In which our hero begins to learn the secret to much of computing: the ability to create higher level abstractions and to hide complexity via the amazing and simple concept of the function.

13.1 The Humble Programmer

Edsger Dijkstra was a famous computer scientist, mostly for his work on algorithms (e.g., shortest paths [D59]) and concurrency (e.g., semaphores [D65]), and for these works he was recognized with the highest honor in the field: the Turing Award.

Part of the tradition of the winner of this award is to give a speech; Dijkstra’s was entitled the “Humble Programmer” [D72]. The money quote: “We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as the Very Humble Programmer.”

What Dijkstra is getting at here, particularly with the part about the “intrinsic limitations of the human mind” is essential to programming. There is simply no way for humans to contain millions and millions of lines of code in their heads. And yet, we are able to build systems that contain such vast complexity. How?

The answer is that we can manage complexity via abstraction. At every level of a computer system, we create higher-level, simpler ways to reason about what the computer is doing. In fact, abstraction is fundamental to both hardware and software design.

At the lowest level is hardware, made of transistors; these are abstracted into logic gates, which perform simple boolean functions like and, or, and not; these are abstracted further into large functional units such as an adder or a multiplier; eventually, these are presented to the user of a computer as the abstraction of a processor that processes instructions and is connected to a memory, as we have assumed throughout this book.
In software, the same type of hierarchical abstraction is common. For example, in numerical computation, at the base level, you have the language’s ability to provide basic math; on top of this, there is likely a math library which provides additional features, like exponentiation and trigonometric functions; at a higher level of abstraction, you might build a new data type yourself (such as a vector), and provide operations over those vectors (such as addition, multiplication, etc.).

In C programs, the way we build abstractions, and thus the way we enable larger and more complex programs to be built, is via one of the most fundamental programming constructs: the function. Thus, understanding how functions are defined and used in a program is essential.

In this chapter, we introduce the C function, including how to define a function (so you can write your own) and how to call a function (so you can use them!). We also discuss function-call precedence, some common mistakes, and then one critical mystery that we must handle to make functions useful: how to change the values of variables inside the caller.

13.2 Function Definition

Let’s say you want to write a simple function to add two numbers together. Now, it’s true, this is not a particularly useful function; after all, you can just use the + operator and perform the addition directly. So yes, this the much maligned “toy” example you’ve been taught to fear. And here it is!

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

See, that wasn’t so bad. Let’s carefully examine all the parts of this function definition. First is something called the return type of the function, in this case an int. Each function can return a single result to the caller, and thus you must specify what type of thing to expect back. In this case, because the function just adds two numbers together and returns the result, it is not surprising that the result is an integer.

The next piece is the name of the function (add in this example). Legal names follow the same rules previously outlined for variables: lowercase and uppercase letters, digits, and underscore are all legal characters; the name cannot begin with a digit.

Next you always must include parentheses, which surround any arguments (or parameters) that the function takes in. These are the
inputs to the function, separated by a comma, with each input specifying its **type** as well as its **name**.

Finally, we have the **body** of the function, surrounded by curly braces. The body of the function is the list of statements that comprise the work it does. This may include further variable definitions (e.g., `int value;`), computations (e.g., `value=x+y;`), and any of the other constructs you’ve seen thus far (e.g., `if` statements, `while` loops).

However, there is one new keyword you’ll need as well: **return**. The return statement allows you to specify what value to pass back to the **caller** of the function, i.e., the piece of code that invoked this function to get some work done. In this case, the caller gets back the value of the variable `value`, or simply the result of the addition.

Of course, you could rewrite `add()` to be more concise. Here is a shorter version. Note that the **return** keyword can pass back any expression (not just a variable name), as long as it results in an integer value.

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

### 13.3 The Most Special Function: **main()**

Before going further, it’s worth introducing one very special program that is defined in all C programs: the **main** function. This function is special because it defines where the program begins to run; it is the first function in your program that is called to get things going.

We’ll start here with a simple version of `main()` that does not take any arguments; later, we’ll see how to add arguments so that users of your program can pass in values when running the program.

```c
int main() {
    ... // include some statements here
    return 0;
}
```

As you can see, the **main()** function looks like any other function. It returns an integer, which you can access from your shell once the program has finished execution. And, importantly, it tells the C Abstract Machine where to start execution: the first statement within **main()**.

### 13.4 The Function Call (And Return)

From this point forward, many of our code snippets will include a **main()** function (where the computation begins), as well as some
functions that \texttt{main()} calls to get its work done. In this example, we define two functions, \texttt{add()} and \texttt{main()}, and show how \texttt{main()} can call the \texttt{add()} function.

\begin{verbatim}
int add(int x, int y) {
    return x + y;
}

int main() {
    int r;
    r = add(10, 20);
    return r;
}
\end{verbatim}

Inside \texttt{main()}, we first define an integer, \texttt{r}, thus making space for it in memory (in fact, we’ll soon learn more about exactly where in memory such an allocation takes place). In the next statement, we do two things: first, we call the function \texttt{add()}, by writing the text \texttt{add(10, 20)} in the program.

