * max);
```

Write a unit test of `minmax` in a `main` function. □

2.2 Recursion

According to the Church–Turing thesis, you have now learned all the tools necessary to compute anything that is theoretically computable—were memory and time unlimited. Does this statement surprise you? For that matter, have you ever thought about what is and is not computable? An entire branch of knowledge called **computability theory** has evolved from the pioneering work of Gödel, Church, Turing, and others.

To get a taste of just how powerful the combination of the stack, functions, and conditional statements are, let's implement a short function that computes the sum $1 + 2 + \cdots + n$, for a given positive integer n:

```
1  int sum(int n) {
2    int upto = 0;
3    // n must be positive
4    assert (n > 0);
5    if (n == 1)
6      // the sum of 1 is just 1
7      return 1;
8    else {
9      // the sum 1 + ... + n == (the sum 1 + ... + n-1) + n
10     upto = sum(n-1);
11     return upto + n;
12   }
13 }
```

Line 4 asserts that `n` is positive, which is according to the English specification of the function given above. Then, if `n == 1`, the function simply returns 1: the sum of 1 is 1. For the general case, we recognize that

$$1 + \cdots + n = (1 + \cdots + (n-1)) + n \ ,$$

because addition is associative. Thus, to compute the sum $1 + \cdots + n$, `sum` simply needs to compute the sum $1 + \cdots + (n - 1)$ and then add n, which is what lines 10–11 accomplish.

Let's trace through a call to `sum` arising in the following context:

```
1 int main() {
2   int s = sum(3);
3   return 0;
4 }
```

At entry, memory has the following configuration:

`int`	`s`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The call at line 2 causes `sum`'s stack frame to get pushed:

`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

The location `main:2+` refers to the address of the machine instructions that must be executed after `sum` returns, which corresponds to the assignment of the return value to `s`. `sum(3)` executes. The second conditional block is executed because `3 != 1`. Line 10 of `sum` calls `sum` again, so that a second instance of `sum`'s stack frame is pushed:

`int`	`upto`	0	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	⊗	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Notice how, in the new instance, the parameter `n` is initialized to 2 and the program counter is set to be restored to line 10 of `sum` upon return.

Once again, the second conditional block is executed because `2 != 1`, and another stack frame is pushed:

`int`	`upto`	0	1048
`void *`	`pc`	`sum:10+`	1044
`int`	`rv`	⊗	1040
`int`	`n`	1	1036
`int`	`upto`	0	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	⊗	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

This time, the parameter is initialized to 1. Therefore, the first conditional block is executed so that the return value is set to 1:

`int`	`upto`	0	1048
`void *`	`pc`	`sum:10+`	1044
`int`	`rv`	1	1040
`int`	`n`	1	1036
`int`	`upto`	0	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	⊗	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992

Then control returns to the calling context, where `upto` is set to the return value, and the expended stack frame is popped:

`int`	`upto`	1	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	$\otimes$	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	$\otimes$	1008
`int`	`n`	3	1004
`int`	`s`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Control now proceeds to line 11, where the sum `upto + n` is computed and stored in the return value:

`int`	`upto`	1	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	3	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	$\otimes$	1008
`int`	`n`	3	1004
`int`	`s`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Then control returns to the calling context, where `upto` is set to the return value, and the expended stack frame is popped:

`int`	`upto`	3	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	$\otimes$	1008
`int`	`n`	3	1004
`int`	`s`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Control now proceeds to line 11, where the sum `upto + n` is computed and stored in the return value:

`int`	`upto`	3	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	6	1008
`int`	`n`	3	1004
`int`	`s`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Finally, control returns to the calling context, where `s` is set to the return value, the expended stack frame is popped, and `main`'s `rv` is set to 0:

`int`	`s`	6	1000
`void *`	`pc`	"system"	996
`int`	`rv`	0	992

Execution of the program then completes.

Study this section until you understand precisely how the computer executes this program.

This example demonstrates **recursion**, which is the most powerful technique for writing programs that do an amount of work dependent on input.

Exercise 2.6. To make `sum` callable in any context, it would be best to remove the need for the assertion at line 4.

(a) Rename `sum` to `_sum`. Adding an underscore (`_`) at the beginning of a function name is a common naming convention to indicate that it is a function that is not intended to be called outside of a specific context.

(b) Write an entry function called `sum` with the following prototype:

```c
int sum(int n, int * s);
```

The return value should be used to indicate whether the input is malformed, in particular if `n <= 0` or `s == NULL`. As usual, it should return 0 to indicate successful execution and a negative value to indicate an error. The sum itself should be returned via the reference `s`. After checking that the input is well formed, `sum` should call `_sum`, which should perform the main computation.

(c) Remove the protection in `_sum` to optimize the implementation.

Solution. The function `_sum` does the hard work. Unlike the original version of `sum` above, it does not protect itself against spurious input because it is not intended to be called outside of a context in which we can guarantee well formed input:

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/* Helper function that computes the product.  Returns the
 * sum 1 + 2 + 3 + ... + n.  Assumes that n > 0.
 */
int _sum(int n) {
  // Base case: the sum 1 is just 1.
  if (n == 1) return 1;
  // Recursive case: compute (1 + 2 + ... + (n-1)) + n.
  return _sum(n-1) + n;
}
```

The function `sum` checks its input and invokes `_sum` if the input is well formed:

```
/* Interface for computing the sum
 *   1 + 2 + 3 + ... + n
 * Returns -1 if n <= 0 or s is NULL; otherwise, stores the
 * sum in the cell that s references and returns 0.
 */
int sum(int n, int * s) {
  if (n <= 0 || s == NULL)
    return -1;

  // We know that n > 0 at this point, so we can safely
  // call the helper function.
  *s = _sum(n);

  // success
  return 0;
}
```

Although the check that `n > 0` is simple, this pattern of separating the main computation from the external interface is common in situations in which the input check is more complex.

Finally, `main` tests `sum` with both well formed and malformed input. It uses the output function `printf`, which is discussed in depth in Chapter 5, to print the sum to the console:

```
int main() {
  // test the sum function
  int s, err;
  err = sum(5, &s);
  assert (err == 0);
  // print the result to the console
  printf("%d\n", s);
  // test bad input
  err = sum(-3, &s);
  assert (err != 0);
  return 0;
}
```

Compiling and running the program yields the expected output of 15:

```
$ gcc -Wall -Wextra -o sum sum.c
$ ./sum
15
```

□

Exercise 2.7. Write a function to compute the product $1 \times \cdots \times n$, for positive n. Write a `main` function to call it, and illustrate various interesting memory configurations during its execution. Use the protection and naming conventions of Exercise 2.6. □

2.3 Loops

While recursion is necessary for solving some important problems and the most natural looping structure in some widely used programming languages such as lisp and ocaml, the iteration exhibited in the `sum` example is better expressed—in C, anyway—through explicit looping control statements.

Let's revisit the problem of summing $1 + \cdots + n$, for positive integer n. This time we will use a `while` statement:

```
1 int sum(int n) {
2   assert (n > 0);
3   int i = 1, s = 0;
4   while (i <= n) {
5     s = s + i;
6     i = i + 1;
7   }
8   return s;
9 }
```

Line 2 declares a **loop counter**, `i`, that is incremented from `1` to `n` and an **accumulator**, `s`, that is initialized to `0`. Lines 4–7 execute iteratively, as long as `i <= n`. The effect is thus that every integer between `1` and `n` is added to `s` precisely once.

The stack is not the best way to visualize looping, or **iterative**, program behavior. Instead, we construct the following table for an input to `sum` of 5:

	n	i	s
	5	1	0
1	5	2	1
2	5	3	3
3	5	4	6
4	5	5	10
5	5	6	15

The first row of numbers indicates the variables' initial values. Subsequent rows indicate their values at the end of each iteration of the loop. Trace through the code and the table to verify your understanding of the computation. Explain to yourself why `sum(5)` returns `15`. What does `sum(8)` return? What about `sum(0)`?

Once again, we may not be satisfied with the possibility that calling `sum` with a nonpositive value could halt our program: such violent behavior compromises the modularity of the function. Instead, we write the following more modular and more robust function:

```
int sum(int n, int * s) {
  int i = 1;

  // check for well formed input
```

```
  if (n <= 0 || s == NULL)
    // indicate malformed input
    return -1;

  *s = 0;
  while (i <= n) {
    *s += i;  // short for *s = *s + i;
    i++;      // short for i = i + 1;
  }
  // indicate successful execution
  return 0;
}
```

This implementation introduces new operators for accumulating sums. Loop counters are so prevalent in C that the language designers included the operator `++` to increment a variable by 1. Accumulation is also a frequent operation, and the `+=` operator provides a convenient shorthand. Similar operators exist for other arithmetic operations, including `--`, `-=`, `*=`, and `/=`.

Exercise 2.8. Write a version of `product` (see Exercise 2.7) that uses a `while` loop instead of recursion. Draw a table that illustrates values of its variables during execution for a reasonable input. □

The loop of `sum` follows a common pattern that motivates the `for` loop:

```
1  int sum(int n, int * s) {
2    int i;
3
4    // check for malformed input
5    if (n <= 0 || s == NULL) return -1;
6
7    *s = 0;
8    for (i = 1; i <= n; i++)
9      *s += i;
10
11   return 0;
12 }
```

Lines 8–9 compile to exactly the same machine instructions as this loop:

```
  i = 1;
  while (i <= n) {
    *s += i;
    i++;
  }
```

In general, a `for` loop of the form

```
  for (<initialize>; <condition>; <increment>) {
    <body>
  }
```


is exactly the same as a `while` loop of the form

```
  <initialize>
  while (<condition>) {
    <body>
    <increment>
  }
```

Programmer preference dictates when to use a `while` statement and when to use a `for` statement. Readability is the goal.

Exercise 2.9. Rewrite the `product` function of Exercise 2.8 using a `for` loop. □

Exercise 2.10. Write a function to compute the power a^n, where $n \geq 0$. It should have the following prototype:

```
/* Sets *p to the n'th power of a and returns 0, except
 * when n < 0 or p is NULL, in which case it returns -1.
 */
int power(int a, int n, int * p);
```

Write a unit test in a `main` function to test various values. The following code sequence illustrates how to use `printf` to provide informative output:

```
  int x = 3, y = 5, pow;
  power(x, y, &pow);
  printf("%d^%d = %d\n", x, y, pow);
```

□

Exercise 2.11. Mathematical sequences can be computed using loops. Consider, for example, the following sequence:

$$a_0 = 1 \quad \text{and} \quad a_{i+1} = 2 \cdot a_i + 1 \text{ for } i > 0 \ ,$$

whose first elements are $1, 3, 7, 15, 31, 63, \ldots$. This function returns the nth element:

```
int seq(int n) {
  int i, a = 1;
  for (i = 1; i <= n; i++)
    a = 2*a + 1;
  return a;
}
```

For example, `seq(0)` returns 1, `seq(1)` returns 3, and `seq(4)` returns 31.

Write functions to compute the nth elements of the following sequences:

(a) $a_0 = 1$ and $a_{i+1} = 3 \cdot a_i + 2$ for $i > 0$.
(b) $a_0 = 59$ and $a_{i+1} = a_i/2 + 1$ for $i > 0$, where / denotes integer division; in C, use `/`. For example, $3/2 = 1$. The first elements of the sequence are $59, 59/2 + 1 = 29 + 1 = 30, 16, 9, 5, 3, \ldots$.

(c) $a_0 = 1$, $a_1 = 1$, and $a_{i+1} = a_{i-1} + a_i$ for $i > 1$. The first elements of the sequence, called the Fibonacci sequence, are $1, 1, 2, 3, 5, 8, \ldots$.
Solution. This function needs to remember the previous two values:

```
int seq(int n) {
  int i, a = 1, b = 1;
  for (i = 2; i <= n; i++) {
    int t = b;  // temporary variable
    b = a + b;
    a = t;
  }
  return b;
}
```

Verify that this function indeed returns the nth element of the sequence for various n.

(d) $a_0 = 0$, $a_1 = 2$, and $a_{i+1} = 2 \cdot a_{i-1} - a_i$ for $i > 1$.
(e) $a_0 = 7$, $a_1 = 11$, and $a_{i+1} = -a_{i-1} + a_i$ for $i > 1$.
(f) $a_0 = 1$, $a_1 = 1$, $a_2 = 1$, and $a_{i+1} = a_{i-2} + a_i$ for $i > 2$.

□

Exercise 2.12. Mathematical series can be computed using loops. Consider, for example, the following sequence:

$$a_0 = 1 \quad \text{and} \quad a_{i+1} = 2 \cdot a_i + 1 \text{ for } i > 0 .$$

The corresponding series is constructed by computing the partial sums:

$$a_0, \ \sum_{j=0}^{1} a_j, \ \sum_{j=0}^{2} a_j, \ \sum_{j=0}^{3} a_j, \ \ldots .$$

Since the first elements of the sequence are $1, 3, 7, 15, 31, 63, \ldots$, the first elements of the corresponding series are $1, 1{+}3 = 4, 1{+}3{+}7 = 11, 26, 57, 120, \ldots$. This function returns the nth element of the series:

```
int series(int n) {
  int i, a = 1, sum = 1;
  for (i = 1; i <= n; i++) {
    a = 2*a + 1;
    sum += a;
  }
  return sum;
}
```

For example, `series(0)` returns 1, `series(1)` returns 4, and `series(4)` returns 57. Write similar functions to compute the nth elements of series corresponding to the sequences of Exercise 2.11. □

More complex control patterns will come after we have studied more complex data structures. However, all control builds on conditionals, loops, and occasionally recursion.

3

Arrays and Strings

Memory and control come together in **data structures**. A data structure is a program-defined structure in memory with corresponding operations to give it meaning. For example, an `int` variable is a simple data structure when combined with the operations of reading, writing, and basic arithmetic. It has an explicit place in memory—a 32-bit memory cell—and the arithmetic operations give meaning to the data—the 32 bits, or four bytes—that reside there. An `int *` variable, while occupying the same amount of memory as an `int`, is given a different meaning through the operators `*` and `&`, in addition to the arithmetic operators.

These basic data structures can only take us so far. Their fixed size is limiting, for example. (Technically, through recursion, one can program anything that can be programmed using only integer and pointer variables, though the value of such a discipline is questionable.) **Compound data structures** consist of a possibly variable number of basic data structures. They are given meaning through code. In this chapter, we begin our study of compound data structures with the simplest and most fundamental of all: the **array**, which consists of a contiguous range of more basic data structures, all of the same type. An array of `int` data is a typical example. An array is indexable, allowing reading or writing of each of its elements. Besides reading and writing element-wise, iteration over an array can be seen as a fundamental operation; hence, arrays and loops go hand in hand.

One application of arrays is to hold text. Textual data are called **strings** in programming parlance, and we study them in the second half of this chapter.

3.1 Arrays

3.1.1 Introduction to Arrays

A C **array** defines a contiguous region of memory divided into memory cells accessible via indexing:


```
1 int main() {
2   int a[4];
3   a[0] = 1;
4   a[1] = 1;
5   a[2] = a[0] + a[1];
6   a[3] = a[1] + a[2];
7   return 0;
8 }
```

The array `a` declared at line 2 consists of four integer memory cells arranged consecutively in memory:

`int`	`a[3]`	$\otimes$	1012
`int`	`a[2]`	$\otimes$	1008
`int`	`a[1]`	$\otimes$	1004
`int`	`a[0]`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

An array is **indexed** from 0 to 1 less than its size. By the end of line 6, memory is configured as follows:

`int`	`a[3]`	3	1012
`int`	`a[2]`	2	1008
`int`	`a[1]`	1	1004
`int`	`a[0]`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Let's be clear on one point from the beginning: `a` itself is implicitly a pointer. The expression `a` evaluates to the address of the beginning of the array, which in this case is 1000. The following program is almost identical to the one above:

```
int main() {
  int a[4];
  int * x;
  x = a;  // Notice that the right expression is a, not &a!
  x[0] = 1;
  x[1] = 1;
  x[2] = x[0] + x[1];
  x[3] = x[1] + x[2];
  return 0;
}
```

At function entry, memory is configured as follows:

`int *`	`x`	$\otimes$	1016
`int`	`a[3]`	$\otimes$	1012
`int`	`a[2]`	$\otimes$	1008
`int`	`a[1]`	$\otimes$	1004
`int`	`a[0]`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

The expression `a` evaluates to the address of the beginning of the array; hence at line 4, `x` is assigned 1000:

`int *`	`x`	1000	1016
`int`	`a[3]`	$\otimes$	1012
`int`	`a[2]`	$\otimes$	1008
`int`	`a[1]`	$\otimes$	1004
`int`	`a[0]`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

Now, indeed, `x` points to an integer, namely the memory cell at address 1000, which holds `int` data.

At this point, there are two puzzles. First, why does the pointer `x` correspond to a memory cell while the pointer `a` does not? While `a` is implicitly a pointer, it does not have the same functionality as `x`: it can only be read and dereferenced, whereas `x` can also be written, as at line 4. In other words, `a` always refers to the same address relative to `main`'s stack frame, which, in this case, is eight bytes beyond the beginning of the frame (992). No matter what the address of the stack frame is, `a`'s value is a constant offset from that address. The compiler replaces `a` with this stack frame-relative offset. Stack-allocated arrays always behave in this manner.

Second, why do lines 5–8 work? Indexing into an array is just **syntactic sugar** for dereferencing memory: a convenient but unnecessary language feature. The last version of `main` compiles into exactly the same program as the following version:

```
1  int main() {
2    int a[4];
3    int * x;
4    x = a;
5    *x = 1;
6    *(x + 1) = 1;
7    *(x + 2) = *x + *(x + 1);
8    *(x + 3) = *(x + 1) + *(x + 2);
9    return 0;
10 }
```

Cool, right? But it's ugly and unnecessary, so don't write code like this example in practice. Lines 5–8 make heavy use of **pointer arithmetic**. If memory

addresses are just data that look very much like integers, why not add them and subtract them as you would any other integer data? And once a new address has been formed through pointer arithmetic, why not dereference it so as to read from or write to the addressed memory cell?

The only puzzle is why `*(x + 1)` is the same as `x[1]`. Shouldn't we write `*(x + 4)` since the second word of the array is four bytes later in memory? The answer is "no." Pointer arithmetic differs from standard integer arithmetic in one crucial manner: the C compiler takes into account the types of the pointers when it compiles pointer arithmetic. In this case, `x` is an `int *`. Since an `int` occupies one word (four bytes) and `x` is an `int *`, `x + 1` evaluates to the address one word, or four bytes, later in memory than `x` evaluates to. Therefore, `x[1]` and `*(x + 1)` are synonymous: both evaluate to the value in the memory cell one word beyond the address in `x`.

We can use pointer arithmetic on `a` itself, since `a` is implicitly a pointer:

```
int main() {
  int a[4];
  *a = 1;
  *(a + 1) = 1;
  *(a + 2) = *a + *(a + 1);
  *(a + 3) = *(a + 1) + *(a + 2);
  return 0;
}
```

Study the four versions of `main` until you understand precisely how and why they work, and why they effectively describe the same computation.

3.1.2 Looping over Arrays

With the power to declare arbitrary segments of memory for use, the next logical step is to construct loops that modify arbitrarily large arrays.

The Fibonacci sequence is defined as follows. The first two elements of the sequence are 1; then subsequent elements are defined as the sum of their two predecessors:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots\ .$$

In code, we have the following:

```
// defines N to be synonymous with 100
#define N 100

int main() {
  // declare an array of N integers
  int fib[N];
  int i;

  // define the first two elements
```

```
10   fib[0] = 1;
11   fib[1] = 1;
12
13   // define the remaining elements up to the N'th
14   for (i = 2; i < N; i++)
15     fib[i] = fib[i-2] + fib[i-1];
16
17   return 0;
18 }
```

We have used a new feature in this code. `#define N 100` defines `N` to be a synonym for `100`. The use of `#define` allows us to write code that is parametrized by a small set of constants. If we ever want to change a parameter, we need only change its definition. In this case, changing `#define N 100` to `#define N 200` is simpler than changing every occurrence of `100` throughout the code. More importantly, changing one line of code is less likely to introduce bugs than changing many lines of code.

Notice that `fib` is indexed from `0` to `N-1`. In particular, the loop counter `i` ranges between `2` and `N-1`, because the loop only executes while `i < N`. Novice (and even experienced) programmers often introduce **off-by-one** bugs in which the loop condition is incorrectly written as `i <= N`. Such errors can be insidious because one word beyond an array is typically still within the program's allotted memory. Thus, rather than causing a clean segmentation fault, the bug causes memory corruption—which can induce in the young programmer frustration, then anger... fear... aggression. The dark side are they. Or so I've heard, anyway.

We can visualize the first several iterations of the loop as follows:

	i	fib[0]	fib[1]	fib[2]	fib[3]	fib[4]	fib[5]
	2	1	1	⊗	⊗	⊗	⊗
1	3	1	1	2	⊗	⊗	⊗
2	4	1	1	2	3	⊗	⊗
3	5	1	1	2	3	5	⊗
4	6	1	1	2	3	5	8

The first row indicates the variables' values just before the loop executes, but after `i` is initialized to 2. Subsequent rows indicate their values at the end of each iteration. Recall that `i++` is executed after the statement at line 14, but before the condition `i < N` is checked. Hence, `i` has value 3 at the end of the first iteration.

Exercise 3.1. Rewrite the main loop of the Fibonacci computation as a `while` loop.

Solution. The `while` loop form makes the execution table above more clear:

```
  i = 2;
  while (i < N) {
    fib[i] = fib[i-2] + fib[i-1];
    i++;
  }
```

□

3.1.3 Arrays as Parameters

Doing too much in `main` is bad practice. A program tends to grow over time as more is required of it, so it's best to factor code into manageable bundles—that is, functions and, in a few chapters, modules—from the beginning. This modularity pays dividends: it allows modular thinking, where one does not have to recall how exactly a certain function is implemented but only what it accomplishes; it facilitates code reuse, in which a function is called in multiple contexts; and it looks nicer.

Therefore, let's extract the Fibonacci code from `main` and put it in its own function:

```
1  /* Given an array, fib, with length n, computes the first n
2   * elements of the Fibonacci sequence.  Returns 0 to
3   * indicate success and negative values for bad input.
4   */
5  int fibonacci(int * fib, int n) {
6    // check for well-formed input
7    if (fib == NULL)
8      return -1;
9    if (n <= 0)
10     return -2;
11
12   fib[0] = 1;
13   if (n >= 2)
14     fib[1] = 1;
15   int i;
16   for (i = 2; i < n; i++)
17     fib[i] = fib[i-2] + fib[i-1];
18
19   // indicate successful computation
20   return 0;
21 }
```

This well-protected function returns different error codes depending on how the input is malformed: it returns -1 if `fib == NULL` and -2 if `n <= 0`. Notice how lines 7–10 are not structured as an `if`/`else` statement. Because execution of the body of either condition causes the function to return immediately, no `else` is necessary.

Unfortunately, the implementation must make one assumption that cannot be checked: it assumes that `fib` points to a programmer-declared region of memory that extends at least `n` `int` memory cells. Otherwise, it makes no further assumptions. Lines 12–17 are careful to write only to the first `n` memory cells beyond the address held in `fib`. Since array variables and pointer variables are essentially the same thing, array indexing works on the integer pointer `fib`.

Exercise 3.2. Array indexing and `for` loops are "syntactic sugar": convenient but unnecessary. Rewrite `fibonacci` to use pointer arithmetic instead of array indexing and a `while` loop instead of a `for` loop. (For further personal growth through deprivation, replace your keyboard with a punch card interface.) Once you get it right, never, ever write such unnecessarily hideous code again. □

Let's add a proper calling context:

```
1 #define N 3
2
3 int main() {
4   int a[N];
5   int error;
6   error = fibonacci(a, N);
7   assert (!error);
8   return 0;
9 }
```

It's worth visualizing critical memory configurations during the execution of this program. At the function call at line 6, memory is configured as follows:

`int`	`i`	$\otimes$	1032
`void *`	`pc`	`main:6+`	1028
`int`	`rv`	$\otimes$	1024
`int`	`n`	3	1020
`int *`	`fib`	1000	1016
`int`	`error`	$\otimes$	1012
`int`	`a[2]`	$\otimes$	1008
`int`	`a[1]`	$\otimes$	1004
`int`	`a[0]`	$\otimes$	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

In particular, the parameter `fib` holds the address of the beginning of array `a` of `main`, while parameter `n` holds the value 3. As authors of `fibonacci`, we have no choice but to believe the caller that `fib` indeed points to a region of memory with at least three consecutive reserved memory cells. In this case, the assumption is correct.

Upon completion of `fibonacci`, memory is configured as follows:

`int`	`i`	3	1032
`void *`	`pc`	`main:6+`	1028
`int`	`rv`	0	1024
`int`	`n`	3	1020
`int *`	`fib`	1000	1016
`int`	`error`	$\otimes$	1012
`int`	`a[2]`	2	1008
`int`	`a[1]`	1	1004
`int`	`a[0]`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

After `fibonacci` returns, memory has the following configuration:

`int`	`error`	0	1012
`int`	`a[2]`	2	1008
`int`	`a[1]`	1	1004
`int`	`a[0]`	1	1000
`void *`	`pc`	"system"	996
`int`	`rv`	$\otimes$	992

If we desired to emphasize that `fib` can and should be treated as an array, we could write `fibonacci`'s header as follows:

```
int fibonacci(int fib[], int n);
```

Writing `int * fib` or `int fib[]` is a personal preference that does not at all impact the resulting machine code. One of C's peculiarities is how it is frugal in some ways—for example, the meaning of `*` depends on its context: to perform multiplication, to specify a pointer, to dereference a pointer—but lavish in others.

3.1.4 Further Adventures with Arrays

Exercise 3.3. Write a function to copy the elements of one integer array to another, where both have the same length. The function should implement the following specification:

```
/* Copies a to cp and returns 0, unless either is NULL,
 * in which case it returns -1.
 */
int copyArray(int * a, int * cp, int len);
```

Implement a unit test of `copyArray` in a `main` function.

Solution. The strategy is to iterate through the index range and assign each element:

```
#include <assert.h>
#include <stdlib.h>
```

```
3
4 int copyArray(int * a, int * cp, int len) {
5   if (!a || !cp) return -1;
6   int i;
7   for (i = 0; i < len; i++)
8     cp[i] = a[i];
9   return 0;
10 }
11
12 // a unit test of copyArray
13 #define N 5
14 int main() {
15   int a[N], b[N];
16   int i;
17   // initialize the source array
18   for (i = 0; i < N; i++) a[i] = i;
19   // should copy a's elements to b
20   copyArray(a, b, N);
21   // check that the copy indeed occurred
22   for (i = 0; i < N; i++)
23     assert (a[i] == b[i]);
24   // check corner cases
25   assert (copyArray(NULL, a, 0));
26   assert (copyArray(a, NULL, 0));
27   return 0;
28 }
```

Line 5 checks if either of `a` or `cp` is `NULL`; the condition is equivalent to `a == NULL || cp == NULL`. Since `NULL` is address 0, the condition could be written as `a == 0 || cp == 0`. But then we observe that, according to the definition of C's Boolean operator `!`, `a == 0` is equivalent to `!a`. The final form of the condition is a common C idiom. We similarly use C's Boolean facilities in lines 25–26, which assert that the return values are nonzero.

Throughout this chapter, we sometimes take advantage of Boolean operators and sometimes write the more explicit forms of conditions so that you may become accustomed to various patterns; but in later chapters, we prefer the more concise forms. □

Exercise 3.4. Write a function to sum the elements of one integer array. The function should implement the following specification:

```
/* Sums the elements of a, an array of length len, and
 * writes the sum to where sum references.  Returns 0,
 * unless a or sum is NULL, in which case returns -1.
 */
int sumArray(int * a, int len, int * sum);
```

Implement a unit test of `sumArray` in a `main` function. □

Exercise 3.5. Write a function to compute the dot product of two n-dimensional vectors. The dot product of two vectors

$$x = (x_1, x_2, \ldots, x_n) \quad \text{and} \quad y = (y_1, y_2, \ldots, y_n)$$

is

$$x \cdot y = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \ .$$

The function should implement the following specification:

```
/* Computes the dot product of two n-dimensional vectors, x
 * and y, and stores it at address dp.  Returns 0 if
 * successful; -1 if any of x, y, or dp is NULL; and -2 if
 * n <= 0.
 */
int dotProduct(int x[], int y[], int n, int * dp);
```

Solution. Here is one possible implementation:

```
int dotProduct(int x[], int y[], int n, int * dp) {
  // check if input is well-formed
  if (x == NULL || y == NULL || dp == NULL)
    return -1;
  if (n <= 0)
    return -2;

  // compute the dot product
  *dp = 0;
  int i;
  for (i = 0; i < n; i++)
    *dp += x[i] * y[i];

  // indicate success
  return 0;
}
```

□

Exercise 3.6. Write a function to compute the sum of two n-dimensional vectors. The sum of two vectors

$$x = (x_1, x_2, \ldots, x_n) \quad \text{and} \quad y = (y_1, y_2, \ldots, y_n)$$

is the vector

$$(x_1 + y_1,\ x_2 + y_2,\ \ldots,\ x_n + y_n) \ .$$

The function should implement the following specification:

```
/* Computes the sum of two n-dimensional vectors, x and y,
 * and stores it in vector sum.  Returns 0 if successful;
 * -1 if any of x, y, or sum is NULL; and -2 if n <= 0.
 */
int vectorSum(int x[], int y[], int n, int sum[]);
```

Implement a unit test of `sum` in a `main` function. □

Exercise 3.7. Write a function to find the minimum value in an integer array. The function should implement the following specification:

```
/* Computes the minimum element of the array a of length n
 * and stores it in the memory cell referenced by min.
 * Returns 0 if successful; -1 if a or n is NULL; and -2 if
 * n <= 0.
 */
int min(int * a, int n, int * min);
```

Implement a unit test of `min` in a `main` function.

Solution. The strategy is to remember, using a variable, the smallest value seen so far as the function examines each element in turn:

```
1  #include <assert.h>
2  #include <stdlib.h>
3
4  int min(int * a, int n, int * min) {
5    if (a == NULL || min == NULL)
6      return -1;
7    if (n <= 0)
8      return -2;
9    // the minimum value so far is at position 0
10   int m = a[0], i;
11   for (i = 1; i < n; i++)
12     if (a[i] < m)
13       // a[i] is even smaller than previously seen elements
14       m = a[i];
15   *min = m;
16   return 0;
17 }
18
19 int main() {
20   // initializes a to a constant array of 5 elements
21   int a[] = {7, -1, 13, -3, 9};
22   int x;
23   min(a, 5, &x);
24   assert (x == -3);
25   // corner cases
26   assert (min(a, 0, &x) != 0);
27   assert (min(NULL, 0, &x) != 0);
28   assert (min(a, 0, NULL) != 0);
29   return 0;
30 }
```

Line 10 sets `m` to be `a[0]`, essentially saying that `a[0]` is the smallest value seen so far. Then the loop at lines 11–14 inspects each element in turn. If a given element `a[i]` is less than the previously known minimum value, `m`, then

`m` is updated accordingly. Hence, at line 12, `m` is invariably the minimum value for the subarray indexed between 0 and `i-1`, and by line 15, it is the minimum value for the entire array. □

Exercise 3.8. Write a function to compute the minimum and maximum values of an integer array. It should implement the following specification:

```
/* Computes the minimum and maximum elements of the array
 * a of length n, storing them in the memory cells to which
 * min and max, respectively, point.  Returns 0 if
 * successful; -1 if one or more of a, min, or max is NULL;
 * and -2 if n <= 0.
 */
int minmax(int * a, int n, int * min, int * max);
```

Implement a unit test of `minmax` in a `main` function. □

Exercise 3.9. Write a function that computes the range, or the difference between the minimum and maximum values, of an array of integers. It should implement the following specification:

```
/* Computes the range of an array and stores it where rng
 * references.  Returns -1 for erroneous input; and 0
 * otherwise.
 */
int range(int * a, int n, int * rng);
```

Implement a unit test of `range` in a `main` function.

Solution. This function provides an opportunity to reuse previous work, in particular the `minmax` function of Exercise 3.8:

```
#include <assert.h>
#include <stdlib.h>

// Insert minmax here.

int range(int * a, int n, int * rng) {
  if (!a || !rng || n <= 0) return -1;

  int min, max;
  minmax(a, n, &min, &max);
  *rng = max - min;

  return 0;
}

int main() {
  // initializes a to a constant array of 5 elements
  int a[] = {7, -1, 13, -3, 9};
  int r;
```

```
  // test main functionality
  range(a, 5, &r);
  assert (r == 16);
  // corner cases
  assert (!range(NULL, 5, &r));
  assert (!range(a, 5, NULL));
  assert (!range(a, 0, &r));
  return 0;
}
```

□

Exercise 3.10. Write a function that counts the number of occurrences of a given number in a given array. It should implement the following specification:

```
/* Computes the number of occurrences of value v in array a
 * of length n and stores it in occ.  Returns 0 if
 * successful; -1 if either of a or occ is NULL; and -2 if
 * n < 0.
 */
int numOccur(int a[], int n, int v, int * occ);
```

Implement a unit test of `numOccur` in a `main` function. □

Exercise 3.11. Write a function that computes the integer mean of an array of integers. For example, the integer mean of $-1, 4, 2$ is $(-1 + 4 + 2)/3 = 5/3 = 1$, where / denotes integer division. The function should implement the following specification:

```
/* Computes the integer mean of an array and stores it
 * where mn references.  Returns -1 for erroneous input
 * (len <= 0 or NULL array); otherwise returns 0.
 */
int mean(int * a, int len, int * mn);
```

Implement a unit test of `mean` in a `main` function. □

Exercise 3.12. Write a function that concatenates the elements of two arrays into a third one. It should implement the following specification:

```
/* Concatenates arrays a and b, of lengths an and bn,
 * respectively, storing the result in c.  Returns -1 for
 * erroneous input, and 0 otherwise.
 */
int concat(int * a, int an, int * b, int bn, int * c);
```

Solution. We explore several strategies for implementing this function. The first two variants require two loops. In the first, a variable `j` maintains the write position in `c`, while `i` loops through first `a` and then `b`:


```
int concat(int * a, int an, int * b, int bn, int * c) {
  if (!a || !b || !c)
    return -1;
  int i, j = 0;
  for (i = 0; i < an; i++) {
    c[j] = a[i];
    j++;
  }
  for (i = 0; i < bn; i++) {
    c[j] = b[i];
    j++;
  }
  return 0;
}
```

In the second variant, the variable `j` is dropped and instead the length `an` of `a` is used for positioning in the second loop:

```
int concat(int * a, int an, int * b, int bn, int * c) {
  if (!a || !b || !c)
    return -1;
  int i;
  for (i = 0; i < an; i++)
    c[i] = a[i];
  for (i = 0; i < bn; i++)
    c[an + i] = b[i];
  return 0;
}
```

Using linear functions to index into arrays is a common technique.

Notice in the second variation that the placement of `b`'s elements is independent of the placement of `a`'s elements. The third variation therefore fuses the two loops into one. The idea is to iterate sufficiently for the longer of `a` and `b`:

```
int concat(int * a, int an, int * b, int bn, int * c) {
  if (!a || !b || !c)
    return -1;
  int i;
  for (i = 0; i < an || i < bn; i++) {
    if (i < an) c[i] = a[i];
    if (i < bn) c[an + i] = b[i];
  }
  return 0;
}
```

□

Exercise 3.13. Write a function to zip together two arrays of equal length into a third of double the length:

```
/* Zips together two arrays into a third, alternating their
 * values.  E.g.,
 *  a: [1, 2, 3]
 *  b: [4, 5, 6]
 * each of length 3, zip together to form
 *  c: [1, 4, 2, 5, 3, 6]
 * Returns -1 if the input is malformed and 0 otherwise.
 */
int zip(int * a, int * b, int * c, int n);
```

Solution. The trick is to devise the right linear function to index into `c`:

```
int zip(int * a, int * b, int * c, int n) {
  if (!a || !b || !c) return -1;
  int i;
  for (i = 0; i < n; i++) {
    c[2*i] = a[i];
    c[2*i+1] = b[i];
  }
  return 0;
}
```

Operator precedence is the same as elementary-school **PEMDAS**—parentheses, exponents, multiplication, addition, subtraction—except that C lacks an exponentiation operator, since it can be programmed about as efficiently as one could devise a hardware implementation. Hence, `2*i+1` is computed as "multiply i by 2 and then add 1." □

Exercise 3.14. Write a function to unzip an array of length $2n$ into two arrays of length n each:

```
/* Unzips an array into two (opposite of zip).  E.g.,
 *  c: [1, 2, 3, 4, 5, 6]
 * unzips into
 *  a: [1, 3, 5]
 *  b: [2, 4, 6]
 * In this case, n is 3.  Returns -1 if the input is
 * malformed and 0 otherwise.
 */
int unzip(int * a, int * b, int * c, int n);
```

□

3.2 Strings

Text is probably the single most widely used form of data in computer applications. Even scientists or engineers, for whom numbers are fundamental, write

their programs using text editors, typically interact with their programs via textual interfaces, describe their results to their colleagues mainly with text, and summarize their results in bullet points for their managers via (preferably monosyllabic) text.

However, underlying text is its numeric representation as `char`, or **character**, data. Like all data in a computer, text is in the end nothing but numbers—which means that everything that you have learned so far directly applies to textual data.

3.2.1 Strings: Arrays of `char`s

A value of type `char`, short for **character**, requires one byte (eight bits) of storage and thus can be only one of 256 possible values. A character is not a particularly funny or charming kind of value; rather it is supposed to represent a written character, for example, one of `'a'` through `'z'`, `'A'` though `'Z'`, or `'0'` through `'9'`.

ASCII—the *American Standard Code for Information Interchange*—defines the first 128 possible values of a `char` to represent certain characters. For example, ASCII codes 65–90 represent `'A'` to `'Z'`, 97–122 represent `'a'` to `'z'`, and 48–57 represent `'0'` to `'9'`. ASCII code 10 represents the **new line** character. Fortunately, we don't have to remember these codes: the C expression `'p'` evaluates to the corresponding ASCII code for the letter p, while `'\n'` evaluates to the *new line* code.

Assemble a few `char`s in a `char` array, and you have a **string**—almost. A C string is a sequence of `char` values that ends with the **string terminator**, `'\0'`. C provides the convenience of defining constant strings:

```
1 #include <stdio.h>
2
3 int main() {
4   char str[] = "Hello!";
5   printf("%s\n", str);
6   return 0;
7 }
```

Line 1 includes the standard input/output library, `stdio.h`, which we discuss in some detail in Chapter 5. We include it so that at line 5 we can print to the terminal the string `str`. As a preview, the `printf` function is a powerful function for printing formatted text. In this case, the first argument, `"%s\n"` is a format string that specifies that `printf` should print a string, given by the argument `str`, followed by a new line. Notice that both the format string and the second argument, `str`, are text data.

The string `"Hello!"` is compiled into a segment of memory disjoint from the stack that looks like the following:

char	0	506
char	33	505
char	111	504
char	108	503
char	108	502
char	101	501
char	72	500

Notice that each `char` value occupies only one byte, and that the string is terminated with `'\0'`, which corresponds to ASCII code 0. Check that the sequence 72, 101, 108, 108, 111, 33, 0 actually corresponds to the string "Hello!". ASCII tables are easy to find online.

When the program is executed, the stack frame for `main` yields the following initial memory configuration:

`char * str`	500	1000
`void * pc`	"system"	996
`int rv`	⊗	992

Because `str` holds an address, it occupies a word (32 bits). In this case, it holds the address 500, corresponding to the beginning of the constant string `"Hello!"` in memory.

3.2.2 Programming with Strings

By applying the programming tools we have covered so far to strings, we can manipulate strings in some truly interesting ways. (Try the game `nethack`, which is easily installed on most Unix variants, to witness what can be achieved with strings, ambition, and time.)

