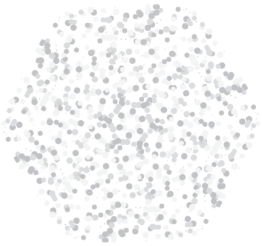
2



C

This chapter introduces C and some of the general concepts behind good programming that script writers often overlook. The function-based approach, stacks of frames, debugging, test functions, and overall good style are immediately applicable to virtually every programming language in use today. Thus, this chapter on C may help you to become a better programmer with any programming language.

As for the syntax of C, this chapter will cover only a subset. C has 32 keywords and this book will only use 18 of them.[1] Some of the other keywords are basically archaic, designed for the days when compilers needed help from the user to optimize code. Other elements, like bit-shifting operators, are useful only if you are writing an operating system or a hardware device driver. With all the parts of C that directly manipulate hexadecimal memory addresses omitted, you will find that C is a rather simple language that is well suited for simulations and handling large data sets.

An outline This chapter divides into three main parts. Sections 2.1 and 2.2 start small, covering the syntax of individual lines of code to make assignments, do arithmetic, and declare variables. Sections 2.3 through 2.5 introduce functions, describing how C is built on the idea of modular functions that are each independently built and evaluated. Sections 2.6 through 2.8 cover *pointers*, a somewhat C-specific means of handling computer memory that complements C's means of handling functions and large data structures. The remainder of the chapter offers some tips on writing bug-free code.

[1] For comparison, C++ has 62 keywords as of this writing, and Java has an even 50.

Tools You will need a number of tools before you can work, including a C *compiler*, a *debugger*, the *make* facility, and a few libraries of functions. Some systems have them all pre-installed, especially if you have a benevolent system administrator taking care of things. If you are not so fortunate, you will need to gather the tools yourself. The online appendix to this book, at the site linked from `http://press.princeton.edu/titles/8706.html`, will guide you through the process of putting together a complete C development environment and using the tools for gathering the requisite libraries.[2]

$\mathbb{Q}_{2.1}$

Check your C environment by compiling and running "Hello, world," a classic first program adapted from Kernighan & Ritchie (1988).

- Download the sample code for this book from the link at `http://press.princeton.edu/titles/8706.html`.
- Decompress the `.zip` file, go into the directory thus created, and compile the program with the command `gcc hello_world.c`. If you are using an *IDE*, see your manual for compilation instructions.
- If all went well, you will now have a program in the directory named either `a.out` or `hello_world`. From the command line, you can execute it using `./a.out` or `./hello_world`.
- You may also want to try the `makefile`, which you will also find in the code directory. See the instructions at the head of that file.

If you need troubleshooting help, see the online appendix, ask your local computing guru, or copy and paste your error messages into your favorite search engine.

2.1 LINES

The story begins at the smallest level: a single line of code. Most of the work on this level will be familiar to anyone who has written programs in any language, including instructions like assignments, basic arithmetic, if-then conditions, loops, and comments. For such common programming elements, learning C is simply a question of the details of syntax. Also, C is a *typed* language, meaning that you will need to specify whether every variable and function is an integer, a real, a vector, or whatever. Thus, many of the lines will be simple type declarations, whose syntax will be covered in the next section.

[2] A pedantic note on standards: this book makes an effort to comply with the ISO C99 standard and the IEEE POSIX standard. The C99 standard includes some features that do not appear in the great majority of C textbooks (like designated initializers), but if your compiler does not support the features of C99 used here, then get a new compiler—it's been a long while since 1999. The POSIX standard defines features that are common to almost every modern operating system, the most notable of which is the *pipe*; see Appendix B for details.

The focus is on `gcc`, because that is what I expect most readers will be using. The command-line switches for the `gcc` command are obviously specific to that compiler, and users of other compilers will need to check the compiler manual for corresponding switches. However, all C code should compile for any C99- and POSIX-compliant compiler. Finally, the `gcc` switch most relevant to this footnote is `-std=gnu99`, which basically puts the compiler in C99 + POSIX mode.

ASSIGNMENT Most of the work you will be doing will be simple assignments. For example,

```
ratio = a / b;
```

will find the value of `a` divided by `b` and put the value in `ratio`. The `=` indicates an assignment, not an assertion about equality; on paper, computer scientists often write this as `ratio` ← `a/b`, which nicely gets across an image of `ratio` taking on the value of `a/b`. There is a semicolon at the end of the line; you will need a semicolon at the end of everything but the few exceptions below.[3] You can use all of the usual operations: `+`, `-`, `/`, and `*`. As per basic algebraic custom, `*` and `/` are evaluated before `+` and `-`, so `4 + 6 / 2` is seven, and `(4 + 6)/2` is five.

※ **TWO TYPES OF DIVISION** There are two ways to answer the question, "What is 11 divided by 3?" The common answer is that $11/3 = 3.\overline{66}$, but some say that it is three with a remainder of two. Many programming languages, including C, take the second approach. Dividing an integer by an integer gives the answer with the fractional part thrown out, while the modulo operator, `%`, finds the remainder. So `11/3` is `3` and `11%3` is `2`.

Is k an even number? If it is, then `k % 2` is zero.[4]

Splitting the process into two parts provides a touch of additional precision, because the machine can write down integers precisely, but can only approximate real numbers like $3.\overline{66}$. Thus, the machine's evaluation of `(11.0/3.0)*3.0` may be ever-so-slightly different from `11.0`. But with the special handling of division for integers, you are guaranteed that for any integers `a` and `b` (where `b` is not zero), `(a/b)*b + a%b` is exactly `a`.

But in most cases, you just want $11/3 = 3.\overline{66}$. The solution is to say when you mean an integer and when you mean a real number that happens to take on an integer value, by adding a decimal point. `11/3` is `3`, as above, but `11./3` is `3.66...` as desired. Get into the habit of adding decimal points now, because integer division is a famous source of hard-to-debug errors. Page 33 covers the situation in slightly more detail, and in the meantime we can move on to the more convenient parts of the language.

---

[3] The number one cause of compiler complaints like "line 41: syntax error" is a missing semicolon on line 40.

[4] In practice, you can check evenness with `GSL_IS_EVEN` or `GSL_IS_ODD`:

```
#include <gsl/gsl_math.h>
...
if (GSL_IS_EVEN(k))
    do_something();
```

INCREMENTING It is incredibly common to have an operation of the form `a = a + b;`—so common that C has a special syntax for it:

```
a += b;
```

This is slightly less readable, but involves less redundancy. All of the above arithmetic operators can take this form, so each of the following lines show two equivalent expressions:

```
a -= b; /*is equivalent to*/ a = a - b;
a *= b; /*is equivalent to*/ a = a * b;
a /= b; /*is equivalent to*/ a = a / b;
a %= b; /*is equivalent to*/ a = a % b;
```

The most common operation among these is incrementing or decrementing by one, and so C offers the following syntax for still less typing:[5]

```
a++;  /*is equivalent to*/ a = a + 1;
a--;  /*is equivalent to*/ a = a - 1;
```

CONDITIONS C has no need for `FALSE` and `TRUE` keywords for Boolean operations: if an expression is zero, then it is false, and otherwise it is true. The standard operations for comparison and Boolean algebra all appear in somewhat familiar form:

```
(a > b)     // a is greater than b
(a < b)     // a is less than b
(a >= b)    // a is greater than or equal to b
(a <= b)    // a is less than or equal to b
(a == b)    // a equals b
(a != b)    // a is not equal to b
(a && b)    // a and b
(a || b)    // a or b
(! a)       // not a
```

- All of these evaluate to either a one or a zero, depending on whether the expression in parens is true or false.

[5] There is also the pre-increment form, `++a` and `--a`. Pre- and post-incrementing differ only when they are being used in situations that are bad style and should be avoided. Leave these operations on a separate line and stick to whichever form looks nicer to you.

- The comparison for equality involves *two* equals signs in a row. One equals sign `(a = b)` will assign the value of `b` to the variable `a`, which is not what you had intended. Your compiler will warn you in most of the cases where you are probably using the wrong one, and you should heed its warnings.
- The `&&` and `||` operators have a convenient feature: if `a` is sufficient to determine whether the entire expression is true, then it won't bother with `b` at all. For example, this code fragment—

```
((a < 0) || (sqrt(a) < 3))
```

—will never take the square root of a negative number. If `a` is less than zero, then the evaluation of the expression is done after the first half (it is true), and evaluation stops. If `a>=0`, then the first part of this expression is not sufficient to evaluate the whole expression, so the second part is evaluated to determine whether $\sqrt{a} < 3$.

Why all the parentheses? First, parentheses indicate the order of operations, as they do in pencil-and-paper math. Since all comparisons evaluate to a zero or a one, both `((a>b)||(c>d))` and `(a>(b||c)>d)` make sense to C. You probably meant the first, but unless you have the order-of-operations table memorized, you won't be sure which of the two C thinks you mean by `(a>b||c>d)`.[6]

Second, the primary use of these conditionals is in flow control: causing the program to repeat some lines while a condition is true, or execute some lines only if a condition is false. In all of the cases below, you will need parentheses around the conditions, and if you forget, you will get a confusing compiler error.

IF-ELSE STATEMENTS Here is a fragment of code (which will not compile by itself) showing the syntax for conditional evaluations:

```
1  if (a > 0)
2      { b = sqrt(a); }
3  else
4      { b = 0; }
```

If `a` is positive, then `b` will be given the value of `a`'s square root; if `a` is zero or negative, then `b` is given the value zero.

- The condition to be evaluated is always in parentheses following the `if` statement, and there should be curly braces around the part that will be evaluated when the

[6] The order-of-operations table is available online, but you are encouraged to not look it up. [If you must, try `man operator` from the command prompt]. Most people remember only the basics like how multiplication and division come before addition and subtraction; if you rely on the order-of-operations table for any other ordering, then you will merely be sending future readers (perhaps yourself) to check the table.

condition is true, and around the part that will be evaluated when the condition is false.

- You can exclude the curly braces on lines two and four if they surround exactly one line, but this will at some point confuse you and cause you to regret leaving them out.
- You can exclude the `else` part on lines three and four if you don't need it (which is common, and much less likely to cause trouble).
- The `if` statement and the line following it are smaller parts of one larger expression, so there is no semicolon between the `if(...)` clause and what happens should it be true; similarly with the `else` clause. If you do put a semicolon after an `if` statement—`if (a > 0);`—then your `if` statement will execute the null statement—`/*do nothing*/;`—when `a > 0`. Your compiler will warn you of this.

$\mathbb{Q}_{2.2}$ Modify `hello_world.c` to print its greeting if the expression `(1 || 0 && 0)` is true, and print a different message of your choosing if it is false. Did C think you meant `((1 || 0) && 0)` (which evaluates to 0) or `(1 || (0 && 0))` (which evaluates to 1)?

LOOPS Listing 2.1 shows three types of loop, which are slightly redundant.

```
1  #include <stdio.h>
2  int main(){
3      int i = 0;
4      while (i < 5){
5          printf("Hello.\n");
6          i ++;
7      }
8
9      for (i=0; i < 5; i++){
10         printf("Hi.\n");
11     }
12
13     i = 0;
14     do {
15         printf("Hello.\n");
16         i ++;
17     } while (i < 5);
18
19     return 0;
20 }
```

Listing 2.1 C provides three types of loop: the `while` loop, the `for` loop, and the `do-while` loop. Online source: `flow.c`.

The simplest is a `while` loop. The interpretation is rather straightforward: while the expression in parentheses on line four is true (mustn't forget the parentheses), execute the instructions in brackets, lines five and six.

Loops based on a counter (`i` $= 0$, `i` $= 1$, `i` $= 2, \ldots$) are so common that they get their own syntax, the `for` loop. The `for` loop in lines 9–11 is exactly equivalent to the `while` loop in lines 3–7, but gathers all the instructions about incrementing the counter onto a single line.

You can compare the `for` and `while` loop to see when the three subelements in the parentheses are evaluated: the first part (`i=0`) is evaluated before the loop runs; the second part (`i<5`) is tested at the beginning of each iteration of the loop; the third part (`i++`) is evaluated at the end of each loop. After the section on arrays, you will be very used to the `for (i=0; i<limit; i++)` form, and will recognize it to mean *step through the array*. There may even be a way to get your text editor to produce this form with one or two keystrokes.

Finally, if you want to guarantee that the loop will run at least once, you can use a `do-while` loop (with a semicolon at the end of the `while` line to conclude the thought). The `do-while` loop in Listing 2.1 is equivalent to the `while` and `for` loops. But say that you want to iteratively evaluate a function until it converges to within 1×10^{-3}. Naturally, you would want to run the function at least once. The form would be something like:

```
do {
    error = evaluate_function();
} while (error > 1e-3);
```

Example: the birthday paradox The birthday paradox is a staple of undergraduate statistics classes.[7] The professor writes down the birth date of every student in the class, and finds that even though there is a 1 in 365 chance that any given pair of students have the same birthday, the odds are good that there is a match in the class overall.

Listing 2.2 shows code to find the likelihood that another student shares the first person's birthday, and the likelihood that any two students share a birthday.