When calling such a function, we need to make sure of two things. The first is that each argument is of the right type; in this case, the call expects two integers, and thus we can pass in either integer literals (i.e., numbers) directly or integer variables – it makes no difference to the function. For example, this usage, which passes \texttt{a} and \texttt{b} to \texttt{add()}, works just like you would think, returning the value 3 and assigning \texttt{r} to it.

\begin{verbatim}
int main() {
    int a = 1;
    int b = 2;
    int r = add(a, b);
    return r;
}
\end{verbatim}

The second important thing is that we use the function in a place in the code where it would make sense to use the return type of the function. For example, \texttt{add()} returns an integer; thus, we can treat it as if it were an integer in the program and everything will work like you might expect. For example, you can include \texttt{add()} in a more complex mathematical expression:

\begin{verbatim}
int sum = add(1, 10) * add(2, 20);
\end{verbatim}

You can even \texttt{nest} these calls inside each other, as follows, arbitrarily:

\begin{verbatim}
int sum = add(add(1, 1), add(2, 2));
\end{verbatim}
Of course, doing so brings up the question of operator precedence; exactly what order do these function calls take place? In C, it turns out, some aspects of this are well defined, and some not so much. Perhaps something to discuss?

### 13.5 Function Call Precedence

The first thing to know here: function call has relatively high precedence. Thus, if you write a math expression with a function call embedded within it:

```c
int x = foo() + 10;
```

The call to `foo()` will take place first, and the result will be added to 10. This precedence is really the only one that makes sense; what would it even mean to perform the addition before the call has returned the value? Figure 13.1 shows that function call is at the highest level, along with postfix increment and decrement.

However, C leaves one very interesting aspect of function call undefined: the evaluation order of the parameters. For example, if you call a function with two arguments, each of which is some kind of expression (e.g., `func(expression1, expression2)`), your code cannot properly rely on `expression1` being evaluated first. The only thing your code can rely on is that all the expressions will be evaluated before the actual call (jump to the function code) takes place; yes, the function call is a **sequence point** that we mentioned before [S08].

Does this matter in practice? Not usually. However, if you are calling a function and trying to get tricky with incrementing a variable as one parameter and then relying upon the incremented value in another, your code might not work in the way you expect.
13.6 But, A Very Big Problem

Although we’ve hinted at it, we have not yet flat out said it, so here we go: C passes parameters by value. When you pass a parameter by value, you make a copy of the value somewhere in memory, and then jump to the function; the function then uses those copies for its duration. When the function call is finished, it returns to where it was called from and the parameter copies are deallocated. But then we have a real problem: what if we want the function that we have called to actually change the value of a parameter such that the change is visible in the caller?

To make this clearer, let’s look at a simple example. In this code, we’ve defined a function, increment, that takes one argument (an integer, x) and increments it. The main() function below allocates an integer f, sets it to 10, and then calls increment(). What is the value of f after the call?

```c
void increment(int x) {
    x = x + 1;
}

int main() {
    int f = 10;
    // value of f before the call: 10
    increment(f);
    // value of f after the call?
    return 0;
}
```

A side note before continuing: we’ve added a new return type here, void. When a function does some work but then has nothing to report about it, a void return type is appropriate, so we use one here.

Back to the question at hand: does calling increment() change the value of f in the caller? As you now might have guessed, the value is still 10, because of those pass-by-value semantics of C; the value in the caller remains unchanged. Given what we now know ...

chapter interrupt

Notice from the book patrol: This chapter has been interrupted by an inability to properly change the values of a variable within a function. The result is a distinct lack of functionality that is required by all programming languages. We thus must halt the progression of this chapter until this dilemma is addressed. We address it by focusing on three critical elements that must come together to help solve the problem: pointers, the stack, and scope.
References

[D59] “A note on two problems in connexion with graphs”
Edsger Dijkstra
Numerische Mathematik, 1, 1959
Who knew you could write about computer algorithms in 1959?

[D65] “Solution of a problem in concurrent programming control”
Edsger Dijkstra
Communications of the ACM, 8(9), September 1965
Pointed to as the first paper of Dijkstra’s where he outlines some of the most fundamental problems in concurrent programming.

[D72] “Turing Award Lecture: The Humble Programmer”
Edsger Dijkstra
www.cs.utexas.edu/users/EWD/EWD340.PDF
Dijkstra collected all of his writings (done by hand!) which you can read through at your leisure. Some interesting tidbits in many of them on the foundations of computing.

[S08] “Parameter evaluation order before a function calling in C”
Answered by Grant Wagner and Robert Gamble
http://stackoverflow.com/questions/376278/parameter-evaluation-order-before-a-function-calling-in-c
Stackoverflow is a great cite for getting answers that are both high quality and very specific. An amazing everyday kind of website, especially when learning a new language.