As a first venture into programming with strings, we implement a function to shout—that is, to capitalize all lowercase letters of a message:

```
1  /* Writes the message of msgIn into msgOut, except with all
2   * capitals.  Returns 0 if successful and -1 if either of
3   * msgIn or msgOut is NULL.
4   */
5  int shout(char * msgIn, char * msgOut) {
6    int i = 0;
7    char c;
8
9    // check for well-formed input
10   if (msgIn == NULL || msgOut == NULL)
11     return -1;
12
13   // loop over msgIn until the string terminator is found
14   while (msgIn[i] != '\0') {
15     // obtain the i'th character of the message
16     c = msgIn[i];
```



```
17     // if it's a lowercase letter, capitalize it
18     if ('a' <= c && c <= 'z')
19       c += 'A' - 'a';
20     // write the character to msgOut
21     msgOut[i] = c;
22     // don't forget to increment i
23     i++;
24   }
25   // terminate msgOut
26   msgOut[i] = '\0';
27
28   // indicate success
29   return 0;
30 }
```

Both `msgIn` and `msgOut` are `char` arrays, which is the same thing as saying that they are `char *`'s. Each holds an address: `msgIn` holds the address of what we can only hope is a well-formed C string, one that is a sequence of characters ending with the string terminator `'\0'`; `msgOut` holds the address of what we can only hope is the beginning of a sufficiently large sector of memory to hold all of `msgIn`. If either assumption (dearly held hope) is false, prepare for memory corruption, anger, fear, the dark side, etc. (Actually, the non-dark (light?) side has powerful weapons, `gdb` and `valgrind`, which we cover in Chapters 4 and 7.)

The loop counter, `i`, iterates over the range of `msgIn` until `msgIn[i] == '\0'`, which indicates the end of the string. In the loop body, the `i`'th character is retrieved from `msgIn`. Lines 18–19 might look like a bit of magic, but they're straightforward once you accept that computers manipulate numbers—nothing more, nothing less. Recall that the ASCII code for `'a'` is 97, that the code for `'z'` is 122, and that the compiler converts `'a'` and `'z'` to these values. Therefore, `'a' <= c && c <= 'z'` is true precisely when `c` holds the code for a lowercase letter. In this case, something should be added to `c` to make it an uppercase letter. But that's easy (if a bit subtle): simply add `'A' - 'a'` to it, the offset between the uppercase and lowercase letters in the ASCII system.[1]

After the loop, the string being constructed in `msgOut` is completed with the string terminator.

Consider the following calling context:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
```

[1] Notice that the ASCII codes for `'A'` and `'a'` are 65 and 97, respectively, and thus have a difference of 32. **Challenge**: Exploit the binary representation of the character codes to develop another, even cleverer, "shout" conversion.

```
6   char msg[] = "Hi!";
7   char out[4];
8   int err = shout(msg, out);
9   assert (!err);
10  printf("%s -> %s\n", msg, out);
11  return 0;
12 }
```

When `shout` is called, memory is configured as follows:

`char`	`c`	⊗	1032
`int`	`i`	0	1028
`void *`	`pc`	`main:7+`	1024
`int`	`rv`	⊗	1020
`char *`	`msgOut`	1004	1016
`char *`	`msgIn`	500	1012
`int`	`err`	⊗	1008
`char`	`out[3]`	⊗	1007
`char`	`out[2]`	⊗	1006
`char`	`out[1]`	⊗	1005
`char`	`out[0]`	⊗	1004
`char *`	`msg`	500	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992
`char`		0	503
`char`		33	502
`char`		105	501
`char`		72	500

The four bytes holding the C string `"Hi!"` that start at address 500 are outside of the program stack, which is indicated by the separation between the final byte and the bottom of the program stack starting at address 1000. Study the addresses carefully. Which memory cells are one byte? Which are four bytes? Why? Study the values that the memory cells hold. Explain why each value makes sense given that control is at the beginning of line 10 of `shout`.

Just before `shout`'s stack frame is popped, memory is configured as follows:

`char`	`c`	33	1032
`int`	`i`	3	1028
`void *`	`pc`	`main:7+`	1024
`int`	`rv`	0	1020
`char *`	`msgOut`	1004	1016
`char *`	`msgIn`	500	1012
`int`	`err`	⊗	1008
`char`	`out[3]`	0	1007
`char`	`out[2]`	33	1006
`char`	`out[1]`	73	1005
`char`	`out[0]`	72	1004
`char *`	`msg`	500	1000
`void *`	`pc`	"system"	996
`int`	`rv`	⊗	992
`char`		0	503
`char`		33	502
`char`		105	501
`char`		72	500

Then `printf` prints the following message to the terminal:

```
Hi! -> HI!
```

Exercise 3.15. Write a function `whisper` that changes every uppercase letter to a lowercase letter. It should implement the following specification:

```
/* Writes the message of msgIn into msgOut, except with all
 * lowercase letters.  Returns 0 if successful and -1 if
 * either of msgIn or msgOut is NULL.
 */
int whisper(char * msgIn, char * msgOut);
```

Illustrate several critical memory configurations of an execution of `whisper` from a context similar to the `main` function above. □

There are other ways of implementing `shout` that take advantage of pointer arithmetic. Recall that for parameter `x` declared either by `int * x` or `int x[]`, the two memory dereferences, `x[1]` and `*(x + 1)`, are identical. The same holds for `char * s`: `s[1]` is the same as `*(s + 1)`. But there is one subtle difference. Memory cells of type `int` occupy four bytes while those of type `char` occupy only one byte. The C compiler translates `x + 1` to an offset of four bytes from address `x`, while it translates `s + 1` to an offset of one byte from address `s`. We don't need to do anything except understand what the compiler does.

Here is an implementation of `shout` that really shouts to the world, "I'm implemented in C!" Other than C++, C's more sophisticated younger sibling, no other widely used language allows this level of direct memory manipulation:

```
1 /* Alternate implementation of shout. */
2 int shout(char * msgIn, char * msgOut) {
3   // check for well-formed input
4   if (!msgIn || !msgOut) return -1;
5 
6   // loop over msgIn until the string terminator is found
7   while (*msgIn != '\0') {
8     // transfer the (possibly modified) character to msgOut
9     if ('a' <= *msgIn && *msgIn <= 'z')
10      *msgOut = *msgIn + ('A' - 'a');
11    else
12      *msgOut = *msgIn;
13    // increment the pointers
14    msgIn++;
15    msgOut++;
16  }
17  // terminate msgOut
18  *msgOut = '\0';
19 
20  // indicate success
21  return 0;
22 }
```

Lines 14–15 apply the `++` operator to the character pointers `msgIn` and `msgOut`, thus incrementing the addresses they hold by one byte. Again, the C compiler figures out the necessary byte offset based on the fact that they are declared as `char *`'s. The rest of the loop is written assuming that `msgIn` points to the byte-size memory cell holding the character that should be read, and that `msgOut` points to the byte-size memory cell to which the new character should be written.

Exercise 3.16. Write another version of `whisper`, from Exercise 3.15, that does not use a loop counter but instead uses pointer arithmetic. □

3.2.3 Further Adventures with Strings

Exercise 3.17. Implement the following specification:

```
/* Returns the length of the string.  Returns 0 if str is
 * NULL and otherwise the length of str.
 */
int strlen(char * str);
```

For example, `strlen("Hello universe!")` should return 15.

Solution. We explore several variations. The first is a straightforward implementation that counts the number of iterations until the string terminator is encountered:


```
int strlen(char * str) {
  if (str == NULL) return 0;
  int n = 0;
  while (str[n] != '\0') {
    n++;
  }
  return n;
}
```

Recall that, because `NULL` has value 0, the expression `str == NULL` is equivalent to the expression `str == 0`, which in turn is equivalent to the expression `!str`. Similarly, the string terminator character `'\0'` has ASCII value 0; hence, the expressions `str[n] != '\0'`, `str[n] != 0`, and `str[n]` are equivalent, yielding the following minor variation:

```
int strlen(char * str) {
  if (!str) return 0;
  int n = 0;
  while (str[n]) n++;
  return n;
}
```

The `while` loop can be restructured as a `for` loop lacking a body:

```
int strlen(char * str) {
  if (!str) return 0;
  int n;
  for (n = 0; str[n]; n++);
  return n;
}
```

A more significant variation relies on pointer arithmetic. At each iteration of the loop, `str` is incremented—by one byte, because `str` is declared as a `char *`. To check if the string terminator has been reached, any of `*str != '\0'`, `*str != 0`, and `*str` can be used:

```
int strlen(char * str) {
  if (!str) return 0;
  int n;
  for (n = 0; *str; n++, str++);
  return n;
}
```

Notice how both `n` and `str` are incremented in the `for` loop by separating the incrementing statements by a comma.

We can drop the counter entirely by once and for all grasping the true nature of addresses:

```
int strlen(char * str) {
  if (!str) return 0;
```

```
  char * start = str;
  for (; *str; str++);
  return str - start;
}
```

In this version, we remember `str`'s initial value with `start`, iterate through the string until the string terminator is encountered, and then return the difference between the final address held by `str` and its initial address, held by `start`. Since a character occupies one byte, this difference is exactly the length of the string. □

Exercise 3.18. Write a function that concatenates two C strings. It should implement the following specification:

```
/* Writes str1 followed by str2 into the memory pointed to
 * by out.  Returns 0 if successful and -1 if any of the
 * parameters are NULL.
 */
int concat(char * str1, char * str2, char * out);
```

Solution. The strategy is to copy `str1` and then `str2` to `out`. The only potential error to watch for is copying `str1`'s terminator to `out` or forgetting to add a terminator to `out` at the ending.

```
int concat(char * str1, char * str2, char * out) {
  // check for well-formed input
  if (str1 == NULL || str2 == NULL || out == NULL)
    return -1;

  // write str1 to out, skipping the terminator
  int i = 0;
  while (str1[i] != '\0') {
    out[i] = str1[i];
    i++;
  }
  // write str2 to out
  int j = 0;
  while (str2[j] != '\0') {
    out[i] = str2[j];
    i++;
    j++;
  }
  // terminate out
  out[i] = '\0';

  // indicate success
  return 0;
}
```


C programmers have many idioms, some of which are tough to read and have nonnegligible odds of introducing bugs, but all of which are, well, awesome. Here is an implementation that uses some of these idioms:

```
1  int concat(char * str1, char * str2, char * out) {
2    // check for well-formed input
3    if (!str1 || !str2 || !out)
4      return -1;
5
6    // write str1 to out, skipping the terminator
7    while (*str1)
8      *out++ = *str1++;
9    // write str2 to out
10   while (*str2)
11     *out++ = *str2++;
12   // terminate out
13   *out = '\0';
14
15   // indicate success
16   return 0;
17 }
```

The conditions at lines 3, 7, and 10 prefer the Boolean shortcuts, which rely on both `NULL`'s and `'\0'`'s equaling 0. The loops use pointer arithmetic, like in the second version of `shout`, but they also use the `++` operator in a most vexing fashion—vexing, that is, until you understand what is happening. Once you do, you'll probably overuse it. But remember: with great power comes great responsibility—or more likely just the overwhelming temptation to abuse it and few consequences to stop you from doing so. In any case, `*out++ = *str1++` yields the same result as the following:

```
  *out = *str;
  out = out + 1;
  str = str + 1;
```

When `++` is used after a variable, it's called a **post-increment**. There is a **pre-increment** version as well: `*(++out) = *(++str1)` yields the same result as the following:

```
  out = out + 1;
  str = str + 1;
  *out = *str;
```

This code sequence yields different results than the post-increment form.

Overusing these idioms can be tempting at times. Here's a puzzle for those who go in for such things. Figure out why the following implementation works:

```
int concat(char * str1, char * str2, char * out) {
  if (!str1 || !str2 || !out) return -1;
  --out;
```

```
4   while (*++out = *str1++);
5   while (*out++ = *str2++);
6   return 0;
7 }
```

The pre- and post-increments are applied before the `*`'s in lines 4–5—but keep in mind that a pre-increment expression evaluates to the original value.

Here is where I'd like to say that I don't write such code in practice; that the best code is correct, efficient, and readable; and that modern compilers optimize so well that every version is equally efficient, thus leaving no reason not to go with the version that is easiest to read. But, alas, I can't—and not because one of the latter two statements is wrong. (**Challenge:** Translate this last paragraph into Boolean logic.) □

Exercise 3.19. Write a function `copyString` that copies a C string `in` to another character array referenced by `out`. We provide a version that uses pointer arithmetic intensively:

```
/* Copy string in into the buffer referenced by out. */
int copyString(char * in, char * out) {
  if (!in || !out) return -1;
  while (*in) *out++ = *in++;
  *out = '\0';
  return 0;
}
```

An expression like `*in++` is executed by first incrementing `in` and then applying `*` to the resulting address. Implement a version that uses array indexing instead of pointer arithmetic, and test it via a `main` function.

In fact, an even more concise version is possible using `do`/`while`:

```
/* Copy string in into the buffer referenced by out. */
int copyString(char * in, char * out) {
  if (!in || !out) return -1;
  do { *out++ = *in++; } while (*in);
  return 0;
}
```

In this version, the assignment occurs before the check, allowing the string terminator to be copied in the loop. □

Exercise 3.20. Do you suffer from a friend or a family member who overuses exclamation marks in textual communication? Write a function called `toneItDown` to convert all exclamation marks to periods.

```
/* Replaces each '!' with a '.'.  Return value indicates
 * erroneous input or success.
 */
int toneItDown(char * in, char * out);
```


The following unit test in `main` exercises its basic functionality:

```
1 #include <assert.h>
2 #include <stdio.h>
3
4 // Write toneItDown here.
5
6 int main() {
7   // unit test
8   char email[] = "Hi friends!  Im so excited about "
9     "programming!!!  Its so kewl!";
10  char out[128];
11  printf("%s\n", email);
12  int err = toneItDown(email, out);
13  assert (!err);
14  printf("%s\n", out);
15  return 0;
16 }
```

Notice how the definition of a string constant can be spread across multiple lines, as in lines 8–9; the C compiler concatenates the parts into one long string. Once `toneItDown` is added, compiling and running yields an improved, though far from perfect, translation of the text message:

```
$ gcc -Wall -Wextra -o tid tid.c
$ ./tid
Hi friends!  Im so excited about programming!!!  Its so kewl!
Hi friends.  Im so excited about programming...  Its so kewl.
```

□

Exercise 3.21. String manipulation functions are often vulnerable to ill-formed C strings: character arrays that lack string terminators. For example, if `copyString` of Exercise 3.19 is given an ill-formed string as `in`, it will read and write through memory until a 0 is found or until a segmentation fault occurs. Write a protected version of `copyString` that transfers at most $n-1$ characters from `in` to `out` and always writes a string terminator to `out`.

```
/* Copies at most n-1 characters of string in into the
 * buffer pointed to by out.  If n is reached, returns -2.
 * Otherwise, returns -1 for malformed input and 0 upon
 * successful completion.
 */
int copyStringN(char * in, char * out, int n);
```

Implement a unit test of `copyStringN` in a `main` function that exercises its full protective functionality. □

Exercise 3.22. Write a function that reverses a string. It should implement the following specification:

```
/* Reverses the string in into the string out.  Returns 0
 * if successful and -1 if in or out is NULL.
 */
int reverse(char * in, char * out);
```

For example, consider this unit test in `main`:

```
#include <assert.h>
#include <stdio.h>

// Write reverse here.

int main() {
  char str[] = "Hello universe!";
  char out[32];
  int err = reverse(str, out);
  assert (!err);
  printf("%s\n%s\n", str, out);
  return 0;
}
```

It should yield the following on the terminal:

```
Hello universe!
!esrevinu olleH
```

Solution. This exercise essentially requires careful thinking about what to do with string terminators.

The first step is to find the end of string `in`:

```
  int i;
  for (i = 0; in[i] != '\0'; i++);
```

The `for` loop does not need a body since all the work is being done in the condition and increment. The corresponding `while` loop is the following:

```
  int i = 0;
  while (in[i] != '\0')
    i++;
```

At this point, `in[i] == '\0'`. We are now ready to read `in` in reverse while simultaneously writing into `out`. Whereas indexing made sense for the first task, a mix of indexing and pointer arithmetic works well for the second:

```
  for (i--; i >= 0; i--) {
    *out = in[i];
    out++;
  }
```

The corresponding `while` loop is the following:


```
  i--;
  while (i >= 0) {
    *out = in[i];
    out++;
    i--;
  }
```

To avoid writing `'\0'` as the first character of `*out` (which would yield a rather short string), `i` is first decremented. Then `in` is read backwards by decrementing `i` as the pointer `out` advances in memory.

Finally, the reversed string must be terminated to really be a string:

```
  *out = '\0';
```

All together, we have the following:

```
int reverse(char * in, char * out) {
  // check for well-formed input
  if (!in || !out) return -1;

  // find the end of the string in
  int i;
  for (i = 0; in[i] != '\0'; i++);

  // i should index the terminator of in
  assert (in[i] == '\0');

  // read in backwards, write out forwards
  for (i--; i >= 0; i--) {
    assert (in[i] != '\0');
    *out = in[i];
    out++;
  }
  // terminate out
  *out = '\0';

  // indicate success
  return 0;
}
```

If you are having trouble understanding this implementation, execute it by hand on a small example string. □

Exercise 3.23. A clichéd trick to passing secret messages is to embed the message in a larger text. For example, one might write a letter in such a way that reading the final word of each line reveals the actual message. Write a function called `decode` that, given a multi-line string (a string with `'\n'` characters within it), prints the message consisting of only the final word of each line. For example, consider this innocuous message:

Hey, old friend. I need to go
to the market today or tonight
to fetch some drinks and food to
bring – maybe also a pint or two
for some jolly times, hey? Oh
I almost forgot: Martha got seven
if you can believe it. From Elmer
no doubt. Cheerio – ST

Applying `decode` should reveal the sinister message, "go tonight to two Oh seven Elmer ST."

Solution. Tackling a complex task like this one requires designing an algorithm before attempting to write the code:

1. Throughout the computation, maintain the pointer `word` so that it points to the beginning of the current word. A word is a sequence of nonspace characters that is either at the start of the message or preceded by a space character, which might be a new line.
2. `str` iterates through subsequent characters until either a space (`' '`) or a newline (`'\n'`) is encountered.
3. If a newline is encountered, copy the string starting at `word` and ending at `str - 1` into the output buffer.
4. If instead a space is encountered, set `word` to the address one character beyond `str`.
5. Return to Step 2 unless the string terminator is encountered, in which case, terminate the output string and return.

Now that we understand what we need to do, we can implement both the function and the unit test inspired by the example above:


```
#include <assert.h>
#include <stdio.h>

/* Given a string str, writes the final word of each line
 * of str into msg.  Returns 0 or -1 to indicate
 * success/input error, as usual.  See the unit test below
 * for an example application.
 */
int decode(char * str, char * msg) {
  if (!str || !msg) return -1;

  // Step 1: points to the word currently being read
  char * word = str;

  while (*str) {
    // reached the end of a line?
    if (*str == '\n') {
      // Step 3: copy the last word into msg
      while (word != str)
        *msg++ = *word++;
      *msg++ = ' ';
      // Step 1
      word++;
    }
    // reached the end of a word?
    else if (*str == ' ') {
      // Steps 4, 1: set word to point to the next position
      word = str + 1;
    }
    // keep reading
    str++;
  }
  // Step 5: don't forget to terminate the string
  *msg = '\0';

  return 0;
}

// unit test of decode
int main() {
  // C allows writing constant strings across multiple
  // lines as follows:
  char * letter = "Hey, old friend.  I need to go\n"
                  "to the market today or tonight\n"
                  "to fetch some drinks and food to\n"
                  "bring -- maybe also a pint or two\n"
                  "for some jolly times, hey?  Oh\n"
                  "I almost forgot: Martha got seven\n"
                  "if you can believe it.  From Elmer\n"
                  "no doubt. Cheerio -- ST\n";
  // buffer to hold decoded message
  char decoded[128];

  int err = decode(letter, decoded);
  assert (!err);
  printf("%s\n%s\n", letter, decoded);
  return 0;
}
```

Compiling and running the program reveals the murderous message:

```
$ gcc -Wall -Wextra test.c
$ ./a.out
Hey, old friend.  I need to go
to the market today or tonight
to fetch some drinks and food to
bring -- maybe also a pint or two
for some jolly times, hey?  Oh
```

```
I almost forgot: Martha got seven
if you can believe it.  From Elmer
no doubt. Cheerio -- ST

go tonight to two Oh seven Elmer ST
```

With the true meaning revealed, it remains only to decide whether we should go tonight to 207 Elmer St. to prevent whatever blood-chilling crime is in the works. □

Exercise 3.24. Implement the following specification:

```
/* Removes all vowels from string in and writes the result
 * to out.  Returns 0 if successful and -1 if either in or
 * out is NULL.
 */
int xvowelize(char * in, char * out);
```

For example, consider this unit test:

```
#include <assert.h>
#include <stdio.h>

// Write xvowelize here.

int main() {
  char str[] = "Hello universe!";
  char out[32];
  int err = xvowelize(str, out);
  assert (!err);
  printf("%s\n%s\n", str, out);
  return 0;
}
```

Executing it should yield the following on the terminal:

```
Hello universe!
Hll nvrs!
```

□

Exercise 3.25. Implement the following specification:

```
/* Returns whether str1 and str2 are equal.  Returns 0 if
 * either str1 or str2 is NULL or if they are not equal;
 * returns 1 if they are equal
 */
int streq(char * str1, char * str2);
```

□

Exercise 3.26. Write a function that determines whether a given string has a given prefix:

```
/* Returns 0 if pre or str is NULL or if pre is not a
 * prefix of str.  Otherwise returns 1.
 */
int prefix(char * pre, char * str);
```

Recall that integer values 0 and 1 correspond to Boolean values "false" and "true," respectively.

Solution. The strategy is to iterate through the strings simultaneously. If ever there is a mismatch, the function returns 0, but if it iterates through all of the prefix and always finds matches, it returns 1.

```
int prefix(char * pre, char * str) {
  if (!pre || !str) return 0;
  int i;
  for (i = 0; pre[i]; i++)
    if (pre[i] != str[i])
      return 0;
  return 1;
}
```

Test this function on several examples. Include examples in which one or the other string is empty, that is, consists of just the string terminator, and in which the prefix is longer than the string.

A version using pointer arithmetic avoids the use of the loop variable `i`:

```
int prefix(char * pre, char * str) {
  if (!pre || !str) return 0;
  while (*pre)
    if (*pre++ != *str++)
      return 0;
  return 1;
}
```

□

Exercise 3.27. Write a function that determines whether a given string has a given suffix:

```
/* Returns 0 if str or suf is NULL or if suf is not a
 * suffix of str.  Otherwise returns 1.
 */
int suffix(char * str, char * suf);
```

To decide if a string has a given suffix, it would be wise to increment backward through the string and the suffix. Review Exercise 3.22 to see another function that reads a string in reverse. □

Exercise 3.28. Implement the following specification:

```
/* Returns whether str contains an instance of substr.
 * Returns 0 if either str or substr is NULL or substr is
 * not in str; returns 1 if substr is in str.
 */
int hasSubstring(char * str, char * substr);
```

For example, `hasSubstring("Hello universe!", "verse")` should return 1. Use the following `main` function to test your code:

```
#include <assert.h>
int main() {
  assert(hasSubstring("Hello universe!", "lo"));
  assert(hasSubstring("Hello universe!", "verse"));
  assert(hasSubstring("Hello universe!", ""));
  assert(hasSubstring("", ""));
  assert(!hasSubstring("Hello universe!", "verses"));
  assert(!hasSubstring("Hello universe!", "loun"));
  assert(!hasSubstring("Hello universe!", "erse!!"));
  return 0;
}
```

This exercise hints at the depth of the subject of computation. While the straightforward implementation is what is intended here, the interested reader should investigate the **Knuth–Morris–Pratt**, or **KMP**, algorithm. □

Exercise 3.29. Implement the following specification:

```
/* Compares str1 and str2 according to "dictionary" (aka,
 * "lexicographic") order, where characters are ordered by
 * their ASCII values. Returns -1 if str1 comes before
 * str2; 0 if either str1 or str2 is NULL or if they are
 * equal; and 1 if str1 comes after str2.
 */
int strcmp(char * str1, char * str2);
```

For example, consider the following unit test in `strcmp_test.c`:


```
#include <stdio.h>

// Write strcmp here.

int main() {
  printf("aardvark, aardwolf %d\n",
         strcmp("aardvark", "aardwolf"));
  printf("AVAST, avast %d\n", strcmp("AVAST", "avast"));
  printf("ahoy, ahoy %d\n", strcmp("ahoy", "ahoy"));
  printf("Watch for aardvarks!, "
         "Watches aren't for aardwolves. %d\n",
         strcmp("Watch for aardvarks!",
                "Watches aren't for aardwolves."));
  printf("zoology, zoo %d\n", strcmp("zoology", "zoo"));
  return 0;
}
```

Once `strcmp` is added, compiling and running indicates the ASCII-based dictionary order of these strings:

```
$ gcc -Wall -Wextra -o strcmp_test strcmp_test.c
$ ./strcmp_test
aardvark, aardwolf -1
AVAST, avast -1
ahoy, ahoy 0
Watch for aardvarks!, Watches aren't for aardwolves. -1
zoology, zoo 1
```

□

4

Debugging

Two aspects of programming frustrate novice programmers: getting the syntax right; and dealing with the many, often simple, bugs that cause program behavior to differ from what was expected. Experience resolves the first issue: braces, semicolons, and funny phrases like `int *` become natural in time, until you find yourself speaking programming in everyday conversation. (For example: "Dude, didn't parse that; can you repeat?" "Yeah, we're neighbors, so my address is just hers plus plus." "And then I'm all, like, you know, int star star, *obviously*." Don't blame me when it happens; I'm just the messenger.)

Experience partially helps with the second issue. Over time, you will introduce fewer novice bugs into your code, although the potential subtlety of the bugs that you do introduce will rise in proportion with the complexity of the code. Hence, even the most experienced programmers encounter bugs regularly. This chapter discusses techniques and tools to minimize the number of bugs and to squash the ones that inevitably get around your defenses.

4.1 Write-Time Tricks and Tips

The easiest way to debug is to avoid introducing bugs in the first place. **Defensive programming** is, as the term suggests, the first line of defense against bugs. While preventing bugs entirely is impossible, defensive practices prevent many simple bugs and help to reveal and to isolate bugs when they do occur.

4.1.1 Build Fences around Functions

Program functions defensively. Write them as if they will be called by someone with malevolent (or at least mischievous) intent.

When appropriate, structure functions as we have been doing for the past few chapters. First, use the return value to indicate erroneous input or if an issue arises during the main computation. Use call-by-reference semantics

to return the actual result of a computation. Second, immediately check that input is well formed. Not everything can be checked, of course. For example, we can check if a pointer is NULL, but if it is simply uninitialized and thus holding a random value, we're out of luck. Similarly, discovering if a supposed string is not well formed is difficult, although Exercise 3.21 offers one preventative technique, and Exercise 7.7 suggests another.

Some functions are too simple or are not part of an exposed interface and thus do not warrant the full treatment. For example, in complex programs, one often writes many functions that together do the actual work and a few functions that are intended to be interfaces. It is convenient to write the worker functions in an unprotected form—in particular, such that they use their return values to return actual computed values rather than to indicate success or failure. They might additionally make unchecked assumptions about their inputs (recall, for example, `_sum` of Exercise 2.6). But an **interface function**—one that separates the internals of how a related set of functions work from the external environment of the rest of the program—should have a tall and sturdy fence.

Use assertions. Whenever you make an assumption or have an expectation *that must always hold*, write an `assert` statement. Think of `assert` as a way of comparing the model in your head against the actual implementation. It often happens that an assumption that was valid for a while becomes invalid when a function is called in a new context.

Let's examine the solution to Exercise 3.22:

```
int reverse(char * in, char * out) {
  // check for well formed input
  if (!in || !out) return -1;

  // find the end of the string in
  int i;
  for (i = 0; in[i]; i++);

  // i should index the terminator of in
  assert (in[i] == '\0');

  // read in backward, write out forwards
  for (i--; i >= 0; i--) {
    assert (in[i] != '\0');
    *out = in[i];
    out++;
  }
  // terminate out
  *out = '\0';

  // indicate success
  return 0;
}
```

This function is a “black box” to the caller: the caller wants to reverse a string but does not care how the reversal is accomplished. Therefore, this function has a fence: line 3 checks the input, and the return value indicates erroneous input or success. Line 10 asserts that line 7, which is somewhat tricky, has achieved the desired result: `i` indexes the end of the C string `in`. This assertion is thus a self-check: “Go, me! This code is so clever. (I guess I better make sure it does the right thing.)” Line 14 asserts our primary hope in the loop—that we don’t accidentally terminate the string that we’re writing in `out` too early. For example, if we had forgotten the decrement of `i` in line 13 the `assert` would be triggered.

This function is still vulnerable to a malformed string. The loop at line 7 would execute forever (well, until a segmentation fault occurs) if `in` lacked a string terminator. One option for making `reverse` more robust is to require the caller to provide a maximum possible length:[1]

```
/* Reverses the C string in into out.  maxLength indicates
 * the maximum possible length of in; if this length is
 * exceeded, returns -2.  Returns 0 if successful, and -1
 * if either in or out is NULL.
 */
int nreverse(char * in, char * out, int maxLength) {
  // check for well formed input
  if (!in || !out) return -1;

  // find the end of the string in
  int i;
  for (i = 0; in[i]; i++) {
    if (maxLength <= 0) return -2;
    maxLength--;
  }

  // The remainder of the code is as in reverse.
```

A return value of -2 alerts the caller that an assumption is incorrect: the string is longer than expected and thus may be lacking a terminator. If you’re both the function writer and the caller, you’ll thank yourself.

4.1.2 Document Code

When it’s too difficult to write an assertion, write a comment that explains what you expect to hold at a given point. In particular, provide a complete function specification at the top of important functions, as in the implementation of `nreverse` above. When programming in a team environment, which

[1] Some functions in the standard string library, `string.h`, require this protective argument.

is typical, comments help team members to detect inconsistent internal models among team members and to isolate bugs that span multiple members’ contributions.

4.1.3 Prefer Readability to Cleverness

Modern compilers can usually compile readable code and “clever” code into machine code with similar performance. Only algorithmic optimizations are typically worth pursuing.[2] Therefore, write readable code. Avoid embedding pre- and post-increments in complex statements. Avoid embedding assignments in conditionals or on the right-hand side of other assignments. (It’s possible! And done!) Write brief comments to explain tricky lines. If a “clever” section of code inspires manic laughter, consider rewriting it.

4.2 Compile-Time Tricks and Tips

As the complexity of our programs increases, we will switch from invoking the compiler, `gcc`, at the command-line to using `make` and **makefiles**. In either case, one easy way to catch trivial but annoying bugs is to up the warning level: `gcc -Wall -Wextra <file>` enables additional warnings.[3] Consider the following (buggy) program, which we will assume is in file `buggyfib.c`:

```
#define N 100
int main() {
  // declare an array of N integers
  int fib[N];
  int i;

  // define the first two elements
  fib[0] = 1;
  fib[1] = 1;

  while (1) {
    fib[i] = fib[i-2] + fib[i-1];
    i++;
    if (i = N) break; // break exits the loop
  }
}
```

[2] An exception is when programming in an underpowered environment, for example, when using a proprietary compiler for an embedded system. Even then, only particular sections of code need be optimized at the statement level, and it may be worth writing those sections in assembly anyway.

[3] `-Wall` means “warnings: all,” but it’s cool that it is pronounced “wall.” `-Wextra` is necessary because `-Wall` doesn’t actually produce *all* warnings.

The `break` statement (line 14) causes control to exit the loop. How many bugs can you spot?

Running without warnings reveals nothing: `gcc buggyfib.c` doesn't report any issues. But here is what `gcc -Wall -Wextra buggyfib.c` finds:

```
buggyfib.c: In function main:
buggyfib.c:14: warning: suggest parentheses around assignment
  used as truth value
buggyfib.c:16: warning: control reaches end of non-void
  function
buggyfib.c:12: warning: i is used uninitialized in this
  function
```

At line 14, the compiler suggests that we use parentheses around the assignment—wait, what?! Assignment? Oh, right, I guess I intended `i == N`, didn't I? Typing `=` when one means `==` is a common mistake.

At line 16, the compiler points out that `main` must return a value since `main` is declared as returning an integer. Easy enough: add `return 0`.

At line 12, the compiler spots a potentially nasty problem: `i` might be uninitialized. Whoops.

Compiling without `-Wall -Wextra` is like riding a racing bicycle with the tires at 40 psi. And ignoring the output of `gcc -Wall -Wextra` is like inflating them to 100 psi but leaving the valves open just for kicks.

By the way, use assertions as much as you want because you can always disable them: `gcc -Wall -Wextra -DNDEBUG` disables assertions. Just be sure that you don't use assertions as follows:

```c
assert(nreverse(str, out, length) == 0);
```

Disabling assertions in this case will remove the entire statement. Instead, write

```c
int err;
err = nreverse(str, out, length);
assert (!err);  // same as assert(err == 0)
```

You might be concerned that `err` will still occupy memory even when assertions are disabled. Rest assured: modern compilers can remove unnecessary variables while juggling swords blindfolded.

Finally, when your program is ready for release, compile with `gcc -O3` to enable all optimizations.

4.3 Runtime Tricks and Tips

4.3.1 GDB: The GNU Project Debugger

Bugs happen. If the error-signaling return values and the assertions are not revealing the source, the next step is to invoke `gdb`. Consider this (buggy) program, which we assume is saved to `sum.c`:

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

#define N 5

int _sum(int n) {
  assert (n > 0);
  if (n == 1)
    return 0;
  else {
    int upto = _sum(n-1);
    return upto + n;
  }
}

int sum(int n, int * s) {
  if (n <= 0) return -1;
  if (!s) return -2;
  *s = _sum(n);
  return 0;
}

int main() {
  int s;
  int err = sum(N, &s);
  assert (!err);
  printf("%d\n", s);
  return 0;
}
```

`gcc -Wall -Wextra sum.c` is silent, but running `./a.out` yields 14, not 15 as expected. You may be able to spot the bug already—and, indeed, one of the most effective debugging techniques is simply to read the code critically—but let's suppose that you haven't.

To prepare for `gdb`, we compile with the `-g` flag, which causes `gcc` to compile debugging information into the binary: `gcc -Wall -Wextra -g sum.c`. Then we fire up `gdb`:

```
$ gdb ./a.out
(gdb)
```

First let's run it:

```
(gdb) run
Starting program: .../a.out
14

Program exited normally.
```

Let's look deeper:

```
(gdb) break sum.c:main
Breakpoint 1 at 0x400606: file sum.c, line 25.
(gdb) list
20   return 0;
21 }
22
23 int main() {
24   int s;
25   int err = sum(N, &s);
```

The command `break sum.c:main` sets a **breakpoint** at the beginning of `main` in file `sum.c`. Typing `break main` would have been sufficient in this case, as there is no ambiguity when there is only one function named `main`. The command `list` lists the (source) code around where the program counter is currently pointing, which is currently at the beginning of `main`.

Now when we `run`, something different happens:

```
(gdb) run
Starting program: .../a.out

Breakpoint 1, main () at sum.c:25
25   int err = sum(N, &s);
(gdb)
```

Let's step through the code:

```
(gdb) step
sum (n=5, s=0x7fffffffe05c) at sum.c:17
17   if (n <= 0) return -1;
(gdb) step
18   if (!s) return -2;
(gdb) step
19   *s = _sum(n);
(gdb) print n
$1 = 5
```

Stepping causes `gdb` to execute one statement at a time. The first `step` enters `sum`, and the next two step past `sum`'s "fence" code. The command `print n` prints the current value of `n`, which is 5 as expected. This value is also indicated by the second line above, which displays the arguments to `sum`.

Control is now at line 19. Stepping once more brings us into the function that does the real work: `_sum`.

```
(gdb) step
_sum (n=5) at sum.c:7
7   assert (n > 0);
(gdb) backtrace
#0  _sum (n=5) at sum.c:7
#1  0x00000000004005f1 in sum (n=5, s=0x7fffffffe05c) at
  sum.c:19
#2  0x0000000000400617 in main () at sum.c:25
```

Executing `backtrace` shows a summary of the stack. Each entry is a stack frame, with `#0` referring to the stack frame at the top of the stack. Let's keep stepping:

```
(gdb) step
8   if (n == 1)
(gdb) step
11     int upto = _sum(n-1);
```

Control reaches the line that recursively calls `_sum`. Let's follow Alice into the rabbit hole:

```
(gdb) step
_sum (n=4) at sum.c:7
7   assert (n > 0);
(gdb) backtrace
#0  _sum (n=4) at sum.c:7
#1  0x00000000004005b8 in _sum (n=5) at sum.c:11
#2  0x00000000004005f1 in sum (n=5, s=0x7fffffffe05c) at
  sum.c:19
#3  0x0000000000400617 in main () at sum.c:25
(gdb) step
8   if (n == 1)
(gdb) step
11     int upto = _sum(n-1);
(gdb) step
_sum (n=3) at sum.c:7
7   assert (n > 0);
(gdb) backtrace
#0  _sum (n=3) at sum.c:7
#1  0x00000000004005b8 in _sum (n=4) at sum.c:11
#2  0x00000000004005b8 in _sum (n=5) at sum.c:11
#3  0x00000000004005f1 in sum (n=5, s=0x7fffffffe05c) at
  sum.c:19
#4  0x0000000000400617 in main () at sum.c:25
```

Notice how each invocation of `_sum` is shown with the value of its parameter. The recursion is evident in the growth of the stack, as revealed by `backtrace`.

The values of `sum`'s parameters are shown as well: it was called with 5 and a pointer to where to write the sum. Let's suppose that we suddenly got curious about `sum`'s parameter `s`:

```
(gdb) frame 4
#4  0x0000000000400617 in main () at sum.c:25
25   int err = sum(N, &s);
(gdb) print s
$2 = 0
(gdb) frame 3
#3  0x00000000004005f1 in sum (n=5, s=0x7fffffffe05c) at sum.c:1
19   *s = _sum(n);
(gdb) print *s
$3 = 0
(gdb) print s
$4 = (int *) 0x7fffffffe05c
(gdb) frame 0
#0  _sum (n=3) at sum.c:7
7   assert (n > 0);
```

The first command focuses on stack frame `#4`, which is `main`'s. Now we can inspect the value of `main`'s local variable `s`. The command `frame 3` changes focus to `sum`'s stack frame, where we can inspect `sum`'s parameter `s`. (As should be plain to you by now, `sum`'s parameter `s` just happens to have the same name as `main`'s local variable `s`; they are otherwise unrelated—except that `sum`'s `s` points to the memory cell associated with `main`'s `s`.) Executing `frame 0` returns focus to the top of the stack.

Nothing seems amiss so far, so let's set a breakpoint to catch when the runtime behavior changes substantially, namely, when `n == 1`:

```
(gdb) break sum.c:8 if n == 1
Breakpoint 2 at 0x40059e: file sum.c, line 8.
(gdb) continue
Continuing.

Breakpoint 2, _sum (n=1) at sum.c:8
8   if (n == 1)
```

The first command sets a breakpoint and a **watch condition**: `gdb` breaks at line 8 only if `n == 1`. Then the command `continue` causes `gdb` to continue running until the next breakpoint is reached or the program halts. In this case, the new breakpoint is reached:

```
(gdb) step
9     return 0;
```


At this point, an alert programmer might wonder why `_sum` is returning 0 instead of 1 when `n == 1`, since the sum of 1 is 1, not 0. But then again, maybe not—or maybe it's 2:00 am, and you're not exactly running at full capacity. Either way, let's continue:

```
(gdb) step
14 }
(gdb) step
12     return upto + n;
(gdb) print upto
$1 = 0
(gdb) print n
$2 = 2
```

Now that pair of values looks strange for sure, especially if you grab a pencil and paper and write out a few sums:

$$
\begin{aligned}
&1 = 1 \\
&1 + 2 = 3 \\
&1 + 2 + 3 = 6
\end{aligned}
$$

Having found the issue, let's clean up:

```
(gdb) quit
A debugging session is active.

Inferior 1 [process 14535] will be killed.

Quit anyway? (y or n) y
```

We change line 9 to `return 1`. Recompiling and running yields the expected value of 15.