- Most of the world's programs never need to take a square root, so functions like `pow` and `sqrt` are not included in the standard C library. They are in the separate math library, which you must refer to on the command line. Thus, compile the program with

[7] It even mystifies TV talk show hosts, according to Paulos (1988, p 36).

```
1  #include <math.h>
2  #include <stdio.h>
3
4  int main(){
5    double no_match = 1;
6    double matches_me;
7    int ct;
8      printf("People\t Matches me\t Any match\n");
9      for (ct=2; ct<=40; ct ++){
10         matches_me = 1− pow(364/365., ct−1);
11         no_match *= (1 − (ct−1)/365.);
12         printf("%i\t %.3f\t\t %.3f\n", ct, matches_me, (1−no_match));
13     }
14     return 0;
15 }
```

Listing 2.2 Print the odds that other students share my birthday, and that any two students in the room share a birthday. Online source: `birthday.c`.

```
gcc birthday.c −lm −o birthday
```

where `-lm` indicates the math library and `-o` indicates that the output program will be named `birthday` (rather than the default `a.out`). More on linking and libraries will follow below.

- Lines 1–7 are introductory material, to be discussed below, including a preface `#include`-ing a few external files, and a list of the dramatis personæ: variables named `no_match`, `matches_me`, and `ct`.
- Line 8 prints a header line labeling the columns of numbers the `for` loop will be producing; it is easy to read once you know that `\t` means *print a tab* and `\n` means *newline*.
- Line 9 tells us that the counter `ct` will start at two, and count up until it reaches 40.
- As for the math itself, it is easier to calculate the complement—the odds that nobody shares a birthday. The odds that one person does not share the first person's birthday is $364/365$; the odds that two people both do not share the first person's birthday is $(364/365)^2$, et cetera.[8] Thus, the odds that among `ct-1` additional people, none have the same birthday as the first person is $1-(364/365)^{\texttt{ct}-1}$. You can see this calculation on line ten.
- As above, the odds that the second person does not share the first person's birthday is $\left(\frac{364}{365}\right)$. The odds that an additional person shares no birthday with the first two given that the first two do not share a birthday is $\left(\frac{363}{365}\right)$, so the odds that the first

[8] We assume away leap years, and the fact that the odds of being born on any given day are not exactly $1/365$—more children are born in the summer.

three do not share a birthday is

$$\left(\frac{364}{365}\right)\left(\frac{363}{365}\right). \tag{2.1.1}$$

This expression is best produced incrementally. In the introductory material, `no_match` was initialized at 1, and on line 11, another element of the sequence headed by Expression 2.1.1 gets multiplied in to `no_match` at each step of the `for` loop.

- Line 12 prints the results. The first input to the `printf` function will be discussed in detail below, but the next inputs indicate what is to be printed: the counter, `matches_me`, and `1-(no_match)`.

$\mathbb{Q}_{2.3}$ Modify the `for` loop to verify that the program prints the correct values for a class of one student.

COMMENTS Put a long block of comments at the head of a file and at the head of each function to describe what the file or function does, using complete sentences. Describe what the function expects to come in, and what the function will put out. The common wisdom indicates that these comments should focus on why your code is doing what it is doing, rather than how, which will be self-explanatory in clearly-written code.[9]

The primary audience of your comment should be you, six months from now. When you are shopping for black boxes to plug in to your next project, or re-auditing your data after the referee finally got the paper back to you, a note to self at the head of each function will pay immense dividends.

```
/* Long comments begin with a slash−star,
   continue as long as you want, and end
   at the first star−slash.
   */
```

The stars and slashes are also useful for *commenting out* code. If you would like to temporarily remove a few lines from your program to see what would happen, but don't want to delete them entirely, simply put a `/*` and a `*/` around the code, and the compiler will think it is a comment and ignore it.

However, there is a slight problem with this approach: what if there is a comment in what you had just commented out? You would have a sequence like this in your code:

[9] The sample code for this book attempts to be an example of good code in most respects, but it has much less documentation than real-world code should have, because this book is the documentation.

```
/* Line A;
   /* Line B */
   Line C;
*/
```

We had hoped that all three lines would be commented out now, but the compiler will ignore everything from the first `/*` until it sees the first `*/`. That means Line A and Line B will be ignored, but

```
    Line C;
*/
```

will be read as code—and malformed code at that.[10]

You will always need to watch out for this when commenting out large blocks of code. But for small blocks, there is another syntax for commenting individual lines of code that deserve a note.

```
this_is_code; //Everything on a line
              //after two slashes
              //will be ignored.
```

Later, we will meet the preprocessor, which modifies the program's text before compilation. It provides another solution for commenting out large blocks that may have comments embedded. The compiler will see none of the following code, because the preprocessor will skip everything between the `#if` statement which evaluates to zero and the `#endif`:

```
#if 0
/*This function does nothing. */
void do_nothing(){ }
#endif
```

PRINTING C prints to anything—the screen, a string of text in memory, a file—using the same syntax. The formatting works much like the Mad Libs party game (Price & Stern, 1988). First, there is a format specifier, showing what the output will be, but with blanks to be filled in:

[10] ℚ: If C didn't have this quirk, and allowed comments inside comments, what different quirk would you have to watch out for instead?

My _____ is very _____.
noun adjective

Then, the user provides a specific instance of the noun and adjective to be filled in (which in this case is left as an exercise for the reader). Since C is a programming language and not a party game, the syntax is a little more terse. Instead of

_____ is number _____ in line.
`string` `int`

`printf` uses:

%s is number %i in line.

Here is a complete example:

```
1  #include <stdio.h>
2
3  int main(){
4      int position = 3;
5      char name[] = "Steven";
6      printf("%s is number %i in line\n", name, position);
7      return 0;
8  }
```

The `printf` function is not actually defined by default—its definition in the standard input/output header must be `#included`, which is what line one does. Lines four and five are variable declarations, defining the *type*s of the variables; these lines foreshadow the next section.

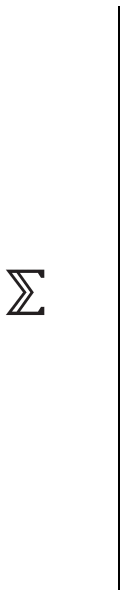
Finally, line six is the actual print statement, which will insert `Steven` into the first placeholder (`%s`), and insert `3` into the second placeholder (`%i`). It will thus print `Steven is number 3 in line` (plus an invisible newline).

Here are the odd characters you will need for almost all of your work.

`%i`	insert an integer here
`%g`	insert a real number in general format here
`%s`	insert a string of text here
`%%`	a plain percent sign
`\n`	begin a new line
`\t`	tab
`\"`	a quote that won't end the text string
\(newline)	continue the text string on the next line

There are many more format specifiers, which will give you a great deal of control; you may want them when printing tables, for example, and can refer to any of a number of detailed references when you need these, such as `man 3 printf` from your command line.

At this point, you may want to flip through this book to find a few examples of `printf` and verify that they will indeed print what they promise to.



- Assignment uses a single equals sign: `assignee = value;`.
- The usual arithmetic works: `ten = 2*3+8/2;`.
- Conditions such as `(a > b)`, `(a <= b)`, and `(a == b)` (two equals signs) can be used to control flow.
- Conditional flow uses the form: `if (condition) {do_if_true;} else {do_if_false;}`.
- The basic loop is a `while` loop: `while (this_is_true) {do_this;}`.
- When iterating through an array, a `for` loop makes the iteration clearer: `for (j=0; j< limit; j++) {printf("processing item %i\n", j);}`.
- Write comments for your future self.

2.2 VARIABLES AND THEIR DECLARATIONS

Having covered the verbs that a line of code will execute, we move on to the nouns—variables.

You would never use x or z in a paper without first declaring, say, 'let $x \in \mathbb{R}^2$ and $z \in \mathbb{C}$'. You could leave the reader to guess at what you mean by x by its first use, but some readers would misunderstand, and your referee would wonder why you did not just come out and declare x. C is a strict referee, and requires that you declare the type of every variable before using it. The declaration consists of listing the type of the variable and then the variable name, e.g.

```
int a_variable, counter=0;
double stuff;
```

- This snippet declared three variables in two lines.
- We could initialize `counter` to zero as it is declared.

- The other variables (such as `a_variable`) have unknown values right now. Assume nothing about what is contained in a declared but uninitialized value.[11]
- Since the first step in using a variable is typically an assignment, and you can declare and initialize on the same line, the burden of declaring types basically means putting a type at the head of the first line where the variable is used, as on lines four and five of the sample code on page 27.

Here is a comprehensive list of the useful basic types for C.

`int`	an integer: $-1, 0, 3$
`double`	a real number: $2.4, -1.3\text{e}8, 27$
`char`	a character: 'a', 'b', 'C'

An `int` can only count to about $2^{32} \approx 4.3$ billion; you may have a simulation that involves five billion agents or other such uses for counting into the trillions, in which case you can use the `long int` type.[12]

There are ways to extend or shrink the size of the numbers, which are basically not worth caring about. A `double` counts up to about $\pm$1e308, which is already significantly more than common estimates of the number of atoms in the universe (circa 1e80), but there is a `long double` type in case you need more precision or size.[13] Section 4.5 offers detailed notes about how numbers are represented.

Finally, notice that the variable names used throughout are words, not letters.[14] Using English variable names is the number one best thing you could do to make your code readable. Imagine how much of your life you have spent flipping back through journal articles trying to remember what μ, M, and m stood for. Why impose that on yourself?

---

[11] I am reluctant to mention this, but later you will see the distinction between global, static, and local variables. Global and static variables are automatically initialized to zero (or `NULL`), while local variables are not. But you will suffer fewer painful debugging sessions if you ignore this fact and get into the habit of explicitly initializing everything that needs initialization.

[12] The `int` type on most 64-bit systems is still 32 bits. Though this norm will no doubt change in the future, the safe bet is to write code under the assumption that an `int` counts to 2^{32}.

[13] The `double` name is short for "double-precision floating-point number," and the `float` thus has half the precision and range of a `double`. Why *floating point*? The computer represents a real using a form comparable to scientific notation: $n \times 10^k$, where n represents the number with the decimal point in a fixed location, and k represents the location of the decimal point. Multiplying by ten doesn't change the number n, it just causes the decimal point to float to a different position.

The `float` type is especially not worth bothering with because the GSL's matrices and vectors default to holding `double`s, and C's floating-point functions internally operate on doubles. For example, the `atof` function (ASCII text to floating-point number) actually returns a `double`.

[14] The exception are indices for counters and `for` loops, which are almost always `i`, `j`, or `k`.

Arrays Much of the art of describing the real world consists of building aggregates of these few basic types into larger structures. The simplest such aggregate is an *array*, which is simply a numbered list of items of the same type. To declare a list of a hundred integers, you would use:

```
int a_list[100];
```

Then, to refer to the items of the list, you would use the same square brackets. For example, to assign the value seven to the last element of the array, you would use: `a_list[99]= 7;`. Why is 99 the last element of the list? Because the index is an *offset* from the first element. The first element is zero items away from itself, so it is `a_list[0]`, not `a_list[1]` (which is the second element). The reasoning behind this system will become evident in the section on pointers.

2-D arrays simply require more indices—`int a_2d_list[100][100]`. But there are details in implementation that make 2-D arrays difficult to use in practice; Chapter 4 will introduce the `gsl_matrix`, which provides many advantages over the raw 2-D array.

Just as you can initialize a scalar with a value at declaration, you can do the same with an array, e.g.:

```
double data[ ] = {2,4,8,16,32,64};
```

You do not have to bother counting how many elements the array has; C is smart enough to do this for you.

$\mathbb{Q}_{2.4}$ Write a program to create an array of 100 integers, and then fill the array with the squares (so `the_array[7]` will hold 49). Then, print a message like "7 squared is 49." for each element of the array. Use the Hello World program as a template from which to start.

$\mathbb{Q}_{2.5}$ The first element of the Fibonacci sequence is defined to be 0, the second is defined to be 1, and then element n is defined to be the sum of elements $n-1$ and $n-2$. Thus, the third element is 0+1=1, the fourth is 1+1=2, the fifth is 1+2=3, then 2+3=5, then 3+5=8, et cetera.
The ratio of the $n\text{th}$ element over the $(n-1)\text{st}$ element converges to a value known as the golden ratio.
Demonstrate this convergence by producing a table of the first 20 elements of the sequence and the ratio of the $n\text{th}$ over the $(n-1)\text{st}$ element for each n.

DECLARING TYPES You can define your own types. For example, these lines will first declare a new type, `triplet`, and then declare two such triplets, `tri1` and `tri2`:

```
typedef double triplet[3];
triplet tri1, tri2;
```

This is primarily useful for designing complex data types that are collections of many subelements, in conjunction with the `struct` keyword. For example:

```
typedef struct {
    double real;
    double imaginary;
} complex;

complex a, b;
```

You now have two variables of type `complex` and can now use `a.real` or `b.imaginary` to refer to the appropriate constituents of these complex numbers.

Listing 2.3 repeats the birthday example, but stores each class size's data in a `struct`.

- Lines 4–7 define the structure: it will hold one variable indicating the probability of somebody matching the first person's birthday, and one variable giving the probability that no two people share a birthday.
- Those lines only defined a type; line 11 declares a variable, `days`, which will be of this type. Since there is a number in brackets after the name, this is an array of `bday_structs`.
- In line 12, the `none_match` element of `days[1]` is given a value. Lines 14 and 15 assign values to the elements of `days[2]` through `days[40]`. Having calculated the values and stored them in an organized manner, it is easy for lines 18–20 to print the values.

Initializing As with an array, you can initialize most or all of the elements of a struct to a value on the declaration line. The first option, comparable to the array syntax above, is to remember the order of the `struct`'s elements and make the assignments in that order. For example,

```
complex one = {1, 0};
complex pioverfour = {1, 1};
```

```
1  #include <math.h>
2  #include <stdio.h>
3
4  typedef struct {
5      double one_match;
6      double none_match;
7  } bday_struct;
8
9  int main(){
10   int ct, upto = 40;
11   bday_struct days[upto+1];
12     days[1].none_match = 1;
13     for (ct=2; ct<=upto; ct ++){
14         days[ct].one_match = 1− pow(364/365., ct−1);
15         days[ct].none_match = days[ct−1].none_match ∗ (1 − (ct−1)/365.);
16     }
17     printf("People\t Matches me\t Any match\n");
18     for (ct=2; ct<=upto; ct ++){
19         printf("%i\t %.3f\t\t %.3f\n", ct, days[ct].one_match, 1−days[ct].none_match);
20     }
21     return 0;
22 }
```

Listing 2.3 The birthday example (Listing 2.2) rewritten using a `struct` to hold each day's data. Online source: `bdaystruct.c`.

would initialize `one` to $1 + 0\mathbf{i}$ and `pioverfour` to $1 + 1\mathbf{i}$. This is probably the best way to initialize a `struct` where there are few elements and they have a well-known order.

The other option is to use *designated initializers*, which are best defined by example. The above two initializations are equivalent to:

```
complex one = {.real = 1};
complex pioverfour = {.imaginary = 1, .real = 1};
```

In the first case, the imaginary part is not given, so it is initialized to zero. In the second case, the elements are out of order, which is not a problem. Designated initializers will prove to be invaluable when dealing with structures like the `apop_model`, which has a large number of elements in no particular order.

Two final notes on designated initializers. They can also be used for arrays, and they can be interspersed with unlabeled elements. The line

```
int isprime[] = {[1]=1, 1, 1, [5]=1, [7]=1, [11]=1};
```

initializes an array from zero to eleven (the length is determined by the last initialized element), setting the elements whose index is a prime number to one. The two ones with no label will go into the 2 and 3 slot, because their index will follow sequentially after the last index given.

Structs are syntactically simple, so there is little to say about them, but much of good programming goes in to designing structs that make sense and are a good reflection of reality. This book will be filled with them, including both purpose-built structures like the `bday_struct` and structures defined by libraries like the GSL, such as the `gsl_matrix` mentioned above.

※ **TYPE CASTING** There are a few minor complications when assigning a value of one type to a variable of a different type. When you assign a `double` value to an integer, such as `int i = 3.2`, everything after the decimal point would be dropped. Also, the range of `floats`, `doubles`, and `ints` do not necessarily match, so you may get unpredictable results with large numbers even when there is nothing after the decimal point.

If you are confident that you want to assign a variable of one type to a variable of another, then you can do so by putting the type to re-cast into in parentheses before the variable name. For example, if `ratio` is a `double`, `(int) ratio` will cast it to an integer. If you want to accept the truncation and assign a floating-point real to an integer, say `int n`, then explicitly tell the compiler that you meant to do this by making the cast yourself; e.g., `n = (int) ratio;`.

Type casting solves the division-by-integers problem from the head of this chapter. If `num` and `den` are both `ints`, then `ratio = (double) num / den` does the division of a real by an integer, which will produce a real number as expected.

There are two other ways of getting the same effect: `(num + 0.0)` is an `int` plus a `double`, which is a `double`. Then `(num + 0.0)/den` is division of a real by an integer, which again works as expected (but don't forget the parens). And as above, if one of the numbers is a constant, then just add a decimal point, because `2` is an `int`, while `2.` is a floating-point real number.

Finally, note that when casting from `double` to `int`, numbers are truncated, not rounded. As a lead-in to the discussion of functions, here is a function that uses type casting to correctly round off numbers:[15]

[15] In the real world, use `rint` (in `math.h`) to round to integer: *`rounded_val = rint(unrounded_number)`*.

```
int round(double unrounded){
/* Input a real number and output the number
    rounded to the nearest integer. */

    if (unrounded > 0)
        return (int) (unrounded + 0.5);
    else
        return (int) (unrounded − 0.5);
}
```

- All variables must be declared before the first use.
- Until a variable is given a value, you know nothing about its value.
- You can assign an initial value to the variable on the declaration line, such as `int i = 3;`.
- Arrays are simply declared by including a size in brackets after the variable name: `int array[100];`. Refer to the elements using an *offset*, so the first element is `array[0]` and the last is `array[99]`.
- You can declare new types, including structures that amalgamate simpler types: `typedef struct {double length, waist; int leg_ct;} pants;`.
- After declaring a variable as a structure, say `pants cutoffs;`, refer to structure elements using a dot: `cutoffs.leg_ct = 1;`.
- An integer divided by an integer is an integer: `9/4 == 2`. By putting a decimal after a whole number, it becomes a real number, and division works as expected: `9./4 == 2.25`.

2.3 FUNCTIONS

The instruction *take the inverse of the matrix* is six words long, but refers to a sequence of steps that typically require several pages to fully describe.

Like many fields, mathematics progresses through the development of new vocabulary like the phrase *take the inverse*. We can comprehensibly express a complex statement like *the variance is* $\sigma^2(\mathbf{X}'\mathbf{X})^{-1}$ because we didn't need to write out exactly how to do a squaring, a transposition $(\mathbf{X}')$ and an inverse.

Similarly, most of the process of writing code is not about describing the procedures involved, but building a specialized vocabulary to make describing the

procedures trivial. Adding new nouns to the vocabulary is a simple task, discussed above using both basic nouns and `structs` that aggregate them to larger concepts. This section covers functions, which are single verbs that encapsulate a larger procedure.

```
1  #include <math.h>
2  #include <stdio.h>
3
4  typedef struct {
5      double one_match;
6      double none_match;
7  } bday_struct;
8
9  int upto = 40;
10 void calculate_days(bday_struct days[]);
11 void print_days(bday_struct days[]);
12
13 int main(){
14     bday_struct days[upto+1];
15     calculate_days(days);
16     print_days(days);
17     return 0;
18 }
19
20 void calculate_days(bday_struct days[]){
21   int ct;
22     days[1].none_match = 1;
23     for (ct=2; ct<=upto; ct ++){
24         days[ct].one_match = 1- pow(364/365., ct-1);
25         days[ct].none_match = days[ct-1].none_match * (1 - (ct-1)/365.);
26     }
27 }
28
29 void print_days(bday_struct days[]){
30   int ct;
31     printf("People\t Matches me\t Any match\n");
32     for (ct=2; ct<=upto; ct ++){
33         printf("%i\t %.3f\t\t %.3f\n", ct, days[ct].one_match, 1-days[ct].none_match);
34     }
35 }
```

Listing 2.4 The birthday example broken into logical functions. Online source: `bdayfns.c`.

The second birthday example, Listing 2.3 can be hard to read, with its mess of multiple `for` loops. Listing 2.4 re-presents the program using one function to do the math and one to print the output. The `main` function (lines 13–18) now describes the procedure with great clarity: declare an array of `bday_structs`, calculate values for the days, print the values, and exit. The functions to which `main` refers—on lines 20–27 and 29–35—are short, and so are easier to read than the long string of

code in Listing 2.3. Simply put, the functions provide structure to what had been a relatively unstructured mess.

Structure takes up space—you can see that this listing is more lines of code than the unstructured version. But consider the format of the book you are reading right now: it uses such stylistic features as paragraphs, chapter headings, and indentation, even though they take up space. Brevity is a good thing, which means that it is typically worth the effort to minimize redundancy and search for simple and brief algorithms. But brevity should never come at the cost of clarity. By eliminating intermediate variables and not using subfunctions, you can sometimes reduce an entire program into a single line of code, but that one-liner may be virtually impossible to debug, modify, or simply understand. No trees have to be killed to add a few lines of white space or a few function headers to your on-screen code, and the additional structure will save you time when dealing with your code later on.

Functional form Have a look at the function headers—the first line of each function, on lines 13, 20 and 29. In parens are the inputs to the function (aka the *arguments*), and they look like the familiar declarations from before. The `main` function takes no arguments, while you will see that many functions take several arguments, in which case the argument declarations are a comma-separated list.

Or consider the function declaration for the `round` function above:

```
int round (double unrounded)
```

If we ignore the argument list in parens, `int round` looks like a declaration as well—and it is. It indicates that this function will return an integer value, that can be used anywhere we need an integer. For example, you could assign the function output to a variable, via `int eight = round(8.3)`.

Declaring a function You can declare the existence of a function separately from the function itself, as per lines 10 and 11 of Listing 2.3. The `main` function thus has an idea of what to expect when it comes across these functions on lines 15 and 16, even though the functions themselves appear later. You will see below that the compiler gets immense mileage out of the declaration of functions, because it can compile `main` knowing only what the other functions take in and return, leaving the inner workings as a black box.

※ *The* `void` *type* If a function returns nothing, declare it as type `void`. Such functions will be useful for side effects such as changing the values of the inputs (like `calculate_days`) or printing data to the screen or an external

file (like `print_days`). You can also have functions which take no inputs, so any of the following are valid declarations for functions:

```
void do_something(double a);
double do_something_else(void);
double do_something_else();
```

The last two are equivalent, but you can't forget the parentheses entirely—then the compiler would think you are declaring a variable instead of a function.

Q2.6 Write a function with header `void print_array(int in_array[ ], int array_size)` that takes in an integer array and the size of the array, and prints the array to the screen. Modify your square-printing program from earlier to use this function for output.

How to write a program Given a blank screen and a program to write, how should you begin? Write an outline, based on function headers.

For example, in the birthday example, you could begin by writing the `main` function, which describes the broad outline of calculating probabilities and then printing to the screen. In writing the outline, you will need to write down the inputs, outputs, and intent of a number of functions. Then you can begin filling in each function. When writing a function's body, you can put the rest of the program out of your mind and focus on making sure that the black box you are working on does exactly what it should to produce the right output. When all of the functions correctly do their job, and the main outline is fully fleshed out, you will have a working program.

You want your black boxes to be entirely predictable and error-free, and the best way to do this is to keep them small and autonomous. Flip through this book and have a look at the structure of the longer sample programs. You will notice that few functions run for more than about fifteen lines, especially after discounting the introductory material about declaring variables and checking inputs.

FRAMES The manner in which the computer evaluates functions also abides by the principle of encapsulating functions, focusing on the context of one function at a time. When a function is called, the computer creates a *frame* for that function. Into that frame are placed any variables that are declared at the top of the file in which the function is defined, including those in files that were `#included` (see below); and copies of variables that are passed as arguments.

The function is then run, using the variables it has in that frame, blithely ignorant of the rest of the program. It does its math, making a note of the return value it

calculates (if any), and then destroys itself entirely, erasing all of the variables created in the frame and copies of variables that had been put in to the frame. Variables that were not passed as an argument but were put in the frame anyway (*global variables*; see below) come out unscathed, as does the return value, which is sent back to the function which had called the frame into existence.

```
1  #include <stdio.h> //printf
2  double globe=1; //a global variable.
3
4  double factorial (int a_c){
5      while (a_c){
6          globe *= a_c;
7          a_c --;
8      }
9       return globe;
10 }
11
12 int main(void){
13     int a = 10;
14     printf("%i factorial is %f.\n", a, factorial(a));
15     printf("a= %i\n", a);
16     printf("globe= %f\n", globe);
17     return 0;
18 }
```

Listing 2.5 A program to calculate factorials. Online source: `callbyval.c`.

One way to think about this is in terms of a *stack* of frames. The base of the stack is always the function named `main`. For example, in the program in Listing 2.5, the computer at first ignores the function `factorial`, instead starting its work at line twelve, where it finds the `main` function. It creates a `main` frame and then starts working, reading the declaration of `a`, and creating that variable in the `main` frame. The global variable declared on line two is also put into the `main` frame.

Then, on line 14, it is told to print the value of `factorial(a)`, which means that it will have to evaluate that expression. This is a *function call*, which commands the program to halt whatever it is doing and start working on evaluating the function `factorial`. So the system freezes the `main` frame, generates a frame for the `factorial` function, and jumps to line four. Think of the new frame as being put on top of the first, leaving only the topmost frame visible and active. The value of `a = 10` will be *copied* into `a_c`, the global variable `globe` will be put into the frame, and the function does its math and returns the calculated value of `globe`.[16]

[16] Why is `globe` a `double`, when the factorial is always an integer? Because the `double` uses exponential notation when necessary, so its range is much larger than that of the `int`, which does not. You could also try using a `long int` (and replacing printf's `int` placeholder `%i` with the `long int` placeholder `%li`), but even that fails after about `a=31`.

Having returned a value, the `factorial` frame and its contents are discarded. Since a copy of the value `a = 10` was sent into the frame, `a` still has the value `10` when the function returns, even though the copy `a_c` was decremented to zero.