This `gdb` session tracked down a computation bug. What happens with an assertion error? Consider this (really buggy) version of `_sum`:

```
int _sum(int n) {
  assert (n > 0);
  int upto = _sum(n-1);
  return upto + n;
}
```

Executing `./a.out` yields

```
a.out: sum.c:7: _sum: Assertion `n > 0' failed.
Aborted
```

Super! The assertion worked. Now let's finish off this bug with `gdb`:

```
(gdb) run
Starting program: .../a.out
a.out: sum.c:7: _sum: Assertion `n > 0' failed.
```

```
Program received signal SIGABRT, Aborted.
0x00007ffff7a8da75 in *__GI_raise (sig=<value optimized out>)
    at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
64 ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file
        or directory.
in ../nptl/sysdeps/unix/sysv/linux/raise.c
(gdb) backtrace
#0  0x00007ffff7a8da75 in *__GI_raise (sig=<value optimized
    out>) at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#1  0x00007ffff7a915c0 in *__GI_abort () at abort.c:92
#2  0x00007ffff7a86941 in *__GI___assert_fail
    (assertion=0x400752 "n > 0", file=<value optimized out>,
    line=7, function=0x400766 "_sum") at assert.c:81
#3  0x000000000040059e in _sum (n=0) at sum.c:7
#4  0x00000000004005ab in _sum (n=1) at sum.c:8
#5  0x00000000004005ab in _sum (n=2) at sum.c:8
#6  0x00000000004005ab in _sum (n=3) at sum.c:8
#7  0x00000000004005ab in _sum (n=4) at sum.c:8
#8  0x00000000004005ab in _sum (n=5) at sum.c:8
#9  0x00000000004005ed in sum (n=5, s=0x7fffffffe05c) at
    sum.c:15
#10 0x0000000000400613 in main () at sum.c:21
(gdb) frame 4
#4  0x00000000004005ab in _sum (n=1) at sum.c:8
8   int upto = _sum(n-1);
```

As usual when working with a complex system, not everything makes sense. What do all the lines concerning `raise.c` and `__GI_abort` mean? Apparently, the computer on which this session was run did not have the Linux source code installed. Nonetheless, typing `backtrace` and carefully filtering out the irrelevant information yields a clue: the stack, which we expected to top out when `n == 1`, shows calls to `_sum` with arguments 5, 4, 3, 2, 1, and 0. The final call is unexpected and points to a lack of a condition for ending the recursion. Inspecting stack frame `#4` reveals that `_sum` is indeed being called when `n == 1`, so that `n-1 == 0`.

The lesson here is twofold. First, don't panic when not everything makes sense; instead, pick out the useful information and discard the rest. Working with computers can be frustrating if you insist on understanding everything that they do. Try to flow instead. Second, assertions and `gdb` play well together: simply run the program within `gdb` until it aborts; then inspect the carnage.

Suppose, though, that even the assertion were missing:


```
int _sum(int n) {
  int upto = _sum(n-1);
  return upto + n;
}
```

Executing this program yields: `Segmentation fault`. Time to fire up `gdb`:

```
(gdb) run
Starting program: .../a.out

Program received signal SIGSEGV, Segmentation fault.
0x000000000040057c in _sum (n=Cannot access memory at address
  0x7fffff5aeffc) at sum.c:6
6 int _sum(int n) {
```

Notice that the value of `_sum`'s parameter cannot even be displayed. That can't be good. Typing `backtrace` is probably a bad idea at this point (try it!). Instead, I carefully and with bated breath type the command `up`, which moves focus to the next stack frame. (It's unfortunate that the command is `up` when we actually intend to move down the stack—an issue of convention.)

```
(gdb) up
#1  0x000000000040058c in _sum (n=-225189) at sum.c:7
7   int upto = _sum(n-1);
```

Whoa, there. It looks like `n == -225189`, indicating that the recursive calls to `_sum` continued just a tad longer than desired. The segmentation fault occurred because of a **stack overflow**: the stack just got too big, which usually indicates an issue with recursion, specifically, a missing base case.

This section merely introduces `gdb`. Use `gdb`'s `help` command to learn more as your expanding programming skills demand greater debugging power. Also, try using `gdb` inside an editor like `emacs` or `vim`; add-on modules to these editors facilitate debugging.

4.3.2 Valgrind

Another powerful tool is `valgrind`, which tracks all memory operations of an executable. Simply run `valgrind ./a.out` and read the resulting report. With various options, it can provide details on reading uninitialized data, reading or writing to undesired places, and more. However, until we cover dynamic memory allocation in Chapter 6, `valgrind` is not terribly useful, so we postpone our discussion of this tool until then.

4.4 A Final Word

Discipline, patience, and critical thinking are the three most powerful tools we have when creating software (or anything, for that matter). Tools help, but only to the extent that we keep our wits about us. When you encounter a particularly nasty bug, take a productive break; then return and apply critical thinking to the task.

5

I/O

A program is only useful to the extent that it communicates the result of a computation to the user. Moreover, the most useful programs are those whose executions vary according to user-provided input. This chapter covers everyday usage of the standard **I/O** (**input/output**) library, `stdio.h`. As with the rest of this text, the coverage of the standard I/O library is not meant to be exhaustive but rather tutorial in nature. Standard references, such as Kernighan and Ritchie's *The C Programming Language*, fill in the details; alternately, technical descriptions of standard library functions—including those used in this chapter: `printf`, `scanf`, and `sscanf`—are easily found online.

5.1 Output

Output to the terminal is accomplished with the `printf` function. The function virtually defines its own programming language, but basic usage is straightforward. One bit of magic is that `printf` accepts a variable number of arguments. (In fact, we can write such functions too, by using the standard argument library, `stdarg.h`. I don't recall any occasion on which I have found this facility to offer the right design choice. It seems to have been designed for a few specific applications, `printf` being one of them.)

Actually, `printf` does not print to the terminal, per se. Rather it prints to a special file handle called `stdout`, short for **standard output**, that is defined in `stdio.h`. Unix shells print `stdout` to the terminal, but they also offer facilities for **redirecting** `stdout` to a file. Try executing, for example, the command `ls` on a terminal. It lists the current directory. Now execute `ls > out.tmp`. Instead of printing to the terminal, it prints to the file `out.tmp`. Open `out.tmp` in an editor to verify that the redirection worked.

Suppose that we need to print out the elements of an integer array:

```
1  /* Prints the n integers of a to stdout.  Returns -1 if a
2   * is NULL or n < 0; otherwise, returns 0.
```



```
3   */
4  int printIntArray(int a[], int n) {
5    // check for well-formed input
6    if (!a || n < 0) return -1;
7
8    // print to stdout: one number per line
9    int i;
10   for (i = 0; i < n; i++)
11     printf("%d\n", a[i]);
12
13   return 0;
14 }
```

The call to `printf` at line 11 has two arguments, a **format string** that defines the format of what is being printed, and an argument that is used to fill in the one placeholder in the format string. The format string `"%d\n"` specifies that an integer should be printed in decimal form, followed by a newline. The character `%` indicates the beginning of a placeholder expression, while `%d` indicates that the placeholder should be filled by an integer. The string `\n` specifies the newline character, as usual.

We could get fancier. Suppose that we want to indicate the index of the element as well. Then we need only replace line 11 with this one:

```
11     printf("%d. %d\n", i, a[i]);
```

In this usage, `printf` takes three arguments because the format string requires two `int` values.

Assume that `fibonacci` is defined as in previous chapters, and consider this calling context:

```
#include <stdio.h>

// Insert fibonacci, printIntArray here.

#define N 5

int main() {
  int fib[N];
  int err = fibonacci(fib, N);
  assert (!err);
  err = printIntArray(fib, N);
  assert (!err);
  return 0;
}
```

The following is printed to the terminal:

```
0. 1
1. 1
```

```
2. 2
3. 3
4. 5
```

If vertical space is in short supply, we could replace lines 10–11 of `printIntArray` with the following:

```
  for (i = 0; i < n; i++) {
    // print the index and element followed by spaces
    printf("%d. %d  ", i, a[i]);
    // print a newline every 4th entry
    if (i % 4 == 3)
      printf("\n");
  }
  // print a newline if one was not just printed
  if (i % 4 != 0)
    printf("\n");
```

Redefining `N`,

```
#define N 44
```

yields the following output:

```
0. 1  1. 1  2. 2  3. 3
4. 5  5. 8  6. 13  7. 21
8. 34  9. 55  10. 89  11. 144
12. 233  13. 377  14. 610  15. 987
16. 1597  17. 2584  18. 4181  19. 6765
20. 10946  21. 17711  22. 28657  23. 46368
24. 75025  25. 121393  26. 196418  27. 317811
28. 514229  29. 832040  30. 1346269  31. 2178309
32. 3524578  33. 5702887  34. 9227465  35. 14930352
36. 24157817  37. 39088169  38. 63245986  39. 102334155
40. 165580141  41. 267914296  42. 433494437  43. 701408733
```

We can finally see that the Fibonacci sequence grows very quickly indeed. One more line of output would have yielded an overflow:

```
44. 1134903170  45. 1836311903  46. -1323752223
47. 512559680
```

Starting at `fib[46]`, the computation is no longer correct. Thus we encounter the problem with fixed-size representations of integers.

Strings are just as easy to print as integers. The only difference is that the placeholder expression is `%s` instead of `%d`, and the corresponding argument should have type `char *` and be a well-formed C string. Recall the functions `shout`, `concat`, and `reverse` of Section 3.2:


```
#include <stdio.h>

// Insert the functions shout, concat, and reverse here.

/* Prints an array of strings.  Return 0 if successful, and
 * -1 if a is NULL, n < 0, or any entry of a is NULL.
 */
int printStringArray(char ** a, int n) {
  // check for well-formed input
  if (!a || n < 0) return -1;

  int i;
  for (i = 0; i < n; i++) {
    // return -1 if a[i] is NULL
    if (!a[i]) return -1;
    printf("%s ", a[i]);
  }
  printf("\n");

  return 0;
}

int main(int argc, char ** argv) {
  char str1[32] = "Hello universe!";
  char str2[32], str3[64];

  // 1. Print the command-line argument array.
  printStringArray(argv, argc);

  // 2. Print the "shouted" version of str1.
  shout(str1, str2);
  printf("%s -> %s\n", str1, str2);

  // 3. Print the concatenation of str1 and str2.
  concat(str1, str2, str3);
  printf("%s + %s =\n  %s\n", str1, str2, str3);

  // 4. Print the reversal of str1.
  reverse(str1, str2);
  printf("%s -> %s\n", str1, str2);

  return 0;
}
```

Assuming that this program is completed with the appropriate functions and resides in `strings.c`, compiling and running yields four lines of output:

```
$ gcc -Wall -Wextra -o strings strings.c
$ ./strings some random command-line arguments
./strings some random command-line arguments
Hello universe! -> HELLO UNIVERSE!
```

```
Hello universe! + HELLO UNIVERSE! =
  Hello universe!HELLO UNIVERSE!
Hello universe! -> !esrevinu olleH
```

Notice the first line of output. The first element of the `argv` array—which, recall, is an array of C strings—is the name of the executable. The next elements are the command-line arguments to the program that we typed on the command line, in this case, `some random command-line arguments`. Generating the next three lines requires mixing text and placeholders in the format strings at lines 32, 36, and 40.

This section merely introduces what is possible with `printf`. We can mix integers, strings, and constant text—as well as `float`s and `double`s, which are number types for representing real numbers and which we will discuss in Chapter 6. Moreover, the format string allows complex specifications to indicate alignment and precision of numerical data.

5.2 Input

Input is a fascinating topic, first, because reading user data is typically a basic requirement of an interesting program; and, second, because one can never be too careful about protecting oneself from mischievous, malevolent, or—most likely—ignorant users. There are two types of input: **command-line arguments** and **terminal** or **file input**. The user provides command-line arguments before executing the program. For example, the Unix shell command `ls` can take a modifier, `-l`, that causes it to print more information; in the command `ls -l`, `-l` is a command-line argument to the program `ls`. Command-line arguments are available to the program via the parameters of `main`: `argc` and `argv`.

In contrast, terminal or file input is read during execution of the program. It is made available to the program via `stdin`, short for **standard input**. In this chapter, we introduce `scanf` as a function for reading `stdin`, although there are many other methods. Whereas command-line arguments are typically short and intended to modify program behavior—think of `ls -l`, where `-l` instructs `ls` to list more information—input through `stdin` can be arbitrarily long and is typically data, such as a text document, a sequence of numbers, or a comma-delimited spreadsheet.

We discuss each of these input types in turn.

5.2.1 Command-Line Input

Command-line arguments are intended to modify the behavior of a program or to provide basic information. We already saw a simple example of processing the command-line in the previous section, which consisted of simply printing it; here, we treat `argc` and `argv` as they are really meant to be used.

Suppose that we would like to write a program to write out the Fibonacci sequence up to a user-provided bound, or up to a set bound if the user does not provide one. Given the issue with overflow, the maximum allowable bound is 46. In the program below, we interpret `argv` so that the user can provide the bound via an option, `-b`. Additionally, the `-h` option causes a usage message to be printed. For example, `./fib -b 13` causes the 0th through 13th elements of the Fibonacci sequence to be printed, while `./fib -h` causes a message to be printed informing the user how to use the `fib` program. Finally, a misuse of the command-line—for example, `./fib -notanoption but oh well`—causes an informational message to the user as well.

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>

// Insert the functions fibonacci and printIntArray here.

#define MAX_N 46

void printUsage() {
  printf("Usage: [-b <bound>] [-h]\n  where <bound>"
         " is a number between 0 and 46\n");
}

int main(int argc, char ** argv) {
  int n = MAX_N;
  int fib[MAX_N];
  int i = 0, numRead = 0;

  // parse command line, skipping argv[0] (program's name)
  for (i = 1; i < argc; i++) {
    // strcmp, defined in string.h, returns 0 if the two
    // strings are equal
    if (strcmp("-h", argv[i]) == 0) {
      // user requested usage message
      printUsage();
    }
    else if (strcmp("-b", argv[i]) == 0) {
      if (i+1 == argc) {
        // -b should be followed by another argument
        printUsage();
        return -1;
      }
      // convert the next argument into the integer n
      numRead = sscanf(argv[i+1], "%d", &n);
      i++;
      // numRead == 0 if the next argument isn't an integer
      if (numRead == 0 || n < 0 || n > MAX_N) {
```

```
38        printUsage();
39        return -1;
40      }
41      assert (numRead == 1);
42    }
43    else {
44      // unrecognized argument
45      printUsage();
46      return -1;
47    }
48  }
49
50  fibonacci(fib, n);
51  printIntArray(fib, n);
52  return 0;
53 }
```

A command-line argument parsing loop typically has this form. The loop counter, `i`, ranges from 1 to `argc-1`, so that each string `argv[i]` can be examined. On each iteration of the loop, each major conditional (at lines 23 and 27) tests whether argument `argv[i]` is equal to one of the interpreted modifiers (`-b` or `-h` in this case) by using the function `strcmp` from the `string.h` standard library: `strcmp(str1, str2)` returns 0 precisely when the two strings `str1` and `str2` are equal.

Lines 27–42 are complicated by the possibility that the user may not use the `-b` option properly. Each of the commands

```
./fib -b
./fib -b -13
./fib -b 99
./fib -b totalnonsense
```

yields a polite usage message and a nonzero return value. In Unix, nonzero return values conventionally indicate that the program did not execute successfully.

Line 34 uses the `sscanf` function, which is similar to the `scanf` function that we study in detail next. As their names suggest, `scanf` and its cousins *scan* a file or a string (`sscanf`, for "string scan"). The format string indicates how the input should be structured. It uses placeholders just like `printf`, except that the corresponding arguments are pointers to where the input should be written.

At line 34, `sscanf` is used to scan the string `argv[i+1]`. The format string `"%d"` indicates that an integer is expected, so the corresponding argument is the address of `int` variable `n`. However, it is reading user-provided data, and the user should be assumed to be a 6-month old. Therefore, we take precautions. In particular, `sscanf` returns the number of placeholders that were processed, which we store in `numRead`; in this case, there is only one

placeholder, `%d`, so that `sscanf` must return either 0 or 1. Line 37 checks if `numRead` is 0, which would indicate that the string `argv[i+1]` does not have the form of an integer. It also checks if `n` is negative or too large. In any of these cases, the program politely informs the user, yet again, of how to use the program. When robot arms become standard equipment on laptops, it may be worth programming a `smack` function to emphasize the point.

This careful handling of input is necessary to create robust programs. Several subsequent examples require reading simpler information from the command line and have correspondingly simpler code; nevertheless, the code must still be written to be robust.

Exercise 5.1. Write a program that takes one argument, a positive integer n, and prints the sum $1 + 2 + \cdots + n$.

Solution. The main task is to use `sscanf` to convert the one argument from a string to an integer; however, the code is complicated by the need to work with — how shall I put it? — *challenged* users.

```
#include <stdio.h>

int main(int argc, char ** argv) {
  if (argc != 2) {
    printf("*cough* Expected precisely one argument.\n");
    return -1;
  }
  int n;
  if (sscanf(argv[1], "%d", &n) == 0) {
    printf("Erm, expected an integer.\n");
    return -1;
  }
  if (n <= 0) {
    printf("You've got to be kidding: positive!\n");
    return -1;
  }

  int i, sum = 0;
  for (i = 1; i <= n; i++) sum += i;
  printf("Sum: %d\n", sum);
  return 0;
}
```

All kidding aside, mishandling user input is a major source of security vulnerabilities in production software, so try to get it right. If users cause your program to crash or misbehave, you're the fool, not them. □

Exercise 5.2. Write a program that takes two arguments, two positive integers m and n such that $m < n$, and prints the sum $m + (m + 1) + \cdots + n$. □

5.2.2 Structured Input: Integer Data

Just as `printf` prints to `stdout`, `scanf` reads from `stdin`, short for **standard input**. Unix shells provide mechanisms for chaining programs together via `stdout` and `stdin`. For example, in a directory containing a mix of C and other files, running the command `ls | grep "\.c$"` prints a list of all `.c` files in the directory. The standard Unix utility `grep` reads from `stdin`. The `|` operator, pronounced "pipe," links `ls`'s output, on `stdout`, to `grep`'s input, on `stdin`. This command runs too fast to see that `grep` runs concurrently with `ls`, a useful feature. Writing to `stdout` and reading from `stdin` are effective ways of building programs that can be used as modules in larger commands. Such programs are called **Unix filters**: they "filter" input data into output data.

Like `printf` and `sscanf`, `scanf` takes a format string, and subsequent arguments must correspond to the placeholders of the format string. Since `scanf`, like `sscanf`, reads rather than writes, the subsequent arguments must tell `scanf` where to write; in other words, they must be addresses.

Two important characteristics of `scanf` are that it reads a data **stream** *incrementally* and *just once*. For example, if a stream of integers comes in through `stdin`, as in the program below, each call of `scanf("%d", &num)`, where `num` is an integer variable, reads precisely one integer. Hence, `scanf` is typically used within a loop that executes as long as `stdin` has data.

Reading input is complicated by the possibility of malformed data. For example, an integer might be expected but an arbitrary string provided instead. Alternately, the data stream may end unexpectedly. To detect such situations, `scanf` returns three types of values:

- A positive integer indicating the number of matches. For example, `scanf("%d", &num)` would return 1 to indicate that an integer was read into `num`.
- 0 to indicate that a match did not occur. For example, if `scanf("%d", &num)` was applied at a point in the data stream with, say, `"banana"`, then it would return 0 (unless we switch to a fruit-based number system) and leave `"banana"` unread.
- **EOF**, an acronym for **End of File**, indicating that the data stream has ended.

Consider this program to compute the integer mean of a list of numbers:


```
1  #include <stdio.h>
2
3  int main() {
4    int sum = 0;
5    int cnt = 0;
6
7    while (1) {
8      // read an integer
9      int num;
10     int ret = scanf("%d", &num);
11     // check if stdin has closed
12     if (ret == EOF)
13       // stdin has closed, so exit the loop
14       break;
15     // check if scanf returned 0, indicating bad input
16     if (ret == 0) {
17       printf("Expected an integer.\n");
18       return -1;
19     }
20     sum += num;
21     cnt++;
22   }
23
24   printf("Sum: %d\nInteger mean: %d\n", sum, sum/cnt);
25   return 0;
26 }
```

Notice at line 10 that the second argument to `scanf` is the address of `num`, allowing `scanf` to write a value to the memory cell associated with the variable `num`. Dropping the `&` would cause `scanf` to write to whatever "address" the (integer) value of `num` corresponds to, a serious and potentially frustrating memory bug.

The constant `EOF` is defined in `stdio.h`.[1] A return value from `scanf` of `EOF` indicates that `stdin` has been closed, likely by the external environment, which indicates that all the numbers that are to be entered have been entered. In this case, the `break` statement causes control to jump out of the loop to line 24. Otherwise, `scanf` returns the number of placeholders that it filled. If that number is 0, then the user provided input other than an integer, so a warning and a clean exit is appropriate.

There are two ways of using this program. One method is to write a list of numbers in a file, say `tmp.in`, and then run `./a.out < tmp.in`, where `<` is the Unix shell redirection operator. It causes the contents of the file `tmp.in` to be accessible to the executable `a.out` as `stdin`. Suppose that `tmp.in` contains the following data:

```
13 29 51
-5 1
129
```

Then `./a.out < tmp.in` yields the following:

```
Sum: 218
Integer mean: 36
```

[1] The integer value of `EOF` is not standardized, but we can find out what it is on a given system. Write code to discover its value on your system.

Spacing in the input does not matter when the format string is "%d".

The second method is to execute ./a.out and then type the numbers directly into the terminal. Pressing Control-D closes stdin, causing the program to exit the loop and print the sum. Notice that, if a non-integer is entered, the program provides a message and then exits.

Exercise 5.3. Write a program that reads one or more integers from stdin and prints the minimum. For example,

```
$ ./min
-5 6 4 -7
Min: -7
```

This example is executed by running min, typing -5 6 4 -7, pressing Enter to keep things tidy, and then pressing Control-D to close stdin. Once stdin is closed, the loop inside the program should terminate upon detecting EOF and then print the minimum value.

Solution. We implement the main loop using a slightly different control structure than in the integer mean example, although with the same effect:

```c
#include <stdio.h>

void printUsage() {
  printf("Usage: min < [data file], where the file is a "
         "nonempty list of integers\n");
}

int main() {
  int min;

  // 1. obtain the first value as min
  if (scanf("%d", &min) != 1) {
    // either empty file or not an integer
    printUsage();
    return -1;
  }

  // 2. scan the rest
  // A busy line of code:
  //  a. call scanf, requesting to scan for an integer that
  //     should be written to val
  //  b. set rc to the return code (EOF, 0, or 1)
  //       EOF - end of file
  //         0 - did not match an integer
  //         1 - matched an integer
  //  c. check if the return code is (not) EOF
  int rc, val;
  while ((rc = scanf("%d", &val)) != EOF) {
    // not EOF, but it might be 0
```



```c
    if (rc == 0) {
      // bad data
      printUsage();
      return -1;
    }
    // good data
    if (val < min)
      min = val;
  }

  // 3. report the min
  printf("Min: %d\n", min);

  return 0;
}
```

□

Exercise 5.4. Write a program that prints the range of a sequence of integers provided through stdin. For example, the range of $-3, 15, -8, 29, 17$ is $29 - (-8) = 37$. □

Exercise 5.5. Write a program that reads one integer n from the command line and then n integers from stdin. It should then print the reverse of the sequence.

Solution. Unlike in previous exercises, this program needs to remember all of the numbers so that it can then reverse them. We can use a feature of C called **variable-length arrays** in order to declare an array of the appropriate size. In Chapter 6, we explore heap-allocated memory, a standard and more powerful alternative.

```c
#include <stdio.h>

void printUsage() {
  printf("Usage: rev [n] < [data file], where the file is "
         "a list of n integers\n");
}

int main(int argc, char ** argv) {
  if (argc != 2) {
    // argument n not provided
    printUsage();
    return -1;
  }
  int n;
  if (sscanf(argv[1], "%d", &n) != 1) {
    // the argument is not an integer
    printUsage();
    return -1;
```

```
  }

  // variable-length array
  int nums[n];

  int i;
  for (i = 0; i < n; ++i) {
    // Tricky!  Tell scanf to write the value directly
    // into the correct position of nums.
    int rc = scanf("%d", nums+i);
    if (rc == EOF) {
      printf("Unexpected end of file.\n");
      printUsage();
      return -1;
    }
    if (rc == 0) {
      printf("Expected an integer.\n");
      printUsage();
      return -1;
    }
  }

  // print the numbers in reverse
  for (i = n-1; i >= 0; --i)
    printf("%d ", nums[i]);
  printf("\n");

  return 0;
}
```

□

5.2.3 Structured Input: String Data

Text data are as easy to read as integers. The invocation `scanf("%s", buf)` tells `scanf` to read characters into `buf` *up to but excluding the next space character*, which might be a space, a tab, or a newline. Hence, calling `scanf` in this way is typically done in a loop; each iteration reads a block of nonspace characters.

For example, recall the `shout` function of Section 3.2. To transform textual data from `stdin` to all capitals, we can write the following program, where line 3 should be replaced by the full text of `shout`:

```
1  #include <stdio.h>
2
3  // Insert shout here.
4
5  int main() {
6    // assume that any word is at most 127 characters
7    char in[128], out[128];
8    while (scanf("%s", in) != EOF) {
9      // read one word, now process it
10     shout(in, out);
11     // print the result
12     printf("%s ", out);
13   }
14   printf("\n");
15   return 0;
16 }
```

Compiling and running the program yields the expected behavior:

```
$ gcc -Wall -Wextra -o shout shout.c
$ ./shout
Let's all use our inside voices.
LET'S ALL USE OUR INSIDE VOICES.
```

To run this program, we type `./shout` and then `Enter` at the command line. The program stops at the call to `scanf` and waits for input. When we type, `Let's all use our inside voices.` followed by `Enter`, the data are passed through `stdin` to the program via `stdin`. Then the text is handled in chunks, one chunk per iteration: `Let's`, `all`, `use`, `our`, `inside`, `voices.` Finally, we press `Control-D` to close `stdin`, which causes `scanf` to return `EOF` and the loop to exit.

Unfortunately, the program is vulnerable: if the user ever types a word with more than 127 characters, we risk a memory corruption at line 8 since `scanf` is unaware of the size of `buf` from the way we called it. One of many solutions is to read single characters. Calling `scanf("%c", &x)`, where `x` is of type `char`, reads a single character from `stdin`. Since the `shout` program only needs to examine a character at a time to achieve its objective, we can implement a safer variant as follows:

```
#include <stdio.h>

int main() {
  char c;
  while (scanf("%c", &c) != EOF) {
    if ('a' <= c && c <= 'z')
      c += 'A' - 'a';
    printf("%c", c);
  }
  printf("\n");
  return 0;
}
```

Exercise 5.6. Write a program according to the following specification:

(1) It has two possible command-line arguments: `-s` indicates "shout" mode, while `-w` indicates "whisper" mode.
(2) It reads an arbitrary list of strings from `stdin`.
(3) It writes the strings to `stdout`, except that it writes all letters in either uppercase or lowercase according to the mode.

□

5.3 Working with Files

While Unix redirection allows reading from a file via `stdin` and writing to a file via `stdout`, it is sometimes appropriate to access files directly. `stdio.h` defines functions for opening and closing files. Assuming that `filename` is a C string holding the name of a file,

- `FILE * inf = fopen(filename, "r")` opens the file for reading;
- `FILE * outf = fopen(filename, "w")` creates the file (and discards an existing one of the same name if necessary) for writing;
- and `FILE * outf = fopen(filename, "a")` opens the file for appending.

The `FILE *` variables `inf` and `outf` are referred to as **file pointers**. Then `fprintf` can be used to write to `outf`, and `fscanf` can be used to read from `inf`. In both cases, the first argument is the file pointer. When reading or writing is complete, `fclose(inf)` (`fclose(outf)`) closes the file.

5.4 Further Adventures with I/O

Exercise 5.7. Numerical simulation is one valuable application of programming. In this exercise, we explore binomially distributed random events.

Consider tossing an unbiased coin n times. What is the probability that k, $0 \le k \le n$, of the tosses turn up heads? One could of course compute this probability analytically. However, in simulations it is common to sample events from a distribution. The function `rand()`, declared in `stdlib.h`, provides random sampling of a uniform distribution: it returns an integer between 0 and `RAND_MAX`, a constant also declared in `stdlib.h`, such that each integer has an equal probability of occurring. We will use this function to simulate sampling from binomial distributions (with parameter n varying and parameter $p = 0.5$, for those with background in probability, which describe n tosses of an unbiased coin).

The main idea is that a sequence of n tosses can be simulated by summing the results of evaluating `rand() % 2` n times. Recall that `m % 2` is 0 or 1, for any integer `m`; `%` is pronounced "modulo." Each evaluation of `rand() % 2` returns 0 (tails) or 1 (heads). Summing the result of n evaluations thus yields

an integer between 0 and n. This process is implemented in the function `toss` below.

Simulating one event is not informative about the distribution. We implement a function, `performExperiment`, to run the experiment many times and record the data. Finally, `printDistribution` uses `printf` creatively to visualize the results, and `main` orchestrates the whole ensemble.

```
#include <stdio.h>
#include <stdlib.h>

/* Simulates tossing an unbiased coin n times.  Returns
 * the number of heads.
 */
int toss(int n) {
  int nHeads = 0, i;
  for (i = 0; i < n; ++i)
    // rand() % 2 yields 0 or 1 with uniform probability
    nHeads += rand() % 2;
  return nHeads;
}

/* Perform nTrials of an nTosses coin-tossing experiment
 * and store the results in nOccur.
 */
void performExperiment(int * nOccur, int nTosses,
                       int nTrials)
{
  if (!nOccur) return;
  int i;
  // Initialize nOccur.
  for (i = 0; i <= nTosses; ++i)
    nOccur[i] = 0;
  // Perform nTrials of the experiment.
  for (i = 0; i < nTrials; ++i)
    // 1. toss(nTosses) returns the outcome of one trail.
    // 2. Increment the count for that outcome.
    nOccur[toss(nTosses)]++;
}

/* Given an array of occurrence data of size sz
 * representing the results of nTrials of an experiment,
 * each instance of which yields an integer in the range
 * [0, sz), prints a distribution labeled with the outcomes
 * and the percentages (as an int) of trials that yielded
 * those outcomes.
 */
void printDistribution(int * nOccur, int sz, int nTrials) {
  if (!nOccur) return;
  int i, j;
```

```
  for (i = 0; i < sz; ++i) {
    int percent = (100 * nOccur[i]) / nTrials;
    printf("%2d %2d ", i, percent);
    for (j = 0; j < percent; ++j)
      printf("*");
    printf("\n");
  }
}

/* Prints usage. */
void printUsage() {
  printf("Usage: binomial [# tosses] [# trials]\n");
}

/* Graphs a distribution for nTrials of a coin-tossing
 * experiment, where each trial consists of nTosses coin
 * tosses, and the number of heads is counted.  Explores
 * the binomial distribution.
 */
int main(int argc, char ** argv) {
  // Zeroth argument: name of the executable
  // First argument: nTosses
  // Second argument: nTrials
  // If different, print usage and quit.
  if (argc != 3) {
    printUsage();
    return 0;
  }

  // Obtain the input.  Protect against malformed input.
  int nTosses, nTrials, numRead;
  numRead = sscanf(argv[1], "%d", &nTosses);
  if (numRead != 1 || nTosses <= 0) {
    printUsage();
    return 0;
  }
  numRead = sscanf(argv[2], "%d", &nTrials);
  if (numRead != 1 || nTrials <= 0) {
    printUsage();
    return 0;
  }

  // Set up the occurrence array, which maps the number of
  // heads out of nTosses to the number of trials that had
  // precisely that number of heads.
  int nOccur[nTosses+1];

  // Perform the experiment.
  performExperiment(nOccur, nTosses, nTrials);
```



```

  // Visualize the result of the experiment.
  printDistribution(nOccur, nTosses+1, nTrials);

  return 0;
}
```

Compiling and running the program with various command-line arguments yields about what one would expect of binomially distributed data. Notice that the distribution becomes more ideal as the number of trials increases.

```
$ gcc -Wall -Wextra -o binomial binomial.c
$ ./binomial
Usage: binomial [# tosses] [# trials]
$ ./binomial 20 1000
 0  0
 1  0
 2  0
 3  0
 4  0
 5  1 *
 6  4 ****
 7  7 *******
 8 12 ************
 9 16 ****************
10 16 ****************
11 18 ******************
12 11 ***********
13  6 ******
14  2 **
15  1 *
16  0
17  0
18  0
19  0
20  0
$ ./binomial 20 10000
 0  0
 1  0
 2  0
 3  0
 4  0
 5  1 *
 6  3 ***
 7  7 *******
 8 12 ************
```

```
 9 16 ****************
10 17 *****************
11 15 ***************
12 12 ************
13  7 *******
14  3 ***
15  1 *
16  0
17  0
18  0
19  0
20  0
```

□

Exercise 5.8. Write a program according to the following specification:

(1) It reads an arbitrary list of strings from `stdin`.
(2) It records the number of vowels encountered.
(3) It prints the number of occurrences of each vowel to `stdout`.

□

Exercise 5.9. Write a program, called `hide`, according to the following specification:

(1) It has two possible arguments: `-encrypt` indicates encryption mode, while `-decrypt` indicates decryption mode.
(2) It reads an arbitrary list of strings from `stdin`.
(3) It applies a cypher to the strings. You may invent your own, but a simple one is to shift the letters by a constant amount (for example, `'a'` becomes `'d'`, and `'z'` becomes `'c'`). It either encrypts or decrypts the strings ("shifts" or "deshifts" the letters) depending on the mode.
(4) It prints the encrypted or decrypted text to `stdout`.

At minimum, it should be able to handle text consisting only of lowercase letters. For example, if the message

```
attention home planet stop prepare invasion stop earth is
ripe for the taking stop cu soon full stop
```

is in file `msg.txt`, then

```
$ ./hide -encrypt < msg.txt > msge.txt
```

would produce the following cyphertext in file `msge.txt` if `hide` is using a shift of 12:

```
mffqzfuaz tayq bxmzqf efab bdqbmdq uzhmeuaz efab qmdft ue
dubq rad ftq fmwuzs efab og eaaz rgxx efab
```


Then

```
$ ./hide -decrypt < msge.txt
```

would yield the original message. Using Unix piping would also result in the output of the original message:

```
$ ./hide -encrypt < msg.txt | ./hide -decrypt
```

To achieve the proper shift, use the following formula:

```
'a' + (((c - 'a') + sh) % 26)
```

The idea is to find `c`'s position in the alphabet (`c - 'a'`), add the shift (`(c - 'a') + sh`) modulo 26 (`((c - 'a') + sh) % 26`), and finally translate the character back into the ASCII range for lowercase letters.

To unshift, set `sh` to `26 - sh` and use the same formula. For example, if `sh` is 12, then

```
'a' + (((c - 'a') + 12) % 26)
```

yields `'q'` if `c == 'e'` since `'q'` is 12 characters later than `'e'`; and `'f'` if `c == 't'` since `'f'` is 12 characters later than `'t'` modulo 26. □

Exercise 5.10. Write a program that reads strings from stdin and computes the integer mean of their lengths. □

Exercise 5.11. Write a program to determine word-length frequencies in a text file read through stdin. Reserve one category for all words of length 32 or greater. Output the frequencies in a useful way, which could include rendering a chart as in Exercise 5.7. □

6

Memory: The Heap

In any complex system—whether engineered or natural—products are constructed, distributed, used, and finally recycled. The lifetimes of such products are typically independent of the manufacturing process. Data structures are the primary (intermediate) products of complex programs, and data structures require memory. However, the memory that we have used so far—the stack—is not well suited for creating data structures whose existence is independent of functions' execution periods. Stack frames form the stack, and the lifetimes of stack frames, by definition, correspond to function execution periods: a frame is pushed on the stack at the beginning of a function's execution and popped from the stack at the end of the function's execution; any data structure that resides in the stack frame is then lost.

This chapter introduces a sector of program memory, called the **heap**, specifically intended for producing data structures whose lifetimes are independent of the execution periods of the functions that create and manipulate them. **Dynamic memory allocation** is the process of obtaining segments of memory from the heap for use.

As a motivating application, this chapter focuses on implementing a **library** of functions for creating, manipulating, and disposing of matrices and vectors. Along the way, we will cover basic data types for representing real numbers and a mechanism for defining new data types. In Chapter 7, we will refine the matrix library into an **abstract data type**.

In Chapters 9–11, we will *use* Matlab's basic matrix data structure to accomplish computational tasks. How it works will be irrelevant; only how to use it will matter. In contrast, this and the next two chapters provide the complementary perspective, that of the library *designer* and *implementer*. In this context, we care how the library is intended to be used, which motivates the choice of functions that are offered, which in turn dictates what functions we must implement.

6.1 Review of Matrices

An $m \times n$ **matrix** M consists of m rows of n real numbers, while an m-dimensional **vector** is a $m \times 1$ matrix. For concreteness, consider the following matrices and vectors:

$$A = \begin{bmatrix} 1 & .25 & -.1 \\ 0 & .4 & .3 \\ 0 & .1 & -.3 \end{bmatrix} \;,\quad B = \begin{bmatrix} .5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \;,\quad c = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \;,\quad d = \begin{bmatrix} 0 \\ .5 \\ .5 \end{bmatrix} \;.$$

We refer to elements of each using indexing: $A_{3,1}$ refers to the bottom-left element 0 of matrix A, while c_1 refers to the top element 1 of c. Notice that rows and columns of a matrix are numbered starting at 1 rather than at 0, as in arrays. While this difference is annoying and causes minor complications in the matrix library, it is in keeping with standard practice; for example, matrices in Matlab are indexed starting from 1.

The **transpose** of a matrix essentially swaps indices: if A' is the transpose of A, then element $A_{i,j}$ corresponds to $A'_{j,i}$. For example,

$$A' = \begin{bmatrix} 1 & 0 & 0 \\ .25 & .4 & .1 \\ -.1 & .3 & -.3 \end{bmatrix} \qquad c' = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \;.$$

Matrix addition requires two matrices of equal **dimensions**—that is, they must have the same number of rows and columns. Matrices are summed element-wise: if $S = A + B$, then $S_{i,j} = A_{i,j} + B_{i,j}$. For example,

$$\begin{bmatrix} 1 & .25 & -.1 \\ 0 & .4 & .3 \\ 0 & .1 & -.3 \end{bmatrix} + \begin{bmatrix} .5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1.5 & 25 & -.1 \\ 0 & 2.4 & .3 \\ 0 & .1 & .7 \end{bmatrix} \;,$$

and

$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ .5 \\ .5 \end{bmatrix} = \begin{bmatrix} 1 \\ .5 \\ 1.5 \end{bmatrix} \;.$$

Matrix multiplication is more complicated. To form the product $P = AB$, the number of columns of A must match the number of rows of B: if A has dimensions $\ell \times m$, and B has dimensions $m \times n$, then the product AB has dimensions $\ell \times n$. Furthermore, element $P_{i,j}$ is defined as follows:

$$P_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j} \;. \qquad (6.1)$$

For example, to compute the top-left element of the product of the matrices A and B above, compute

$$A_{1,1}B_{1,1} + A_{1,2}B_{2,1} + A_{1,3}B_{3,1} = 1 \cdot .5 + .25 \cdot 0 + -.1 \cdot 0 = .5 \;.$$

The **dot product** of m-dimensional vectors c and d is computed as the matrix product $c'd$. For example,

$$c'd = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ .5 \\ .5 \end{bmatrix} = 1 \cdot 0 + 0 \cdot .5 + 1 \cdot .5 = .5 \ .$$

6.2 Matrix: A Specification

Our goal is to implement functions to create matrices, to read and write their values, to obtain their dimensions, to print them, to compute their transposes, and to calculate their sums and products. As the matrices should hold real values, we use the basic type `double` to represent elements. A `double` value is a double-word (that is, 8-byte) IEEE floating point value, which represents real numbers with high precision. We will define a type called `matrix`, and with this type we will implement the following **application programming interface** (**API**), which defines operations for manipulating matrices. In general, an API is an interface to a possibly complex set of functions and data types; it hides the complexity behind (ideally) straightforward function and data type specifications.