The `main` frame is now at the top of the stack, so it can pick up where it had left off, printing the value of `a` and 10! to the screen, using the calls to the `printf` function—which each create their own frames in turn. Finally, the `main` function finishes its work, and its frame is destroyed, leaving an empty stack and a finished program.

Call-by-value A common error is to forget that global variables are put in all function frames, but only *copies* of the variables in the function's argument list are put in the frame.

When the `factorial` function is called, the system puts a copy of `a` into `a_c`, and then the function modifies the copy, `a_c`. Meanwhile, `globe` is not a copy of itself, but the real thing, so when it is changed inside the function, it changes globally. This is why the output you got when you ran the program showed `a=10`, not `a=0`.

On the one hand, the `factorial` function could mangle `a_c` without affecting the original; on the other hand, we sometimes want functions to change their inputs. This may make global variables tempting to you, but resist. Section 2.6 will give a better alternative (and explain why the `bday_struct` examples worked).

※ *Static variables* There is one exception to the rule that all local variables are destroyed with their frame: you can define a `static` variable. When the function's frame is destroyed, the program makes a note of the value of the static variable, and when the function is called again, the static variable will start off with the same value as before. This provides continuity within a function, but only that function knows about the variable.

Static variable declarations look just like other declarations but with the word `static` before them. You can also put an initialization on the declaration line, which will be taken into consideration only the first time the function is called. Here is a sample function to enter data points into an array. It assumes that the calling function knows the length of `survey_data` and does bounds-checking accordingly.

```
void add_a_point(double number, double survey_data[]){
    static int count_so_far = 0;
    survey_data[count_so_far] = number;
    count_so_far++;
}
```

The first time this function is called, `count_so_far` will be initialized at zero, the number passed in will be put in `survey_data[0]`, and `count_so_far` will be incremented to one. The second time the function is called, the program will remember that `count_so_far` is one, and will thus put the second value in `survey_data[1]`, where we would want it to be.

※ *The* `main` *function* All programs must have one and only one function named `main`, which is where the program will begin executing—the base of the stack of frames. The consistency checks are now with the operating system that called the program, which will expect `main` to be declared in one of two forms:

```
int main(void);
int main(int argc, char **argv);
```

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

int main(int argc, char **argv){
    if (argc==1){
        printf("Give me a command to run.\n");
        return 1;
    }
    int return_value = system(argv[1]);
    printf("The program returned %i.\n", return_value);
    return return_value;
}
```

Listing 2.6 A *shell* is a program that is primarily intended for the running of other programs; this is a very rudimentary one. Online source: `simpleshell.c`.

The second form will be discussed on page 206, but for now, Listing 2.6 provides a quick example of the use of inputs to `main`. It uses C's `system` function to call a program. The usage of this program would be something like

```
./simpleshell "ls /a_directory"
```

Conceptually, there is little difference between calling a function that you wrote and calling the `main` function of a foreign program. In this case, the `system` function would call `ls`, effectively putting the `ls` program's `main` function on top of the current stack. More generally, you can think of your computer's entire functioning, from boot to shutdown, as the evaluation of a set of stacks of frames. At boot, the system starts a C program named `init`, and every other program is a child of `init` (or a child of a child of init, or a child of a child of a child, et cetera).

There is also a `fork` function that generates a second stack that runs concurrently with the parent, which is how a system runs several programs at once.

The `system` function will pass back the return value of the subprogram's `main`. The general custom is that if `main` returns 0 then all went well, while a positive integer indicates a type of error.[17] Because so many people are not concerned with the return value of `main`, the current C standard assumes that `main` returns zero if no indication is given otherwise, which is how most of the programs in this book get away with not having a `return` statement in their `main` function.

SCOPE When one function is running, only the variables in that frame are visible: all of the variables in the rest of the program are dormant and inaccessible. This is a good thing, since you don't want to have to always bear in mind the current state of all the variables in your program, the GNU Scientific Library, the standard library, and who knows what else.

A variable's *scope* is the set of functions that can see the variable. A variable declared inside a function is visible only inside that function. If a variable is declared at the top of a file, then that variable is *global* to the file, and any function in that file can see that variable. If declared in a *header file* (see below, including an important caveat on page 50), then any function in a file that `#includes` that header can see the variable.

The strategy behind deciding on the scope of a variable is to keep it as small as possible.

Block scope

You can declare variables inside a loop.

- `while(...){int i;... }` works as you expect, declaring `i` only once.
- `while(...){int i=1;... }` will re-set `i` to one for every iteration of the loop.
- `for (int i=0; i<max; i++){...}` works, but `gcc` may complain about it unless you specify the `-std=c99` or `-std=gnu99` flags for the compiler.
- A variable declared inside a bracketed block (or at the header for a pair of curly braces, like a `for` loop) is destroyed at the end of the bracketed code. This is known as *block scope*. Function-level scope could be thought of a special case of block scope.

Block scope is occasionally convenient—especially the `for (int i;...)` form—but bear in mind that you won't be able to refer to a block-internal variable after the loop ends.

- If only one function uses a variable, then by all means declare the variable inside the function (possibly as a `static` variable).
- If a variable is used by only a few functions, then declare the variable in the `main` function and pass it as an argument to the functions that use it.
- If a variable is used throughout a single file and is infrequently changed, then let it be globally available throughout the file, by putting it at the top of the file, outside all the function bodies.

[17] In the bash shell (the default on many POSIX systems), the return value from the last program run is stored in `$?`, so `echo $?` will print its value.

- Finally, if a variable is used throughout all parts of a program consisting of multiple files, then declare it in a header file, so that it will be globally available in every file which `#includes` that header file (see page 50).[18]

There is often the temptation to declare every variable as global, and just not worry about scope issues. This makes maintaining and writing the code difficult: are you sure a tweak you made to the black box named `function_a` won't change the workings inside the black box named `function_b`? Next month, when you want to use `function_a` in a new program you have just written, you will have to verify that nothing in the rest of the program affects it, so what could have been a question of just cutting and pasting a black box from one file to another has now become an involved analysis of the original program.

- ➤ Good coding form involves breaking problems down into functions and writing each function as an independent entity.
- ➤ The header of a function is of the form *`function_type function_name(p1_type p1_name, p2_type p2_name, ...)`*.
- ➤ The computer evaluates each function as an independent entity. It maintains a stack of frames, and all activity is only in the current top frame.
- ➤ When a program starts, it will first build a frame for the function named `main`; therefore a complete program must have one and only one such function.
- ➤ Global variables are passed into a new frame, but only copies of parameters are passed in. If a variable is not in the frame, it is out of scope and can not be accessed.

[18] This is the appropriate time to answer a common intro-to-C question: What is the difference between C and C++? There is much confusion due to the almost-compatible syntax and similar name—when explaining the name C-double-plus, the language's author references the Newspeak language used in George Orwell's *1984* (Orwell, 1949; Stroustrup, 1986, p 4).

The key difference is that C++ adds a second scope paradigm on top of C's file- and function-based scope: object-oriented scope. In this system, functions are bound to objects, where an object is effectively a `struct` holding several variables and functions. Variables that are *private* to the object are in scope only for functions bound to the object, while those that are *public* are in scope whenever the object itself is in scope.

In C, think of one file as an object: all variables declared inside the file are private, and all those declared in a header file are public. Only those functions that have a declaration in the header file can be called outside of the file.

But the real difference between C and C++ is in philosophy: C++ is intended to allow for the mixing of various styles of programming, of which object-oriented coding is one. C++ therefore includes a number of other features, such as yet another type of scope called *namespaces*, templates and other tools for representing more abstract structures, and a large standard library of templates. Thus, C represents a philosophy of keeping the language as simple and unchanging as possible, even if it means passing up on useful additions; C++ represents an all-inclusive philosophy, choosing additional features and conveniences over parsimony.

2.4 THE DEBUGGER

The *debugger* is somewhat mis-named. A better name would perhaps be the *interrogator*, because it lets you interact with and ask questions of your program: you can look at every line as it is being executed, pause to check the value of variables, back up or jump ahead in the program, or insert an extra command or two. The main use of these powers is to find and fix bugs, but even when you are not actively debugging, you may still want to run your program from inside the debugger.

The joy of segfaults

There are a few ways in which your program can break. For example, if you attempt to calculate `1/0`, there is not much for the computer to do but halt.

Or, say that you have declared an array, `int data[100]` and you attempt to read to `data[1000]`. This is a location somewhere in memory, 901 `int`s' distance past the end of the space allocated for the array. One possibility is that `data[1000]` happens to fall on a space that has something that can be interpreted as an integer, and the computer processes whatever junk is at that location as if nothing were wrong. Or, `data[1000]` could point to an area of protected memory, such as the memory that is being used for the operating system or your dissertation. In this case, referring to `data[1000]` will halt the program with the greatest of haste, before it destroys something valuable. This is a *segmentation fault* (*segfault* for short), since you attempted to refer to memory outside of the segment that had been allocated for your program. Below, in the section on pointers, you will encounter the *null pointer*, which by definition points to nothing. Mistakenly trying to read the data a null pointer is pointing to halts the program with the complaint *attempting to dereference a null pointer*.

A segfault is by far the clearest way for the computer to tell you that you miscoded something and need to fire up the debugger.[19] It is much like refusing to compile when you refer to an undeclared variable. If you declared `receipts` and `data[100]`, then setting `reciepts` or `data[999]` to a value is probably an error. A language that saves you the trouble of making declarations and refuses to segfault will just produce a new variable, expand the array, and thus insert errors into the output that you may or may catch.

[19] In fact, you will find that the worst thing that can happen with an error like the above read of `data[1000]` would be for the program to *not* segfault, but to continue with bad data and then break a hundred lines later. This is rare, but when such an event becomes evident, you will need to use a memory debugger to find the error; see page 214.

The debugging process

To debug the program *run_me* under the debugger, type `gdb run_me` at the command line. You will be given the gdb prompt.[20]

You need to tell the compiler to include the names of the variables and functions in the compiled file, by adding the `-g` flag on the compiler command line. For example, instead of `gcc hello.c`, use `gcc -g hello.c`. If the debugger complains that it can't find any debugging symbols, then that means that you forgot the `-g` switch. Because `-g` does not slow down the program but makes debugging possible, you should use it every time you compile.

If you know the program will segfault or otherwise halt, then just start `gdb` as above, run the program by typing `run` at `gdb`'s prompt, and wait for it to break. When it does, you will be returned to gdb's prompt, so you can interrogate the program.

The first thing you will want to know is where you are in the program. You can do this with the `backtrace` command, which you can abbreviate to either `bt` or `where`. It will show you the stack of function calls that were pending when the program stopped. The first frame is always `main`, where the program started. If `main` called another function, then that will be the next frame, et cetera. Often, your program will break somewhere in the internals of a piece of code you did not write, such as in a call to `mallopt`. Ignore those. You did not find a bug in `mallopt`. Find the topmost frame that is in the code that you wrote.

At this point, the best thing to do is look at a listing of your code in another window and look at the line the debugger pointed out. Often, simply knowing which line failed is enough to make the error painfully obvious.

If the error is still not evident, then go back to the debugger and look at the variables. You need to be aware of which frame you are working in, so you know which set of variables you have at your disposal. You will default to the last frame in the stack; to change to frame number three, give the command `frame 3` (or `f 3` for short). You can also traverse the stack via the `up` and `down` commands, where `up` goes to a parent function and `down` goes to the child function.

Once you are in the frame you want, get information about the variables. You can get a list of the local variables using `info locals`, or information about the arguments to the function using `info args` (though the argument information is already in the frame description). Or, you can print any variable that you think may be in the frame using `print var_name`, or more briefly, `p var_name`. You

[20] Asking your favorite search engine for *gdb gui* will turn up a number of graphical shells built around gdb. Some are stand-alone programs like `ddd` and others are integrated into IDEs. They will not be discussed here because they work exactly like gdb, except that they involve using the mouse more.

Also, GDB itself offers many conveniences not described here. See Stallman *et al.* (2002) for the full story.

can print the value of any expression in scope: `p sqrt(var)` or `p apop_show_matrix(m)` will display the square root of `var` and the matrix `m`, provided the variables and functions are available to the scope in which you are working. Or, `p stddev = sqrt(var)` will set the variable `stddev` to the given value and print the result. Generally, you can execute any line of C code that makes sense in the given context via the `print` command.

GDB has a special syntax for viewing several elements of an array at once. If you would like to see the first five elements of the array `items`, then use: `p *items@5`.

Breaking and stepping If your program is doing things wrong but is not kind enough to segfault, then you will need to find places to halt the program yourself. Do this with the `break` command. For a program with only one file of code, simply give a line number: `break 35` will stop the program just before line 35 is evaluated.

- For programs based on many files, you may need to specify a file name: `break file2.c:35`.
- Or, you can specify a function name, and the debugger will stop at the first line after the function's header. E.g, `break calculate_days`.
- You may also want the program to break only under certain conditions, such as when an iterator reaches 10,000. I.e., `break 35 if counter > 10000`.[21]
- All breakpoints are given a number, which you can list with `info break`. You can delete break point number three with the command `del 3`.

Once you have set the breakpoints, `run` (or just `r`) will run the program until it reaches a break point, and then you can apply the interrogation techniques above. You may want to carefully step through from there:

- `s` will step to the next line to be evaluated, which could mean backing up in the current function or going to a subfunction.
- `next` or `n` will step through the function (which may involve backtracking) but will run without stopping in any subframes which may be created (i.e., if subfunctions are called).
- `until` or `u` will keep going until you get to the next line in the function, so the debugger will run through subfunctions and loops until forward progress is made in the current function.
- `c` will continue along until the next break point or the end of the program.[22]

[21] You can also set *watchpoints*, which tell gdb to watch a variable and halt if that variable changes, e.g., `watch myvar`. Watchpoints are not as commonly used as breakpoints, and sometimes suffer from scope issues.

[22] A mnemonic device for remembering which is which: `s` is the slowest means of stepping, `n` slightly faster, `u` still faster, and `c` the fastest. In this order, they spell `snuc`, which is almost an English word with implications of stepping slowly.

- `jump lineno` will jump to the given line number, so you can repeat a line with some variables tweaked, or skip over a few lines. Odd things will happen if you jump out of the frame in which you are working, so use this only to jump around a single function.
- `return` will exit the given frame and resume in the parent frame. You can give a return value, like `return` *`var`*.
- Just hitting <enter> will repeat the last command, so you won't have to keep hitting `n` to step through many lines.

Q2.7

Break `bdayfns.c` (from Listing 2.4) and debug it.

- Modify line 13 from `days[1]` to `days[-1]`.
- Recompile. Be sure to include the `-g` flag.
- Run the program and observe the output (if any).
- Start the debugger. If the program segfaulted, just type `run` and wait for failure; otherwise, insert a breakpoint, `break calculate_days`, and then `run`.
- Check the backtrace to see where you are on the stack. What evidence can you find that things are not right?

A note on debugging strategy Especially for numeric programs, the strategy in debugging is to find the first point in the chain of logic where things look askew. Below, you will see that your code can include *assertions* that check that things have not gone astray, and the debugger's break-and-inspect system provides another means of searching for the earliest misstep. But if there are no intermediate steps to be inspected, debugging becomes very difficult.

Say you are writing out the roots of the quadratic equation, $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, and erroneously code the first root as:

```
firstroot = -b + sqrt(b*b - 4 *a *c)/2*a; //this is wrong.
```

There are basically no intermediate steps: you put in `a`, `b`, and `c`, and the system spits out a bad value for `firstroot`. Now say that you instead wrote:

```
1 firstroot = -b;
2 firstroot += sqrt(b*b - 4*a*c);
3 firstroot = firstroot/2*a; //still wrong.
```

If you know that the output is wrong, you can interrogate this sequence for clues about the error. Say that `a=2`. As you step through, you find that the value of `firstroot` does not change after line three runs. From there, the error is obvious: the line should have been either `firstroot = firstroot/(2*a)` or `firstroot /= 2*a`. Such a chain of logic would be impossible with the one-line version of the routine.[23]

However, for the typical reader, the second version is unattractively over-verbose. A first draft of code should err on the side of inelegant verbosity and easy debugability. You can then incrementally tighten the code as it earns your trust by repeatedly producing correct results.

Q 2.8

A triangular number is a number like 1 ($\cdot$), 1+2=3 ($\begin{smallmatrix}\cdot\\ \cdot\,\cdot\end{smallmatrix}$), 1+2+3=6 ($\begin{smallmatrix}\cdot\\ \cdot\,\cdot\\ \cdot\,\cdot\,\cdot\end{smallmatrix}$), 1+2+3+4=10 ($\begin{smallmatrix}\cdot\\ \cdot\,\cdot\\ \cdot\,\cdot\,\cdot\\ \cdot\,\cdot\,\cdot\,\cdot\end{smallmatrix}$), et cetera. Fermat's polygonal number theorem states that any natural number can be expressed as the sum of at most three triangular numbers. For example, 13 = 10+3 and 19=15+3+1. Demonstrate this via a program that finds up to three triangular numbers for every number from 1 to 100.

- Write a function `int triangular(int i)` that takes in an index and returns the ith triangular number. E.g., `triangular(5)` would return 1+2+3+4+5=15. Write a `main` to test it.
- Use that function to write a function `int find_next_triangular(int in)` that returns the index of the smallest triangular number larger than its input. Modify `main` to test it.
- Write a function `void find_triplet(int in, int out[])` that takes in a number and puts three triangular numbers that sum to it in `out`. You can use `find_next_triangular` to find the largest triangular number to try, and then write three nested `for` loops to search the range from zero to the maximum you found. If the loop-in-a-loop-in-a-loop finds three numbers that sum to `in`, then the function can `return`, thus cutting out of the loops.
- Finally, write a `main` function that first declares an array of three `int`s that will be filled by `find_triplet`, and then runs a `for` loop that calls `find_triplet` for each integer from 1 to 100 and prints the result.

[23] The one-line version also has a second error; spotting it is left as a quick exercise for the reader.

- The debugger will allow you to view intermediate results at any point along a program's execution.
- You can either wait for the program to segfault by itself, or use `break` to insert breakpoints.
- You can execute and print any expression or variable using `p variable`.
- Once the program has stopped, use `s`, `n`, and `u` to step through the program at various speeds.

2.5 COMPILING AND RUNNING

The process of compiling program text into machine-executable instructions relies heavily on the system of frames. If function A calls function B, the compiler can write down the instructions for creating and executing function A without knowing anything about function B beyond its declaration. It will simply create a frame with a series of instructions, one of which is a call to function B. Since the two frames are always separate, the compiler can focus on creating one at a time, and then link them later on.

What to type

To this point, you have been using a minimal command line to compile programs, but you can specify much more. Say that we want the compiler to

- include symbols for debugging (`-g`),
- warn us of all potential coding errors (`-Wall`),
- use the C99 and POSIX standards (`-std=gnu99`),
- compile using two source files, `file1.c` and `file2.c`, plus
- the sqlite3 and standard math library (`-lsqlite3 -lm`), and finally
- output the resulting program to a file named `run_me` (`-o run_me`).

You could specify all of this on one command line:

```
gcc -g -Wall -std=gnu99 file1.c file2.c -lsqlite3 -lm -o run_me
```

This is a lot to type, so there is a separate program, `make`, which is designed to facilitate compiling. After setting up a makefile to describe your project, you will be able to simply type `make` instead of the mess above. You may benefit from reading Appendix A at this point. Or, if you decide against using `make`, you could

write yourself an alias in your shell, write a batch file, or use an IDE's compilation features.

Multiple windows also come in handy here: put your code in one window and compile in another, so you can see the inevitable compilation errors and the source code at the same time. Some text editors and IDEs even have features to compile from within the program and then step you through the errors returned.

The components Even though we refer to the process above as *compilation*, it actually embodies three separate programs: a preprocessor, a compiler, and a linker.[24]

The three sub-programs embody the steps in developing a set of frames: the preprocessor inserts header files declaring functions to be used, the compilation step uses the declarations to convert C code into machine instructions about how to build and execute a standalone frame, and the linking step locates all the disparate frames, so the system knows where to look when a function call is made.

THE PREPROCESSING STEP The *preprocessor* does nothing but take text you wrote and convert it into more text. There are a dozen types of text substitutions the preprocessor can do, but its number one use is dumping the contents of header files into your source files. When the preprocessor is processing the file `main.c` and sees the lines