```
/* Creates a ``rows by cols'' matrix with all values 0.
 * Returns NULL if rows <= 0 or cols <= 0 and otherwise a
 * pointer to the new matrix.
 */
matrix * newMatrix(int rows, int cols);

/* Copies a matrix.  Returns NULL if mtx is NULL.
 */
matrix * copyMatrix(matrix * mtx);

/* Deletes a matrix.  Returns 0 if successful and -1 if mtx
 * is NULL.
 */
int deleteMatrix(matrix * mtx);

/* Sets the (row, col) element of mtx to val.  Returns 0 if
 * successful, -1 if mtx is NULL, and -2 if row or col are
 * outside of the dimensions of mtx.
 */
int setElement(matrix * mtx, int row, int col, double val);

/* Sets the reference val to the value of the (row, col)
 * element of mtx.  Returns 0 if successful, -1 if either
 * mtx or val is NULL, and -2 if row or col are outside of
 * the dimensions of mtx.
 */
int getElement(matrix * mtx, int row, int col,
               double * val);

```



```
/* Sets the reference n to the number of rows of mtx.
 * Returns 0 if successful and -1 if mtx or n is NULL.
 */
int nRows(matrix * mtx, int * n);

/* Sets the reference n to the number of columns of mtx.
 * Returns 0 if successful and -1 if mtx is NULL.
 */
int nCols(matrix * mtx, int * n);

/* Prints the matrix to stdout.  Returns 0 if successful
 * and -1 if mtx is NULL.
 */
int printMatrix(matrix * mtx);

/* Writes the transpose of matrix in into matrix out.
 * Returns 0 if successful, -1 if either in or out is NULL,
 * and -2 if the dimensions of in and out are incompatible.
 */
int transpose(matrix * in, matrix * out);

/* Writes the sum of matrices mtx1 and mtx2 into matrix
 * sum. Returns 0 if successful, -1 if any of the matrices
 * are NULL, and -2 if the dimensions of the matrices are
 * incompatible.
 */
int sum(matrix * mtx1, matrix * mtx2, matrix * sum);

/* Writes the product of matrices mtx1 and mtx2 into matrix
 * prod.  Returns 0 if successful, -1 if any of the
 * matrices are NULL, and -2 if the dimensions of the
 * matrices are incompatible.
 */
int product(matrix * mtx1, matrix * mtx2, matrix * prod);

/* Writes the dot product of vectors v1 and v2 into
 * reference prod.  Returns 0 if successful, -1 if any of
 * v1, v2, or prod are NULL, -2 if either matrix is not a
 * vector, and -3 if the vectors are of incompatible
 * dimensions.
 */
int dotProduct(matrix * v1, matrix * v2, double * prod);
```

Just as we write unit tests of individual functions, we must write unit tests of libraries. Here is a unit test of this specification:

```
int main() {
  matrix * A, * Ac, * B, * c, * d, * M, * ct, * mdp;
  double dp;
```

```

  A = newMatrix(3, 3);
  setElement(A, 1, 1, 1.0);
  setElement(A, 1, 2, .25);
  setElement(A, 1, 3, -.1);
  setElement(A, 2, 2, .4);
  setElement(A, 2, 3, .3);
  setElement(A, 3, 2, .1);
  setElement(A, 3, 3, -.3);
  printf("Matrix A:\n");
  printMatrix(A);

  Ac = copyMatrix(A);
  printf("\nCopy of A:\n");
  printMatrix(Ac);

  B = newMatrix(3, 3);
  setElement(B, 1, 1, .5);
  setElement(B, 2, 2, 2.0);
  setElement(B, 3, 3, 1.0);
  printf("\nMatrix B:\n");
  printMatrix(B);

  c = newMatrix(3, 1);
  setElement(c, 1, 1, 1.0);
  setElement(c, 3, 1, 1.0);
  printf("\nVector c:\n");
  printMatrix(c);

  d = newMatrix(3, 1);
  setElement(d, 2, 1, 1.0);
  setElement(d, 3, 1, 1.0);
  printf("\nVector d:\n");
  printMatrix(d);

  M = newMatrix(3, 3);
  transpose(A, M);
  printf("\nA':\n");
  printMatrix(M);

  ct = newMatrix(1, 3);
  transpose(c, ct);
  printf("\nc':\n");
  printMatrix(ct);

  sum(A, B, M);
  printf("\nA + B:\n");
  printMatrix(M);

  product(A, B, M);
  printf("\nA * B:\n");
  printMatrix(M);

  mdp = newMatrix(1, 1);
  product(ct, d, mdp);
  getElement(mdp, 1, 1, &dp);
  printf("\nDot product (1): %.2f\n", dp);

  dotProduct(c, d, &dp);
  printf("\nDot product (2): %.2f\n", dp);

  product(A, c, d);
  printf("\nA * c:\n");
  printMatrix(d);

  deleteMatrix(A);
  deleteMatrix(Ac);
  deleteMatrix(B);
  deleteMatrix(c);
  deleteMatrix(d);
  deleteMatrix(M);
  deleteMatrix(ct);
  deleteMatrix(mdp);

  return 0;
}
```

This unit test not only shows that the library, as designed, offers the necessary functionality to perform basic matrix arithmetic but also will become the first test of the eventual implementation of the API. In general, a unit test is a test program that exercises the functionality of a programming unit independent of the rest of the program—whether that unit be a function, a library, or a set of related libraries. Unit tests usually encode a set of usage scenarios; hence, writing a unit test can sometimes reveal deficiencies in an API's design. In large engineering efforts involving software development, the developers typically create their own unit tests while the product analysts design and execute **system tests** that test many modules at once. Subgroups may also design and execute **integration tests** to exercise several modules together.

Once we implement the specification, we should get the following output for this unit test:

```
Matrix A:
  1.00   0.25  -0.10
  0.00   0.40   0.30
  0.00   0.10  -0.30

Copy of A:
```

```
  1.00   0.25  -0.10
  0.00   0.40   0.30
  0.00   0.10  -0.30

Matrix B:
  0.50   0.00   0.00
  0.00   2.00   0.00
  0.00   0.00   1.00

Vector c:
  1.00
  0.00
  1.00

Vector d:
  0.00
  1.00
  1.00

A':
  1.00   0.00   0.00
  0.25   0.40   0.10
 -0.10   0.30  -0.30

c':
  1.00   0.00   1.00

A + B:
  1.50   0.25  -0.10
  0.00   2.40   0.30
  0.00   0.10   0.70

A * B:
  0.50   0.50  -0.10
  0.00   0.80   0.30
  0.00   0.20  -0.30

Dot product (1): 1.00

Dot product (2): 1.00

A * c:
  0.90
  0.30
 -0.30
```


6.3 Matrix: An Implementation

6.3.1 Defining the Data Structure

The primary attributes of a matrix are the number of rows, the number of columns, and the elements themselves. These attributes can be organized into a single new datatype called `matrix` as follows:

```
typedef struct {
  int rows;
  int cols;
  double * data;
} matrix;
```

A C `struct`, short for "structure," is the primary mechanism for creating complex data structures. This declaration is actually a C idiom; the long version is as follows:

```
1 struct _matrix {
2   int rows;
3   int cols;
4   double * data;
5 };
6
7 typedef struct _matrix matrix;
```

Lines 1–5 declare the type `struct _matrix`. Then in line 7, the `typedef` statement, read as "define `matrix` as short for `struct _matrix`," provides the simpler name `matrix` for the type `struct _matrix`.

The first two **fields** of `matrix` are self-evident; the third is a `double *` because it is intended to be an array of `double`s. However, we will use dynamic memory allocation to obtain the actual memory for the array.

To access the fields of an instance of the `matrix` structure, we use the `.` (**dot**) operator:

```
{
  matrix mtx;
  mtx.rows = 3;
  mtx.cols = 3;
}
```

Structures are laid out as one block of memory, so accessing a field is compiled into a constant memory offset from the beginning of a structure's block. For example, a `matrix` structure has the following layout:

`double *`	`data`		8
`int`	`cols`		4
`int`	`rows`		0

In the code snippet above, the stack looks as follows at the beginning of execution:

`double *`	`mtx.data`	$\otimes$	1008
`int`	`mtx.cols`	$\otimes$	1004
`int`	`mtx.rows`	$\otimes$	1000
`void *`	`pc`	header	996
`int`	`rv`	$\otimes$	992

Therefore, `mtx.cols` refers to the memory cell at address 1004, an offset of four bytes from the beginning of the `matrix` structure.

Another method of accessing a structure's fields is via a pointer to the structure:

```
{
  matrix m;
  matrix * mtx = &m;  // mtx points to a matrix structure
  mtx->rows = 3;
  mtx->cols = 3;
}
```

The operator `->` (**arrow**) is convenient but unnecessary: `mtx->cols` is equivalent to `(*mtx).cols`—a dereference of the pointer `mtx` followed by an access of the field `cols`. The `int` memory cell that is four bytes offset from the address stored in `mtx` is accessed in this case.

The type declaration describes what a `matrix` looks like, while the functions `newMatrix` and `deleteMatrix` actually create and destroy instances of `matrix`. **Dynamic memory allocation** is required:

```
  matrix * m = (matrix *) malloc(sizeof(matrix));
```

The standard library, `stdlib.h`, defines `malloc`, which is actually a call to the operating system. It returns a generic pointer (of type `void *`) to a segment of memory containing the number of bytes specified as the argument. Here we use `sizeof(matrix)` to specify the number of bytes. The compiler replaces `sizeof(matrix)` with the actual number of bytes, which in this case is `sizeof(int) + sizeof(int) + sizeof(double *)`, or $4+4+4 = 12$ bytes. The final peculiar notation of this allocation is the **typecast**, `(matrix *)`, preceding `malloc`. It casts the `void *` type generically returned by `malloc` to the specific type `matrix *`, which matches the type of `m`; hence, the typecast allows the left and right sides of the assignment to have the same type, as desired.

The allocated memory is located not on the stack but in the heap. This memory remains allocated for arbitrarily long after `newMatrix` returns—until, in fact, `free(m)` is called. The function `free` is also defined in `stdlib.h`. Every call to `malloc` should correspond to precisely one call to `free`. A bug in which dynamically allocated memory is never freed is referred to as a **memory leak**. Long-running programs with memory leaks can eventually crash or—worse—compromise the performance of the entire system. Another type of bug, a **double-free** bug, is when `free` is called twice on the same allocated memory;

it may or may not crash the system. Both types of bugs can be detected with `valgrind`, which we discuss later.

Let's take a look at the implementation of `newMatrix`:

```
1  matrix * newMatrix(int rows, int cols) {
2    if (rows <= 0 || cols <= 0) return NULL;
3
4    // allocate a matrix structure
5    matrix * m = (matrix *) malloc(sizeof(matrix));
6
7    // set dimensions
8    m->rows = rows;
9    m->cols = cols;
10
11   // allocate a double array of length rows * cols
12   m->data = (double *) malloc(rows*cols*sizeof(double));
13   // set all data to 0
14   int i;
15   for (i = 0; i < rows*cols; i++)
16     m->data[i] = 0.0;
17
18   return m;
19 }
```

Line 5 allocates the `matrix`, but it does not allocate the `data` field of the `matrix`. This allocation is accomplished at line 12. Lines 8–9 and 15–16 initialize the `matrix` to be the `rows` $\times$ `cols` zero matrix.

To make it absolutely clear that heap memory is separate from stack memory, let's visualize `newMatrix`'s stack frame:

`int`	`i`		20
`matrix *`	`m`		16
`void *`	`pc`		12
`matrix *`	`rv`		8
`int`	`cols`		4
`int`	`rows`		0

Every memory cell of the stack frame occupies one word. When the assignment at line 5 occurs, `m` is set to the allocated address.

How can we use an array to represent a matrix? In other words, how do we map a matrix's two-dimensional existence onto one-dimensional memory? We have to be clever. The idea is to decide on a policy for laying out the elements of the matrix. One policy—the one that we adopt—is to concatenate the columns of the matrix into one long list.[1] For example, matrix A from above is represented as a sequence of nine `double`s:

```
1.00 0.00 0.00 0.25 0.40 0.10 -0.10 0.30 -0.30
```

[1] This policy is referred to as **column major**, which is a standard policy for dense matrix representations.

The middle element, $A_{2,2}$, is at index 4 in the flat representation. Notice that we must translate between two indexing standards: C arrays are indexed starting at 0 since indices represent explicit memory offsets, while mathematical matrices are indexed starting at 1. In general, we access the element at row `row` and column `col` of matrix `mtx` as follows:

```
mtx->data[(col - 1) * mtx->rows + (row - 1)]
```

In the case of A, element $A_{2,2}$ corresponds to index

$$(2-1)\cdot 3 + (2-1) = 4\ .$$

Verify that this formula correctly maps the elements of A to their positions in the flat representation above.

To isolate this policy decision to one place, we encode it as a C **macro**:

```
#define ELEM(mtx, row, col) \
  mtx->data[(col-1) * mtx->rows + (row-1)]
```

A macro is expanded during compilation and is thus an efficient means of gaining modularity without losing efficiency. For example,

```
  ELEM(mtx1, row, k) = 0.0;
```

expands to

```
  mtx1->data[(k-1) * mtx1->rows + (row-1)] = 0.0;
```

during compilation.

Having decided on a definition of a matrix—both its type `matrix` and the data layout policy—we need to implement the remaining functions that define a matrix to the user. First, `deleteMatrix` provides a way of de-allocating a `matrix`:

```
int deleteMatrix(matrix * mtx) {
  if (!mtx) return -1;
  // free mtx's data
  assert (mtx->data);
  free(mtx->data);
  // free mtx itself
  free(mtx);
  return 0;
}
```

Next, `copyMatrix` creates a separate `matrix` instance that is initialized with the same values as the given `matrix`:


```
1  matrix * copyMatrix(matrix * mtx) {
2    if (!mtx) return NULL;
3
4    // create a new matrix to hold the copy
5    matrix * cp = newMatrix(mtx->rows, mtx->cols);
6
7    // copy mtx's data to cp's data
8    memcpy(cp->data, mtx->data,
9           mtx->rows * mtx->cols * sizeof(double));
10
11   return cp;
12 }
```

The function `memcpy` is defined in the standard string library, `string.h`. (Why `string.h`? Good question.) A call to `memcpy(to, from, nBytes)` copies the `nBytes` of memory starting at address `from` to the `nBytes` of memory starting at address `to`. An alternative to lines 8–9 is the following:

```
  int i;
  for (i = 0; i < mtx->rows * mtx->cols; i++)
    cp->data[i] = mtx->data[i];
```

Yet another alternative is the following:

```
  int row, col;
  for (col = 1; col <= mtx->cols; col++)
    for (row = 1; row <= mtx->rows; row++)
      ELEM(cp, row, col) = ELEM(mtx, row, col);
```

Notice that these two alternatives write values to `cp->data` in exactly the same order because of the column major layout. It is likely that the implementation based on `memcpy` is the most efficient, followed by the first alternative. The final method requires multiplication (see the definition of `ELEM`).

The next four functions provide access to a `matrix`'s dimensions and elements:

```
int setElement(matrix * mtx, int row, int col, double val)
{
  if (!mtx) return -1;
  assert (mtx->data);
  if (row <= 0 || row > mtx->rows ||
      col <= 0 || col > mtx->cols)
    return -2;

  ELEM(mtx, row, col) = val;
  return 0;
}

int getElement(matrix * mtx, int row, int col,
               double * val) {
  if (!mtx || !val) return -1;
  assert (mtx->data);
  if (row <= 0 || row > mtx->rows ||
      col <= 0 || col > mtx->cols)
    return -2;
```

```

  *val = ELEM(mtx, row, col);
  return 0;
}

int nRows(matrix * mtx, int * n) {
  if (!mtx || !n) return -1;
  *n = mtx->rows;
  return 0;
}

int nCols(matrix * mtx, int * n) {
  if (!mtx || !n) return -1;
  *n = mtx->rows;
  return 0;
}
```

The majority of the implementation of each function focuses on protecting against bad input, which is appropriate since these are interface functions—that is, functions that can be called by a less-than-informed user.

Exercise 6.1. Heap-allocated memory allows a program to store an unbounded amount of data. Implement a program that reads and stores a given number, `n`, of strings from `stdin`. To show that the text was indeed saved, make it print the strings just before freeing all allocated memory and exiting. As usual, it is reasonable to assume that the longest word has fewer than 128 characters.

Solution. The main data structure is a `char *` array, `strings`, with `n` elements, each of which points to a `char` array that holds a string. The program iteratively reads a string into a temporary buffer, `buf`, of size 128; allocates a new `char` array according to the length of the string, using `strlen` from `string.h` (see also Exercise 3.17); and copies the string from `buf` to the newly allocated array using `strcpy` from `string.h` (see also Exercise 3.19).

One essential and often missed detail is that the allocated character array must have one more byte than the length of the string, as returned by `strlen`, to hold the string terminator.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char ** argv) {
6   // 1. Obtain number of strings.
7   if (argc != 2) {
8     printf("Expected one integer argument.\n");
9     return -1;
10  }
11
```



```
12  int n;
13  if (sscanf(argv[1], "%d", &n) != 1 || n <= 0) {
14    printf("Expected a positive integer.\n");
15    return -1;
16  }
17
18  // 2. Read n strings.
19  // Array of char *'s to hold strings.
20  char ** strings = (char **) malloc(n * sizeof(char *));
21  // Temporary buffer.
22  char buf[128];
23
24  int i;
25  for (i = 0; i < n; ++i) {
26    // 2a. Scan for each string.
27    if (scanf("%127s", buf) == EOF) {
28      printf("Unexpected end of input.\n");
29      return -1;
30    }
31    // 2b. Allocate space to hold string permanently.
32    // Notice the extra byte to hold the string terminator.
33    strings[i] = (char *) malloc(strlen(buf) + 1);
34    // 2c. Copy string from buffer to its space.
35    strcpy(strings[i], buf);
36  }
37
38  // 3. Do something with strings.  In this case, print.
39  for (i = 0; i < n; ++i)
40    printf("%s\n", strings[i]);
41
42  // 4. Free allocated memory.
43  for (i = 0; i < n; ++i)
44    free(strings[i]);
45  free(strings);
46
47  return 0;
48 }
```

Notice at line 27 the format string `"%127s"`. It tells `scanf` to read at most 127 characters, even if the string is longer. The remaining characters are read subsequently. This format string makes the assumption at line 22 safe. ☐

Exercise 6.2. The `realloc` function in `stdlib` allows growing a region of memory. Suppose that `strings` is a `char **` variable pointing to an array of `n` strings. The statement

```
  strings = realloc(strings, 2*n);
```

reallocates the array to be twice its original size while preserving the existing data, although its base address may be changed. Use `realloc` to implement a version of the program of Exercise 6.1 that does not take an argument specifying the number of strings, instead storing as many strings as are given.

Solution. A standard strategy is to allocate an initial array, here called `strings`, of a default size, `n`, and then to double the size of the array—and `n`—each time more space is required. This strategy guarantees that less than twice as much memory as required is used and that a number of reallocations only logarithmic in the amount of data are executed. For example, if `n` is initially 1, and 600 strings are read, `strings` will have sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and finally 1,024 during execution.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  // Allocate initial array of char *'s to hold strings.
  int n = 1;  // size of array
  char ** strings = (char **) malloc(n * sizeof(char *));

  int nstrings = 0;  // number of strings read
  char buf[128];
  while (scanf("%127s", buf) != EOF) {
    // Is there space in strings for another string?
    if (nstrings == n) {
      // No, so double size of strings.
      n *= 2;
      strings = realloc(strings, n * sizeof(char *));
    }
    // Allocate space to hold string permanently.
    strings[nstrings] = (char *) malloc(strlen(buf) + 1);
    // Copy string from buffer to its space.
    strcpy(strings[nstrings], buf);
    // Increment the number of strings read.
    ++nstrings;
  }

  // Do something with strings.  In this case, print.
  int i;
  for (i = 0; i < nstrings; ++i)
    printf("%s\n", strings[i]);

  // Free allocated memory.
  for (i = 0; i < nstrings; ++i)
    free(strings[i]);
  free(strings);

  return 0;
}
```


6.3.2 Manipulating the Data Structure

Having defined the functionality to create, copy, destroy, read from, and write to matrices, we can now implement higher level functionality. We use the `ELEM` macro to access the elements of **`matrix`**es so that the memory layout policy impacts as little of the code as possible. If we were to change the policy at some point, we would not have to change the following code.

Our first function is `printMatrix`. Here we employ some of the sophisticated formatting features of `printf`. Perhaps the one new programming idea here is the double loop in lines 5–15. The outer loop iterates over the rows, while the inner loop iterates over the columns of each row:

```
1  int printMatrix(matrix * mtx) {
2    if (!mtx) return -1;
3
4    int row, col;
5    for (row = 1; row <= mtx->rows; row++) {
6      for (col = 1; col <= mtx->cols; col++) {
7        // Print the floating-point element with
8        //  - either a - if negative or a space if positive
9        //  - at least 3 spaces before the .
10       //  - precision to the hundredths place
11       printf("% 6.2f ", ELEM(mtx, row, col));
12     }
13     // separate rows by newlines
14     printf("\n");
15   }
16   return 0;
17 }
```

The output at the end of Section 6.2 provides many examples of this function in action.

The functions `transpose` and `sum` are fairly straightforward implementations of definitions (see Section 6.1):

```
int transpose(matrix * in, matrix * out) {
  if (!in || !out) return -1;
  if (in->rows != out->cols || in->cols != out->rows)
    return -2;

  int row, col;
  for (row = 1; row <= in->rows; row++)
    for (col = 1; col <= in->cols; col++)
      ELEM(out, col, row) = ELEM(in, row, col);
  return 0;
}

int sum(matrix * mtx1, matrix * mtx2, matrix * sum) {
  if (!mtx1 || !mtx2 || !sum) return -1;
```

```
  if (mtx1->rows != mtx2->rows ||
      mtx1->rows != sum->rows ||
      mtx1->cols != mtx2->cols ||
      mtx1->cols != sum->cols)
    return -2;

  int row, col;
  for (col = 1; col <= mtx1->cols; col++)
    for (row = 1; row <= mtx1->rows; row++)
      ELEM(sum, row, col) =
        ELEM(mtx1, row, col) + ELEM(mtx2, row, col);
  return 0;
}
```

In contrast, the implementation of `product` is not exactly straightforward, although it does follow from the standard definition (see Section 6.1):

```
1  int product(matrix * mtx1, matrix * mtx2, matrix * prod) {
2    if (!mtx1 || !mtx2 || !prod) return -1;
3    if (mtx1->cols != mtx2->rows ||
4        mtx1->rows != prod->rows ||
5        mtx2->cols != prod->cols)
6      return -2;
7
8    int row, col, k;
9    for (col = 1; col <= mtx2->cols; col++)
10     for (row = 1; row <= mtx1->rows; row++) {
11       double val = 0.0;
12       for (k = 1; k <= mtx1->cols; k++)
13         val += ELEM(mtx1, row, k) * ELEM(mtx2, k, col);
14       ELEM(prod, row, col) = val;
15     }
16   return 0;
17 }
```

Find the correspondence between the code and Equation (6.1). Trace through the execution of this function for line 53 of the unit test, which computes the product AB.

Notice the triple loop. For an $n \times n$ matrix, n^3 scalar products are computed at line 13. Oddly enough, matrix multiplication can be done faster: in 1969, Strassen surprised the linear algebra world with an algorithm requiring approximately $n^{2.807}$ multiplications; and the Coppersmith–Winograd algorithm, introduced in 1990, theoretically requires about $n^{2.376}$ multiplications although is not practical.

When computing a dot product of two vectors, it is inconvenient to transpose one vector, allocate a 1×1 matrix, compute the product of the transposed vector with the other vector, and then extract the one element from the product matrix, as in lines 44–45, 57–59 of the unit test. The `dotProduct` function computes the dot product of two vectors directly:


```
int dotProduct(matrix * v1, matrix * v2, double * prod) {
  if (!v1 || !v2 || !prod) return -1;
  if (v1->cols != 1 || v2->cols != 1) return -2;
  if (v1->rows != v2->rows) return -3;

  *prod = 0;
  int i;
  for (i = 1; i <= v1->rows; i++)
    *prod += ELEM(v1, i, 1) * ELEM(v2, i, 1);
  return 0;
}
```

Exercise 6.3. Implement a function to return the $n \times n$ identity matrix, which is a square matrix with 1s on its diagonal and 0s everywhere else.

Solution. At least two interfaces are possible. In the first version, the user provides an integer n for the size of the desired identity matrix and receives a new matrix in return:

```
1  matrix * identity(int n) {
2    if (n <= 0) return NULL;
3    matrix * m = newMatrix(n, n);
4    int i;
5    for (i = 1; i <= n; i++)
6      ELEM(m, i, i) = 1.0;
7    return m;
8  }
```

Since `newMatrix` returns an all-0 matrix, lines 5–6 need only set the diagonal elements.

In the second version, the user provides a matrix. If it is square, then it is set to be the identity matrix:

```
int identity(matrix * m) {
  if (!m || m->rows != m->cols) return -1;
  int row, col;
  for (col = 1; col <= m->cols; col++)
    for (row = 1; row <= m->rows; row++)
      if (row == col)
        ELEM(m, row, col) = 1.0;
      else
        ELEM(m, row, col) = 0.0;
  return 0;
}
```

This version allows the user to control when memory is allocated. □

Exercise 6.4. Implement a function that returns whether a given matrix is a diagonal matrix, that is, square and 0 everywhere except possibly on the diagonal.

Solution. The strategy is to check each off-diagonal element in turn—except that if the matrix is not even square, then it can't be diagonal. If ever a nonzero value is encountered, then the function can immediately return 0 (false). If the search concludes without finding a nonzero value, then the function concludes that the matrix is indeed diagonal.

```c
int isSquare(matrix * mtx) {
  return mtx && mtx->rows == mtx->cols;
}

int isDiagonal(matrix * mtx) {
  if (!isSquare(mtx)) return 0;
  int row, col;
  for (col = 1; col <= mtx->cols; col++)
    for (row = 1; row <= mtx->rows; row++)
      // if the element is not on the diagonal and not 0
      if (row != col && ELEM(mtx, row, col) != 0.0)
        // then the matrix is not diagonal
        return 0;
  return 1;
}
```

□

Exercise 6.5. Implement a function that returns whether a given matrix is upper triangular, that is, square and with all 0s below the diagonal.

Solution. We use the `isSquare` function of Exercise 6.4. The strategy is to check the below-diagonal elements; if any is nonzero, then the matrix is not upper triangular.

```c
int isUpperTriangular(matrix * mtx) {
  if (!isSquare(mtx)) return 0;
  int row, col;
  // looks at positions below the diagonal
  for (col = 1; col <= mtx->cols; col++)
    for (row = col+1; row <= mtx->rows; row++)
      if (ELEM(mtx, row, col) != 0.0)
        return 0;
  return 1;
}
```

Notice the initialization of the inner loop. □

Exercise 6.6. Implement a function that returns whether a given matrix is lower triangular, that is, square and with all 0s above the diagonal. □

Exercise 6.7. Implement the following specification:

```c
/* Return 1 if mtx is square and symmetric and 0 otherwise
 * (including if mtx is NULL).
```



```c
 */
int symmetric(matrix * mtx);
```

□

Exercise 6.8. Implement the following specification:

```c
/* Returns column col of mtx as a new vector.  Returns NULL
 * if mtx is NULL or col is inconsistent with mtx's
 * dimensions.
 */
matrix * getColumn(matrix * mtx, int col);

/* Returns row row of mtx as a row vector.  Returns NULL if
 * mtx is NULL or row is inconsistent with mtx's
 * dimensions.
 */
matrix * getRow(matrix * mtx, int row);
```

□

Exercise 6.9. Implement a function that constructs a diagonal matrix from a given vector. For example,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{yields the matrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} .$$

Solution. The following version accepts a vector and a matrix from the user and then, if they are of the correct dimensions, sets the matrix to be diagonal with the vector's elements on the diagonal.

```c
int diagonal(matrix * v, matrix * mtx) {
  if (!v || !mtx ||
      v->cols > 1 || v->rows != mtx->rows ||
      mtx->cols != mtx->rows)
    return -1;
  int row, col;
  for (col = 1; col <= mtx->cols; col++)
    for (row = 1; row <= mtx->rows; row++)
      if (row == col)
        ELEM(mtx, row, col) = ELEM(v, col, 1);
      else
        ELEM(mtx, row, col) = 0.0;
  return 0;
}
```

Implement a version that returns a fresh matrix given a vector:

```c
matrix * diagonal(matrix * v);
```

□

Exercise 6.10. Implement a function that sets a matrix to its product with a scalar:

```
/* Sets each element of mtx to the product of that element
 * with s.  Returns -1 if mtx is NULL and 0 otherwise.
 */
int scalarProduct(double s, matrix * mtx);
```

□

Exercise 6.11. Implement the following specification:

```
/* Writes the pow'th power of square matrix mtx into out.
 * Returns 0 if successful, -1 if mtx or out is NULL, -2
 * if mtx is not square, and -3 if pow < 0.
 */
int power(matrix * mtx, int pow, matrix * out);
```

For $n \times n$ matrix A, A^0 is the identity matrix, and $A^{n+1} = A \times A^n = A^n \times A$. □

Exercise 6.12. Challenge: Implement Gaussian elimination. □

Exercise 6.13. Challenge: Implement a less-than-naive version of `power` by exploiting the binary representation of `pow`. □

Exercise 6.14. Challenge: Sophisticated implementations of dense matrix operations are complicated by a critical memory-access optimization: they exploit the caching behavior of modern architectures. In particular, a memory access causes a block of memory to be transferred from main memory (RAM, for "random access memory") to an on-chip cache, unless the accessed address is already mirrored in the cache because of a prior access to the same or nearby address. As the cache is limited in size, such a transfer typically causes another cached memory segment to be evicted from the cache. Cache-aware code tries to maximize the computational work accomplished for each main-memory transfer, often yielding substantial performance gains over naive code.

Implement a version of `product` that exploits the cache. In particular, notice that line 13 of `product` accesses `mtx2` in a manner that, along with the memory layout policy defined by `ELEM`, plays well with the cache. However, its access pattern for `mtx1` is about as bad as it can get: for large matrices, each arithmetic operation corresponds to one RAM-to-cache transfer.

Additionally, implement the loop counters so that the multiplication in `ELEM` is not required.[2] □

[2] This question was suggested by Andrew Bradley.

6.4 Debugging Programs that Use the Heap

Manipulating heap memory is prone to the same bugs as manipulating stack memory:

- Dereferencing `NULL`
- Dereferencing an uninitialized pointer
- Reading uninitialized memory
- Off-by-one indexing
- Getting hosed by a malformed C string

and some new ones:

- Failing to `free` allocated memory (memory leak)
- Accessing `free`d memory
- Double-`free`ing a pointer
- Exhausting the heap and failing to notice when `malloc` returns `NULL`

This list contains typical bugs. One can be alarmingly creative in "inventing" new categories of bugs.

With so many ways of introducing memory bugs into our code, what are we to do? Fortunately, there is an amazing (and open-source) tool available: `valgrind`. According to the `valgrind` website, it's named for the entrance to "Valhalla," where Norse heroes headed after one heroic act too many.

Let's take a rusty scalpel to lines 69–76 of the `main` function that defines our unit test:

```
69   deleteMatrix(A);
70   deleteMatrix(Ac);
71   //deleteMatrix(B);
72   deleteMatrix(c);
73   deleteMatrix(d);
74   //deleteMatrix(M);
75   deleteMatrix(ct);
76   deleteMatrix(mdp);
```

After compiling, running `valgrind ./a.out` yields the following report:

```
HEAP SUMMARY:
    in use at exit: 176 bytes in 4 blocks
  total heap usage: 16 allocs, 12 frees, 496 bytes allocated

LEAK SUMMARY:
   definitely lost: 32 bytes in 2 blocks
   indirectly lost: 144 bytes in 2 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory
```

```
For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from
4)
```

The good news is that the `ERROR SUMMARY` indicates a clean bill of health. The bad news is that 32 bytes were definitely lost in 2 blocks, and another 144 bytes were indirectly lost in 2 blocks. (For fun, let's do the arithmetic. On my 64-bit machine, each `int` of `matrix` requires 4 bytes, and the `double *` requires 8 bytes, yielding 16 bytes per `matrix` structure. Each `matrix` is 3×3 and each `double` requires 8 bytes, so each `matrix`'s `data` field points to $3 \cdot 3 \cdot 8 = 72$ bytes. It checks out.) Running `valgrind --leak-check=full ./a.out` as recommended yields the following additional information:

```
88 (16 direct, 72 indirect) bytes in 1 blocks are definitely
     lost in loss record 3 of 4
   at 0x4C274A8: malloc (vg_replace_malloc.c:236)
   by 0x40072D: newMatrix (matrix.c:21)
   by 0x4010C1: main (matrix.c:212)

88 (16 direct, 72 indirect) bytes in 1 blocks are definitely
     lost in loss record 4 of 4
   at 0x4C274A8: malloc (vg_replace_malloc.c:236)
   by 0x40072D: newMatrix (matrix.c:21)
   by 0x40120E: main (matrix.c:231)
```

The report is clear in pointing out that two instances of `matrix` were not freed—both reports indicate that `newMatrix` was the source of the lost memory—but we still have to track down which instances they are.

Let's wield the rusty scalpel again:

```
69  deleteMatrix(A);
70  deleteMatrix(Ac);
71  deleteMatrix(B);
72  deleteMatrix(c);
73  deleteMatrix(d);
74  deleteMatrix(M);
75  deleteMatrix(ct);
76  deleteMatrix(mdp);
77  deleteMatrix(A);  // double-free
```

Recompiling and executing the program goes just fine on my system, even with `-O3`. But what does `valgrind` have to say?

```
Invalid read of size 8
   at 0x400857: deleteMatrix (matrix.c:55)
   by 0x4013F6: main (matrix.c:269)
 Address 0x51b0048 is 8 bytes inside a block of size 16
     free'd
   at 0x4C270BD: free (vg_replace_malloc.c:366)
```



```
   by 0x400894: deleteMatrix (matrix.c:58)
   by 0x401396: main (matrix.c:261)

Invalid read of size 8
   at 0x40087D: deleteMatrix (matrix.c:56)
   by 0x4013F6: main (matrix.c:269)
 Address 0x51b0048 is 8 bytes inside a block of size 16
     free'd
   at 0x4C270BD: free (vg_replace_malloc.c:366)
   by 0x400894: deleteMatrix (matrix.c:58)
   by 0x401396: main (matrix.c:261)

Invalid free() / delete / delete[]
   at 0x4C270BD: free (vg_replace_malloc.c:366)
   by 0x400888: deleteMatrix (matrix.c:56)
   by 0x4013F6: main (matrix.c:269)
 Address 0x51b0090 is 0 bytes inside a block of size 72
     free'd
   at 0x4C270BD: free (vg_replace_malloc.c:366)
   by 0x400888: deleteMatrix (matrix.c:56)
   by 0x401396: main (matrix.c:261)

Invalid free() / delete / delete[]
   at 0x4C270BD: free (vg_replace_malloc.c:366)
   by 0x400894: deleteMatrix (matrix.c:58)
   by 0x4013F6: main (matrix.c:269)
 Address 0x51b0040 is 0 bytes inside a block of size 16
     free'd
   at 0x4C270BD: free (vg_replace_malloc.c:366)
   by 0x400894: deleteMatrix (matrix.c:58)
   by 0x401396: main (matrix.c:261)
```

The first two reports indicate that `deleteMatrix` is chomping on memory that has already been `free`d; the latter two reports indicate double-`free`ing. Line numbers don't correspond to the text, but, for example, lines 261 and 269 correspond to lines 69 and 77 above. Perhaps only having a TA leaning over your shoulder pointing directly to the buggy line could possibly make the issue any clearer. The lesson here is that `valgrind` is worth running even when the program seems to run fine.

You may be wondering why we need to `free` memory when the program is about to exit. We technically don't. However, in more complex programs, data that we intentionally—or, rather, lazily—decide not to free can mask valid reports of leaked memory that indicate true bugs.

It's probably unnecessary to provide further evidence of `valgrind`'s capabilities. You'll surely discover them for yourself.

7

Abstract Data Types

The `matrix` type, with its supporting functions, is the first complex data structure that we have encountered. But there are several aspects of the implementation that are unsatisfying. First, the entire implementation, including the unit test, is in one file, yet the type and its functions are clearly intended to be used as a library in larger programs—similar to the way we use, for example, the standard I/O library in many programs. Second, the `struct` defining a `matrix` is visible to anyone, which, if nothing else, is esthetically displeasing. More to the point, it encourages an unmodular style of programming in which any part of a program can access data that are essentially private to the `matrix` module. Furthermore, it prevents the possibility of offering multiple implementations of the same interface, for example, dense or sparse matrix representations and manipulations.

Modern programming languages provide facilities for separating public and private aspects of interfaces. While C does not explicitly provide facilities, there is a way of organizing code that yields this separation. Data types defined in this way are called **abstract data types**, or **ADTs** for short.

The idea of an ADT becomes apparent when one considers built-in types, such as `int`. An `int` variable, up to certain technicalities, holds integer values; it can be manipulated using arithmetic operations. A knowledge of how an `int` value is represented in memory, or how arithmetic on `int`s is implemented, is unnecessary to use `int` data. Indeed, two computer architectures may implement `int` operations in different ways. Now suppose, for example, that one might want to manipulate matrices, coordinates, or complex numbers, none of which are part of C. We must define these types and their corresponding operations. An ADT is a programmatic method of defining new data types in a modular and elegant fashion.

The specification of an ADT resides in a **header file**. It consists of the declaration of the ADT itself and a list of **function prototypes**, also called **function signatures**; each prototype specifies the name, input types, and output type of a function. The **implementation** of an ADT resides in a different file: the memory layout for the type and the implementations of each

function declared in the header file must be provided. Finally, various projects can include the header file, just as we have included `stdio.h`, in order to have access to the new type.

7.1 Revisiting Matrices

Let's put aside the rusty scalpel of Section 6.4 and pull out a freshly sharpened one. The goal is to make minor modifications to the matrix module to convert it into an ADT.

The first step is to create a **header file** called `matrix.h`, whose purpose is to define the public interface for the matrix module:

```
#ifndef _MATRIX_H_
#define _MATRIX_H_

/* The type declaration of the ADT. */
typedef struct _matrix * matrix;

/* Creates a rows x cols matrix with all values 0. */
matrix newMatrix(int rows, int cols);

/* Copies a matrix. */
matrix copyMatrix(matrix mtx);

/* Deletes a matrix. */
void deleteMatrix(matrix mtx);

/* Sets the (row, col) element of mtx to val.  Returns 0 if
 * successful, and -1 if row or col are outside of the
 * dimensions of mtx.
 */
int setElement(matrix mtx, int row, int col, double val);

/* Sets the reference val to value of the (row, col)
 * element of mtx.  Returns 0 if successful, -1 if val is
 * NULL, and -2 if row or col are outside of the dimensions
 * of mtx.
 */
int getElement(matrix mtx, int row, int col, double * val);

/* Returns the number of rows of mtx. */
int nRows(matrix mtx);

/* Returns the number of columns of mtx. */
int nCols(matrix mtx);

/* Prints the matrix to stdout. */
```

```
36 void printMatrix(matrix mtx);
37
38 /* Writes the transpose of matrix in into matrix out.
39  * Returns 0 if successful, and -1 if the dimensions of in
40  * and out are incompatible.
41  */
42 int transpose(matrix in, matrix out);
43
44 /* Writes the sum of matrices mtx1 and mtx2 into matrix sum.
45  * Returns 0 if successful, and -1 if the dimensions of the
46  * matrices are incompatible.
47  */
48 int sum(matrix mtx1, matrix mtx2, matrix sum);
49
50 /* Writes the product of matrices mtx1 and mtx2 into matrix
51  * prod.  Returns 0 if successful, and -1 if the dimensions
52  * of the matrices are incompatible.
53  */
54 int product(matrix mtx1, matrix mtx2, matrix prod);
55
56 /* Writes the dot product of vectors v1 and v2 into
57  * reference prod.  Returns 0 if successful, -1 if prod is
58  * NULL, -2 if either matrix is not a vector, and -3 if
59  * the vectors are of incompatible dimensions.
60  */
61 int dotProduct(matrix v1, matrix v2, double * prod);
62
63 #endif
```

There are several differences between this specification and the original. First, lines 1, 2, and 62 are **C preprocessor** instructions that prevent multiple inclusions of `matrix.h` even if several files include `matrix.h` (through a `#include "matrix.h"` statement). One reads such instructions as follows: if the constant `_MATRIX_H_` is not yet defined (line 1), then define it (line 2) and read everything through line 62; otherwise (if `_MATRIX_H_` is already defined), skip everything through line 62. Preprocessor instructions are executed during compilation and direct the compilation process itself.

Second, line 5 defines the type `matrix` as short for `struct _matrix *`. Subsequently, each `matrix *` of the original specification is converted into simply `matrix`, as the type itself is a pointer. From the user's point of view, the type being defined is simply called `matrix`, and except for a slight leak of information—that a `matrix` is actually a `struct _matrix *`—the user cannot deduce from the file `matrix.h` how a matrix is actually represented in memory. The definition of `struct _matrix` itself will be provided shortly in the implementation file `matrix.c`.

With implementation information hidden, we can design the library to be more convenient to use; in particular, `deleteMatrix`, `nRows`, `nCols`,

`printMatrix`, `transpose`, `sum`, `product`, and `dotProduct` need not be designed in such a way as to alert the user that a matrix argument is `NULL`. Of course, a mischievous user can always find a way to undermine an interface, but a well-meaning user will still be protected.

The file `matrix.c` contains the implementation:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "matrix.h"

struct _matrix {
  int rows;
  int cols;
  double * data;
};

matrix newMatrix(int rows, int cols) {
  // allocate a matrix structure
  matrix m = (matrix) malloc(sizeof(struct _matrix));

  // set dimensions
  m->rows = rows;
  m->cols = cols;

  if (rows > 0 && cols > 0) {
    // allocate a double array of length rows * cols
    m->data = (double *) malloc(rows*cols*sizeof(double));
    // set all data to 0
    int i;
    for (i = 0; i < rows*cols; i++)
      m->data[i] = 0.0;
  }
  else
    m->data = NULL;

  return m;
}

void deleteMatrix(matrix mtx) {
  if (mtx->data) free(mtx->data);
  free(mtx);
}

// ...
```

The implementation continues with minor differences relative to the original implementation. Notice, however, that `newMatrix` always returns a matrix structure, even when the number of specified rows or columns is nonpositive. This implementation ensures that `NULL` is never returned to the user, and so a `NULL` matrix can never be passed as an argument. However, because the `data` field may be set to `NULL`, `deleteMatrix` must take this possibility into account at line 37.