```
#include <gsl/gsl_matrix.h>
#include "a_file.h"
```

it finds the `gsl_matrix.h` and the `a_file.h` *header files*, and puts their entire contents verbatim at that point in the file. You will never see the expansion (unless you run `gcc` with the `-E` flag); the preprocessor just passes the expanded code to the compiler. For example, the `gsl_matrix.h` header file declares the `gsl_matrix` type and a few dozen functions that act on it, and the preprocessor inserts those declarations into your program, so you can use the structure and its functions as if you'd written them yourself.

The angle-bracket form, `#include <gsl/gsl_matrix.h>` indicates that the preprocessor should look at a pre-specified include path for the header; use this for the headers of library files, and see Appendix A for details. The `#include "a_file.h"` form searches the current directory for the header; use this for header files you wrote yourself.[25]

[24] As a technical detail which you can generally ignore in practice, the preprocessor and compiler are typically one program, and the linker is typically a separate program.

[25] The `#include "a_file.h"` form searches the include path as well, so you could actually use it for both home-grown and system `#includes`. In practice, the two forms serve as an indication of where one can find the given header file, so most authors use the <> form even though it is redundant.

Header aggregation

The Apophenia library provides a convenience header that aggregates almost every header you will likely be using. By placing
`#include <apop.h>`
at the top of your file, you should not need to include any of the other standard headers that one would normally include in a program for numerical analysis (`stdio.h`, `stdlib.h`, `math.h`, `gsl_anything.h`). This means that you could ignore the headers at the top of all of the code snippets in this chapter.

Of course, you will still need to include any headers you have written, and if the compiler complains about an undeclared function, then its header is evidently not included in `apop.h`.

Many programming languages have a way to declare variables as having *global scope*, meaning that every function everywhere can make use of the variable. Technically, C has no such mechanism. Instead, the best you can do is what I will call *file-global scope*, meaning that every function in a single file can see any variable declared above it in that file.

Header files allow you to simulate truly global scope, but with finer control if you want it. If some variables should be global to your entire program, then create a file named `globals.h`, and put all declarations in that file (but see below for details). By putting `#include "globals.h"` at the top of every file in your project, all variables declared therein are now project-global. If the variables of `process.c` are used in only one or two other code files, then project-global scope is overkill: `#include "process.h"` only in those few code files that need it.

Q2.9

In prior exercises, you wrote a program with one function to create an array of numbers and their squares (page 30), and another function, `print_array`, to print those values (page 37).

- Move `print_array` to a new text file, `utility_fns.c`.
- Write the corresponding one-line file `utility_fns.h` with `print_array`'s header.
- `#include "utility_fns.h"` in the main square-printing program.
- Modify the square-calculating code to call `print_array`.
- Compile both files at once, e.g., `gcc your_main.c utility_fns.c`.
- Run the compiled program and verify that it does what it should.

※ *Variables in headers* The system of putting file-scope variables in the base `.c` file and global-scope variables in the `.h` file has one ugly detail. A function declaration is merely advice to the compiler that your function

has certain inputs and outputs, so when multiple files reread the declaration, the program suffers only harmless redundancy. But a variable declaration is a command to the compiler to allocate space as listed. If `head.h` includes a declaration `int x;`, and `file1.c` includes `head.h`, it will set aside an `int`'s worth of memory and name it `x`; if `file2.c` includes the same header, it will also set aside some memory named `x`. So which bit of memory are you referring to when you use `x` later in your code?

C's solution is the `extern` keyword, which tells the compiler that the declaration that follows is not for memory allocation, but is purely informative. Simply put it in front of the normal declaration: `extern int x; extern long double y;` will both work. Then, in one and only one `.c` file, declare the variables themselves, e.g., `int x; long double y = 7.0`. Thus, all files that `#include` the header know what to make of the variable `x`, so `x`'s scope is all files with the given header, but the variable is allocated only once.

To summarize: function declarations and `typedef`s can go into a header file that will be included in multiple `.c` files. Variables need to be declared as usual in one and only one `.c` file, and if you want other `.c` files to see them, re-declare them in a header file with the `extern` keyword.

THE COMPILATION STEP The *compilation* stage consists of taking each `.c` file in turn and writing a machine-readable *object file*, so `file1.c` will result in `file1.o`, and `file2.c` will compile to `file2.o`. These object files are self-encapsulated files that include a table of all of the symbols declared in that file (functions, variables, and types), and the actual machine code that tells the computer how to allocate memory when it sees a variable and how to set up and run a frame when it sees a function call. The preprocessor inserted declarations for all external functions and variables, so the compiler can run consistency checks as it goes.

The instructions for a function may include an instruction like *at this point, create a frame for* `gsl_matrix_add` *with these input variables*, but executing that instruction does not require any knowledge of what `gsl_matrix_add` looks like—that is a separate frame in which the current frame has no business meddling.

THE LINKING STEP After the compilation step, you will have on hand a number of standalone frames. Some are in `.o` files that the compiler just created, and some are in libraries elsewhere on the system. The *linker* collects all of these elements into a single executable, so when one function's instructions tell the computer to go evaluate `gsl_matrix_add`, the computer will have no problem locating and loading that function. Your primary interaction with the linker will be

in telling it where to find libraries, via `-l` commands on the compiler command line (`-lgsl -lgslcblas -lm`, et cetera).

Note well that a library's header file and its object file are separate entities—meaning that there are two distinct ways in which a call to a library function can go wrong. To include a function from the standard math library like `sqrt`, you will need to (1) tell the preprocessor to include the header file via `#include <math.h>` in the code, and (2) tell the linker to link to the math library via a `-l` flag on the command line, in this case `-lm`. Appendix A has more detail on how to debug your `#include` statements and `-l` flags.

※ *Finding libraries* An important part of the art of C programming is knowing how to find libraries that will do your work for you, both online and on your hard drive.

- The first library to know is the standard library. Being standard, this was installed on your computer along with the compiler. If the documentation is not on your hard drive (try `info glibc`), you can easily find it online. It is worth giving the documentation a skim so you know which wheels to not reinvent.
- The GNU/UNESCO website (`gnu.org`) holds a hefty array of libraries, all of which are free for download.
- `sourceforge.net` hosts on the order of 100,000 projects (of varying quality). To be hosted on Sourceforge, a project must agree to make its code public, so you may fold anything you find there into your own work.
- Finally, you can start writing your own library, since next month's project will probably have some overlap with the one you are working on now. Simply put all of your functions relating to *topic* into a file named *topic*`.c`, and put the useful declarations into a separate header file named *topic*`.h`. You already have a start on creating a utility library from the exercise on page 50.

∑

➤ Compilation is a three-step process. The first step consists of text expansions like replacing `#include <`*header*`.h>` with the entire contents of *header*`.h`.

➤ Therefore, put public variable declarations (with the `extern` keyword) and function declarations in header files.

➤ The next step consists of compilation, in which each source (`.c`) file is converted to an object (`.o`) file. ⋙

⋙

- Therefore, each source file should consist of a set of standalone functions that depend only on the file's contents and any declarations included via the headers.
- The final step is linking, in which references to functions in other libraries or object files are reconciled.
- Therefore, you can find and use libraries of functions for any set of tasks you can imagine.

2.6 POINTERS

Pointers will change your life. If you have never dealt with them before, you will spend some quantity of time puzzling over them, wondering why anybody would need to bother with them. And then, when you are finally comfortable with the difference between data and the location of data, you will wonder how you ever wrote code without them.

Pointers embody the concept of the *location of data*—a concept with which we deal all the time. I know the location `http://nytimes.com`, and expect that if I go to that location, I will get information about today's events. I gave my colleagues an email address years ago, and when they have new information, they send it to that location. When so inclined, I can then check that same location for new information. Some libraries are very regimented about where books are located, so if you need a book on probability (Library of Congress classification QA273) the librarian will tell you to go upstairs to the third bookshelf on the left. The librarian did not have to know any information about probability, just the location of such information.

Returning to the computer for a moment, when you declare `int k`, then the computer is going to put `k` somewhere in memory. Perhaps with a microscope, you could even find it: there on the third chip, two hundred transistors from the bottom. You could point to it.

Lacking a finger with which to point, the computer will use an illegible hexadecimal location, but you will never have to deal with the hexadecimal directly, and lose nothing by ignoring the implementation and thinking of pointers as just a very precise finger, or a book's call number.

The confusing part is that the location of data is itself data. After all, you could write "QA273" on as slip of paper as easily as "$P(A \cap B) = P(A|B)P(B)$."

Further, the location of information may itself have a location. Before computers took over, there was a card catalog somewhere in the library, so you would have to go to the card catalog—the place where location data is stored—and then look up the location of your book. It sometimes happens that you arrive at the QA273 shelf and find a wood block with a message taped to it saying "oversized books are at the end of the aisle."

In these situations, we have no problem distinguishing between information that is just the location of data and data itself. But in the computing context, there is less to guide us. Is 8,049,588 just a large integer (data), or a memory address (the location of data)?[26] C's syntax will do little to clear up the confusion, since a variable like `k` could be integer data or the location of integer data. But C uses the location of data to solve a number of problems, key among them being function calls that allow inputs to be modified and the implementation of arrays.

Call-by-address v call-by-value First, a quick review of how functions are called: when you call a function, the computer sets up a separate frame for the function, and puts into that frame *copies* of all of the variables that have been passed to the function. The function then does its thing and produces a return value. Then, the entire frame is destroyed, including all of the copies of variables therein. A copy of the return value gets sent back to the main program, and that is all that remains of the defunct frame.

This setup, known as call-by-value since only values are passed to the function, allows for a more stable implementation of the paradigm of standalone frames. But if `k` is an array of a million `doubles`, then making a copy every time you call a common function could take a noticeable amount of time. Also, you will often want your function to change the variables that get sent to it.

Pointers fix these problems. The trick is that instead of sending the function a copy of the variable, we send in a copy of the location of the variable: we copy the book's call number onto a slip of paper and hand that to the function. In Figure 2.7, the *before* picture shows the situation before the function call, in the main program: there is a pointer to a location holding the number six. Then, in the *during* picture, a function is called with the pointer as an argument, via a form like `fn_call(pointer)`. There are now two fingers, original and copy, pointing to the same spot, but the function knows only about the copy. Given its copy of a finger, it is easy for the function to change the value pointed to to seven. When the function returns, in the *after* picture, the copy of a finger is destroyed but the changes are not undone. The original finger (which hasn't changed and is pointing to the same place it was always pointing to) will now be pointing to a modified value.

[26] This number is actually an address lifted from my debugger, where it is listed as `0x08049588`. The `0x` prefix indicates that the number is represented in hexadecimal.

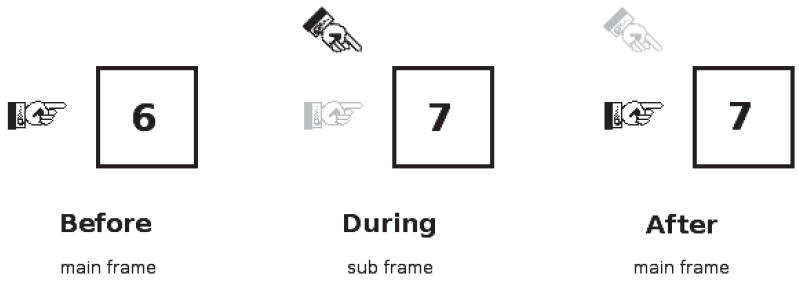


Figure 2.7 Before, during, and after a function call that modifies a pointed-to value

Returning to C syntax, here are the rules for using pointers:

- To declare a pointer to an integer, use `int *k`.
- Outside the declarations, to refer to the integer being pointed to, use `*k`.
- Outside the declarations, to refer to the pointer itself, use `k`.

The declaration `int *p, i` means that `p` will be a pointer to an integer and `i` is an integer, but in a non-declaration line like `i = *p`, `*p` refers to the integer value that `p` points to. There is actually a logical justification for the syntax, which I will not present here because it tends to confuse more often than it clarifies. Instead, just bear in mind that the star effectively means something different in declarations than in non-declaration use.

To give another example, let us say that we are declaring a new pointer `p2` that will be initialized to point to the same address as `p`. Then the declaration would be `int *p2 = p`, because `p2` is being *declared* as a pointer, and `p` is being *used* as a pointer.

The spaces around our stars do not matter, so use whichever of `int *k`, `int* k`, or `int * k` you like best. General custom prefers the first form, because it minimizes the chance that you will write `int * k, b` (allocate a pointer named `k` and an `int` named `b`) when you meant `int *k, *b` (allocate two pointers). The star also still means multiply. There is never ambiguity, but if this bothers you, use parentheses.

Listing 2.8 shows a sample program that uses C's pointer syntax to implement the call-by-address trick from Figure 2.7.

```
1  #include <stdio.h> //printf
2  #include <malloc.h> //malloc
3
4  int globe=1; //a global variable.
5
6  int factorial (int *a_c){
7        while (*a_c){
8              globe *= *a_c;
9              (*a_c) --;
10       }
11        return globe;
12 }
13
14 int main(void){
15    int *a = malloc(sizeof(int));
16       *a = 10;
17       printf("%i factorial ...", *a);
18       printf(" is %i.\n", factorial(a));
19       printf("*a= %i\n", *a);
20       printf("globe= %i\n", globe);
21       free(a);
22       return 0;
23 }
```

Listing 2.8 A version of the factorial program using call-by-address. Online source: `callbyadd.c`.

- In the `main` function, `a` is a pointer—the address of an integer—as indicated by the star in its declaration on line 15; the `malloc` part is discussed below.
- To print the integer being pointed to, as on line 19, we use `*a`.
- The header for a function is a list of declarations, so on line 6, `factorial(int *a_c)` tells us that the function takes a pointer to an integer, which will be named `a_c`.
- Thus, in non-declaration use like lines eight and nine, `*a_c` is an integer.

Now for the call-by-address trick, as per Figure 2.7. When the call to `factorial` is made on Line 18, the pointer `a` gets passed in. The computer builds itself a frame, using a copy of `a`—that is, a copy of the location of an integer. Both `a` (in the `main` frame) and `a_c` (in the `factorial` frame) now point to the same piece of data. Line 9, `(*a_c)--`, tells the computer to go to the address `a_c` and decrement the value it finds there. When the frame is destroyed (and `a_c` goes with it), this will not be undone: that slot of memory will still hold the decremented value. Because `*a`—the integer `a` points to—has changed as a side effect to calling the `factorial` function, you saw that line 19 printed `*a=0` when you ran the program.

Dealing with memory Finally, we must contend with the pointer initialization on line 15:

```
int *a = malloc(sizeof(int));
```

Malloc, a function declared in `stdlib.h`, is short for *memory allocate*. By the library metaphor, `malloc` builds bookshelves that will later hold data. Just as we have no idea what is in an `int` variable before it is given a value, we have no idea what address `a` points to until we initialize it. The function `malloc()` will do the low-level work of finding a free slot of memory, claiming it so nothing else on the computer uses it, and returning that address. The input to `malloc` is the quantity of memory we need, which in this case is the size of one integer: `sizeof(int)`.

There are actually three characteristics to a given pointer: the location (where the finger is pointing), the type (here, `int`), and the amount of memory which has been reserved for the pointer (`sizeof(int)` bytes—enough for one integer). The location is up to the computer—you should never have to look at hexadecimal addresses. But you need to bear in mind the type and size of your pointer. If you treat the data pointed to by an `int` pointer as if it is pointing to a `double`, then the computer will read good data as garbage, and if you read twenty variables from a space allocated for fifteen, then the program will either read garbage or segfault.

The ampersand

Every variable has an address, whether you declared it as a pointer or not. The ampersand finds that address: if `count` is an integer, then `&count` is a pointer to an integer. The ampersand and star are inverses: `*(&count) == count`, which may imply that they are symmetric, but the star will appear much more often in your code than the ampersand, and an ampersand will never appear in a declaration or a function header. As a mnemonic, *ampersand*, *and sign*, and *address of* all begin with the letter A.

By the way, `int *k = 7` will fail—the initialization on the declaration line is for the pointer, not the value the pointer holds. Given that k is a pointer to an integer, all of these lines are correct:

```
int *k = malloc(sizeof(int));
*k = 7;
k = malloc(sizeof(int));
```

One convenience that will help with allocating pointers is `calloc`, which you can read as *clear and allocate*: it will run `malloc` and return the appropriate address, and will also set everything in that space to zero, running `*k = 0` for you. Sample usage:

```
int *k = calloc(1, sizeof(int));
```

The syntax requires that we explicitly state that we want one space, the size of an integer. You need to give more information than `malloc` because the process of putting a zero in a `double` pointer may be different from putting a zero in a `int` pointer. Thus `calloc` requires two arguments: `calloc(`*`element_count`*`, sizeof(`*`element_type`*`))`.

Finally, both allocation and de-allocation are now your responsibility. The de-allocation comes simply by calling `free(k)` when you are done with the pointer `k`. When the program ends, the operating system will free all memory; some people free all pointers at the end of `main` as a point of good form, and some leave the computer to do the freeing.

$\mathbb{Q}_{2.10}$

Write a function named `swap` that takes two pointers to `int` variables and exchanges their values.

- First, write a `main` function that simply declares two `int`s (not pointers) `first` and `second`, gives them values, prints the values and returns. Check that it compiles.
- Then, write a `swap` function that accepts two pointers, but does nothing. That is, write out the header but let the body be `{ }`.
- Call your empty function from `main`. Do you need to use `&first` (as per the box on page 57), `*first`, or just `first`? Check that the program still compiles.
- Finally, write the swap function itself. (*Hint*: include a local variable `int temp`.) Add a `printf` to the end of `main` to make sure that your function worked.

$\mathbb{Q}_{2.11}$

Modify your swap program so that the two variables in `main` are now pointers to `int`s.

- Add allocations via `malloc`, either in the declaration itself or on another line.
- Which of `&first`, `*first`, or `first` will you need to send to `swap` now? Do you need to modify the `swap` function itself?

$\mathbb{Q}_{2.12}$

Modify `callbyadd.c` so that the declaration on line 15 is for an integer, not a pointer. That is, replace the current `int *a = malloc(...);` with `int a;`. Make the necessary modifications to `main` to get the program running. Do not modify the `factorial` function.

- A variable can hold the address of a location in memory.
- By passing that address to a function, the function can modify that location in memory, even though only copies of variables are passed into functions.
- A star in a declaration means the variable is a pointer, e.g., `int *k`. A star in a non-declaration line indicates the data at the location to which the pointer points, e.g., `two_times = *k + *k`.
- The space to which a pointer is pointing needs to be prepared using `malloc`, e.g., `int *integer_address = malloc(sizeof(int));`.
- When you are certain a pointer will not be used again, free it, e.g., `free(integer_address)`.

2.7 ARRAYS AND OTHER POINTER TRICKS

You can use a pointer as an array: instead of pointing to a single integer, for example, you can point to the first of a sequence of integers. Listing 2.9 shows some sample code to declare an array and fill it with square numbers:

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

int main(){
  int array_length=1000;
  int *squares = malloc (array_length * sizeof(int));
    for (int i=0; i < array_length; i++)
          squares[i] = i * i;
}
```

Listing 2.9 Allocate an array and fill it with squares. Online source: `squares.c`.

The syntax for declaring the array exactly matches that of allocating a single pointer, except we needed to allocate a block of size `1000 * sizeof(int)` instead of just a single `sizeof(int)`. Referring to an element of `squares` uses identical syntax to the automatically-declared arrays at the beginning of this chapter. Internally, both types of array are just a sequence of blocks of memory holding a certain data type.

$\mathbb{Q}_{2.13}$ The listing in `squares.c` is not very exciting, since it has no output. Add a second `for` loop to print a table of squares to the screen, by printing the index `i` and the value in the `squares` array at position `i`.

$\mathbb{Q}_{2.14}$ After the `for` loop, `squares[7]` holds a plain integer (49). Thus, you can refer to that integer's address by putting a `&` before it. Extend your version of `squares.c` to use your swap function to swap the values of `squares[7]` and `squares[8]`.

But despite their many similarities, arrays and pointers are not identical: one is automatically allocated memory and the other is manually allocated. Given the declarations

```
double a_thousand_doubles[1000];
// and
double *a_thousand_more_doubles = malloc(1000 * sizeof(double));
```

the first declares an automatically allocated array, just as `int i` is automatically allocated, and therefore the allocation and de-allocation of the variable is the responsibility of the compiler. The second allocates memory that will be at the location `a_thousand_doubles` until you decide to free it.

In function arguments, you can interchange their syntaxes. These are equivalent:

```
int a_function(double *our_array);
//and
int a_function(double our_array[]);
```

But be careful: if your function `frees` an automatically allocated array passed in from the parent, or assigns it a new value with `malloc`, then you are stepping on C's turf, and will get a segfault.

ARRAYS OF STRUCTS Before, when we used the `struct` for complex numbers, we referred to its elements using a dot, such as `a.real` or `b.imaginary`. For a pointer to a structure, use `->` instead of a dot. Here are some examples using the definition of the `complex` structure from page 31.

```
complex *ptr_to_cplx = malloc (sizeof(complex));
ptr_to_cplx->real = 2;
ptr_to_cplx->imaginary = -2;
complex *array_of_cplxes = malloc (30 * sizeof(complex));
array_of_cplexes[15]->real = 3;
```

If you get an error like *request for member 'real' in something not a structure or union* then you are using a dot where you should be using `->` or vice versa. Use that feedback to understand what you misunderstood, then switch to the other and try again.

REALLOCATING If you know how many items you will have in your array, then you probably won't bother with pointers, and will instead use the `int fixed_list[300]` declaration, so you can leave the memory allocation issues to the computer. But if you try to put 301 elements in the list (which, you will recall, means putting something in `fixed_list[300]`), then you will be using memory that the machine hadn't allocated for you—a segfault.

If you are not sure about the size of your array, then you will need to expand the array as you go. Listing 2.10 is a program to find *prime numbers*, with a few amusing tricks thrown in. Since we don't know how many primes we will find, we need to use `realloc`. The program runs until you hit <ctrl-c>, and then dumps out the complete list to that point.

- Line 13: `SIGINT` is the signal that hitting <ctrl-c> sends to your program. By default, it halts your program immediately, but line 13 tells the system to call the one-line function on line 9 when it receives this signal. Thus, the `while` loop beginning at line 14 will keep running until you hit <ctrl-c>; then the program will continue to line 26.
- Lines 16–17: Check whether `testme` is evenly divisible by `primes[i]`. The second element of the `for` loop, the run-while condition, includes several conditions at once.
- Line 19: Computers in TV and movies always have fast-moving counters on the screen giving the impression that something important is happening, and line 19 shows how it is done. The newline `\n` is actually two steps: a carriage return (go to beginning of line) plus a line feed (go to next line). The `\r` character is a carriage return with no line feed, so the current line will be rewritten at the next `printf`. The `fflush` function tells the system to make sure that everything has been written to the screen.
- Lines 20–24: Having found a prime number, we add it to the list, which is a three-step process: reallocate the list to be one item larger, put the element in the last space, and add one to the counter holding the size of the array. The first argument to `realloc` is the pointer whose space needs resizing, and the second argument is the new size. The first part of this new block of memory will be the `primes` array so far, and the end will be an allocated but garbage-filled space ready for us to fill with data.

```
1  #include <math.h>
2  #include <stdio.h>
3  #include <signal.h>
4  #include <malloc.h>
5
6  int ct =0, keepgoing = 1;
7  int *primes = NULL;
8
9  void breakhere(){ keepgoing = 0; }
10
11 int main(){
12    int i, testme = 2, isprime;
13       signal(SIGINT, breakhere);
14       while(keepgoing){
15            isprime = 1;
16            for (i=0; isprime && i< sqrt(testme) && i<ct; i++)
17                 isprime = testme % primes[i];
18            if (isprime){
19                 printf("%i \r", testme); fflush(NULL);
20                 primes = realloc(primes, sizeof(int)*(ct+1));
21                 primes[ct] = testme;
22                 ct ++;
23            }
24            testme ++;
25       }
26       printf("\n");
27       for (i=0; i< ct; i++)
28          printf("%i\t", primes[i]);
29       printf("\n");
30 }
```

Listing 2.10 Find prime numbers and put them in an array. Online source: `primes.c`.

SOME COMMON FAUX PAS Figure 2.11 shows two errors in pointer handling. The first step picks up from the second step of Figure 2.7: a function has been called with a pointer as an argument. Then, the function frees what the copy of a pointer is pointing to—and thus frees what the original finger was pointing to. Next, it allocates new space, moving the copy of a finger to point to a new location. But when the function finishes, depicted in the final step, the copy of a pointer is destroyed, so there is no way to refer to the `malloc`ed space in the main program. We now have a pointer in the main frame with no space and a space with no pointer.

Returning to the library metaphor for a moment, the calling function wrote down a call number on a slip of paper, handed it to the function, and the function then went into the shelves and moved things around. But since the function has no mechanism of telling the caller where it put things, the shelves are now a mess.

Listing 2.12 is a repeat of the prime-finding code, with the cute tricks removed and

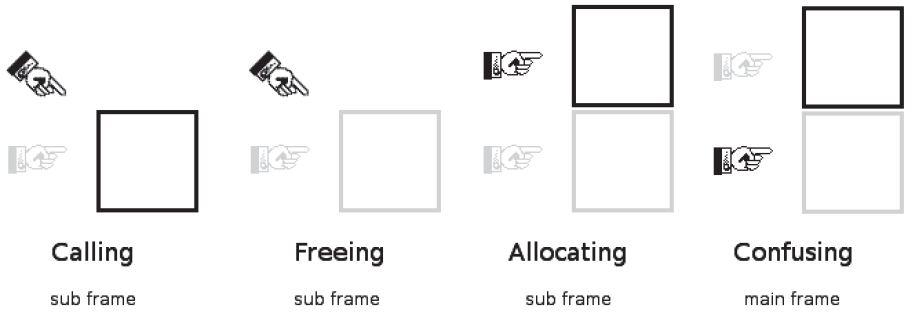


Figure 2.11 How to mess up your pointers

```
1  #include <math.h>
2  #include <stdio.h>
3  #include <malloc.h>
4
5  void add_a_prime(int addme, int *ct, int **primes){
6      *primes = realloc(*primes, sizeof(int)*(*ct+1));
7      (*primes)[*ct] = addme;
8      (*ct)++;
9  }
10
11 int main(){
12    int ct =0, i, j, testme = 2, isprime, max = 1000;
13    int *primes = NULL;
14       for (j=0; j< max; j++){
15            isprime = 1;
16            for (i=0; isprime && i< sqrt(testme) && i<ct; i++)
17                 isprime = testme % primes[i];
18            if (isprime)
19                 add_a_prime(testme, &ct, &primes);
20            testme ++;
21       }
22       for (i=0;i< ct; i++)
23          printf("%i\t", primes[i]);
24       printf("\n");
25 }
```

Listing 2.12 Find prime numbers and put them in an array. Online source: `primes2.c`.

the process of adding a prime relegated to a separate function (as it should be). The wrong way to implement the function in lines 5–9 would be

```
void add_a_prime_incorrectly(int addme, int *ct, int *primes){
    primes = realloc(primes, sizeof(int)*(*ct+1));
    primes[*ct] = addme;
    (*ct)++;
}
```

This commits the above faux pas of changing the value of the *copy* of `primes` and allocating new data at the new location—but whatever called that function has no idea about these changes in `primes`, and will keep on pointing to what may now be an invalid location.

The correct method is shown in Listing 2.12. It passes a pointer to the `primes` pointer—the location of the location of data. The calling function sends in the location of the card catalog, and the called function can then revise the locations listed in the card catalog when it makes changes on the shelves.

The syntax may seem confusing, but compare it to how `ct` is treated. Because the function will modify it, line 19 sends in its location, `&ct`, and the function header has an extra star: instead of `int ct`, it refers to the address `int *ct`. Similarly, the function call sends in the array's address, `&primes`, and the function adds a star to the header, `int **primes`.

$\mathbb{Q}_{2.15}$

Kernighan & Pike (1999) point out that `realloc` can be slow, so you are better off not calling it every time an array is extended. Instead, they suggest that every time an array of length n is `realloc`ed, its size be doubled to $2n$. Thus, if an array will eventually have ten elements, it will be `realloc`ed when adding the first, second, fourth, and eighth data points, for a total of four reallocations instead of ten.
Rewrite the code in figure `primes2` to implement this method of selective reallocation.

$\sum$

- Arrays are internally represented as pointers. The `int` *`sarray`*`[100]` form creates an automatically-allocated array, where the computer creates and destroys the array; the `int *`*`harray`*`= malloc(sizeof(int) * 100);` form creates a manually-allocated array that is yours to allocate and deallocate.
- Refer to array elements in both cases using the same square-brackets notation, such as `harray[14]`.
- Arrays of structs can be declared just as with arrays of basic variables. Refer to the elements in a pointer-to-struct using `->`.
- You can expand a manually-allocated array using `realloc`.
- Just as copies of normal variables are passed to functions, copies of pointers are sent in. Therefore, be careful when modifying a pointer in a subfunction.

2.8 STRINGS

C's handling of text *strings* is simple, elegant, unpleasant and awkward. Although this book is oriented toward programs about manipulating numbers rather than building Web pages or other such text, words and phrases are inevitable in even the mathiest of programs. So this section will cover the basics of how a system deals with variable-length text.

There are three approaches to dealing with C's awkwardness regarding text. You can (and should) skim the standard library documentation, to get to know what functions are always available for the most common string operations. You can use a higher-level library-provided data type for text, such as Glib's `GString` type (see Chapter 6 on Glib). Or, you can leave C entirely and do text manipulation via a number of command-line tools or a text-focused language like Ruby or Perl (see Appendix B). Nonetheless, all of these methods are based on C's raw string-handling at their core, and the raw C methods are always at hand, so it is worth getting to know C's string-handling even if you prefer higher-level methods.

C implements lines of text such as `"hello"` as arrays of individual characters, followed by an invisible null character, written as `\0`. That is, you may think of the above single word as a shorthand for the array `{'h', 'e', 'l', 'l', 'o', '\0'}`. This means that you have to think in terms of arrays when dealing with strings of characters.

The first implication of the elegant use of arrays to represent text is that your expectations about assignment won't work. Here are some examples:

```
1 char hello[30];
2 char hello2[] = "Hi.";
3 hello = hello2; //This is probably not what you meant.
4 hello = "Hi there"; //Nor is this.
```

- Line one shows that strings are declared with array-style declarations, either of the static form here or via `malloc`.
- Line two shows that, as with arrays of integers, we can specify a list to put in to the array when we initialize the array, but not later.
- But it is a common error to think that line three will copy the text `"Hi."` into `hello`, but as with pointers to integers, the actual function of third line is a pointer operation: instead of copying the data pointed to by `hello2` to the location pointed to by `hello`, it simply copies the location of `hello2`. When you change the text at one pointer later on, the other (now pointing to the same location) will change as well. Along a similar vein, line four also does not behave as you may expect.[27]

[27] Line four will compile, because the string `"Hi there."` is held in memory somewhere, so the pointer `hello` can point to it. However, a pointer to literal text is a `const` pointer, meaning that the first time you try to change `hello`, the program will crash.

Instead, there are a series of functions that take in strings to copy and otherwise handle strings.

`strlen` There are two lengths associated with a string pointer: the first is the space `malloc`ed for the pointer, and the second is the number of characters until the string-terminating `'\0'` character appears. Size of free memory is your responsibility, but `strlen(`*your_string*`)` will return the number of text characters in *your_string*.

`strncpy` Continuing the above example, this is the right way to copy data into `hello`:

```
#include <string.h>
strncpy(hello, "Hi there.", 30);
strncpy(hello, hello2, 30);
```

The third argument is the total space `malloc`ed to *hello*, not `strlen(`*hello*`)`.