Notice also that the implementation includes `matrix.h` so that it becomes aware of the type `matrix`. Line 16 uses both types `struct _matrix` and `matrix`: `matrix` is the convienient name for referring to the pointer type, while `struct _matrix` must be used explicitly for obtaining the size of the structure.

Finally, we place `main`, which implements a unit test, in its own file, `matrix_test.c`:

```
#include <stdio.h>
#include "matrix.h"

int main() {
  matrix A, Ac, B, c, d, M, ct, mdp;
  double dp;

  A = newMatrix(3, 3);
  setElement(A, 1, 1, 1.0);
  // ...
```

To obtain access to the library, the unit test simply includes `matrix.h`. The difference in inclusion style—`"matrix.h"` with quotes, `<stdio.h>` with brackets—tells the compiler where to look for the files. Brackets indicate standard or system header files, which typically reside in system-level directories such as `/usr/include`, while quotes indicate project header files, which reside in the same or a nearby directory.

Because `matrix.c` and not `matrix.h` has the definition of `struct _matrix`, the data layout is as invisible to the user as are the implementations of the functions, achieving true separation of interface and implementation.

One can compile the multiple files manually:

```
$ gcc -Wall -Wextra -c matrix.c
$ gcc -Wall -Wextra -c matrix_test.c
$ gcc -o matrix_test matrix.o matrix_test.o
```

The `-c` flag tells `gcc` to compile but not to link; `gcc` yields the two **object files** `matrix.o` and `matrix_test.o` in this mode. The final invocation of `gcc` links the two object files together; the `-o matrix_test` option tells `gcc` to call the final executable `matrix_test` rather than `a.out`. The first call to `gcc` also checks the syntax of `matrix.h` because `matrix.c` includes it; however, one can explicitly check the file—for example, after writing the interface but before implementing it—by running `gcc` on it:


```
$ gcc -Wall -Wextra -c matrix.h
```

A more convenient option is to define a **makefile**, which is typically called `Makefile`:

```
1 CC     = gcc                # sets gcc as the compiler
2 CFLAGS = -Wall -Wextra -g   # our standard arguments to gcc
3
4 # to make matrix_test, we need matrix.o and matrix_test.o
5 matrix_test: matrix.o matrix_test.o
6
7 # "make clean" removes the executable and the object files
8 clean:
9         rm -f matrix_test *.o
```

Now we simply run `make`, yielding the executable `matrix_test` as well as the following output:

```
$ make
gcc -Wall -Wextra -g   -c -o matrix_test.o matrix_test.c
gcc -Wall -Wextra -g   -c -o matrix.o matrix.c
gcc   matrix_test.o matrix.o   -o matrix_test
```

By using the standard variables `CC`, for "C Compiler," and `CFLAGS`, for "C Flags," the `make` utility does most of the work for us. We just provide the **target** executable (`matrix_test`) and the object files that it depends on (`matrix.o` and `matrix_test.o`) at line 5. The second target, `clean`, is executed via `make clean`; it deletes the executable and object files. Executing `./matrix_test` then yields the output at the end of Section 6.1.

Separate compilation highlights the separation of implementation from specification that ADTs offer. The reason that `gcc` can compile `matrix_test.c` without having the definition of `struct _matrix` is because `matrix` is a pointer type. All pointers occupy the same number of bytes; hence, `gcc` can compute stack offsets, among many other tasks, without knowing about `struct _matrix`.

Exercise 7.1. Implement an abstract data type for representing and manipulating complex numbers. For two complex numbers $a + b\mathrm{i}$ and $c + d\mathrm{i}$,

- $(a + b\mathrm{i}) + (c + d\mathrm{i}) = (a + c) + (b + d)\mathrm{i}$
- $(a + b\mathrm{i})(c + d\mathrm{i}) = ac + ad\mathrm{i} + bc\mathrm{i} + bd\mathrm{i}^2 = (ac - bd) + (ad + bc)\mathrm{i}$

Solution. In `complex.h`, we write the following interface, which defines how a user creates, manipulates, and destroys complex numbers:

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

/* The type declaration of the ADT. */
typedef struct _complex * complex;
```

```

/* Creates a complex number, initially 0. */
complex newComplex();

/* Deletes a complex number. */
void deleteComplex(complex c);

// "setters"

/* Sets the real component of c. */
void setReal(complex c, double r);

/* Sets the imaginary component of c. */
void setImaginary(complex c, double i);

// "getters"

/* Returns the real component of c. */
double getReal(complex c);

/* Returns the imaginary component of c. */
double getImaginary(complex c);

// basic arithmetic

/* Adds b to a, with the result being stored in a. */
void addTo(complex a, complex b);

/* Multiplies b by a, with the result being stored in a. */
void multiplyBy(complex a, complex b);

/* Multiplies complex a by real b, with the result being
 * stored in a.
 */
void multiplyByReal(complex a, double b);

/* Prints in a + bi form. */
void printComplex(complex c);

#endif
```

Even before implementing the interface, we can test if the interface itself is "complete." Does it allow us to perform the work that we would like to accomplish? The test of the interface later becomes a unit test of the implementation. We implement the test in `complex_test.c`:

```
#include "complex.h"
#include <stdio.h>

```



```
int main() {
  complex c1 = newComplex();
  complex c2 = newComplex();

  // create c1 with value 1
  setReal(c1, 1.0);
  setImaginary(c1, 0.0);
  printComplex(c1);
  printf(", ");

  // create c2 with initial value i
  setReal(c2, 0.0);
  setImaginary(c2, 1.0);
  printComplex(c2);
  printf("\n");

  // set c1 = c1 * c2, which is i
  multiplyBy(c1, c2);
  printComplex(c1);
  printf("\n");

  // negate c1 so that it becomes -i
  multiplyByReal(c1, -1);
  printComplex(c1);
  printf("\n");

  // set c1 = c1 + c2, which is -i + i, or 0
  addTo(c1, c2);
  printComplex(c1);
  printf("\n");

  // clean up
  deleteComplex(c1);
  deleteComplex(c2);

  return 0;
}
```

If all goes well with the implementation, we expect to see the following printed to the terminal when we run the unit test:

```
1.000000 + 0.000000i, 0.000000 + 1.000000i
0.000000 + 1.000000i
-0.000000 + -1.000000i
0.000000 + 0.000000i
```

Having established that the interface for manipulating complex numbers is usable, we turn to the task of implementing the ADT in `complex.c`:

```
#include "complex.h"
```

```
#include <stdio.h>
#include <stdlib.h>

// represents a complex number as a pair of doubles
struct _complex {
  double r;  // the real part
  double i;  // the imaginary part
};

complex newComplex() {
  complex c = (complex) malloc(sizeof(struct _complex));
  c->r = 0.0;
  c->i = 0.0;
  return c;
}

void deleteComplex(complex c) {
  free(c);
}

void setReal(complex c, double r) {
  c->r = r;
}

void setImaginary(complex c, double i) {
  c->i = i;
}

double getReal(complex c) {
  return c->r;
}

double getImaginary(complex c) {
  return c->i;
}

void multiplyByReal(complex c, double r) {
  c->r = r * c->r;
  c->i = r * c->i;
}

void addTo(complex a, complex b) {
  a->r += b->r;
  a->i += b->i;
}

void multiplyBy(complex a, complex b) {
  double r = a->r * b->r - a->i * b->i;
  double i = a->r * b->i + a->i * b->r;
```



```
  a->r = r;
  a->i = i;
}

void printComplex(complex c) {
  printf("%f + %fi", c->r, c->i);
}
```

Each function is relatively straightforward, but all together, they define a powerful new data type.

To compile the three files into a unit test, we write the following in `Makefile`:

```
CC     = gcc
CFLAGS = -Wall -Wextra -g

all: complex_test

complex_test: complex.o complex_test.o

clean:
        rm -f complex_test *.o
```

At the command line, we do the following:

```
$ make
$ ./complex_test
1.000000 + 0.000000i, 0.000000 + 1.000000i
0.000000 + 1.000000i
0.000000 + -1.000000i
0.000000
```

In practice, the `complex` ADT would be used in a more complicated program with its own `Makefile` and `main` function. For example, a program that implemented the discrete Fourier transform (see Chapter 11) would require a representation of complex numbers. □

Exercise 7.2. Implement the following interface for the abstract data type of two-dimensional coordinates. The interface provides access to a coordinate using both Cartesian and polar coordinates.

Cartesian coordinate (x, y) corresponds to polar coordinate $(r = \sqrt{x^2 + y^2},\ \theta = \operatorname{atan2}(y, x))$, where $\operatorname{atan2}(y, x)$ returns the angle between 0 and 2π, exclusive, corresponding to (x, y). Conversely, polar coordinate (r, θ) corresponds to Cartesian coordinate $(x = r\cos(\theta),\ y = r\sin(\theta))$.

```
#ifndef _COORD_H_
#define _COORD_H_

/* The type declaration of the ADT. */
```

```c
typedef struct _coord * coord;

/* Creates a coordinate. */
coord newCoord();

/* Deletes a coordinate. */
void deleteCoord(coord c);

// "getters"

/* For Cartesian coordinates. */
double getX(coord c);
double getY(coord c);

/* Returns the radius component. */
double getR(coord c);

/* Returns the angle component through the reference th.
 * The angle is undefined if the corresponding Cartesian
 * coordinate is (0, 0), so it returns -1 in this case;
 * otherwise, it returns 0.
 */
int getTheta(coord c, double * th);

// "setters"

/* For Cartesian coordinates. */
void setX(coord c, double x);
void setY(coord c, double y);

/* Set the radius/angle components if possible and return
 * 0.  However, neither can be set if the corresponding
 * Cartesian coordinate is (0, 0), so they leave the
 * coordinate unmodified and return -1 in this case.
 */
int setR(coord c, double r);
int setTheta(coord c, double th);

#endif
```

As an example of the ADT's usage, consider the following unit test:

```c
#include "coord.h"
#include <stdio.h>

int main() {
  coord c = newCoord();
  double th;

  setX(c, 1.0);
  // c is (1, 0), so th should be 0
  getTheta(c, &th);
  printf("%f\n", th);

  setY(c, .5);
  // c is (1, .5), so th should be atan(.5/1)
  getTheta(c, &th);
  printf("%f\n", th);

  setX(c, 0.0);
  setR(c, 1.0);
  // c is (0, 1)
  printf("%f %f\n", getX(c), getY(c));

  setTheta(c, 3.14159265);
  // c is (-1, 0)
  printf("%f %f\n", getX(c), getY(c));

  deleteCoord(c);

  return 0;
}
```

To implement the ADT, use the trigonometric functions `sin`, `cos`, and `atan2` defined in `math.h`. You can read about them online. Compiling when using the standard math library requires the library inclusion flag `-lm`:

```makefile
CC     = gcc
CFLAGS = -Wall -Wextra -g
LIBS   = -lm

all: coord_test

coord_test: coord.o coord_test.o
        gcc -o $@ $^ $(CFLAGS) $(LIBS)

clean:
        rm -f coord_test *.o
```

When implementing the ADT, think carefully about the basic representation. There are two obvious possibilities:

- Represent the coordinate using Cartesian coordinates only, so that `struct _coord` has two `double` fields: `x` and `y`.
- Represent the coordinate using polar coordinates only, so that `struct _coord` has two `double` fields: `r` and `theta`.

The first representation is fast if the user mostly uses the Cartesian functionality but relatively slow if the user mostly uses the polar functionality; the opposite is true of the second representation. As a challenge, devise a representation that is fast whenever the user only uses the Cartesian functions or only uses the polar functions; in other words, it adapts to how the user applies the library. □

7.2 FIFO Queue: A Specification

A **queue** is a data structure into which one can **put** elements and from which one can subsequently **get** elements. The policy of the queue dictates in what order elements are retrieved. A **first-in first-out**, or **FIFO**, queue is one in which elements are retrieved in the same order that they were added. A line at the grocery store is a FIFO. A **last-in first-out**, or **LIFO**, queue (sometimes called a **first-in last-out**, or **FILO**, queue) has the opposite policy. Another name for a LIFO queue is a **stack**.

Queues of both types are frequently used data structures in complex programs. For example, FIFO queues are used to buffer sensory input in embedded systems, to orchestrate software pipelines in multicore systems, and as a basis for breadth-first search in graph-based algorithms. LIFO queues are used to implement general recursion with loops, in compilers to implement variable scoping, in interpreters to provide a program stack, and as a basis for depth-first search in graph-based algorithms. The program stack that we have been using is, of course, a LIFO—although one that is implicit in the programming model rather than explicit as a data structure.

In some applications, FIFOs can be effectively unbounded, while other applications require FIFOs with a maximum capacity. When that capacity is reached, the client program must implement its own policy. For example, inessential sensory information in an embedded system might be ignored. In contrast, the arrival of vital sensory information when a FIFO is full might trigger a different mode of behavior that is intended to handle the vital information as soon as possible. The FIFO module itself need only provide a mechanism for alerting the client module that the FIFO is full. In this chapter, we explore an implementation of the FIFO ADT that has a user-specified maximum capacity; in Chapter 8, we discuss a new basic data structure that enables an unbounded implementation of FIFOs.

The file `fifo.h` specifies the abstract data type of `fifo`:

```
1  #ifndef _FIFO_H_
2  #define _FIFO_H_
3
4  /* Defines the ADT of First-In First-Out queues. */
5
6  /* The type declaration of the ADT. */
7  typedef struct _fifo * fifo;
8
9  /* Returns a new fifo with the given maximum capacity. */
```



```
10 fifo newFifo(int capacity);
11
12 /* Deletes a fifo. */
13 void deleteFifo(fifo q);
14
15 /* Returns whether q is empty -- 1 (true) or 0 (false). */
16 int isEmptyFifo(fifo q);
17
18 /* Adds element e to q.  Returns 0 if successful and -1 if
19  * e could not be added to q because q is full.
20  */
21 int putFifo(fifo q, void * e);
22
23 /* Sets e to point to the first element of q and removes
24  * the element from q.  Returns 0 if successful and -1 if e
25  * is NULL.  If q is empty, returns -2 and sets *e to NULL.
26  */
27 int getFifo(fifo q, void ** e);
28
29 /* Specification of user-provided printing function. */
30 typedef void (* printFn)(void *);
31
32 /* Prints the elements of q in order.  Requires a printFn,
33  * a user-provided function that prints an element.
34  * Returns 0 if successful and -1 if f is NULL.
35  */
36 int printFifo(fifo q, printFn f);
37
38 #endif
```

There are several new programming concepts in this specification. The first is the use of the pointer type `void *`, which is used to indicate a pointer to data with an unknown structure. From the `fifo`'s perspective, the form of the data does not matter. However, a user of a `fifo` must **cast** data to be of type `void *` to avoid a lot of compiler warnings:

```
  char * in = "Gallia est omnis divisa in partes tres...";
  putFifo(q, (void *) str);
```

More generally, a FIFO is a **container ADT**: it stores user-provided data, and it should work for any type of data. Some languages, like C++ and Java, provide advanced facilities for writing container types like FIFOs, but C does not. Therefore, it makes sense that we use a "generic" type like `void *` when implementing a container ADT: it says that the `fifo` neither knows nor cares about what the data are, but it will do a good job of storing them and returning them in the same order that they were given.

The second new concept is the use of a **function pointer**. A function pointer is, as its name suggests, a pointer to a function. The type declaration at line 30 declares the type `printFn` to describe a pointer to a function that accepts one argument, data, and returns nothing (`void`). The need for a function pointer is simple: we would like to provide a function to print the state of the `fifo`, but the `fifo` does not have any knowledge of what the data that it holds look like. Hence, the user provides a function that prints one data element. We provide an example usage shortly.

We have reached a point where I must assume that you have developed a certain level of programming maturity in order to continue the exposition. In particular, the use of `void *`, the consequent typecasting, and the use of function pointers are only the first of a set of "advanced" C techniques that we will use in the next two chapters. If you feel that your understanding of the foundations is inadequate or that the advanced material requires too big of a jump, now is a good time to allocate extra time to shore up the foundations.

Without further ado, we consider a unit test for the `fifo` module, which we put in the file `fifo_test.c`. The first half of the test uses a `fifo` to hold data of type `long`, which is an integer type that on many, but not all, platforms occupies the same number of bytes as a pointer and which may or may not occupy more bytes than an `int`.[1] Therefore, we typecast values of type `long` to be values of type `void *`, a seemingly hacky thing to do. A **hack** is a kludge, an inelegant widget, an application of duct tape, a programming no-no—in short, a line or two of code that you hope nobody notices but that gets the job done. But what makes a hack a hack is that it's not commonly accepted practice—or if it is, it's at least frowned upon. This kind of cast is common, and while it may cause a raised eyebrow, it probably should not induce a frown.

The second half of the test exercises a `fifo` that holds strings. Both `fifo`s are initialized to have a maximum capacity of three elements.

Finally, the functions `printLong` and `printString` are passed to `printFifo`. They provide the interface between the user and the library in order to print out the state of the `fifo`. Notice that the `void *` datum that is passed to these function must be typecast to `long` and `char *`, respectively.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "fifo.h"

static void printLong(void * e) {
  // %ld tells printf to print a long integer
  printf("%ld", (long) e);
}

static void printString(void * e) {
  printf("%s", (char *) e);
```

[1] On 32-bit and 64-bit Unix platforms, a `long` occupies 4 and 8 bytes, respectively, the same as a pointer.


```
}

int main() {
  fifo longq, stringq;
  void * e;

  // test with longs

  longq = newFifo(3);

  assert(isEmptyFifo(longq));

  printf("longq (empty): ");
  printFifo(longq, printLong);

  assert(!putFifo(longq, (void *) 1));
  assert(!putFifo(longq, (void *) 2));
  assert(!putFifo(longq, (void *) 3));

  assert(putFifo(longq, (void *) 4));

  printf("longq (3 elements): ");
  printFifo(longq, printLong);

  assert(!getFifo(longq, &e));
  printf("from longq (1): %ld\n", (long) e);

  assert(!putFifo(longq, (void *) 4));

  printf("longq (3 elements): ");
  printFifo(longq, printLong);

  assert(!getFifo(longq, &e));
  printf("from longq (2): %ld\n", (long) e);
  assert(!getFifo(longq, &e));
  printf("from longq (3): %ld\n", (long) e);
  assert(!getFifo(longq, &e));
  printf("from longq (4): %ld\n", (long) e);

  assert(isEmptyFifo(longq));
  assert(getFifo(longq, &e));
  assert(!e);

  deleteFifo(longq);

  // test with strings

  stringq = newFifo(3);

```

```
  assert(isEmptyFifo(stringq));

  printf("stringq (empty): ");
  printFifo(stringq, printString);

  assert(!putFifo(stringq, (char *) "Hello"));
  assert(!putFifo(stringq, (char *) "there"));
  assert(!putFifo(stringq, (char *) "universe"));

  assert(putFifo(stringq, (char *) "!"));

  printf("stringq (3 elements): ");
  printFifo(stringq, printString);

  assert(!getFifo(stringq, &e));
  printf("from stringq (Hello): %s\n", (char *) e);

  assert(!putFifo(stringq, (char *) "!"));

  printf("stringq (3 elements): ");
  printFifo(stringq, printString);

  assert(!getFifo(stringq, &e));
  printf("from stringq (there): %s\n", (char *) e);
  assert(!getFifo(stringq, &e));
  printf("from stringq (universe): %s\n", (char *) e);
  assert(!getFifo(stringq, &e));
  printf("from stringq (!): %s\n", (char *) e);

  assert(isEmptyFifo(stringq));
  assert(getFifo(stringq, &e));
  assert(!e);

  deleteFifo(stringq);

  return 0;
}
```

Running this unit test yields the following output:

```
longq (empty):
longq (3 elements):  1:1 2:2 3:3
from longq (1): 1
longq (3 elements):  1:2 2:3 3:4
from longq (2): 2
from longq (3): 3
from longq (4): 4
stringq (empty):
stringq (3 elements):  1:Hello 2:there 3:universe
```



```
from stringq (Hello): Hello
stringq (3 elements):  1:there 2:universe 3:!
from stringq (there): there
from stringq (universe): universe
from stringq (!): !
```

It is customary in unit tests to indicate parenthetically the expected output as well as to use an abundance of `assert`s.

Exercise 7.3. Write a specification for a LIFO module in file `lifo.h`. Run `gcc -Wall -Wextra -c lifo.h` to check for syntax errors. □

Exercise 7.4. Write a unit test for a LIFO module in file `lifo_test.c`. Run `gcc -Wall -Wextra -c lifo_test.c` to check for syntax errors. □

7.3 FIFO Queue: A First Implementation

In this and the next chapters, we cover two implementations of the specification given in `fifo.h`. The implementation that is chosen at link time determines the runtime behavior, although the functionality looks almost identical—*almost*, because the second implementation allows the user to specify an unbounded queue—from a `fifo` user's perspective. This section focuses on an implementation based on a **circular buffer**, which we place in a file called `cbuffer.c`.

A circular buffer is simply a bit of logic built on top of an array.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "fifo.h"

struct _fifo {
  unsigned capacity;
  unsigned head;
  unsigned tail;
  void * data[0];
};
```

A `head` index, a `tail` index, and the `data` buffer itself form the circular buffer. The `capacity` must also be recorded for reasons that will become clear shortly. The type `unsigned` is short for "unsigned integer." Data in memory cells of type `unsigned` are interpreted as nonnegative integers.

The `data` field is declared as a 0-length array of `void *` elements. In fact, in `newFifo` below, we allocate memory in such a way that the `data` array has a number of elements equal to the `capacity`. This cryptic but idiomatic declaration allows us to allocate one contiguous chunk of memory to hold both the `struct _fifo` structure and the data. We could have declared `data` to be a `void **`, as in the `matrix` implementation, but then we would have to allocate one chunk of memory to hold the `struct _fifo` structure and another to hold the data.

Before delving into the implementation, let's consider circular buffers functionally and pictorially. Here is a partially full circular buffer:

$\boxed{\overline{2}}\boxed{3}\boxed{4}\boxed{\underline{\otimes}}$

The overline in cell 0 indicates that `tail` is 0, while the underline in cell 3 indicates that `head` is 3. This circular buffer has three elements, and they are placed in the first three cells. The symbol $\otimes$ indicates that cell 3 does not hold valid data. Here is another partially full circular buffer:

$\boxed{8}\boxed{\underline{\otimes}}\boxed{\otimes}\boxed{\overline{7}}$

Notice how `head < tail` in this configuration. A circular buffer is circular in the sense that indexing is modulo its `capacity`. The valid data range is between `tail` and `head - 1`, modulo `capacity`. In the second configuration, the cells with valid data are thus 3 and 0.

A new element is added by placing it in the `head` cell and then setting `head` to `(head + 1) % capacity`. If a new element were added to the first configuration, it would be the case that `head == tail` in the resulting configuration. Yet when `head == tail`, the valid data range is empty. Therefore, a circular buffer is full when `(head + 1) % capacity == tail`. Circular buffers are slightly inefficient in that one cell is always garbage. Adding an element, say 9, to the second configuration yields a full buffer:

$\boxed{8}\boxed{9}\boxed{\underline{\otimes}}\boxed{\overline{7}}$

If a circular buffer is nonempty—that is, `head != tail`—then an element can be removed from the `tail` cell: the value is returned, and `tail` is set to `(tail + 1) % capacity`. For example, removing an element from the configuration above yields the element 7 and the following new configuration:

$\boxed{\overline{8}}\boxed{9}\boxed{\underline{\otimes}}\boxed{\otimes}$

Of course, the $\otimes$ in cell 3 is technically still 7; there is no reason to explicitly delete that datum.

Exercise 7.5. How large a circular buffer is required for the following sequence of actions to succeed: put 1, put 2, put 3, get, put 4, get, get, put 5, put 6, put 7? Which value will the next "get" yield with this sufficiently large buffer?

Solution. By analyzing the sequence of puts and gets, we see that the most elements—four of them—are in the circular buffer after the 7 is put. Given that circular buffers have one wasted cell, we apparently need a buffer of size five. Let's visualize the sequence to verify that five is indeed correct:

initial: $\boxed{\overline{\underline{\otimes}}}\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}$

put 1: $\boxed{\overline{1}}\boxed{\underline{\otimes}}\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}$

put 2: $\boxed{\overline{1}}\boxed{2}\boxed{\underline{\otimes}}\boxed{\otimes}\boxed{\otimes}$

put 3: $\boxed{\overline{1}}\boxed{2}\boxed{3}\boxed{\underline{\otimes}}\boxed{\otimes}$

get: $\boxed{\otimes}\boxed{\overline{2}}\boxed{3}\boxed{\underline{\otimes}}\boxed{\otimes}$

put 4: $\boxed{\otimes}\boxed{\overline{2}}\boxed{3}\boxed{4}\boxed{\underline{\otimes}}$

get: $\boxed{\otimes}\boxed{\otimes}\boxed{\overline{3}}\boxed{4}\boxed{\underline{\otimes}}$

get: $\boxed{\otimes}\boxed{\otimes}\boxed{\otimes}\boxed{\overline{4}}\boxed{\underline{\otimes}}$

put 5: $\boxed{\underline{\otimes}}\boxed{\otimes}\boxed{\otimes}\boxed{\overline{4}}\boxed{5}$

put 6: $\boxed{6}\boxed{\underline{\otimes}}\boxed{\otimes}\boxed{\overline{4}}\boxed{5}$

put 7: $\boxed{6}\boxed{7}\boxed{\underline{\otimes}}\boxed{\overline{4}}\boxed{5}$

The next "get" will yield 4. □

With this visual introduction to circular buffers, let's see how the implementation plays out. First, `newFifo` allocates and initializes the circular buffer:

```
1  fifo newFifo(int capacity) {
2    assert (capacity > 0);
3
4    // The capacity of a circular buffer is one less than one
5    // would think: if the user wants a given capacity, the
6    // required array is one cell larger.
7    capacity++;
8
9    // allocate one chunk of memory
10   fifo q = (fifo) malloc(sizeof(struct _fifo) +
11                          capacity * (sizeof(void *)));
12   q->capacity = (unsigned) capacity;
13   q->head = 0;
14   q->tail = 0;
15   return q;
16 }
```

Notice first that `capacity` is incremented at line 7. Recall that the actual capacity of a circular buffer is one fewer than its number of cells—`head == tail` indicates an empty buffer, so that `(head + 1) % capacity == tail` indicates a full buffer.

Next, observe the allocation trick that we employ at lines 10–11. Rather than allocating two separate chunks of memory, one of `sizeof(struct _fifo)` bytes and the other of `capacity * sizeof(void *)` bytes, we allocate a single chunk. Therefore, `q->capacity`, `q->head`, `q->tail`, and elements `q->data[0]` through `q->data[q->capacity-1]` of the `q->data` array are all within the one chunk of allocated memory.

Deleting a circular buffer is comparatively easy, especially since there is only one chunk of memory to `free`:

```
void deleteFifo(fifo q) {
  assert (q);
  free(q);
}
```

The implementations of the next three functions, `isEmptyFifo`, `putFifo`, and `getFifo`, follow directly from the discussion of circular buffers and the specifications of the functions in `fifo.h`:

```
int isEmptyFifo(fifo q) {
  assert (q);
  return (q->head == q->tail);
}

int putFifo(fifo q, void * e) {
  assert (q);
  if ((q->head+1) % q->capacity == q->tail) // full?
    return -1;
  q->data[q->head] = e;
  q->head = (q->head+1) % q->capacity;
  return 0;
}

int getFifo(fifo q, void ** e) {
  assert (q);
  if (!e) return -1;
  if (isEmptyFifo(q)) {
    *e = NULL;
    return -2;
  }
  *e = q->data[q->tail];
  q->tail = (q->tail+1) % q->capacity;
  return 0;
}
```

Finally, `printFifo` applies the user-supplied `printFn` to every valid cell, in order:

```
int printFifo(fifo q, printFn f) {
  assert (q);
  if (!f) return -1;
```



```

  unsigned i, cnt = 1;
  for (i = q->tail; i != q->head; i = (i+1) % q->capacity)
  {
    printf(" %d:", cnt);
    f(q->data[i]);
    cnt++;
  }
  printf("\n");
  return 0;
}
```

Notice how modulo is used in the loop increment.

To compile `fifo.h`, `cbuffer.c`, and `fifo_test.c`, we write the following `Makefile`:

```
CC     = gcc
CFLAGS = -Wall -Wextra -g

cbuffer_test: cbuffer.o fifo_test.o
        $(CC) -o cbuffer_test cbuffer.o fifo_test.o

clean:
        rm -f cbuffer_test *.o
```

This `Makefile` is slightly more complicated than the one for the `matrix` module because the target, `cbuffer_test`, has a different name than the file containing the function `main`. We use a different name because we intend to augment this `Makefile` with another target that uses the alternate `fifo` implementation of the next chapter but the same `fifo_test.c`.

The result of running `./cbuffer_test` is at the end of Section 7.2. Not satisfied, we also run `valgrind -v ./cbuffer_test`, which yields the following satisfying report:

```
All heap blocks were freed -- no leaks are possible

ERROR SUMMARY: 0 errors from 0 contexts
```

Exercise 7.6. Implement in file `buffer.c` your specification from Exercise 7.3 of the LIFO queue using an array (as part of a `struct`) as the basic underlying data structure. Test it using your unit test from Exercise 7.4. The implementation is simpler than than of the circular buffer, so if you're introducing `head` and `tail` indices and trying to apply modulo addition, retreat and regroup. □

Exercise 7.7. Accessing basic C arrays can be dangerous because they are nothing more than regions of memory. If the wrong size is passed to a function, or a string is missing its string terminator, a loop over an array can easily read or write beyond the allocated memory. A segmentation fault is then the best case; a subtle and occasional memory corruption is far worse; and a security vulnerability is worse still.

Design and implement an ADT of protected arrays of `void *` elements. A user should be able to create and destroy arrays of given sizes, set and get their elements, and get its size. One of the key decisions is how to handle the case when a user provides an index outside the domain of an array. Whatever you decide, it better not result in a memory corruption.

As motivation, consider the following function that uses such a library:

```
void printNums(parray a) {
  int i;
  for (i = 0; i < size(a); i++)
    printf("%ld ", (long) get(a, i));
  printf("\n");
}
```

Your interface may of course have a different type name than `parray` and different functions than `size` and `get`. □

8

Linked Lists

While arrays are probably the most used data structure, they have their limits. In particular, whether stack or heap allocated, an array's size is fixed. For some applications, such as matrix-based computations or real-time control in embedded systems, this fixed size is appropriate. But many other applications require data structures that grow and shrink throughout their lifetimes according to demands. The **linked list** is among the most widely used data structure in such applications.

8.1 Introduction to Linked Lists

The Henry Ford Museum in Dearborn, Michigan, has a section on bicycles. One of the displays documents a turn-of-the-century (19^{th} to 20^{th}, that is) bicycling club that awarded a pin for the first century (100-mile ride) that a cyclist completed and a smaller medallion for each subsequent century. Each medallion linked via a hook to the previous one, and the pin itself had a hole for the first medallion, thus allowing the proud cyclist to display for everyone to see a list of his or her accomplishments. As I had reached that region of the country via my bicycle in a relatively short period, my first thought was that riding centuries is apparently much easier now than it was then. My second was — Behold! A linked list!

A **singly linked list** consists of a **head** pointer (the pin) followed by an arbitrary number of **nodes** (the medallions), each pointing to the next. Here is one possible definition of a `node`:

```
1 typedef struct _node {
2   struct _node * next;
3   void * e;
4 } * node;
```

The `next` pointer is intended to hold the address of the next `struct _node` in the list or `NULL` if it is the last, while the `e` field is intended to hold a

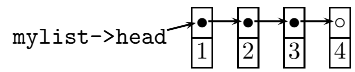
(generic) element of data. This data structure is **recursively defined**: the type of `next` is a pointer to an instance of the very same structure in which the `next` field resides. At line 2, the short name `node` for a `struct _node *` is not yet known, so the full name, provided on line 1, must be used.

A list can be as simple as a structure with a single field of type `node`:

```
typedef struct _llist {
  node head;
} * llist;
```

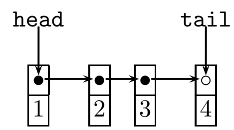
Or it could be more complex, potentially including an `int` field to hold the size of the list, another `node` to hold the tail of a list, or other information.

Linked lists are easy to visualize:

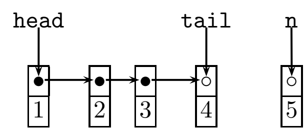


This list represents the data consisting of the sequence 1, 2, 3, 4. The final `node`'s `next` field is `NULL`.

Manipulating linked lists is the fun part. Let's suppose that we have one `node` called `head` that points to the beginning of the list and one called `tail` that points to the end:

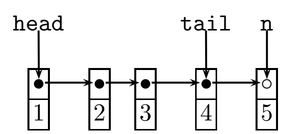


We want to append the datum 5 to the list. After creating a new node, referenced by `n`, whose `next` field is `NULL` and whose `e` field is 5,

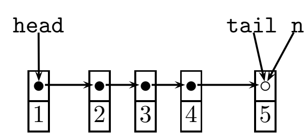


we append it to the list:

- Assign `tail->next = n`:

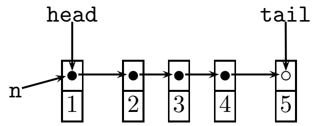


- Assign `tail = n`:

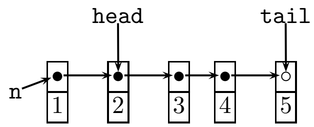


Let's suppose now that we want to obtain and remove the first element of the list. After obtaining the datum via `head->e`, we remove the first node:

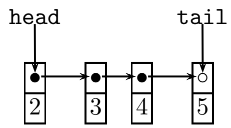
- Assign `node n = head`:



- Assign `head = head->next`:



- Free the node pointed to by `n`.



These two operations are the ones required for implementing a FIFO queue using linked lists.

Exercise 8.1. Declare the type of a comparison function that should take two `void *` elements and return one of -1, 0, or 1 to indicate the first element is less than, equal to, or greater than the second element, respectively.

Solution.

```
/* Type of user-defined comparison function.  Should return
 *   -1 - first element is less than second
 *    0 - the two elements are equally valued
 *    1 - the second element is greater than the first
 */
typedef int (* compareFn)(void *, void *);
```

□

Exercise 8.2. Using the `llist` type declared above and the `compareFn` type of Exercise 8.1, implement a function to decide if a given list is sorted in ascending order according to the provided comparison function:

```
/* Returns 0 for false or if f is NULL, 1 for true. */
int isSorted(llist ll, compareFn f);
```

Solution. The strategy is to compare adjacent elements. There are two corner cases: if the list is empty or if the list has one element. In both cases, the list is sorted independent of `f`.


```
int isSorted(llist ll, compareFn f) {
  node n = ll->head;

  // empty?
  if (!n) return 1;
  // single element?
  if (!n->next) return 1;

  if (!f) return 0;

  while (n->next) {
    // If any adjacent pair are in the wrong order...
    if (f(n->e, n->next->e) > 0)
      // ... the list is not sorted.
      return 0;
    n = n->next;
  }
  // All adjacent pairs are ordered; hence, so is the list.
  return 1;
}
```

□

Exercise 8.3. Using the `llist` type declared above, implement a function to reverse one linked list into another:

```
/* Reverses the elements of ll1 into ll2.  For example, if
 * ll1 is [0, 1, 2] and ll2 is [3, 4, 5], then after
 * running, ll1 will be empty and ll2 will be
 * [2, 1, 0, 3, 4, 5].
 */
void reverse(llist ll1, llist ll2);
```

Solution. This function plays the juggling game typical of linked-list manipulation. A node is carefully pulled from the front of `ll1` in a way so as not to forget the node's successor and then inserted at the front of `ll2`:

```
void reverse(llist ll1, llist ll2) {
  node n = ll1->head;
  while (n) {
    node next = n->next;
    n->next = ll2->head;
    ll2->head = n;
    n = next;
  }
  ll1->head = NULL;
}
```

It may help to sketch several iterations. □

Exercise 8.4. Using the `llist` type declared above, implement a function to concatenate linked lists:

```
/* Concatenates the elements of ll1 with ll2.  For example,
 * if ll1 is [0, 1, 2] and ll2 is [3, 4, 5], then after
 * running, ll1 will be [0, 1, 2, 3, 4, 5], and ll2 will be
 * empty.
 */
void concat(llist ll1, llist ll2);
```

Solution. See Exercise 8.14. □

8.2 FIFO Queue: A Second Implementation

Before reading this section, review the specification of the ADT `fifo` from Section 7.2.

Since linked lists are an integral part of the implementation, we first implement a definition of the `node` data type and functions for creating and deleting nodes in `llist.c`:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "fifo.h"

typedef struct _node {
  struct _node * next;
  void * e;
} * node;

static node newNode(void * e) {
  node n = (node) malloc(sizeof(struct _node));
  n->next = NULL;
  n->e = e;
  return n;
}

static void deleteNode(node n) {
  assert (n);
  free(n);
}
```

As these functions are internal to the module—that is, not intended to be called by a user—we use the `static` qualifier to hide them from other files. A file implementing a complex ADT's specification can have many `static`, or private, functions; only interface functions are non-`static`.

We next define `struct _fifo`. Similar to the circular buffer implementation, it has a `capacity`. Unlike circular buffers, however, linked lists can be of arbitrary length, so we also need to track the list's `size`. Finally, `head` and `tail` pointers are intended to be used as in the figures of Section 8.1: elements are removed via the `head` and added via the `tail`.

```
struct _fifo {
  int capacity;
  int size;
  node head;
  node tail;
};
```

The `newFifo` implementation takes advantage of a linked list's ability to be of arbitrary size. If `capacity <= 0`, we set `q->capacity = -1` to indicate unbounded capacity:

```
fifo newFifo(int capacity) {
  fifo q = (fifo) malloc(sizeof(struct _fifo));
  if (capacity <= 0) capacity = -1;
  q->capacity = capacity;
  q->size = 0;
  q->head = NULL;
  q->tail = NULL;
  return q;
}
```

Deleting a linked list is tricky, so we save its implementation for later.

A `fifo` is empty when its `size` parameter is 0:

```
int isEmptyFifo(fifo q) {
  assert (q);
  return (q->size == 0);
}
```

Another characteristic of an empty `fifo` based on linked lists is that both the `head` and the `tail` pointers are `NULL`. Hence, if we decided not to have a `size` field, we could implement `isEmptyFifo` based on checking whether `head` (alternately, `tail`) is `NULL`.

Now we get to the heart of the linked list implementation. For `putFifo`, we need to translate the illustrated steps in Section 8.1 of appending a new node holding the value `e` to the end of the list:

```
int putFifo(fifo q, void * e) {
  assert (q);
  if (q->size == q->capacity)
    // Full?  Impossible if q->capacity == -1.
    return -1;

  node n = newNode(e);
```

```
 8  if (q->size == 0) {
 9    // Both the head and the tail should be NULL.
10    assert (!q->head);
11    assert (!q->tail);
12    // Set them both to point to n.
13    q->head = n;
14    q->tail = n;
15  }
16  else {
17    // The tail node should be the last one.
18    assert (!q->tail->next);
19    // Append n and make it the new tail.
20    q->tail->next = n;
21    q->tail = n;
22  }
23  q->size++;
24
25  return 0;
26 }
```

Line 3 checks if the queue is full. If `capacity == -1`, the queue can never be full. Lines 13–14 handle the case in which the queue is empty, while lines 20–21 handle the nonempty case. Read these lines carefully. Draw your own illustrations for key assignments and for both empty and nonempty situations.

The implementation of `getFifo` similarly follows the illustrated steps of removing the first node of the list:

```
 1 int getFifo(fifo q, void ** e) {
 2   assert (q);
 3   if (!e) {
 4     // Nowhere to write result.
 5     return -1;
 6   }
 7   if (isEmptyFifo(q)) {
 8     // Nothing to get.
 9     *e = NULL;
10     return -2;
11   }
12   // Should be nonempty at this point.
13   assert (q->head);
14
15   node n = q->head;
16   // Write the element.
17   *e = n->e;
18   if (q->size == 1) {
19     // n should not have a successor.
20     assert (!n->next);
21     // Set both head and tail to NULL (empty list).
22     q->head = NULL;
```



```
23     q->tail = NULL;
24   }
25   else {
26     // Set the head to n's successor.
27     q->head = n->next;
28   }
29   deleteNode(n);
30   q->size--;
31
32   return 0;
33 }
```

Lines 22–23 handle the special case in which the queue has one element. Again, draw your own illustrations for key assignments and for both one-element and multi-element situations.

The implementation of `printFifo` is interesting in that it uses a common programming idiom: iterating over a linked list (lines 7–13):

```
 1 int printFifo(fifo q, printFn f) {
 2   assert (q);
 3   if (!f) return -1;
 4
 5   int cnt = 1;
 6   node n;
 7   for (n = q->head; n != NULL; n = n->next) {
 8     // Print the index of the element.
 9     printf(" %d:", cnt);
10     // Call user-provided f to print the element.
11     f(n->e);
12     cnt++;
13   }
14   printf("\n");
15
16   return 0;
17 }
```

Exercise 8.5. Illustrate the execution of the following loop:

```
  node n;
  for (n = q->head; n != NULL; n = n->next) {
    // do something
  }
```

Consider both empty and nonempty queues.

An idiomatic form drops the comparison with `NULL`:

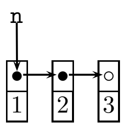
```
  node n;
  for (n = q->head; n; n = n->next) {
    // do something
  }
```

□

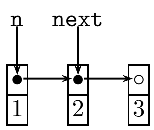
Finally, we must implement `deleteFifo`. The tricky part of this function is that we must `free` not only the `struct _fifo` instance but also any node that is in its list. Iterating over a linked list while `free`ing the nodes requires some pointer juggling:

```
void deleteFifo(fifo q) {
  assert (q);
  node n = q->head;
  while (n != NULL) {
    node next = n->next;
    deleteNode(n);
    n = next;
  }
  free(q);
}
```

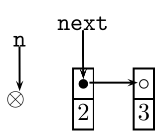
Let's visualize one iteration of the loop. We start with `n` pointing to the first (remaining) member of the list:



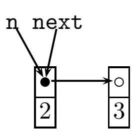
Then the assignment `next = n->next` occurs:



The node referenced by `n` is deleted:



Finally, `n` is set to `next`:



The iterations continue.

Exercise 8.6. Illustrate the final iteration, when `next` is assigned `NULL`. □

With two implementations of the same specification, we can augment the `Makefile` to allow us to choose which implementation to use:

```
1  CC     = gcc
2  CFLAGS = -Wall -Wextra -g
3
4  all: cbuffer_test llist_test
5
6  cbuffer_test: cbuffer.o fifo_test.o
7          $(CC) -o cbuffer_test cbuffer.o fifo_test.o
8
9  llist_test: llist.o fifo_test.o
10         $(CC) -o llist_test llist.o fifo_test.o
11
12 clean:
13         rm -f cbuffer_test llist_test *.o
```

Executing **`make all`** creates both versions of **`fifo_test`**—one called **`cbuffer_test`** and the other called **`llist_test`**—while executing **`make cbuffer_test`** or **`make llist_test`** makes one or the other. Running **`valgrind llist_test`** indicates a clean bill of health, which is a good sign given the tricky code.[1]

Exercise 8.7. Implement your specification from Exercise 7.3 of the LIFO queue using a linked list as the basic underlying data structure. Test it using your unit test from Exercise 7.4. □

8.3 Priority Queue: A Specification

FIFO and LIFO queues have simple policies that are sufficient for many situations. But what if some values are more important than others? For example, in embedded systems, some sensory data are more important than others. In general, in many applications, one needs to impose an order on data other than order of arrival. A **priority queue** accepts a user-defined comparison function, and the `getPQueue` function returns the datum with the highest priority according to that comparison function.

The following specification is in file `pqueue.h`:

[1] It may relieve you to know that I did not simply type this module, compile it, and run it without a problem. For your edification, I confess to the following issues: (1) multiple syntax errors, (2) an initially incorrect implementation of **`deleteFifo`**, (3) a forgotten call to **`printf`** at line 14 of **`printFifo`**, and (4) a forgotten call to **`deleteNode`** at line 29 of **`getFifo`**. While I caught problems (2) and (4) myself, **`valgrind`** would have indicated them. I also had a copy–paste error in **`Makefile`**: line 10 initially compiled in **`cbuffer.o`**, so that I wasn't even testing the linked list implementation at first. This final issue took longer to discover, although the fact that everything seemed to be working fine upon first execution should have been a good indicator that I had messed up the **`Makefile`**.

```
#ifndef _PQUEUE_H_
#define _PQUEUE_H_

/* Defines the ADT of Priority Queue. */

typedef struct _pqueue * pqueue;

/* Definition of a comparison function:
 *  -1: e1 has higher priority than e2
 *   0: e1 and e2 have equal priorities
 *   1: e1 has lower priority than e2
 */
typedef int (* compareFn)(void * e1, void * e2);

/* Returns a new pqueue ordered by f. */
pqueue newPQueue(compareFn f);

/* Deletes a pqueue. */
void deletePQueue(pqueue q);

/* Returns 1 if q is empty and otherwise 0. */
int isEmptyPQueue(pqueue q);

/* Adds element e to q. */
void putPQueue(pqueue q, void * e);

/* Sets e to point to the element of q with the highest
 * priority and removes that element from q.  Returns 0 if
 * successful and -1 if e is NULL.  If q is empty, returns
 * -2 and sets *e to NULL.  If there are multiple elements
 * with equal priorities, only one of them is returned, but
 * which one is implementation-dependent.
 */
int getPQueue(pqueue q, void ** e);

/* Prints the elements of q in order.  Requires a pointer
 * to a function that prints an element.  Returns 0 if
 * successful and -1 if f is NULL.
 */
typedef void (* printFn)(void *);
int printPQueue(pqueue q, printFn f);

#endif
```

The `pqueue` specification is similar to the `fifo` specification, except that `newPQueue` requires the user to provide a `compareFn`, and there is no way of limiting the capacity of the queue.

The following unit test, in file `pqueue_test.c`, exercises the functionality of an implementation of `pqueue.h`. Notice how it uses a command-line argument, if one is provided, to modify its behavior:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "pqueue.h"

// for printPQueue
static void printLong(void * e) {
  printf("%ld", (long) e);
}

// defines priorities over long data
static int compareLong(void * e1, void * e2) {
  if ((long) e1 < (long) e2)
    return -1;
  else if ((long) e1 == (long) e2)
    return 0;
  else
    return 1;
}

int main(int argc, char ** argv) {
  int i, nElements = 5;  // default value for nElements
  pqueue q;

  // Did the user provide an integer argument?
  if (argc > 1) {
    int n;
    if (sscanf(argv[1], "%d", &n))
      // If so, use it as nElements.
      nElements = n;
  }

  q = newPQueue(compareLong);

  // insert nElements random longs
  for (i = 0; i < nElements; ++i) {
    // rand() is provided by stdlib.h
    long e = (long) (rand() % 32);
    printf("putPQueue: %ld\n", e);
    putPQueue(q, (void *) e);
  }

  printf("State of the queue:\n");
  printPQueue(q, printLong);
```

```
  
  // get and print the elements
  while (!isEmptyPQueue(q)) {
    long e;
    assert (!getPQueue(q, (void **) &e));
    printf("getPQueue: %ld\n", e);
  }

  deletePQueue(q);

  return 0;
}
```

A correct implementation should yield output similar (up to variations in `rand()`) to the following if no argument is provided on the command line:

```
putPQueue: 7
putPQueue: 6
putPQueue: 9
putPQueue: 19
putPQueue: 17
State of the queue:
 1:6 2:7 3:9 4:17 5:19
getPQueue: 6
getPQueue: 7
getPQueue: 9
getPQueue: 17
getPQueue: 19
```

Exercise 8.8. Augment the unit test to test a priority queue of strings. Use the `strcmp` function of Exercise 3.29 or `string.h`. □

8.4 Priority Queue: An Implementation

Since the implementation is based on linked lists, we require the same definitions of `node`, `newNode`, and `deleteNode` in `pqueue.c` as in the linked list-based FIFO implementation. Additionally, we place the following code in `pqueue.c`:

```
struct _pqueue {
  compareFn cmp;
  node head;
};

pqueue newPQueue(compareFn f) {
  pqueue q = (pqueue) malloc(sizeof(struct _pqueue));
  q->cmp = f;
```



```
  q->head = NULL;
  return q;
}

void deletePQueue(pqueue q) {
  assert (q);
  node n = q->head;
  while (n) {
    node next = n->next;
    deleteNode(n);
    n = next;
  }
  free(q);
}
```

Notice that `deletePQueue` is identical to `deleteFifo`; it's worth studying again.

A `pqueue` is empty if `q->head` is `NULL`:

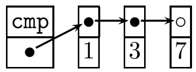
```
int isEmptyPQueue(pqueue q) {
  assert (q);
  return (q->head == NULL);
}
```

Now we arrive at the interesting functions. In this implementation, `putPQueue` applies the user-provided `compareFn`, stored in `q->cmp`, to find where to insert a new node with the supplied datum:

```
1  void putPQueue(pqueue q, void * e) {
2    assert (q);
3
4    node nn = newNode(e);
5    node n = q->head;
6    node * np = &(q->head);
7    while (n) {
8      if (q->cmp(e, n->e) < 0) break;
9      np = &(n->next);
10     n = n->next;
11   }
12
13   nn->next = n;
14   *np = nn;
15 }
```

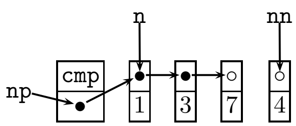
The twist in this implementation is that `np` is a `node *` (and recall that a `node` is itself a pointer) so that it can point either to the `head` field of `q` (line 6) or to the `next` field of a `node` (line 9). Recall that the `break` statement at line 8 causes control to exit the loop and then execute line 13.

To illustrate the `putPQueue` operation, let's consider inserting the `long` value 4 into the following priority queue, which is prioritized according to the `compareLong` function of `pqueue_test.c`:

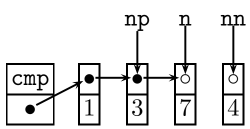


The first structure is a `struct _pqueue`, which, recall, has a `compareFn` field named `cmp` (top) and a `node` field named `head` (bottom). The other structures are nodes. As we walk through the process, pay attention to how `np` is used.

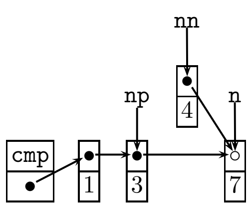
- Create the new node and set `nn` to it; set `np = &(q->head)` and `n = q->head`.



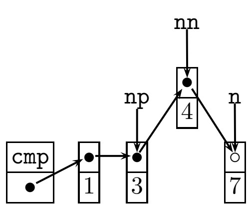
  Notice that `np` holds the address of the `head` field of the `pqueue`.
- Find `nn`'s place in the list:



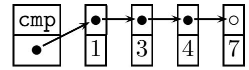
  Here, `np` holds the address of the `next` field of the node holding 3. Recall that the `next` field is of type `node`, and `np` is of type `node *`.
- Assign `nn->next = n`:



- Assign `*np = nn`:



Upon return, `putPQueue` yields this new configuration of the priority queue:



Because the hard work—placing the new node according to its element's priority—is done in `putPQueue`, `getPQueue` is comparatively straightforward:

```
int getPQueue(pqueue q, void ** e) {
  assert(q);
  if (!e) return -1;
  if (!q->head) {
    *e = NULL;
    return -2;
  }

  *e = q->head->e;
  node n = q->head;
  q->head = n->next;
  deleteNode(n);

  return 0;
}
```

The implementation is reminiscent of the linked list implementation of `getFifo`.

Exercise 8.9. Illustrate the operation of `getPQueue`. □

We compile this implementation and the unit test file with the following `Makefile`:

```
CC     = gcc
CFLAGS = -Wall -Wextra -g

all: pqueue_test

pqueue_test: pqueue.o pqueue_test.o

clean:
        rm -f pqueue_test *.o
```

The product is the executable `llist_test`. Running `valgrind ./llist_test` with various arguments (none, 0, 15, etc.) indicates a solid implementation.

Exercise 8.10. Implement the following specification:

```
/* Returns the number of values in q whose priorities equal
 * that of e.
 */
int countPQueue(pqueue q, void * e);
```

Solution. This function needs to perform a standard traversal of the list:

```
int countPQueue(pqueue q, void * e) {
  int cnt = 0;
  node n;
  for (n = q->head; n; n = n->next)
    if (q->cmp(n->e, e) == 0)
      cnt++;
  return cnt;
}
```

□

Exercise 8.11. Implement the following specification:

```
/* Removes all values from q whose priorities are equal to
 * that of e.
 */
void removePQueue(pqueue q, void * e);
```

Solution. Here is one possible implementation:

```
1 void removePQueue(pqueue q, void * e) {
2   assert (q);
3 
4   // Iterate over the list...
5   node n = q->head;
6   // ... while maintaining a pointer to what points to n.
7   node *np = &(q->head);
8   while (n) {
9     if (q->cmp(n->e, e) == 0) {
10      // Remove n.
11      *np = n->next;
12      deleteNode(n);
13      // Advance n...
14      n = *np;
15      // ... but np is already just behind n.
16    }
17    else {
18      // Advance n and np.
19      np = &(n->next);
20      n = n->next;
21    }
22  }
23 }
```

However, it only tests for equality at line 9, whereas q->cmp returns comparison information. Optimize it to use all of q->cmp's possible return values. □

Exercise 8.12. Illustrate the operation of `removePQueue` from Exercise 8.11. As in `putPQueue`, you need to handle `np` carefully because it is a `node *`. □

Exercise 8.13. The implementation presented here has the following characteristics: `putPQueue` takes time proportional to the queue size, while `getPQueue` takes constant time. Provide a new implementation of `pqueue.h` in which `putPQueue` takes constant time and `getPQueue` takes time proportional to the size of the queue. *Hint*: `putPQueue` should just insert the new node at the beginning of the list, while `getPQueue` should search the list for a maximum-priority value and then remove its corresponding node. □

8.5 Further Adventures with Linked Lists

Exercise 8.14. The use of `node *` variables—which, given that `node` is short for `struct _node *`, is a double-pointer type—in `putPQueue` and in `removePQueue` of Exercise 8.11 is a simple trick, which I call the **chaser pointer** technique, for implementing complex manipulations of pointer-based data structures. The chaser pointer references the field *in the data structure* that points to the node that the loop `node` variable points to. In other words, it chases the `node` variable.

For example, consider the `concat` function of Exercise 8.4. One possible implementation is the following:

```
void concat(llist ll1, llist ll2) {
  if (!ll1->head)
    ll1->head = ll2->head;
  else {
    node n;
    // position n so that it points to the final node
    for (n = ll1->head; n->next; n = n->next);
    // now append ll2
    n->next = ll2->head;
  }
  ll2->head = NULL;
}
```

The cases of `ll1`'s being empty and nonempty must be treated separately: in the former case, the `head` field of `ll1` is updated directly; in the latter, the `next` field of the last node of `ll1` is updated. Re-implement `concat` using a chaser pointer to avoid this case analysis.

Solution. The idea is to initialize the chaser pointer `np` to point to the `head` field of `ll1` and then iterate through `ll1`'s list. Upon completing, `np` will point to the `next` field of the final node of `ll1`'s list, which is exactly the field that must be updated:

```
void concat(llist ll1, llist ll2) {
  node n = ll1->head;
  node * np = &(ll1->head);  // np chases n
  while (n) {
    // go to the end of ll1
```

```
    np = &(n->next);          // np continues to chase n
    n = n->next;
  }
  // np points to the next field of the final node of ll1
  *np = ll2->head;
  ll2->head = NULL;
}
```

□

Exercise 8.15. Illustrate the execution of the two versions of `concat` of Exercise 8.14. □

Exercise 8.16. Using the `llist` type declared above, implement a function to copy a linked list:

```
/* Copies the list. */
llist copy(llist l);
```

Solution. Once again we use a chaser point, though in a slightly different way. Here, the chaser pointer references the final `node` field in the new list that is being created:

```
llist copy(llist l) {
  // create the new linked list
  llist cl = (llist) malloc(sizeof(struct _llist));
  cl->head = NULL;

  // copy each node of l and add to cl
  node n = l->head;
  // np "chases" the node to be created
  node * np = &(cl->head);
  while (n) {
    node cn = newNode(n->e);
    *np = cn;
    n = n->next;
    np = &(cn->next);
  }
  return cl;
}
```

□

Exercise 8.17. Illustrate the execution of `copy` of Exercise 8.16. □

Exercise 8.18. Using the `llist` type declared above, implement a function to "zip" together two linked lists:

```
/* Zips together the lists ll1 and ll2.  For example, if
 * ll1 is [0, 1, 2] and ll2 is [3, 4, 5, 6, 7], then after
 * running, ll3 will be [0, 3, 1, 4, 2, 5, 6, 7], and both
```



```
 * ll1 and ll2 will be empty.
 */
void zip(llist ll1, llist ll2, llist ll3);
```

□

Exercise 8.19. Using the `llist` type declared above, implement a function to "unzip" a linked list into two:

```
/* Unzips the list ll1 into ll2 and ll3.  For example, if
 * ll1 is [0, 1, 2, 3, 4, 5, 6] and ll2 and ll3 are empty,
 * then after running, ll1 will be empty, ll2 will be
 * [0, 2, 4, 6], and ll3 will be [1, 3, 5].
 */
void unzip(llist ll1, llist ll2, llist ll3);
```

□

Exercise 8.20. Implement a version of Exercise 6.2 that uses a linked list, rather than a growing array, to hold the strings. How does memory usage compare between the two versions? How do the number of allocations or re-allocations compare? Which data structure is more appropriate for the application? Describe a scenario in which the alternative becomes more appropriate. □

9

Introduction to Matlab

Variables, functions, parameters, call-by-value and call-by-reference semantics, control, data structures, ADTs, algorithms, modularity, design—these are *programming* concepts, not *C-specific* concepts. An accomplished programmer in any language, such as C, can learn any other programming language with little effort. In this chapter, we explore a **high-level programming language** embedded within a powerful engineering tool: Matlab. A high-level language is one in which a single statement can instigate an enormous amount of work—the complete opposite of a low-level language like C, in which each statement compiles to a small number of machine instructions.

High-level languages allow fast program development in specific domains. For example, developing numerical software is typically easier in Matlab than in C. Matlab provides built-in data structures for complex numbers and matrices; a concise and expressive language for their manipulation; and a vast library of functions for performing higher level computations, such as solving ordinary differential equations (Chapter 10), analyzing and manipulating time- and frequency-domain signals (Chapters 9 and 11), and many others relevant to engineers. As another example, the programming languages Python, Perl, and Ruby have elements that make system-level development simple: they provide powerful tools for analyzing and manipulating strings, interacting with the operating system, and writing network-level applications.

The typical trade-off of a high-level language is a sometimes significant decrease in performance. That said, a given language can be high performing for some applications and be appropriate for a wide range of applications. For example, `NumPy` is a Python package for programming fast numerical computations in Python. Also, all practical high-level languages allow writing performance-critical modules in a low-level language such as C or C++.

The final chapters of this text have three goals. The first is to make you a more flexible programmer by forcing you to translate important programming concepts from C to Matlab. In other words, there is a "meta-learning" opportunity: you should learn how to learn a new language. Engineers who write software learn (and sometimes forget) many languages over their careers. The

second goal is to switch from *creating* ADTs and libraries (as in Chapters 7 and 8) to *using* ADTs and libraries. At the same time, you should critically analyze the libraries that you use—with the eyes of a developer. What are their flaws? What are their strengths? Hence, the first two goals are in line with the primary focus of this book: learning to program.

The final goal is to introduce several engineering applications of high-level programming. This and Chapter 11 focus on time- and frequency-domain analysis and manipulation, with the fun motivation of understanding and creating music mathematically. Chapter 10 introduces the numerical approach to solving ordinary differential equations (ODEs) in the context of simulating orbital dynamics. Teaching all of the necessary fundamentals of these applications is well beyond the scope of this text; however, in future or concurrent courses that cover these topics, you should recall these applications and challenge yourself to identify opportunities to apply programming to help you understand new mathematical concepts and to obtain more general—and more impressive—results than can be obtained by hand.

9.1 The Command-Line Interface

High-level programming languages typically have **command-line interfaces** that allow users to construct relatively complex computations on the fly.

Suppose that we want to solve the following set of linear equations:

$$\begin{aligned} x_1 - x_2 &= 1 \\ \tfrac{1}{2}x_2 + x_3 &= 0 \\ -x_1 - x_2 - x_3 &= 2 \end{aligned}$$

As you have learned in your linear algebra course, we can view this system as a matrix equation of the form $Ax = b$:

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & \frac{1}{2} & 1 \\ -1 & -1 & -1 \end{bmatrix} x = \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}$$

In a linear algebra course, you would perhaps at this point solve the matrix equation using Gaussian elimination. As engineers, however, we can turn to Matlab, as matrix manipulation is one area where it excels. Let's fire up its command-line interface by running `matlab`:

```
>> A = [1 -1 0; 0 1/2 1; -1 -1 -1]

A =

    1.0000   -1.0000         0
         0    0.5000    1.0000
   -1.0000   -1.0000   -1.0000
```

```
>> b = [1; 0; 2]

b =

   1
   0
   2

>> A\b

ans =

    -1
    -2
     1
```

Semicolons suppress output:

```
>> A = [1 -1 0; 0 1/2 1; -1 -1 -1];
>> b = [1; 0; 2];
>> A\b

ans =

    -1
    -2
     1
```

On the language-level spectrum, if C is at sea level (*sea* level—get it?), then Matlab's language is somewhere above Mt. Everest.

Deconstructing the above three input lines, we see that the first line defines matrix `A`, the second defines column vector `b`, and the third applies the **left division**, or **backslash**, operator to solve the matrix equation $Ax = b$. The left division operator is a one-character interface to a library of horrendously complicated code. Defining a matrix or a vector is simple: spaces or commas separate elements of a row, and semicolons separate rows. Based on our exploration of a naive matrix library in Chapters 6 and 7, you can imagine the fair amount of code that underlies even the first two straightforward lines.

Exercise 9.1. Play around with the following matrix operators and functions to discover how they work: `\` (left division operator), `'` (transpose), `*` (matrix product), `.*` (element-wise product), `+`, `-`, `eye`, `ones`, `zeros`, `size`, and `length`. Use the command `help`, as in `help eye`, to learn more about each function. For punctuation-based operators (`'`, `*`, etc.), typing `help *` yields a menu of further help topics by name, next to their associated operators. □

Since a matrix is one of Matlab's primary data types, the language has a sophisticated facility for manipulating matrices. For matrix `A` defined as follows,

```
>> A = ones(4,3)

A =

     1     1     1
     1     1     1
     1     1     1
     1     1     1
```

we can obtains its dimensions,

```
>> [m,n] = size(A)

m =

     4


n =

     3
```

set every element of row 2 to 0,

```
>> A(2,:) = 0

A =

     1     1     1
     0     0     0
     1     1     1
     1     1     1
```

and then multiply the resulting matrix's third column by 2,

```
>> A(:,3) = A(:,3) * 2

A =

     1     1     2
     0     0     0
     1     1     2
     1     1     2
```

form the **indicator matrix** of those elements that are greater than 1,

```
>> indA = A > 1

indA =

     0     0     1
     0     0     0
     0     0     1
     0     0     1
```

multiply all elements that are at most 1 by the scalar value 3,

```
>> 3 * (A <= 1) .* A

ans =

     3     3     0
     0     0     0
     3     3     0
     3     3     0
```

set a submatrix to all -1,

```
>> A(2:3,1:2) = -1

A =

     1     1     2
    -1    -1     0
    -1    -1     2
     1     1     2
```

and much more. Like most high-level languages, the fun of Matlab is in writing short, clever code to accomplish a given task.

Exercise 9.2. At the Matlab command line, type `doc colon` and read the resulting documentation. □

As a few more examples, recall from Chapters 6 and 7 that our naive matrix ADT provides two methods of computing the dot product of two vectors. The same two methods are encoded in Matlab as follows:

```
>> v = [1; 2; -1];
>> v' * v

ans =

     6

>> sum(v .* v)
```



```
ans =

     6
```

Recall also the power function:

```
>> A = diag([1;2;3]); A(1,3) = 1; A(3,1) = 1/2

A =

    1.0000         0    1.0000
         0    2.0000         0
    0.5000         0    3.0000

>> A^3

ans =

    3.5000         0   13.5000
         0    8.0000         0
    6.7500         0   30.5000
```

Exercise 9.3. Write short Matlab command sequences to accomplish the following tasks:

(a) Create the following matrix:

```
A =

     0     0     0     1
     0     0     0     2
     0     0     0     3
     0     0     0     4
```

Solution. `A = zeros(4); A(:,4) = 1:4`

(b) Create the following matrix:

```
A =

     0     0     0     4
     0     0     0     3
     7     9    11     2
     0     0     0     1
```

Solution. `A = zeros(4); A(:,4) = 4:-1:1; A(3,1:3) = 7:2:11`

(c) Multiply the odd elements of a matrix by 2 and the even elements by 3 (*hint*: `help mod`); for example, applying this operation to `A` of the previous problem yields the following matrix:

```
ans =

     0     0     0    12
     0     0     0     6
    14    18    22     6
     0     0     0     2
```

Solution. The idea is to form two indicator matrices, `mod(A, 2) == 0` and `mod(A, 2) == 1`, which yield complementary 1s and 0s. The first matrix should be multiplied element-wise by `3 * A`, while the second should be multiplied element-wise by `2 * A`; the results should then be summed:

```
A .* (3 * (mod(A, 2) == 0)) + A .* (2 * (mod(A, 2) == 1))
```

(d) Create the following matrix:

```
A =

         0         0         0         0    1.0000
         0    2.5000         0         0    1.0000
         0         0    5.0000         0    1.0000
         0         0         0    7.5000    1.0000
         0         0         0         0   10.0000
```

(e) Create a table of powers of 2 of arbitrary size, for example,

```
ans =

     1     2     4     8    16    32
```

(f) Replace every element of a matrix whose value is less than -1 by -1.
Solution. The trick is to use two complementary indicator matrices:

```
(A < -1) * -1 + (A >= -1) .* A
```

The first term yields a matrix of -1s and 0s, where the -1s are at the positions at which `A` has elements less than -1. The second term yields a matrix like `A` except that each position at which `A` has an element less than -1, it has a 0 instead.

(g) Scale the negative elements of a matrix by 2.

(h) **Challenge:** Decide if a matrix is symmetric. (Use `help` to learn about `all`, `any`, `==`, and `&&`.)

□

Matlab enables easy visualization, as we'll see in several applications in the next few chapters. But to get started, consider the function $e^{-\frac{t}{5}} \cos\theta + 3$. To plot it over the interval $[0, 6\pi]$ in Matlab, simply select a sample of points in the interval:

```
>> s = 0:pi/100:6*pi; plot(s, exp(-s/5) .* cos(s) + 3);
```


The resulting plot is shown in Figure 9.1.[1] Notice that, just as `s` is a row vector, `exp(-s/5)`, `cos(s)`, `exp(-s/5) .* cos(s)`, and `exp(-s/5) .* cos(s) + 3` are all row vectors as well, which is why the element-wise operator `.*` is used.

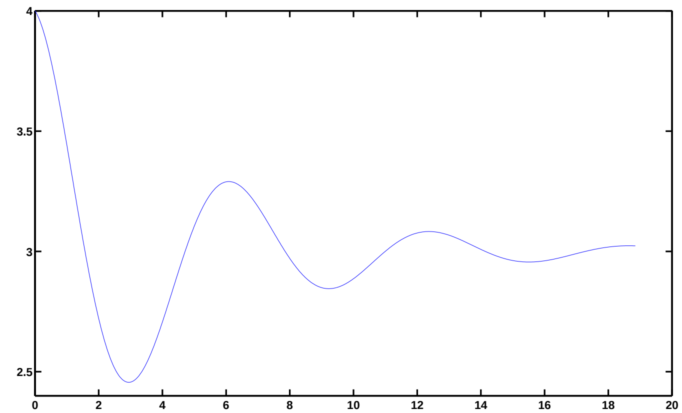


Fig. 9.1. `plot(s, exp(-s/5) .* cos(s) + 3)`

Exercise 9.4. Consider the vector function

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} e^{-t/3} \cos 3t \\ e^{-t/10} \sin t + 1 \end{bmatrix} .$$

Plot the described trajectory over the interval $[0, 10\pi]$. You should create a plot that looks similar to the one in Figure 9.2. □

Matlab is a huge system encompassing a powerful set of built-in functions, extension packages, and open-source modules from the Matlab user community. Besides learning to use Matlab, you should learn *how to learn to use a new tool*: use `help`, `doc`, and Internet search engines extensively.

9.2 Programming in Matlab

Matlab's capabilities will become more relevant to you as you advance through your academic career and learn about the engineering applications that require

[1] If the use of the colon operator, `:`, in the statement above is unfamiliar, return to Exercise 9.2.

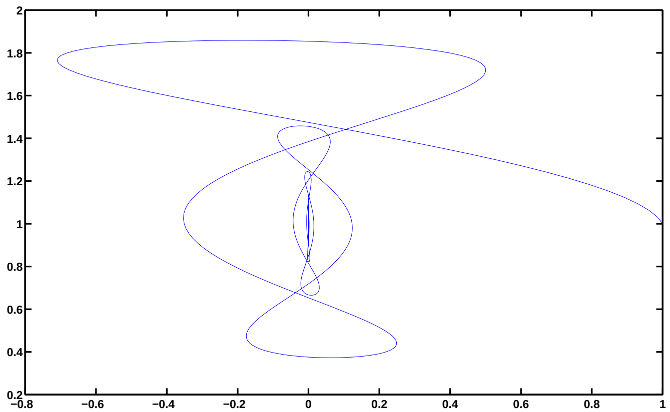


Fig. 9.2. Plot from Exercise 9.4

its computational power. However, now is an excellent time to learn how to program well in its language.

As a motivating example, we write a module, called `song.m`, that defines a function to translate a musical score into a `.wav` file that can be played by any audio player. Matlab provides a function, `wavwrite`, that converts a sampled signal into a `.wav` file, so we need only construct the signal.

Fundamentally, sound is generated by periodic mechanical motion that generates pressure waves in the surrounding medium, which propagate through the medium to strike our ear, which causes structures in our ear to vibrate accordingly, which our nervous system translates into electrical signals that our brains interpret as sound. Our appreciation of sound as music is probably a consequence of our incessant recognizing of patterns, so it is perhaps not surprising that the basic physics of music is fairly simple mathematically.

9.2.1 Generating a Pure Tone

A wave's frequency determines the pitch that we hear. For example, middle A of the modern Western chromatic scale has a frequency of 440 Hz: a pressure wave that peaks 440 times per second is interpreted by our ears as middle A. Mathematically, we can represent middle A by a **sine wave** with a frequency of 440 Hz. In general, a pure tone of frequency f corresponds to the trigonometric function $\sin(2\pi ft)$, where t ranges over time, so middle A corresponds to $\sin(2\pi \cdot 440t)$.

Computers work in discrete time, not in continuous time. Therefore, we cannot manipulate $\sin(2\pi ft)$ directly to generate sound. Instead, we **sample**

the function at some frequency—ideally at a frequency at least double that of the function that we're sampling, according to Nyquist and Shannon.

Suppose, then, that we want to produce a pure middle A tone for one second using a computer. We decide on a **sampling frequency** that is at least double the frequency of the tone but that is not so high that the computer cannot keep up. We use 8,192 Hz as our sampling frequency throughout this chapter, which is sufficient to produce music for the human ear. First we produce an array of times at which the function should be sampled:

```
>> sampleTimes = (0:8192-1)/8192;
```

This command produces a row vector of 8,192 elements (that is, a $1 \times 8{,}192$ matrix) that, in floating point, approximates the matrix

$$\left[0 \;\frac{1}{8192}\; \frac{2}{8192}\; \cdots\right] .$$

Because 8,192 is a power of two, the following statement produces the same row vector:

```
>> sampleTimes = 0:1/8192:1-1/8192;
```

However, because floating points are approximate and repeated summation yields ever larger errors, it is better to create sample vectors using `(0:nsamples-1)/nsamples` rather than `0:1/nsamples:1-1/nsamples` when `nsamples` is not a power of two.

Then we produce the samples of the function $\sin(2\pi \cdot 440t)$ at these times:

```
>> samples = sin(2 * pi * 440 * sampleTimes);
```

This command produces a row vector of length 8,192 whose elements range between -1 and 1 and approximate 440 cycles of a sine wave. Finally, we produce the tone:

```
>> wavwrite(samples', 'middle_a.wav');
```

Since `wavwrite` expects a column vector, we apply the transpose operator, `'`, to `samples`. The resulting file can be played by any music player.

Let's package these operations as a function, which we write in `tone.m`:

```
1 function rv = tone(duration, freq)
2 % Generates the sampled sine wave for the given 'freq' and
3 % 'duration'.  The sampling rate is 8192 Hz.
4   sampleTimes = (0:duration*8192-1)/8192;
5   rv = sin(2*pi*freq*sampleTimes);
6 end
```

The Matlab programming language's syntax differs from C's, but its structures are similar. In particular, line 1 declares the function `tone` to have two parameters, `duration` and `freq`, and to return a value that, at line 5, is apparently a row vector of samples. The Matlab language is **dynamically typed**, whereas C is **statically typed**. Types are "discovered" during execution, and type mismatches result in runtime errors rather than compile-time errors as in C. Notice that, because the variable `rv` is declared as the return value on line 1, there is no explicit return statement.

Comments immediately after the function header are read by Matlab's `help` command. In this case, the comments at lines 2–3 are printed if `help tone` is executed. Given Matlab's lack of static typing, it is important to describe the parameters and return value.

Running Matlab from the directory in which `tone.m` resides allows us to generate tones easily:

```
>> wavwrite(tone(1, 440)', 8192, 'middle_a.wav');
```

At a graphical Matlab console, one can also use the `sound` function, which, according to `help sound`, assumes a default sampling rate of 8,192 Hz:

```
>> sound(tone(1, 440));
```

An audible tone should play.

Listening to the tone is one way to "visualize" the function. Another is to plot the generated function:

```
>> midA = tone(1.0, 440);
>> plot(midA);
```

The result is in Figure 9.3. This plot is not terribly useful. The relevant part of the x-axis ranges over the indices of `midA`, 1 to 8,192, while values on the y-axis of course lie between -1 and 1. With a frequency of 440 Hz and a duration of 1 s, there are 440 peaks and troughs of the sine wave, explaining why we essentially have a gray box—with interesting moiré patterns, to be sure, but still rather uninformative.

We can extract a portion of the vector `midA` using range notation; for example, `midA(1:10)` selects the first 10 components. With a sampling frequency of 8,192 Hz, one cycle of the sine function is between index 1 and somewhere around $\frac{8{,}192}{440}$. We use the `ceil` function (for "ceiling") to yield the least integer greater than $\frac{8{,}192}{440}$, although it's not strictly necessary (try it without):

```
>> plot(midA(1:ceil(8192/440)));
```

The plot, displayed in Figure 9.4, is somewhat misleading because it suggests that we are plotting a continuous function. In fact, `plot` is in line mode and so is connecting the dots. The format option `'-o'` tells `plot` to draw the discrete samples as dots, in addition to adding connecting lines:

```
>> plot(midA(1:ceil(8192/440)), '-o');
```

This plot, shown in Figure 9.5, reveals 19 discrete samples.

Finally, we can get a sense of the function by plotting multiple cycles:

```
>> plot(midA(1:ceil(10*8192/440)));
```

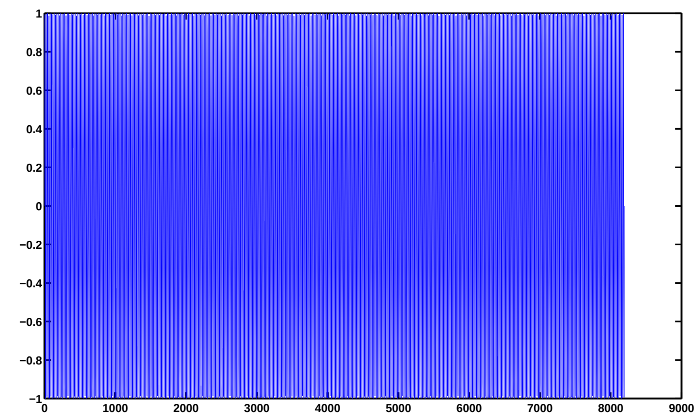



Fig. 9.3. `plot(midA)`

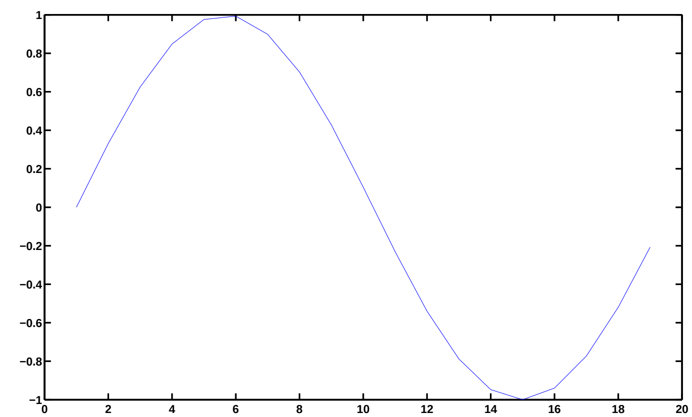


Fig. 9.4. `plot(midA(1:ceil(8192/440)))`

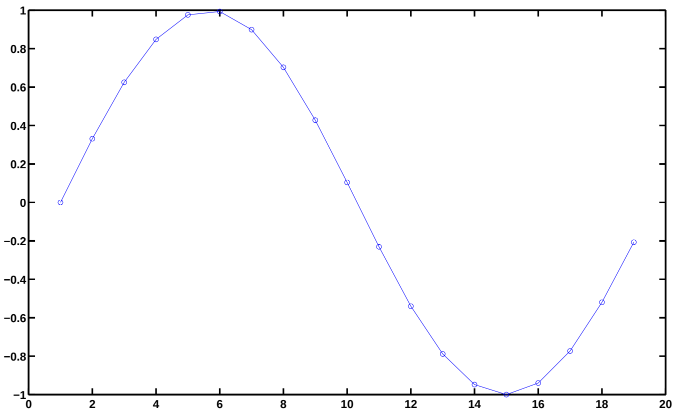


Fig. 9.5. `plot(midA(1:ceil(8192/440)), '-o')`

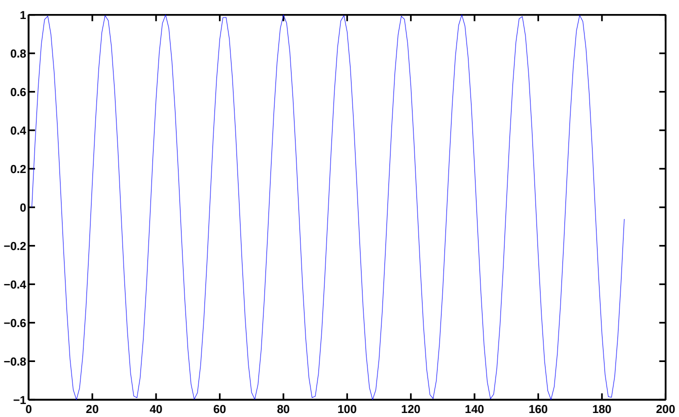


Fig. 9.6. `plot(midA(1:ceil(10*8192/440)))`

The result is displayed in Figure 9.6.

Exercise 9.5. Figures 9.3-9.6 plot sample indices versus amplitude. Instead, generate `sampleTimes` as above and plot time versus amplitude. It may help to read `help plot`, in particular, about how to plot given x values versus y values. □

Exercise 9.6. Changing the amplitude of the function affects volume. Experiment with multiplying `midA` by values between, say, 0 and 2. Visualize the results by both generating and listening to `.wav` files and plotting segments. Once scalars become boring, use the `.*` operator (element-wise multiplication) to multiply the samples element-wise by some function, such as e^{-3t} or $\cos 10t$. Generate a tone that gradually becomes quieter and another tone whose volume pulses. □

9.2.2 Making Music

While generating tones has its uses, such as in modems, it is not all that interesting. Generating music is our goal. We first introduce how to generate notes of the Western chromatic scale; we then write a Matlab module, `song.m`, that exports a function, `song`, that converts a restricted form of musical score to a sampled signal.

The Western chromatic scale is generated by raising frequencies to powers. One octave consists of the following 12 notes:

0	1	2	3	4	5	6	7	8	9	10	11
A	$A^\sharp/B^\flat$	B	C	$C^\sharp/D^\flat$	D	$D^\sharp/E^\flat$	E	F	$F^\sharp/G^\flat$	G	$G^\sharp$

An **octave** of a note with frequency f is defined as that note whose frequency is $2f$. Furthermore, the chromatic scale consists of 12 notes that are each a **semitone** different from its neighbor. In other words, the ratio r between adjacent notes is constant. If $f * r^{12} = 2f$, then it must be that $r = 2^{\frac{1}{12}}$. Therefore, to compute the frequency of a given note, we need only compute its distance (in semitones) from middle A.

For example, the $C^\sharp$ above middle A has index 4 and is thus four semitones beyond middle A. Therefore, its frequency is $440 \cdot (2^{\frac{1}{12}})^4 = 440 \cdot 2^{\frac{4}{12}}$; and its corresponding function is $\sin(2\pi \cdot 440 \cdot 2^{\frac{4}{12}} t)$. In general, note i of the octave above middle A is generated by the function $\sin(2\pi \cdot 440 \cdot 2^{\frac{i}{12}} t)$. A note i of the next octave has frequency $440 \cdot 2^{\frac{12+i}{12}} = 440 \cdot 2 \cdot 2^{\frac{i}{12}}$, while the same note of the preceding octave has frequency $440 \cdot 2^{\frac{-12+i}{12}} = 440 \cdot 2^{-1} \cdot 2^{\frac{i}{12}}$.

Exercise 9.7. Chords can be generated by summing multiple scaled sample vectors together. Generate several notes using the `tone` function of `tone.m`; then sum them together and divide by the number of notes. Examine the results both aurally and visually. □

A note can thus be characterized by its octave relative to middle A (-1 for the prior octave, 2 for two octaves higher), its index within its octave (0 to 11), and its duration (a nonnegative real number). In Matlab, a musical score can be represented by a $N \times 3$ matrix, where each of the N rows specifies a note: the first column specifies the *octave* relative to the middle octave (an integer), the second the *note* within the octave (an integer between 0 and 11, inclusive), and the third the *duration* of the tone (a nonnegative real number). Given such a row, the desired tone is the one with frequency $440 \cdot 2^{octave + \frac{note}{12}}$ and the given *duration*.

One complication is that we need to generate rests (silences of a given duration) as well. We therefore decide that, if a row's *note* column is -1, then that row specifies a rest of the given duration. Calling `tone` with the duration and a frequency of 0 generates the desired rest.

In `song.m`, we write the following code to meet this specification:

```
1  function samples = song(score)
2  % song(score)
3  %   Constructs a sampled signal corresponding to the song
4  %   specified by 'score'.  The format of 'score' is an
5  %   N x 3 matrix, where each row corresponds to a note
6  %   specification:
7  %     [ octave, note, duration ]
8  %   where
9  %    - octave specifies the number of octaves away from
10 %      middle A (440 Hz);
11 %    - note specifies one of the 12 pitches of the
12 %      chromatic scale, 0-11;
13 %    - duration specifies the length in seconds.
14 %   If the 'note' element is -1, the row specifies a rest
15 %   of the given duration and the octave specifier is
16 %   ignored.
17
18   % The matrix to hold the signal, which consists of
19   % concatenated sine wave samples.
20   samples = [];
21   % Extract the number of notes.
22   [N, width] = size(score);
23   if (width ~= 3)  % ~= is 'not equal'
24     % malformed input
25     return;
26   end
27   % For each note specification...
28   for n = 1:N
29     % ... extract the components of the specification...
30     octave = score(n, 1);
31     note = score(n, 2);
32     duration = score(n, 3);
33     % ... compute the frequency...
```



```
34     freq = 0;
35     if (note >= 0)
36       % it's not a rest
37       freq = 2^(octave + note/12) * middleA;
38     end
39     % ... and concatenate its wave.
40     samples = [samples tone(duration, freq)];
41   end
42 end
43
44 function rv = middleA
45 % Defines the frequency of middle A.  Constants in Matlab
46 % are generated in this peculiar way.
47   rv = 440;
48 end
49
50 function rv = tone(duration, freq)
51 % Generates the sampled sine wave for the given 'freq' and
52 % 'duration'.  The sampling rate is 8192 Hz.
53   sampleTimes = (0:duration*8192-1)/8192;
54   rv = sin(2*pi*freq*sampleTimes) * .999; % scale
55 end
```

The first function in a `.m` file is the only one that can be called publicly and must have the same name as the file. The other functions are only visible within the file.

While the syntax is new, I bet that you can read and understand it fairly easily, given your knowledge of C. However, there is one major, yet subtle, point that this code illustrates and that is not an issue in C: the difference between explicitly "looping" code, as in lines 28–41, and implicitly "looping"—or **vectorized**—code, as in lines 53–54. The program itself is interpreted by software, so loop iteration is orders of magnitude slower than loop iteration in C. However, vectorized functions—like `sin`—which can act on either scalar values or vector values, cause Matlab to execute highly optimized C, C++, or Fortran functions internally. A good strategy is to use explicit looping for high-level operations and implicit looping for mathematical operations.

At line 54, the generated tone is multiplied by `.999` so that the resulting function ranges between -1 and 1, exclusive. Matlab's functions `wavwrite` and `sound` expect signals within that range and produce aberrations otherwise.

In Matlab, we execute the following:

```
>> bs5 = [0 -1 1/4; 0 10 1/4; 0 10 1/4; 0 10 1/4; 0 6 1;
          0 -1 1/4; 0 8 1/4; 0 8 1/4; 0 8 1/4; 0 5 2];
>> sound(song(bs5));
```

The resulting music probably sounds familiar, if a bit unemotional.

Exercise 9.8. A note can also be characterized by its volume. Scaling a sampled function by a value between 0 and 1 yields a quieter tone. Augment `song.m` to take a score defined by an $N \times 4$ matrix, where the fourth column specifies volume. Modify the `bs5` score to produce a less-unemotional song. □

Exercise 9.9. Notes played on an instrument such as a piano fade over time. Using the ideas explored in Exercise 9.6, modify `song.m` so that each note decays over the period that it is played. □

Exercise 9.10. Write a Matlab function, `chord`, that takes two arguments: a chord specification as an $N \times 2$ matrix, where the first column specifies the octave and the second column specifies the note; and a duration in seconds. It should produce a signal sampled at 8,192 Hz of the corresponding chord. To avoid problems with clipping, the signal should be scaled to be between -1 and 1, exclusive. Use Matlab's `sound` function to play several common chords.

Solution. In `chord.m`, we implement the following function:

```
function rv = chord(spec, dur)
  % obtain number of notes in chord
  [N, width] = size(spec);
  if (width ~= 2)
    % malformed input
    rv = [];
    return
  end

  % sample times
  t = (0:8192*dur-1)/8192;
  % initialize signal and accumulate notes into it
  rv = zeros(1, length(t));
  for j = 1:N
    f = 440 * 2^(spec(j,1) + spec(j,2)/12);
    rv = rv + sin(2*pi*f*t);
  end
  % scale the signal to within (-1, 1)
  rv = rv/N * 0.999;
end
```

In general, to produce the signal corresponding to two signals being played at once, we simply have to add them; this observation is an example of the **superposition principle**. □

Exercise 9.11. For a pure tone of a given frequency f, its **harmonics** are tones at frequencies that are integer multiples of f: $2f$, $3f$, $4f$, and so on; f itself is called the **fundamental**. Playing some of the harmonics of a fundamental adds depth to the resulting sound.

Implement a Matlab function, `hchord`, that takes three arguments: the first two are as in Exercise 9.10, while the third is a row vector whose elements sum to 1. Each element specifies the contribution of a given harmonic to the overall contribution of a note of the chord.

For example, `hchord([0 0; 0 3; 0 7], 0.25, [0.7 0.05 0.15 0.1])` specifies the chord ACE, to be played for a quarter of a second, and such that each note be played with 0.7 contribution from the fundamental, 0.05 contribution from the first harmonic, 0.15 from the second, and 0.1 from the third. Hence, the note A will yield signals at frequencies 440, 880, 1,340, and 1,760 Hz, and most of its contribution will come from the 440 Hz signal.

Plot the signals for several common chords; compare them to the signals produced by the `chord` function of Exercise 9.10. Use Matlab's `sound` function to play the signals; try various harmonic specifications until you find one that is pleasing. □

Exercise 9.12. Augment `song.m` to generate songs with chords. □

Exercise 9.13. More interesting tones can be created by playing a fundamental with tones—called **overtones** instead of harmonics—that are very slightly different from its harmonics. Develop a specification for these differences, and implement a Matlab function to generate chords with off-harmonic overtones. Experiment to find a pleasing result. □

Exercise 9.14. Use `help` to learn about the `rand` and `floor` functions. For example, `floor(12 * rand)` generates a random integer between 0 and 11, inclusive. Write a function to generate random music. Try various strategies to yield more pleasing results. For example, one method of creating melodic music is to construct an overall structure to the piece by randomly assembling a set of standard chord progressions for a given key. This structure directly yields the harmony. Then add the melody by sampling within each chord progression. Use techniques from Exercises 9.6, 9.8, 9.11, and 9.13 to add complexity to the music. □

10

Exploring ODEs with Matlab

Many physical processes, both natural and engineered, are best described by **ordinary differential equations** (**ODEs**), which relate time derivatives of particular quantities to each other. A mathematics course on ODEs would likely focus on developing techniques to solve ODEs analytically. But computers offer the option of solving ODEs numerically for fixed initial values. As you will see, numerical methods actually provide an enlightening perspective on ODEs. Solving an ODE numerically is sometimes called "simulating" it, so numerical methods can be seen as a methodology for programming simulations of physical processes.

In this chapter, we continue our exploration of Matlab in the context of numerical methods. From two well-known physical laws—Newton's second law of motion ($F = ma$) and Newton's law of universal gravitation ($F = G\frac{Mm}{r^2}$)—we develop an ODE to describe the orbits of satellites around planets. We then study and apply various numerical methods to solve numerically for an orbit given a satellite's initial position and velocity. Our explorations will yield one universal truth of numerical methods: no one method works best on all problems. In order to determine which is the best for this application, we will rely on some common sense reasoning to make predictions about what we expect to see for certain initial conditions.

10.1 Developing an ODE Describing Orbits

10.1.1 Developing the ODE

Consider the following two equations describing physical laws: Newton's second law of motion relating force (F), mass (m), and acceleration (a),

$$F = ma \; ;$$

and Newton's law of universal gravitation between two point masses, M and m, where G is the gravitational constant and r is the distance between the centers of the two masses,

$$F = G\frac{Mm}{r^2} \; .$$

These equations should be familiar from a physics course.

In the two-dimensional setting, F, a, and r are two-dimensional vectors. Let the mass M be at the origin, and let x be the position (in two dimensions) of m relative to M. Then $\dot{x}$, which is sometimes written $\frac{\mathrm{d}x}{\mathrm{d}t}$, denotes the velocity of mass m relative to M; and $\ddot{x}$, which is sometimes written $\frac{\mathrm{d}^2x}{\mathrm{d}t^2}$, denotes the acceleration of mass m relative to M. We can thus write the second law of motion as

$$F = m\ddot{x} \; ,$$

and the law of universal gravitation as

$$F = -G\frac{Mm}{x'x}\frac{x}{|x|} = -G\frac{Mm}{x'x}\frac{x}{\sqrt{x'x}} = -G\frac{Mm}{(x'x)^{\frac{3}{2}}}x \; ,$$

where x' is the transpose of x, as in Matlab. In the universal law of gravitation, we simply multiply the scalar value given by $G\frac{Mm}{r^2} = G\frac{Mm}{x'x}$ by the unit vector $\frac{-x}{|x|}$, where $|x| = \sqrt{x'x} = \sqrt{x_1^2 + x_2^2}$, which indicates the direction of the force—toward the central mass.

Notice that these equations are now vector equations:

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = m \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = G\frac{Mm}{(x_1^2 + x_2^2)^{\frac{3}{2}}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \; .$$

Setting their right sides equal and dividing out m yields

$$\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} = G\frac{M}{(x_1^2 + x_2^2)^{\frac{3}{2}}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \; ,$$

or, more concisely,

$$\ddot{x} = -G\frac{M}{(x'x)^{\frac{3}{2}}}x \; .$$

This final equation is an ordinary differential equation (ODE) relating a satellite's position x, relative to a point mass M, to its acceleration $\ddot{x}$. In particular, the force that M exerts on the satellite m is towards it—recall that M is at the origin, so that $-x$ is the vector pointing from the satellite's location (also x) to M—and proportional in magnitude to $G\frac{M}{(x'x)^{\frac{3}{2}}}$. Via $F = m\ddot{x}$, this force manifests itself as acceleration $\ddot{x}$ on the satellite.

As a simplification, we will choose units so that $GM = 1$, yielding our final ODE:

$$\ddot{x} = -(x'x)^{-\frac{3}{2}}x \; . \tag{10.1}$$

While the satellite's position x and acceleration $\ddot{x}$ are explicit in the equation, its velocity is not. Yet for a fixed position, the satellite's velocity has a major influence on where the satellite goes next. We must thus specify the **initial condition** of the satellite: its initial velocity and position. Thereafter, the ODE determines its path, as $\ddot{x}$ influences $\dot{x}$, and $\dot{x}$ influences x.

10.1.2 Converting into a System of First-Order ODEs

ODE (10.1) is of second order: it relates acceleration (the second derivative of position) to position. For some applications, including numerical solving, it is better to present such an ODE as a **system of first-order ODEs**. Doing so is simple. Rather than looking at the problem in two (position) dimensions and taking the second derivative, we instead look at the problem in four dimensions—two for position and two for velocity—and take only a first derivative.

Let y be a four-dimensional vector. We relate y to x as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} . \tag{10.2}$$

That is, the first two elements of y describe the position of the satellite, and the latter two elements of y describe the velocity of the satellite.

With Definition (10.2), we can build $\dot{y}$. First, by stepping through the definition, we find that $\dot{y}_1 = \dot{x}_1 = y_3$ and that $\dot{y}_2 = \dot{x}_2 = y_4$. In words, components y_3 and y_4 describe the satellite's velocity, as expected. Second, $\dot{y}_3 = \ddot{x}_1$ and $\dot{y}_4 = \ddot{x}_2$ form the acceleration vector of the satellite, which is given by ODE (10.1) in terms of the satellite's position. After translating x_1 to y_1 and x_2 to y_2 according to Definition (10.2), the result is the following:

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \\ \dot{y}_4 \end{bmatrix} = \begin{bmatrix} y_3 \\ y_4 \\ -(y_1^2 + y_2^2)^{-\frac{3}{2}} y_1 \\ -(y_1^2 + y_2^2)^{-\frac{3}{2}} y_2 \end{bmatrix} , \tag{10.3}$$

which is a system of first-order ODEs. We numerically solve this system throughout the remainder of the chapter.

We can encode this ODE as a function in Matlab:

```
function ydot = orbit(t, y)
% Returns the vector of the derivative of y at time t.
  r = sqrt(y(1:2)' * y(1:2));
  ydot = [y(3);              ... % an ellipsis continues the
          y(4);              ... % statement to the next line
          -1/r^3 * y(1); ...
          -1/r^3 * y(2)];
end
```

This function adheres to a standard way of encoding ODEs in Matlab: a time `t` and a vector `y` are given, and the first-derivative vector `ydot` is returned. In ODE (10.3), t does not appear explicitly and so the argument `t` is not used.

10.2 Numerical Integration

For a given initial position and velocity of the satellite, we wish to determine its future positions and velocities. That is, we would like to plot its orbit around the central mass—or determine that it does not enter into an orbit but instead shoots off into space. Furthermore, we would like to plot its velocity over time. Ideally, we would like to observe circular and elliptical orbits as well as parabolic paths leading into the depths of space. In short, we would like to integrate $\dot{y}$ over time.

We derive our first numerical method simply by understanding what the system of ODEs says. Consider the vector $y(t)$ as detailing the **state** of the satellite, which consists of its position and its velocity, at time t. Its state in the "next time instant"—if we can so discretize time—is determined by its current state and the influence of the central mass, as ODE (10.3) so clearly show: how y changes is given by $\dot{y}$. Suppose that time moves discretely in increments of ΔT. Then one way of estimating the state after ΔT units of time—that is, at time $t + \Delta T$, if t is the current time—based on $y(t)$ is by assuming that $y(t)$ changes at the constant rate of $\dot{y}(t)$ throughout the period $[t,\ t + \Delta T]$. Hence,

$$y(t + \Delta T) = y(t) + \dot{y}(t)\Delta T .$$

In words, an estimate for the state at time $t + \Delta T$ is the current state plus the rate of change at time t times the period ΔT. For system (10.3), the estimate is the following:

$$\begin{aligned} \begin{bmatrix} y_1(t+\Delta T) \\ y_2(t+\Delta T) \\ y_3(t+\Delta T) \\ y_4(t+\Delta T) \end{bmatrix} &= \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \\ y_4(t) \end{bmatrix} + \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \dot{y}_3(t) \\ \dot{y}_4(t) \end{bmatrix} \Delta T \\ &= \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \\ y_4(t) \end{bmatrix} + \begin{bmatrix} y_3(t) \\ y_4(t) \\ -(y_1(t)^2 + y_2(t)^2)^{-\frac{3}{2}} y_1(t) \\ -(y_1(t)^2 + y_2(t)^2)^{-\frac{3}{2}} y_2(t) \end{bmatrix} \Delta T . \end{aligned}$$

This method is known as **Euler's method**, after Leonhard Euler, an 18th century mathematician.

All that remains is to implement Euler's method into Matlab and then apply it to our system, as previously encoded in the function `orbit`. We implement the following function in `euler_solve.m`:

```
function sol = euler_solve(ydot, init, t)
% Input:
%   ydot - a function for computing ydot given t and y
%   init - the initial condition
%   t    - a row vector of times at which to solve
% Output: a length(init) x length(t) matrix giving the
```

```
 7 %  solutions at the specified time steps t
 8   n = length(init);   % determine the dimension
 9   steps = length(t);  % determine how many time steps
10   % create solution matrix
11   % #columns is # of discrete time steps
12   % #rows is # of dimensions
13   sol = zeros(n, steps);
14 
15   % at time t(1), the state is init
16   sol(:,1) = init;
17   % iterate through time
18   for i = 1:length(t)-1
19     % add slope * time-step to current values
20     sol(:,i+1) = sol(:,i) + ...
21       (t(i+1)-t(i)) * ydot(t(i), sol(:,i));
22   end
23 end
```

Recall from Exercise 9.1 that the built-in `length` function returns the length of a row or a column vector and that `zeros` makes a 0-matrix of the given number of rows and columns. Also recall from Exercise 9.2 that Matlab takes indexing to a new level with inline matrix slices: `sol(:,1)` refers to column 1 of matrix `sol`, and lines 20–21 use matrix slicing in both read and write contexts. For vectors, like `t`, only one index is required.

Having implemented a basic numerical method, we now employ it to plot orbits. We implement the following functions in `plot_orbit.m`; recall that only the function `plot_orbit` itself is callable from outside.

```
 1 function plot_orbit(y0, T, s, solve)
 2 % Input:
 3 %  y0    - the initial state of the satellite
 4 %  T     - the maximum time to solve to
 5 %  s     - the step size (delta-T)
 6 %  solve - a function to a solver
 7 % Plots the orbit and the velocity vs. time for the
 8 % satellite system, using the provided solver.
 9 
10   % Solve the system.
11   sol = solve(@orbit, y0, 0:s:T);
12 
13   % Clear the plot window.
14   clf;
15   % Plot the orbit and the velocity w.r.t. time.
16   % plot 1: the orbit
17   subplot(2, 1, 1);
18   hold on;
19   title('Position');
20   xlabel('X');
21   ylabel('Y');
```



```
22   % central mass
23   plot([0], [0], 'or');
24   % orbit
25   plot(sol(1,:), sol(2,:), '-b');
26   axis('equal');
27   % plot 2: the velocity w.r.t. time
28   v = velocity(sol);
29   subplot(2, 1, 2);
30   hold on;
31   title('Velocity');
32   xlabel('Time');
33   ylabel('Absolute velocity');
34   plot(0:s:T, v, '-b');
35 end
36 
37 function ydot = orbit(t, y)
38 % Returns the vector of the derivative of y at time t.
39   r = sqrt(y(1:2)' * y(1:2));
40   ydot = [y(3);           ... % an ellipsis continues the
41           y(4);           ... % statement to the next line
42           -1/r^3 * y(1); ...
43           -1/r^3 * y(2)];
44 end
45 
46 function V = velocity(sol)
47 % Returns the vector of the velocities of the satellite at
48 % the timesteps.
49   V = sqrt(sol(3,:).*sol(3,:) + sol(4,:).*sol(4,:));
50 end
```

The function `plot_orbit` takes four arguments: the initial condition, the period over which to solve, the step size (ΔT), and a **function handle** to the solver to use. So far we have only implemented `euler_solve`, but we will explore other methods in the next section. A function handle is conceptually like a function pointer in C: the user of `plot_orbit` is expected to provide a solver, and the function calls it (`solve`) at line 11. In fact, the `solve` function itself requires a function handle to the function that describes the ODE, which is `orbit` in our case.[1]

This program uses many built-in functions. Rather than describing them here, let me encourage you once again to use Matlab's `help` function: `help subplot`, `help hold`, etc. It also uses complex matrix slicing. Line 25, for example, plots the first row of the solution matrix, which has four rows and as many columns as requested time steps, against the second row of the solution matrix.

Figure 10.1 displays the result of an invocation with initial condition

[1] Invoke `help punct` to read about punctuation-based operators in Matlab, in particular the use of `@` as the operator to pass a function handle.

$$y_0 = \begin{bmatrix} 0 \\ 100 \\ -.1 \\ 0 \end{bmatrix} ,$$

that is, with initial position $(0, 100)$ and initial velocity $(-.1, 0)$. The use of `axis('equal')` at line 26 reveals that the orbit is possibly circular—except that the satellite is apparently spiraling away from the central mass. Common sense indicates that this predicted behavior cannot possibly be right.

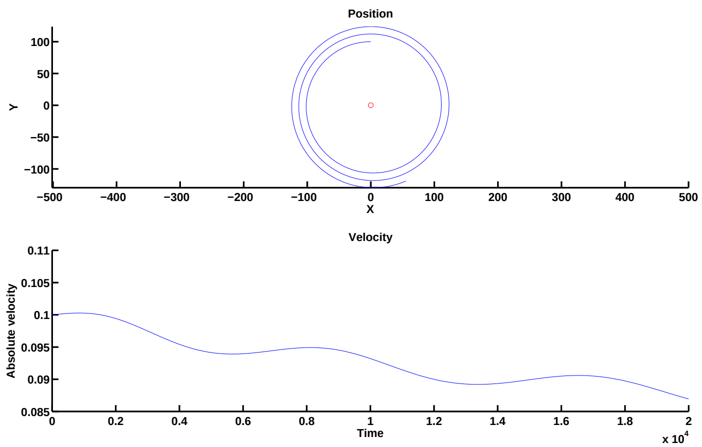


Fig. 10.1. `plot_orbit([0; 100; -.1; 0], 20000, 10, @euler_solve)`

10.3 Comparing Numerical Methods

Numerical methods differ. Euler's method, it turns out, yields a bizarre "solution" for system (10.3), one in which (as explored in Exercise 10.2) the satellite gains energy over time. In this section, we explore two other relatively simple numerical methods that are known to yield good results on Hamiltonian systems like (10.3): the semi-implicit Euler method, also known as the symplectic method, and the leapfrog method.

Euler's method uses the current state's position, velocity, and acceleration to predict the satellite's trajectory over the next ΔT time units. The **symplectic Euler method**, in contrast, uses a two-phase approach:

- It uses the current-state acceleration to predict the velocity at the end of ΔT time units.

- It then uses the new velocity to predict the position after ΔT time units.

It is "semi-implicit" in that next-state information appears on both sides of the equations that describe the method.

In detail, the method works as follows. First it computes the next-state velocity:

$$\begin{bmatrix} y_3(t+\Delta T) \\ y_4(t+\Delta T) \end{bmatrix} = \begin{bmatrix} y_3(t) \\ y_4(t) \end{bmatrix} + \begin{bmatrix} \dot{y}_3(t) \\ \dot{y}_4(t) \end{bmatrix} \Delta T \,. \tag{10.4}$$

Then it computes the next-state position using the next-state velocity:

$$\begin{bmatrix} y_1(t+\Delta T) \\ y_2(t+\Delta T) \end{bmatrix} = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} + \begin{bmatrix} \dot{y}_1(t+\Delta T) \\ \dot{y}_2(t+\Delta T) \end{bmatrix} \Delta T \,.$$

For any system obtained via the transformation to a system of first-order ODEs, the first-order terms on the right side can be replaced (see Definition (10.2)):

$$\begin{bmatrix} y_1(t+\Delta T) \\ y_2(t+\Delta T) \end{bmatrix} = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} + \begin{bmatrix} y_3(t+\Delta T) \\ y_4(t+\Delta T) \end{bmatrix} \Delta T \,. \tag{10.5}$$

Now, even though the right side of (10.5) refers to information from the next time step, that information is already available from (10.4). In some applications of the semi-implicit method and in applications of fully implicit methods, solving linear equations is required at each step.

The following code, in `symplecticEuler_solve.m`, implements the numerical method described by Equations (10.4) and (10.5) in Matlab. Unlike `euler_solve`, this implementation is dimension dependent: lines 11, 17, and 19–20 only work for a system of ODEs configured like ours, that is, in which y consists of two position elements followed by two velocity elements.

```
1  function sol = symplecticEuler_solve(ydot, init, t)
2  % Input:
3  %  ydot - a function for computing ydot given t and y
4  %  init - the initial condition
5  %  t    - a row vector of times at which to solve
6  % Output: a length(init) x length(t) matrix giving the
7  %  solutions at the specified time steps t
8    steps = length(t);
9    % only works for this problem: 2 position dimensions,
10   % 2 velocity dimensions
11   sol = zeros(4, steps);
12
13   sol(:,1) = init;
14   for i = 1:length(t)-1
15     dot = ydot(t(i), sol(:,i));
16     % Compute the next-time velocity...
17     sol(3:4,i+1) = sol(3:4,i) + (t(i+1)-t(i)) * dot(3:4);
18     % ... and use to compute the next-time position.
19     sol(1:2,i+1) = sol(1:2,i) + ...
```

```
      (t(i+1)-t(i)) * sol(3:4,i+1);
  end
end
```

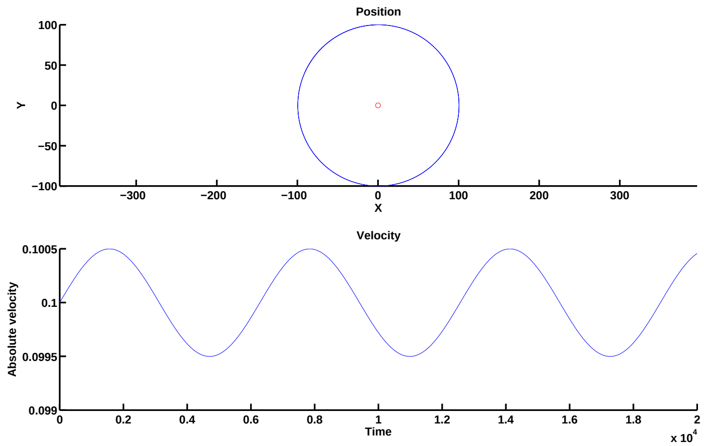


Fig. 10.2. `plot_orbit([0;100;-.1;0], 20000, 10, @symplecticEuler_solve)`

Figure 10.2 illustrates the result of calling `plot_orbit` using `symplecticEuler_solve` instead of `euler_solve` with the same initial conditions as in Figure 10.1. The orbit is now clearly circular. However, the velocity oscillates around 0.1 by a small amount, whereas common sense indicates that the velocity of a satellite in a circular orbit should be constant.

Euler's method and the symplectic Euler method are both **first-order** numerical methods, as their defining equations relate quantities separated only by one derivative: acceleration updates velocity, and velocity updates position. The next method we examine is of second order because it relates position, velocity, and acceleration in a single equation. It is called the **leapfrog method** because the computation of position and velocity "leapfrog" over each other in time.

Like the symplectic Euler method, the next-state values are computed in two phases. Compared with the two previous methods, the major difference in this new method is the use of acceleration in computing the next-state position:

$$\begin{aligned}\begin{bmatrix} y_1(t+\Delta T) \\ y_2(t+\Delta T)\end{bmatrix} &= \begin{bmatrix} y_1(t) \\ y_2(t)\end{bmatrix} + \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t)\end{bmatrix}\Delta T + \begin{bmatrix} \ddot{y}_1(t) \\ \ddot{y}_2(t)\end{bmatrix}\frac{\Delta T^2}{2} \\ &= \begin{bmatrix} y_1(t) \\ y_2(t)\end{bmatrix} + \begin{bmatrix} y_3(t) \\ y_4(t)\end{bmatrix}\Delta T + \begin{bmatrix} \dot{y}_3(t) \\ \dot{y}_4(t)\end{bmatrix}\frac{\Delta T^2}{2}\,.\end{aligned} \tag{10.6}$$

Notice how Definition (10.2) allows us to replace $\ddot{y}_1(t)$ with $\dot{y}_3(t)$ and similarly for $\ddot{y}_2(t)$, which is essentially independent of the form of the original ODE. That is, converting any other 2D second-order system to a first-order system would yield the same equation $\ddot{y}_1 = \dot{y}_3$. Also notice how the acceleration component is multiplied by $\frac{\Delta T^2}{2}$, intuitively corresponding to the fact that acceleration is the second derivative of position.

With the next-state position computed, the next phase is to compute the next-state velocity. In these equations, the average of the current-state and the next-state accelerations is used:

$$\begin{bmatrix} y_3(t+\Delta T) \\ y_4(t+\Delta T)\end{bmatrix} = \begin{bmatrix} y_3(t) \\ y_4(t)\end{bmatrix} + \left(\begin{bmatrix} \dot{y}_3(t) \\ \dot{y}_4(t)\end{bmatrix} + \begin{bmatrix} \dot{y}_3(t+\Delta T) \\ \dot{y}_4(t+\Delta T)\end{bmatrix}\right)\frac{\Delta T}{2}\,. \tag{10.7}$$

The right side refers to $\dot{y}_3(t+\Delta T)$ and $\dot{y}_4(t+\Delta T)$, which have not yet been computed. However, expanding the first-derivative terms according to ODE (10.3) reveals that the necessary information is indeed available from (10.6):

$$\begin{bmatrix} y_3(t) \\ y_4(t)\end{bmatrix} + \left(\begin{matrix}\begin{bmatrix} -(y_1(t)^2+y_2(t)^2)^{-\frac{3}{2}}y_1(t) \\ -(y_1(t)^2+y_2(t)^2)^{-\frac{3}{2}}y_2(t)\end{bmatrix} \\ \\ +\begin{bmatrix} -(y_1(t+\Delta T)^2+y_2(t+\Delta t)^2)^{-\frac{3}{2}}y_1(t+\Delta T) \\ -(y_1(t+\Delta T)^2+y_2(t+\Delta t)^2)^{-\frac{3}{2}}y_2(t+\Delta T)\end{bmatrix}\end{matrix}\right)\frac{\Delta T}{2}\,.$$

Because the expansion based on system (10.3) is required, we expect to see two calls to `ydot` per iteration in the Matlab implementation of this method.

The following function implements the leapfrog method as described by Equations (10.6) and (10.7):

```
function sol = leapfrog_solve(ydot, init, t)
% Input:
%   ydot - a function for computing ydot given t and y
%   init - the initial condition
%   t    - a row vector of times at which to solve
% Output: a length(init) x length(t) matrix giving the
%   solutions at the specified time steps t
  steps = length(t);
  % only works for this problem: 2 position dimensions,
  % 2 velocity dimensions
  sol = zeros(4, steps);

  sol(:,1) = init;
  for i = 1:length(t)-1
    step = t(i+1)-t(i);
```

```
    % Compute next-time position using
    %  1. current-time velocity
    %  2. current-time acceleration
    dot1 = ydot(t(i), sol(:,i));
    sol(1:2,i+1) = sol(1:2,i) + step * sol(3:4,i) + ...
      step*step/2 * dot1(3:4);
    % Compute next-time velocity using
    %  1. current-time acceleration
    %  2. next-time acceleration (which requires next-time
    %     position from above)
    dot2 = ydot(t(i+1), sol(:,i+1));
    sol(3:4,i+1) = sol(3:4,i) + ...
      step * (dot1(3:4) + dot2(3:4))/2;
  end
end
```

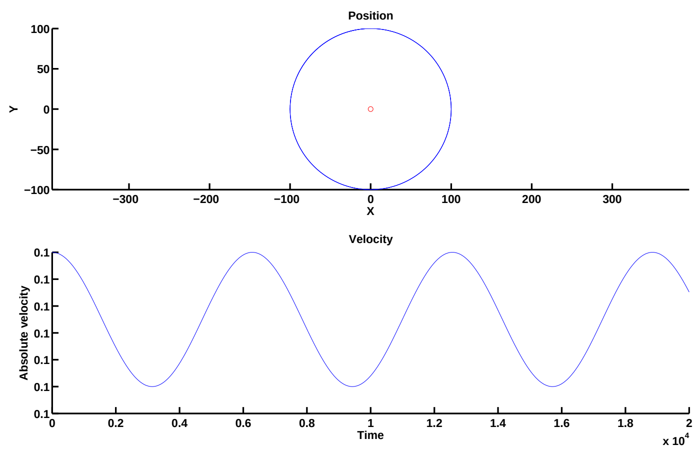


Fig. 10.3. `plot_orbit([0; 100; -.1; 0], 20000, 10, @leapfrog_solve)`

Figure 10.3 illustrates the result of calling `plot_orbit` using `leapfrog_solve` with the same initial conditions as in Figure 10.1. The orbit is again clearly circular, and the vertical scale of the velocity plot indicates that the oscillations around a velocity of 0.1 are much smaller in amplitude than the oscillations produced by the symplectic Euler method. Exercise 10.1 confirms this observation.

As a final point of comparison, the following function, in `matlab_solve.m`, provides access to Matlab's built-in function, `ode15s`, using the same arguments as our numerical methods:


```
function sol = matlab_solve(ydot, init, t)
% Input:
%  ydot - a function for computing ydot given t and y
%  init - the initial condition
%  t    - a row vector of times at which to solve
% Output: a length(init) x length(t) matrix giving the
%  solutions at the specified time steps t
  % ignore the time-step part of the output (dummy)
  [dummy, sol] = ode15s(ydot, t, init);
  % transpose it to be like the output of the other solvers
  sol = sol';
end
```

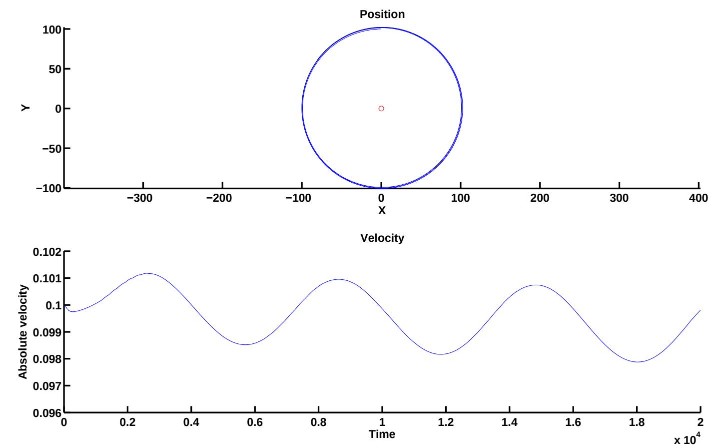


Fig. 10.4. `plot_orbit([0; 100; -.1; 0], 20000, 10, @matlab_solve)`

Figure 10.4 illustrates the result of calling `plot_orbit` using `matlab_solve` with the same initial conditions as in Figure 10.1. The velocity plot indicates a downward trend—not what common sense predicts. Invoking `help ode15s` reveals that Matlab has a quiver of ODE solvers: `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode45`, `ode23`, etc. An expert in numerical methods knows the advantages and disadvantages of each. Clearly, `ode15s` is not the right method for our application, although it does much better than Euler's method.

Exercise 10.1. Modify `plot_orbit` to create a function `compare` that plots the results of all four methods on the same position and velocity plots. Read `help plot` to learn how to specify the line characteristics. The result should be similar to the plots in Figure 10.5 for the specified initial condition.

Try a variety of initial conditions. Find initial conditions for which the numerical methods yield strikingly different qualitative results—for example, certain initial conditions cause Euler's method to predict a parabolic (non-orbital) trajectory, contrary to the predictions of the other methods. Describe what happens as the step size is varied. □

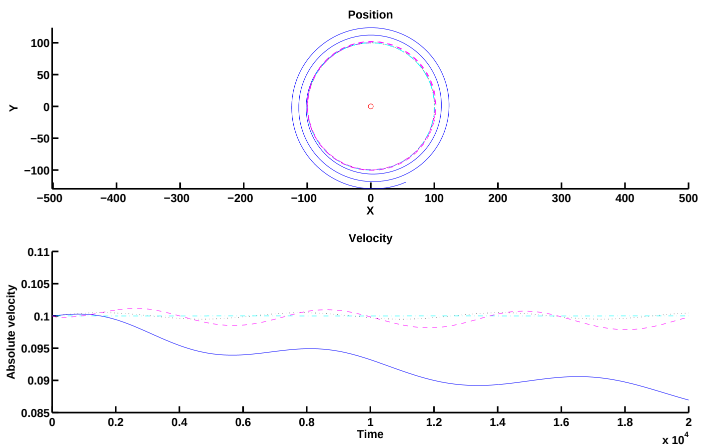


Fig. 10.5. `compare([0; 100; -.1; 0], 20000, 10)`

Having established that the leapfrog method is seemingly the best among the four numerical methods for our specific application, we can explore further qualitative characteristics of the satellite–central mass system. In particular, Figure 10.6 displays an elliptical orbit, while Figure 10.7 reveals a parabolic trajectory in which a space probe's course is influenced by the central mass, yet the mass fails to capture the probe into an orbit.

Exercise 10.2. The energy of the satellite–central mass system is given by

$$E = \frac{1}{2}m\dot{x}'\dot{x} - G\frac{Mm}{\sqrt{x'x}}\ ,$$

that is, the sum of the kinetic and the potential energies, where x is the position vector of system (10.1). Using our assumption that $GM = 1$ and factoring out m, the energy of the system is proportional to the quantity

$$E \propto \frac{1}{2}\dot{x}'\dot{x} - \frac{1}{\sqrt{x'x}}\ .$$

In an ideal system, energy should remain constant, and this ideal approximation works well in practice for orbital mechanics. Modify `plot_orbit` to

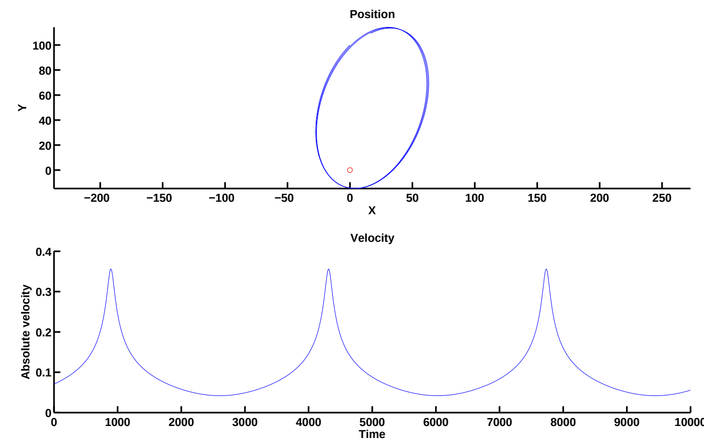


Fig. 10.6. `plot_orbit([0; 100; -.05; -.05], 10000, 10, @leapfrog_solve)`

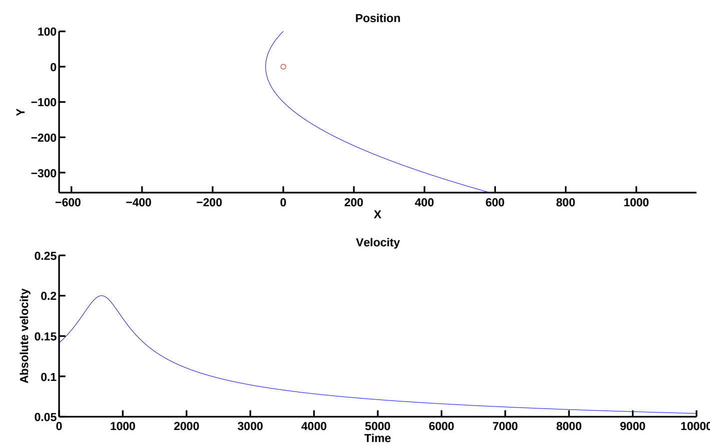


Fig. 10.7. `plot_orbit([0; 100; -.1; -.1], 10000, 10, @leapfrog_solve)`

generate a third graph that illustrates energy over time. Try the various numerical methods on several initial conditions. Which of the methods best captures the expected ideal behavior? □

Exercise 10.3. Modify `plot_orbit` and at least one of the numerical methods `symplecticEuler_solve` or `leapfrog_solve` to solve the three-dimensional version of the satellite–central mass system. First, derive the first-order system of ODEs for the three-dimensional system, which should have three position components and three velocity components. Then modify the `orbit` function, which encodes the system of ODEs into Matlab, to reflect the changes. Next, use `help plot3` to learn the basic features of Matlab's 3D plotting capabilities, and modify the remainder of `plot_orbit.m` accordingly; test the modifications using `matlab_solve`, which is dimension independent. Finally, modify one of `symplecticEuler_solve` or `leapfrog_solve`. As an example, Figure 10.8 displays the result for the given initial condition. □

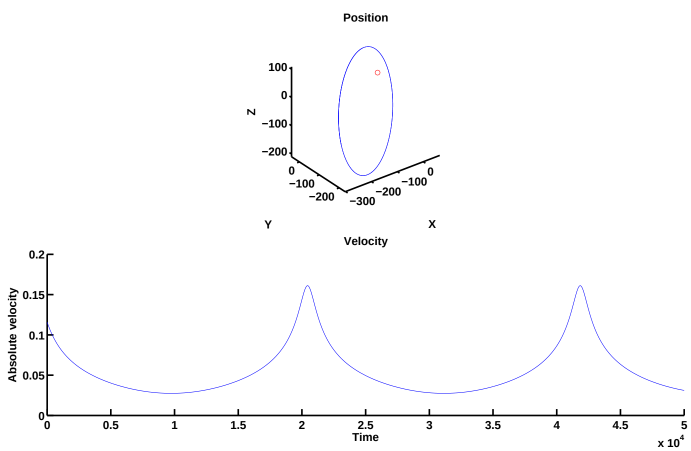


Fig. 10.8. `plot_orbit3d([-50; 10; 100; -.1; -.05; .03], 50000, 10, @leapfrog3d_solve)`

Exercise 10.4. Consider the following specification of a numerical method:

$$\begin{bmatrix} y_1(t+\Delta T) \\ y_2(t+\Delta T) \end{bmatrix} = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} + \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \end{bmatrix} \Delta T + \begin{bmatrix} \ddot{y}_1(t) \\ \ddot{y}_2(t) \end{bmatrix} \frac{\Delta T^2}{2}$$

and

$$\begin{bmatrix} y_3(t+\Delta T) \\ y_4(t+\Delta T) \end{bmatrix} = \begin{bmatrix} y_3(t) \\ y_4(t) \end{bmatrix} + \begin{bmatrix} \dot{y}_3(t) \\ \dot{y}_4(t) \end{bmatrix} \Delta T \,.$$

Implement it in Matlab, and compare it with the other methods explored in this chapter in the context of the orbit system. □

11

Exploring Time and Frequency Domains with Matlab

Physical processes often evolve periodically over time, making frequency-domain analysis a powerful engineering tool for characterizing and designing a system's behavior. This chapter introduces the basic concepts of the time domain, the frequency domain, and transformations between the two in the context of our continuing study of Matlab. Subsequent engineering courses study the subject in great depth, so our goal is to use Matlab to develop a foundational understanding.

11.1 Time and Frequency Domains

A graph of a signal in the **time domain** plots the amplitude of a signal against time. For example, consider the discretely sampled A major chord: on a guitar, it consists of (in descending order) E, $C^\sharp$, and A (at 440 Hz), and, from one octave lower, E and A. From our study of the Western chromatic scale in Chapter 9, we calculate the following frequencies:

$$\begin{aligned}
E &= 440 \cdot 2^{\frac{6}{12}} \approx 622 \text{ Hz} \\
C^\sharp &= 440 \cdot 2^{\frac{4}{12}} \approx 554 \text{ Hz} \\
A &= 440 \text{ Hz} \\
E &= 440 \cdot 2^{-1+\frac{7}{12}} \approx 330 \text{ Hz} \\
A &= 440 \cdot 2^{-1} = 220 \text{ Hz}
\end{aligned}$$

As usual, let us assume a sampling rate of 8,192 Hz. In Matlab, we carefully construct exactly 8,192 sample times over one second:

```
>> t = (0:8192-1)/8192;
```

We then construct the signal of the A major chord with a duration of one second:

```
>> f = (sin(2*pi*622*t) + sin(2*pi*554*t) + ...
        sin(2*pi*440*t) + sin(2*pi*330*t) + ...
        sin(2*pi*220*t)) / 5;
```


We divide by 5 so that the overall signal is normalized to have a maximum absolute amplitude of 1. Plotting the signal in the time domain,

```
>> plot(t, f)
```

yields the graph in Figure 11.1; a more instructive plot is obtained by plotting only a portion of the signal,

```
>> plot(t(1:128), f(1:128))
```

as displayed in Figure 11.2. To hear the chord, use Matlab's `sound` function, which assumes a sampling rate of 8,192 Hz:

```
>> sound(f);
```

Exercise 11.1. For contrast, construct and play the A minor chord, which is similar to the A major chord, except that the $C^\sharp$ is instead a C. □

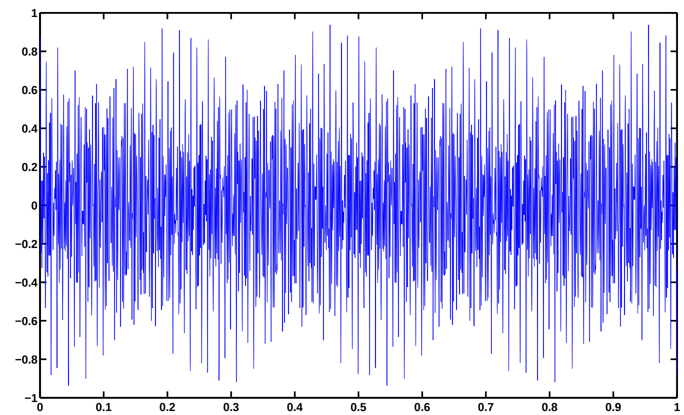


Fig. 11.1. `plot(t, f)`

The time-domain plots are "interesting" at best and a mess at worst. You might think that there *must* be a more informative way of visualizing signals—and you would be right!

In the **frequency domain**, one plots the amplitudes of discrete frequencies. In the case of the A major chord, we would hope that its frequency-domain plot would reveal its component notes. We will later get into the mathematics of constructing the frequency-domain plot, but for now let's simply put Matlab to work:

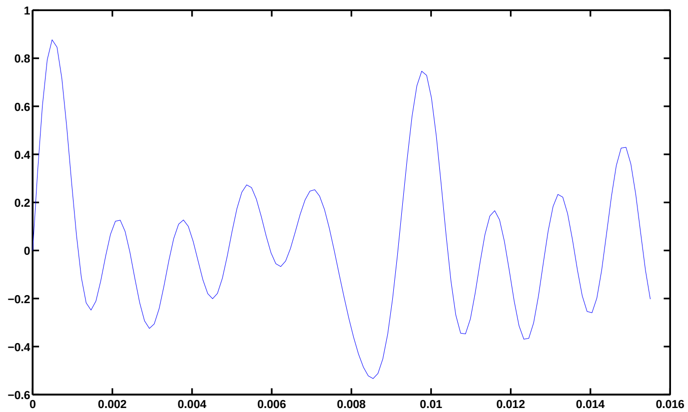


Fig. 11.2. `plot(t(1:128), f(1:128))`

```
>> F = fft(f);
>> ssas = abs([F(1) 2*F(2:4096)])/8192;
>> plot(0:4095, ssas);
```

The resulting plot, called the **single-sided amplitude spectrum** (explaining the variable name `ssas`), is shown in Figure 11.3. The units of the x-axis are Hz; the y-axis, while without units, shows the magnitude of the contribution of each frequency. A single-sided amplitude spectrum shows the amplitudes of component frequencies between 0—a signal without periodicity, sometimes called **DC** for **direct current**—and about one-half of the number of samples, $\frac{8{,}192}{2} - 1$ in this case. Hence, the x-axis actually has the units of "cycles per sample period." Because the sample period is 1 second and is sampled at 8,192 Hz in our case, we end up with the units of Hz.

Figure 11.4 shows a zoomed view of the plot in Figure 11.3 so as to reveal the frequencies at which the function is nonzero. Rather satisfyingly, the plot reveals five frequencies with amplitude 0.2—which makes sense when you recall that we divided the sum of five magnitude-one sine functions by 5. Moreover, the five frequencies are exactly those of the A major chord—recovered from analyzing a time-sampled trigonometric function.

Exercise 11.2. Plot the single-sided amplitude spectrum for the A minor chord. □

The magic behind this transformation from the time to the frequency domain is the **discrete Fourier transform** (**DFT**), as implemented in the **fast Fourier transform** (**FFT**). And the magic works in two directions: the **inverse DFT**, as implemented in the **inverse FFT**, maps a function in the

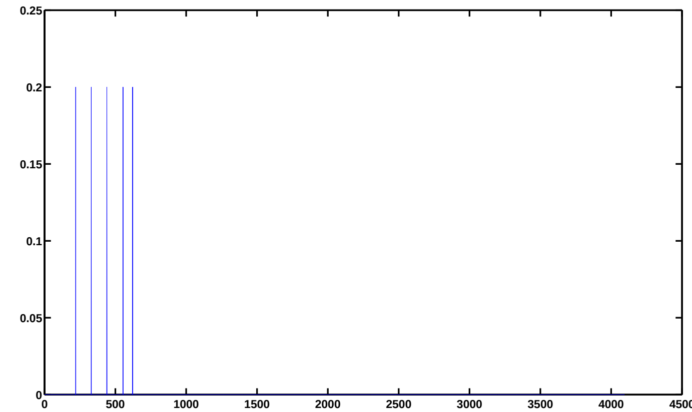


Fig. 11.3. `plot(0:4095, ssas)`

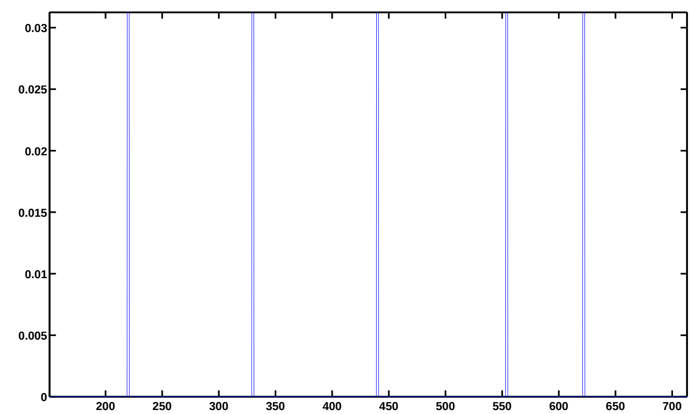


Fig. 11.4. Zoom of `plot(0:4095, ssas)`

frequency domain to a function in the time domain. Suppose that we want to build the D major chord:

$$\begin{aligned}
F^\sharp &= 440 \cdot 2^{\frac{9}{12}} \approx 740 \text{ Hz} \\
D &= 440 \cdot 2^{\frac{5}{12}} \approx 587 \text{ Hz} \\
A &= 440 \text{ Hz} \\
D &= 440 \cdot 2^{-1+\frac{5}{12}} \approx 370 \text{ Hz}
\end{aligned}$$

Rather than building the signal in the time domain as we did for the A major chord, we'll build it in the frequency domain:

```
>> ssas = zeros(1, 4096);
>> ssas(740+1) = 0.25; % +1 b/c of Matlab indexing
>> ssas(587+1) = 0.25;
>> ssas(440+1) = 0.25;
>> ssas(370+1) = 0.25;
>> plot(0:4095, ssas);
>> F = [ssas(1), ssas(2:4096)/2, 0, ssas(4096:-1:2)/2]*8192;
>> f = real(ifft(F));  % eliminate residual Im component
>> plot(t(1:128), f(1:128));
>> sound(f);
```

The frequency domain plot is shown in Figure 11.5. It should not be a surprise given that we explicitly constructed `F` to have nonzero (amplitude 0.25) frequencies at 370 Hz, 440 Hz, 587 Hz, and 740 Hz. The mathematics behind the seventh and eighth lines will become clear later. Notice now, however, that `a(r:-1:l)` is a Matlab idiom for reversing a vector `a` in the range `[l, r]`, so that `F` contains `ssas` and its reverse, both scaled by 4,096. The eighth line applies the inverse FFT to compute the time-domain signal, whose residual imaginary components are removed via `real`. A portion of the resulting signal is shown in Figure 11.6.

Exercise 11.3. Based on the discussion above, implement a function `chord` in `chord.m` that, given a row vector of frequencies, constructs the corresponding time-domain signal of duration one second via the inverse FFT. For example, `chord([370, 440, 587, 740])` should return the signal `f`, from above, of the D major chord. □

11.2 The Discrete Fourier Transform

Consider a signal sampled at n uniformly spaced intervals to yield the n-vector f. We assume that f is normalized to have a maximum absolute value of 1. The discrete Fourier transform (DFT) constructs an n-vector F of frequencies, expressed in cycles per n-step period, as follows:

$$F_{k+1} = \sum_{m=0}^{n-1} f_{m+1} \mathrm{e}^{-\frac{2\pi \mathrm{i}}{n} km} \quad \text{for } k \in \{0, 1, \ldots, n-1\}.$$

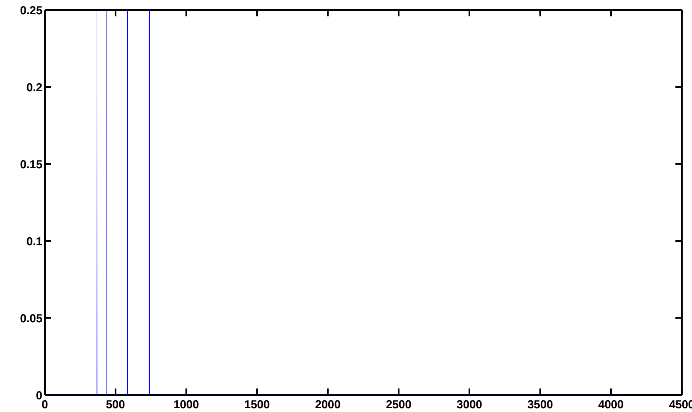


Fig. 11.5. D major: frequency domain

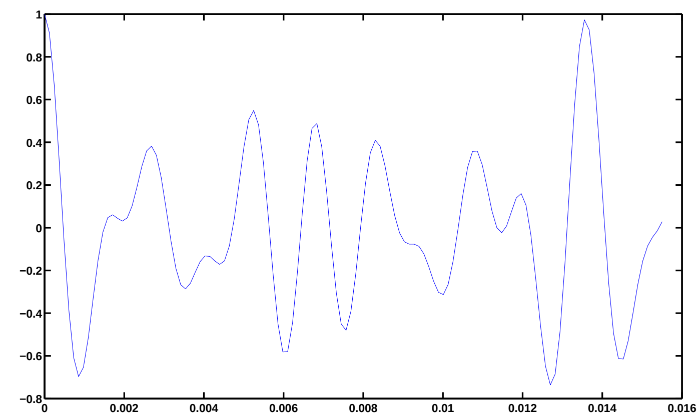


Fig. 11.6. D major: time domain

F_{k+1} is the magnitude of the "k cycles per n-period" frequency component.

Totally clear? I didn't think so. Let's delve deeper into the meaning of this definition. For convenience, we assume that $n = 8$ throughout the discussion, so that f is a row vector of 8 samples of the original analog signal, and F is a row vector representing frequency components "0 cycles per period," "1 cycle per period," ..., "7 cycles per period."

Let's first try to understand the term $\mathrm{e}^{-\frac{2\pi\mathrm{i}}{n}km}$. From Euler's formula,

$$\mathrm{e}^{\mathrm{i}\theta} = \cos\theta + \mathrm{i}\sin\theta\ ,$$

we see that this term cycles clockwise around the unit circle in the complex plane at a frequency given by $\frac{2\pi}{n}k$.

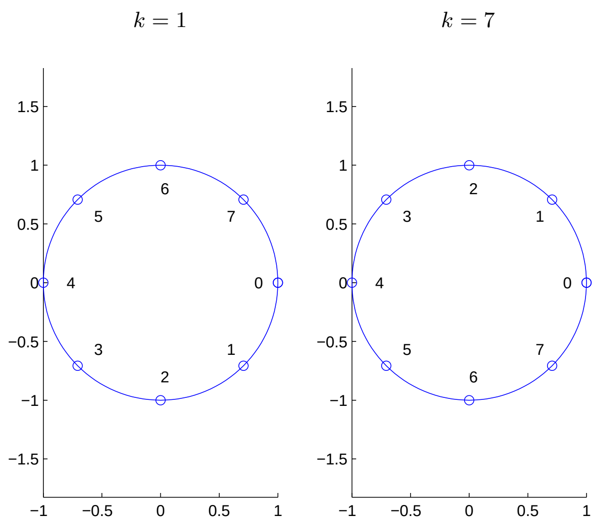


Fig. 11.7. One cycle per 8-step period

Figure 11.7 visualizes this periodicity for $k = 1$ (left) and $k = 7$ (right). In the figure, the x-axis represents the real component and the y-axis represents the imaginary component. The numbers around each circle indicate the values of m. For $k = 1$, the angles are given by $-\frac{2\pi}{8}m$, for $m \in \{0, 1, \ldots, 7\}$: 0, $-\frac{\pi}{4}$, $-\frac{\pi}{2}$, and so on. When $k = 7$, the reverse cycling occurs: rotating by $-\frac{14\pi}{8}$ radians is the same as rotating by $\frac{2\pi}{8}$ radians. Notice that, in both cases, precisely one traversal of the unit circle is achieved during the 8-step period; hence, $k = 1$ and $k = 7$ correspond to a frequency of one cycle per 8-step period.

The values $k = 2$ and $k = 6$ (Figure 11.8), and $k = 3$ and $k = 5$ (Figure 11.9) are similarly related. In general, k and $n - k$ correspond to similar

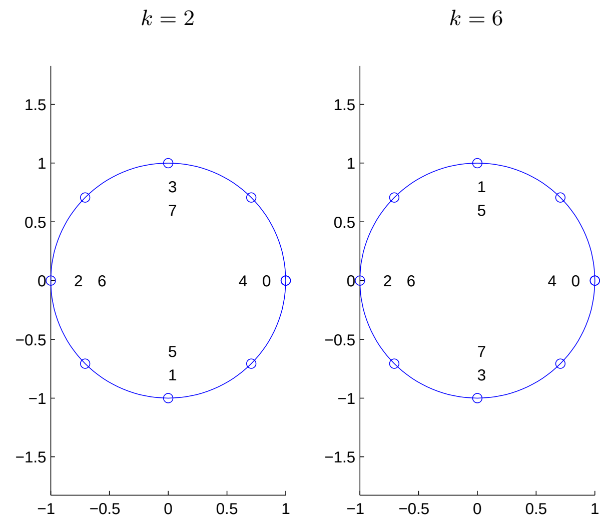


Fig. 11.8. Two cycles per 8-step period

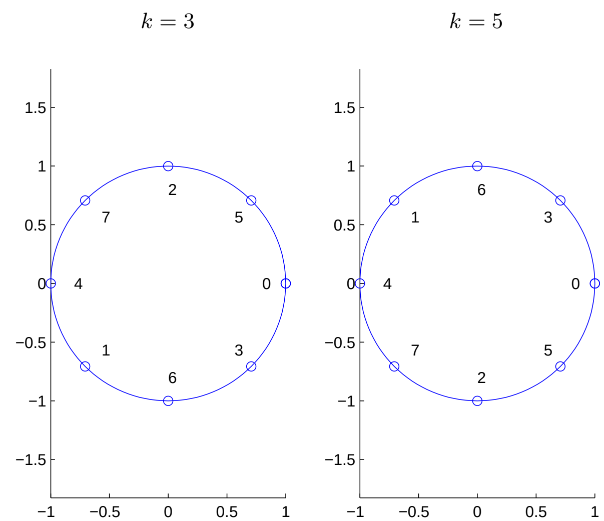


Fig. 11.9. Three cycles per 8-step period

frequencies for $k \in \{1, 2, \ldots, \frac{n}{2} - 1\}$, except that $n - k$ corresponds to the "negative frequency" of n. Furthermore, for $k = 2$ and $k = 6$, two traversals are made in the 8-step period, yielding a frequency of two cycles per 8-step period; and for $k = 3$ and $k = 5$, three traversals are made, yielding a frequency of three cycles per 8-step period.

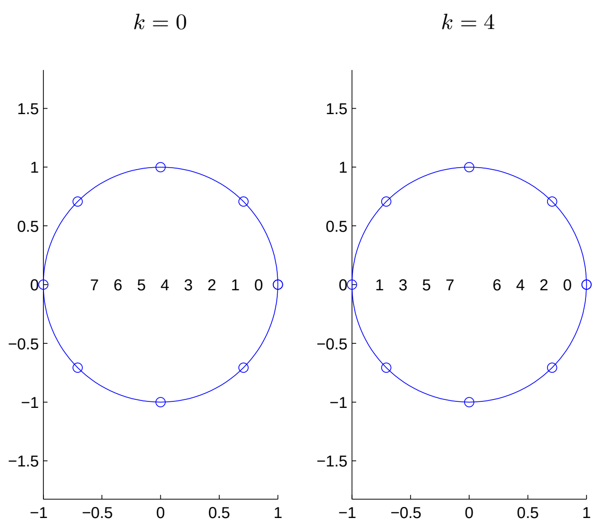


Fig. 11.10. DC and Nyquist frequencies

Two outliers are $k = 0$ and $k = 4$ (in general, $\frac{n}{2}$; see Figure 11.10). The former does not cycle; it corresponds to a DC signal, that is, a nonperiodic element such as the constant term 0.1 in the function $0.1 + \sin 2\pi t$. The latter corresponds to the **Nyquist frequency**—one-half the sampling frequency. In a realistic situation of a sampling frequency of 8,192 Hz, the Nyquist frequency is $\frac{8{,}192}{2} = 4{,}096$. No k corresponds to a higher frequency. We discuss the meaning of the Nyquist frequency in further depth momentarily.

Thus, the summation

$$\sum_{m=0}^{n-1} f_{m+1} \mathrm{e}^{-\frac{2\pi \mathrm{i}}{n} km}$$

can be understood as a cyclic traversal of the time-domain signal f that yields the degree to which the frequency component corresponding to k contributes to the overall signal f. This contribution is computed as component F_{k+1} of the DFT.

Notice that F_{k+1} is a complex number. The **absolute value** (in the complex sense: $|a + b\mathrm{i}| = \sqrt{a^2 + b^2}$) of F_{k+1} corresponds to the **amplitude** of the corresponding frequency component, while its **argument**[1] corresponds to the **phase** of the component. In this chapter, we consider only the amplitude. Therefore, the construction of the amplitude spectrum must compute the absolute value of each F_{k+1}. And, indeed, recall from the Matlab computations in Section 11.1 the use of the `abs` (absolute value) function in constructing the amplitude spectrum.

One element has yet to be explained: in the computation of `ssas` in Section 11.1, we scale by $\frac{2}{n}$. The reason for $\frac{1}{n}$ is simple, as the $k = 0$ case reveals: the sum of n values that range between -1 and 1 can be between $-n$ and n, so dividing the amplitudes by n normalizes them to have absolute values at most 1.

The reason for multiplying by 2 is less obvious though also readily explained. From our discussion above, we know that the k and $n-k$ components are related; in fact, they represent the same frequency, so that the magnitude of that frequency's contribution is spread between the two. The result is obvious once one sees a plot. For example, consider again the frequency-domain analysis of the A major chord in Section 11.1. This time, we simply normalize:

```
>> asf = abs(fft(f))/8192;
>> plot(0:8191, asf);
```

Figure 11.11 shows the result. There is a clear symmetry around $\frac{8{,}192}{2} = 4{,}096$.

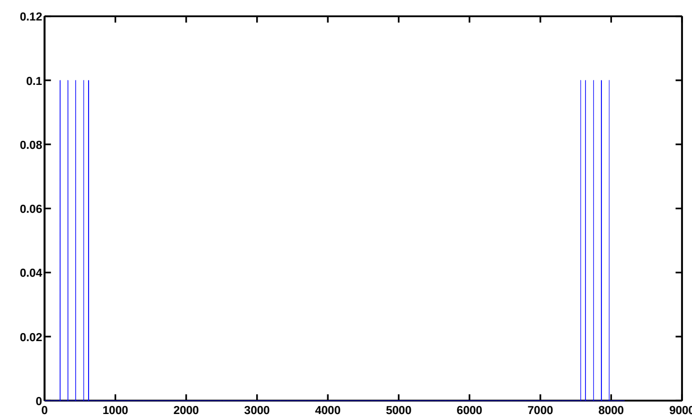


Fig. 11.11. Raw amplitude spectrum

[1] $\arg(a + b\mathrm{i}) = \arctan(\frac{b}{a})$ when $a, b > 0$; it is similarly defined for other signs of a and b.

The **single-sided amplitude spectrum** eliminates this symmetric redundancy by dropping the right half of the DFT and scaling most of the left half by 2. However, the DC frequency component should not be scaled by 2 since it is represented precisely once in the DFT. To construct the single-sided amplitude spectrum `ssas` from time-domain signal `f` thus requires computing the FFT of the signal and then extracting and scaling the components as follows, where `length(f)` is assumed to be divisible by 2:

```
>> F = fft(f);
>> ssas = abs([F(1) 2*F(2:length(f)/2)])/length(f);
```

This structure is also apparent in the inverse DFT computation in the construction of the D major chord of Section 11.1, in particular at line 7, where the symmetry is artificially induced into `F`, to which `ifft` is then applied. In general, from a single-sided amplitude spectrum `ssas`, one constructs the time-domain signal `f` as follows:

```
>> F = [ssas(1), ...                    % DC
        ssas(2:length(ssas))/2, ...
        0, ...                          % Nyquist
        ssas(length(ssas):-1:2)/2] * (2*length(ssas));
>> f = real(ifft(F));
```

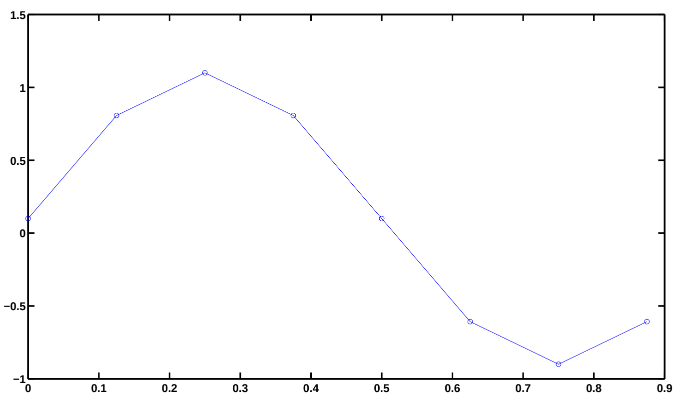


Fig. 11.12. `plot(t, f, '-o')`

To make these ideas more concrete, consider the function $0.1 + \sin 2\pi t$ sampled uniformly in the unit interval $[0, 1)$:

```
>> t = (0:7)/8};
```



```
>> f = 0.1 + sin(2*pi*t);
>> plot(t, f, '-o');
```

The function is plotted in Figure 11.12. Visually, we see that the function has a DC component (0.1): the local (absolute) maxima are 1.1 at time 0.25 and 0.9 at time 0.75. It also has a frequency-1 component, that is, a component with frequency one cycle per sample period. It does not have any higher-frequency components.

To compute F_1, which corresponds to $k = 0$, notice that $\mathrm{e}^{-\frac{2\pi\mathrm{i}}{n}0m} = 1$. Therefore, simply summing the values of $0.1 + \sin 2\pi t$ at the sample times `t` and then dividing by 8 (for eight samples) will yield a normalized F_1:

```
>> sum(f)/8

ans =

    0.1000
```

From the original function, $0.1 + \sin 2\pi t$, we see that the DC component is indeed 0.1.

To compute F_2, which corresponds to $k = 1$ and the "one cycle per sample period" frequency (see Figure 11.7), we must use the definition of the transform directly:

```
>> sum(f .* exp(-2*pi*i/8*(0:7)*1))/8

ans =

    0.0000 - 0.5000i
```

The amplitude of this component is thus $|-0.5\mathrm{i}| = \sqrt{0^2 + (-0.5)^2} = 0.5$. But the amplitude of this frequency component in the original function, $0.1 + \sin 2\pi t$, is clearly 1. Recall, though, that half of its amplitude is detected by component $n - k$ and is thus stored in F_8:

```
>> sum(f .* exp(-2*pi*i/8*(0:7)*7))/8

ans =

    -0.0000 + 0.5000i
```

The absolute value of this complex number is also 0.5, and summing the two absolute values yields the expected amplitude of 1.

Let's examine one more pair, F_3 and F_7, corresponding to $k = 2$ and $k = 6$:

```
>> sum(f .* exp(-2*pi*i/8*(0:7)*2))/8

ans =
```

```
   -9.1056e-18 + 1.3878e-17i

>> sum(f .* exp(-2*pi*i/8*(0:7)*6))/8

ans =

   2.8702e-16 + 9.7145e-17i
```

Both answers are close enough to 0 to be 0. Hence, the original signal apparently does not contain a frequency-2 component, and indeed $0.1+\sin 2\pi t$ does not.

Exercise 11.4. Compute the frequency components F_4, F_5, and F_6. □

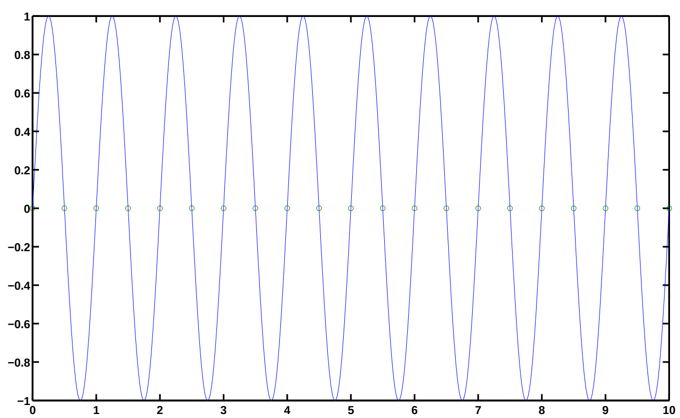


Fig. 11.13. A 10 Hz signal sampled at 20 Hz

Finally, we must understand a fundamental limitation of the DFT, expressed as the Nyquist frequency, which is half of the sampling frequency. In Figure 11.13, a 10 Hz signal is sampled (represented by the circles) at 20 Hz—yielding a discrete signal that is 0 everywhere, rather than the expected 10 Hz signal. But Nyquist explained the problem: at a sampling rate of 20 Hz, one is not sampling sufficiently frequently to capture frequencies above $\frac{20}{2} = 10$ Hz. In general, any frequency at or above half the sampling rate will not be detected correctly.

The inverse DFT is computed similarly to the DFT:

$$f_{k+1} = \frac{1}{n}\sum_{m=0}^{n-1} F_{m+1} e^{\frac{2\pi i}{n}km} \quad \text{for } k \in \{0, 1, \ldots, n-1\}.$$

The intuition for the inverse transform is that a sampled signal can be represented by the finite sum of a properly scaled set of periodic functions—indeed, by the sum of at most as many periodic functions as samples. In the construction of the A major chord, for example, we explicitly sum five scaled sine functions, each of a component frequency.

Exercise 11.5. While Matlab's implementations of the DFT (`fft`) and the inverse DFT (`ifft`) are difficult to compete with, it is still edifying to implement one's own naive versions. Using the basic definitions of these transforms, implement `mydft` and `myidft`, and verify that they produce the expected results when used in the computations of Section 11.1. □

Exercise 11.6. Implement a Matlab function, `ssas`, that takes a time-domain signal and returns its single-sided amplitude spectrum. You may assume that the signal's length is even. □

Exercise 11.7. Implement a Matlab function, `signal`, that takes a single-sided amplitude spectrum and returns the corresponding time-domain signal. □

11.3 De-hissing a Recording

Tape recordings are subject to "hissing": high-frequency white noise. In this section, we apply the DFT, first, to simulate hissing on a track and, second, to de-hiss the track. The technique used in this section is about as naive as one can get. In future courses, you will learn much more about time- and frequency-domain operations, particularly the convolution operator, that are necessary to implement nonnaive digital signal processing functions.

Matlab installations come with a file that defines a segment of Händel's "Hallelujah Chorus":

```
>> load handel;
>> f = y';
>> sound(f);
```

Lovely.

The following function makes it considerably less lovely:

```
function rv = hiss(f, th_freq)
% Input:
%  f       - sampled signal
%  th_freq - threshold frequency beyond which to add
%            white noise
% Output:
%  original signal with high frequency white noise added

  % make the magnitude of the white noise proportional
```

```
10   % to the maximum magnitude of f
11   noise_lvl = max(abs(f))/3;
12
13   % add high-frequency white noise
14   w = randn(1, 8192);   % normally distributed
15   W = fft(w);           % transform of white noise
16   % zero out frequencies at and below th_freq
17   W(1:th_freq) = 0;
18   W((8192-th_freq):8192) = 0;
19   % transform back to obtain high-frequency white noise
20   hfw = noise_lvl * real(ifft(W));
21
22   % add high-frequency white noise to each segment of f in
23   % increments of 8192 samples
24   rv = zeros(1, length(f));
25   for i = 0:8192:length(f)
26     sz = min(8192, length(f)-i);
27     rv(i+1:i+sz) = f(i+1:i+sz) + hfw(1:sz);
28   end
29 end
```

Lines 13–20 create high-frequency, normally distributed **white noise**. Line 14 creates a signal in the time domain of normally distributed white noise. Then line 15 transforms it to the frequency domain, where certain frequency components are canceled out in lines 17–18. Finally, line 20 transforms the signal back as `hfw`, for "high-frequency (white) noise."

With the noise constructed, lines 24–28 add the noise to the provided signal, `f`, one 8,192-sample window at a time. Let's apply it to the lovely music:

```
>> nzf = hiss(f, 3000);  % add white noise above 3000 Hz
>> sound(nzf);
```

Not so lovely.

Exercise 11.8. Complete the following function, in `dehiss.m`, to cancel frequency components beyond the threshold frequency.

```
function rv = dehiss(f, th_freq)
% Input:
%  f       - sampled signal
%  th_freq - threshold frequency beyond which to cancel
%            frequency components
% Output:
%  original signal with high frequency components canceled

  % modify one "window" of 8192 samples at a time
  for i = 0:8192:length(f)
    if i+8192 <= length(f)
      lo = i+1;
      hi = i+8192;

      % eliminate hiss in f(lo:hi)
      % YOUR CODE HERE
    end
  end
end
```


The code currently ignores the rightmost incomplete "window" of the signal. For an additional challenge, make the code handle this window as well.

Apply your implementation to `nzf`. While this naive approach leaves the resurrected music sounding somewhat hollow, you should nevertheless hear the "hallelujahs" clearly. Adjust `th_freq` and `noise_lvl` in `hiss`. What happens as `th_freq` becomes low? □

Exercise 11.9. Challenge: Write a Matlab program to generate Figures 11.7–11.10. □

Exercise 11.10. Percussion instruments typically produce a lot of white noise. Using the DFT, implement a Matlab function to produce a percussion-based beat of a specified duration. Add a beat track to the random music generated by your program from Exercise 9.14. □

Index