`strncat` Rather than overwriting one string with another, you can also append (i.e., concatenate) one string to another, using

```
strncat(base_string, addme, freespace);
```

For example, `strncat(hello, hello2, 30)` will leave `"Hi there. Hello."` in `hello`.

Q2.16 The key problem with pointers-as-strings is that editing a string often becomes a three-step problem: measure the length the string will have after being changed, then `realloc` the string to the appropriate size, then finally make the change to the string.
Write a function `astrncpy(char **base, char *copyme)`, that will copy `copyme` to `*base`. Internally, it will use `strlen`, `realloc`, and `strncpy` to execute the three steps of string extension. All of the above discussion regarding re-pointing pointers inside a function applies here, which is why the function needs to take in a pointer-to-pointer as its first argument. Once this function is working, write a function `astrncat` that executes the same procedure for string concatenation. After you've tested both functions, add them to your library of utilities.

snprintf The snprintf function works just like the printf statements above, but prints its output to a string instead of the screen.[28] The fill-in-the-blanks syntax for printf works in exactly the same manner with strings.

```
#include <string.h>
...
int string_length = 1000;
char write_to_me[string_length];
char name[] = "Steven";
int position = 3;
snprintf(write_to_me, string_length, "person %s is number %i in line\n", name, position);
```

However, the three-step process of measuring the new string, calling realloc, and then finally modifying the string is especially painful for snprintf, because it is hard to know how long the printf-style format specifier will be after all its blanks are filled in. One way to make sure there is enough room for adding more text would be to simply allocate an absurd amount of space for each string, like just under a megabyte of memory: char hello[1000000]. You won't notice the wasted memory on a modern computer, but this method is also error-prone: what if you have a brilliant idea about a for loop that will add a little text for each of a million data points into the string?

asprintf If you are using the GNU C library, BSD's standard C library, or Apophenia, you could use asprintf, which allocates a string that is just big enough to handle the inputs, and then runs snprintf. Here is a simple example, with no memory allocation in sight, to print to line.

```
char *line;
asprintf(&line, "%s is number %i in line.", "Steven", 7);
```

Once again, because asprintf will probably move line in memory, we need to send the location of the pointer, not the pointer itself.

You can comfortably put asprintf into a for loop without worrying about overflow. Here is a snippet to write a string that counts to a hundred:[29]

[28] The other nice feature of snprintf is that it is more secure: other common functions like strcpy and sprintf do not check the length of the input, and so make it easy for you to inadvertently overwrite important bits of memory with the input string, including the location of the next instruction to be executed. Unsafe string-handling functions are thus a common security risk, allowing the execution of malicious code.

[29] Because asprintf does not free the current space taken up by string before reallocating the new version, this for loop is a memory leak. In many situations it isn't enough of a leak to matter, but if it is, you will need to use:

```
for (int i =0; i< 100; i++){
     char *tmp = string;
     asprintf(&string, "%s %i", string, i);
     free(tmp);
}
```

```
char *string = NULL;
asprintf(&string, ""); //initialize to empty, non-NULL string.
for (int i =0; i< 100; i++)
    asprintf(&string, "%s %i", string, i);
```

If you don't have `asprintf` on hand, you can hack your way through by guessing the final length of the filled-in string and running the measure/`realloc`/write procedure directly, e.g.:[30]

```
char *string = NULL;
char int_as_string[10000];
for (int i =0; i< 100; i++){
    int newlen = strlen(string) + 10000;
    string = realloc(string, newlen);
    snprintf(int_as_string, 10000, " %i", i);
    strncat(string, int_as_string, newlen);
}
```

- The C standard defines `sizeof(char)==1`, so it is safe to write `newlen` in the place of `sizeof(char)*newlen`.[31]

See Listing 3.7, page 112, for another example of extending a string inside a loop.

$\mathbb{Q}_{2.17}$ Modify `primes2.c` to write to a string named `primes_so_far` rather than printing to the screen. As the last step of the program, `printf(primes_so_far)`.

$\mathbb{Q}_{2.18}$ Modify the program in the last exercise to print only primes whose last digit is seven. (*Hint*: if you have written a number to `pstring`, you can compare the last element in `pstring`'s array of characters to the single character `'7'`.)

`strcmp` Because strings are arrays, the form `if("Joe"=="Jane")` does not make sense. Instead, the `strcmp` function goes through the two arrays of characters you input, and determines character-by-character whether they are equal.

You are encouraged to *not* use `strcmp`. If `s1` and `s2` are identical strings, then `strcmp(s1, s2)==0`; if they are different, then `strcmp(s1, s2)!=0`. There is

[30] It would be nice if we could use `snprintf(string, newlen, "%s %i", string, i)`, but using `snprintf` to insert `string` into `string` behaves erratically. This is one more reason to find a system with `asprintf`.

[31] ISO C standard, Committee Draft, ISO/IEC 9899:TC2, §6.5.3.4, par 3.

a rationale for this: the `strcmp` function effectively subtracts one string from the other, so when they are equal, then the difference is zero; when they are not equal, the difference is nonzero. But the great majority of humans read `if (strcmp(s1, s2))` to mean 'if `s1` and `s2` are the same, do the following.' To translate that English sentiment into C, you would need to use `if (!strcmp(s1, s2))`. Experienced programmers the world over regularly get this wrong.

$\mathbb{Q}_{2.19}$ Add a function to your library of convenience functions to compare two strings and return a nonzero value if the strings are identical, and zero if the strings are not identical. Feel free to use `strcmp` internally.

$\sum$

- Strings are actually arrays of `chars`, so you must think in pointer terms when dealing with them.
- There are a number of functions that facilitate copying, adding to, or printing to strings, but before you can use them, you need to know how long the string will be after the edit.

2.9 ※ ERRORS

The compiler will warn you of syntax errors, and you have seen how the debugger will help you find runtime errors, but the best approach to errors is to make sure they never happen. This section presents a few methods to make your code more robust and reliable. They dovetail with the notes above about writing one function at a time and making sure that function does its task well.

TESTING THE INPUTS

Here is a simple function to take the mean of an input array.[32]

```
double find_means(double *in, int length){
  double mean = in[0];
    for (int i=1; i < length, i++)
        mean += in[i];
    return mean/length;
}
```

What happens if the user calls the function with a `NULL` pointer? It crashes. What happens if `length==0`? It crashes. You would have an easy enough time pulling

[32] Normally, we'd just assign `double mean=0` at first and loop beginning with `i=0`; I used this slightly odd initialization for the sake of the example. How would the two approaches differ for a zero-length array?

out the debugger and drilling down to the point where you sent in bad values, but it would be easier if the program told you when there was an error.

Below is a version of `find_means` that will save you trips to the debugger. It introduces a new member of the `printf` family: `fprintf`, which prints to files and *streams*. Streams are discussed further in Appendix B; for now it suffices to note that writing to the `stderr` stream with `fprintf` is the appropriate means of displaying errors.

```
double find_means(double *in, int length){
    if (in==NULL){
        fprintf(stderr, "You sent a NULL pointer to find_means.\n");
        return NAN;
    }
    if (length<=0){
        fprintf(stderr, "You sent an invalid length to find_means.\n");
        return NAN;
    }
    double mean = in[0];
    for (int i=1; i < length, i++)
        mean += in[i];
    return mean/length;
}
```

This took more typing, and does not display the brevity that mathematicians admire, but it gains in clarity and usability. Listing conditions on the inputs provides a touch of additional documentation on what the function expects and thus what it will do. If you misuse the function, you will know the error in a heartbeat.

The `&&` and `||` are perfect for inserting quick tests, because the left-hand side can test for validity and the right-hand side will execute only if the validity test passes. For example, let us say that the user gives us a list of element indexes, and we will add them to a counter only if the chosen array elements are even. The quick way to do this is to simply use the % operator:

```
if (!(array[i] % 2))
    evens += array[i];
```

But if the array index is invalid, this will break. So, we can add tests before testing for evenness:

```
if (i > 0 && i < array_len && !(array[i]%2))
    evens += array[i];
```

If `i` is out of bounds, then the program just throws `i` out and moves on. When failing silently is OK, these types of tests are perfect; when the system should complain loudly when it encounters a failure, then move on to the next tool in our list: `assert`.

`assert` The `assert` macro makes a claim, and if the claim is false, the program halts at that point. This can be used for both mathematical assertions and for housekeeping like checking for `NULL` pointers. Here is the above input-checking function rewritten using `assert`:

```
#include <assert.h>

double find_means(double *in, int length){
    assert (in!=NULL);
    assert (length>0);
    double mean = in[0];
    for (int i=1; i < length, i++)
        mean += in[i];
    return mean/length;
}
```

If your assertion fails, then the program will halt, and a notice of the failure will print to the screen. On a gcc-based system, the error message would look something like

```
assert: your_program.c:4: find_means: Assertion 'length > 0' failed.
Aborted
```

Some people comment out the assertions when they feel the program is adequately debugged, but this typically saves no time, and defeats the purpose of having the assertions to begin with—are you *sure* you'll never find another bug? If you'd like to compare timing with and without assertions, the `-DNDEBUG` flag to the compiler (just add it to the command line) will compile the program with all the `assert` statements skipped over.

$\mathbb{Q}_{2.20}$ The method above for taking a mean runs risks of overflow errors: for an array of a million elements, `mean` will grow to a million times the average value before being divided down to its natural scale.
Rewrite the function so that it calculates an incremental mean as a function of the mean to date and the next element. Given the sequence $x_1, x_2, x_3, \ldots$, the first mean would be $\mu_1 = x_1$, the second would be $\mu_2 = \frac{\mu_1}{2} + \frac{x_2}{2}$, the third would be $\mu_3 = \frac{2\mu_2}{3} + \frac{x_3}{3}$, et cetera. Be sure to make the appropriate assertions about the inputs. For a solution, see the GSL `gsl_vector_mean` function, or the code in `apop_db_sqlite.c`.

TEST FUNCTIONS The best way to know whether a function is working correctly is to test it, via a separate function whose sole purpose is to test the main function.

A good test function tries to cover both the obvious and strange possibilities: what if the vector is only one element, or has unexpected values? Do the *corner cases*, such as when the input counter is zero or already at the maximum, cause the function to fail? It may also be worth checking that the absolutely wrong inputs, like `find_means(array4, -3)` will fail appropriately. Here is a function to run `find_means` through its paces:

```
void test_find_means(){
  double array1[] = {1,2,3,4};
  int length = 4;
    assert(find_means(array1, length) == 2.5);
  double array2[] = {INFINITY,2,3,4};
    assert(find_means(array2, length) == INFINITY);
  double array3[] = {-9,2,3,4};
    assert(find_means(array3, length) == 0);
  double array4[] = {2.26};
    assert(find_means(array4, 1) == 2.26);
}
```

Writing test functions for numerical computing can be significantly harder than writing them for general computing, but this is no excuse for skipping the testing stage. Say you had to write a function to invert a ten-by-ten matrix. It would take a tall heap of scrap paper to manually check the answer for the typical matrix. But you do know the inverse of the identity matrix (itself), and the inverse of the zero matrix (NaN). You know that $\mathbf{X} \cdot \mathbf{X}^{-1} = \mathbf{1}$ for any $\mathbf{X}$ where $\mathbf{X}^{-1}$ is defined. Errors may still slip through tests that only look at broad properties and special cases, but that may be the best you can do with an especially ornery computation, and such simple diagnostics can still find a surprising number of errors.

Some programmers actually write the test functions first. This is one more manner of writing an outline before filling in the details. Write a comment block explaining what the function will do, then write a test program that gives examples of what the comment block described in prose. Finally, write the actual function. When the function passes the tests, you are done.

Once you have a few test functions, you can run them all at once, via a supplementary test program. Right now, it would be a short program that just calls the `test_find_means` function, but as you write more functions and their tests, they can be added to the program as appropriate. Then, when you add another test, you will re-run all your old tests at the same time. Peace of mind will ensue. For ultimate peace of mind, you can call your test functions at the beginning of your main

analysis. They should take only a microsecond to run, and if one ever fails, it will be much easier to debug than if the function failed over the course of the main routine.

$\mathbb{Q}_{2.21}$

Write a test function for the incremental mean program you'd written above. Did your function pass on the first try?

Some programmers (Donald Knuth is the most famous example) keep a bug log listing errors they have committed. If your function didn't pass its test the first time, you now have your first entry for your bug log.

$\sum$

- Before you have even written a function, you will have expectations about how it will behave; express those in a set of tests that the function will have to pass.
- You also have expectations about your function's behavior at run time, so `assert` your expectations to ensure that they are met.

$\mathbb{Q}_{2.22}$

This chapter stuck to the standard library, which is installed by default with any C compiler. The remainder of the book will rely on a number of libraries that are commonly available but are not part of the POSIX standard, and must therefore be installed separately, including Apophenia, the GNU Scientific Library, and SQLite. If you are writing simulations, you will need the GLib library for the data structures presented in Chapter 6.

Now that you have compiled a number of programs and C source is not so foreign, this is a good time to install these auxiliary libraries. Most will be available via your package manager, and some may have to be installed from C source code. See the online appendix (linked from the book's web site, `http://press.princeton.edu/titles/8706.html`) for notes on finding and installing these packages, and Appendix A for notes on preparing your environment.