In which our hero discovers a much better way to refer to things in memory, and thinks about the importance of names. Sadly, our hero still has no name, at least, of which we know. Finally, our hero recalls Shakespeare and realizes that this current book is not very literary in comparison.

7.1 Is A Rose A Rose?

In the play Romeo and Juliet, Shakespeare famously wrote “What’s in a name? That which we call a rose by any other name would smell as sweet...”. Juliet, who spoke these lines, was making the point that Romeo’s last name (Montague) shouldn’t matter, and that the two of them should be able to be together despite being from warring families (Juliet being a Capulet). And while we ship\(^1\) Romeo and Juliet as much as any other living soul, Juliet was apparently wrong on this point, because (SPOILER ALERT) she ends up dead as a doorknob, as does her dear Romeo. And thus Shakespeare was (indirectly) telling us something else: names do matter [W17]. What was true then in Shakespeare is also true now in computer programs, as it turns out.

So far, we’ve introduced only one kind of naming in our simple C machine: each location (byte) in memory has a name. These “names” are just numerical addresses, from zero to the size of the memory. Pretty boring, no? It’s kind of like each person in the world being referred to by a social security number. Imagine the conversations:

“Hello, I’m 123-45-6789. What’s your name, gorgeous?”

“Yeah, right, like that’s really your name. And I’m 999-99-9999!”

We probably want something better than this – an easy, simple way for a programmer to keep track of the information relevant to a

\(^1\)From the urban dictionary: ship – to endorse a romantic relationship. We are told by our daughters that we probably should not use terms like this, as we are “too old.”
program. And thus we have one of the most basic building blocks of any modern computer program: the ubiquitous variable.

### 7.2 Definitions

Because we thus far only have integers in our restricted C Machine, we probably would like some way to give a name to a certain piece of memory, and make sure there is space created for it. We do this by writing the following text in our program:

```c
int x;
```

And there it is! We call this a definition of the variable $x$; a definition of a variable tells you everything you need to know about a variable, such as its type (an `int`) and its name ($x$), and ensures that language will create space for it (the usual 4 bytes for this `int`) somewhere in memory. And then the beautiful thing: we can now refer to those 4 bytes of memory by using the name $x$ in our code, instead of memory address `0xbf77c008` (or whatever).

The syntax\(^2\) of this definition is simple: the type of the variable (in this case, an `int`), followed by some whitespace (spaces or tabs), followed by its name, followed by more optional whitespace (usually none), followed by a semi-colon. We’ll see that many C statements are terminated by a semi-colon (which allows us to follow it with another C statement, if we’d like to).

The picture in Figure 7.1 shows the chunk of memory where $x$ resides. In the depiction, we show a few 4-byte sized chunks of memory, and we assume that $x$ is indeed referring the 4 bytes starting at address `0xbf77c008`. Note that these are rather high addresses, in what we are presuming is a 32-bit address space (i.e., a memory that can refer to $2^{32}$ different bytes); you can tell the address space is 32 bits long by the way we write it, with eight hex digits (each of which represents 4 bits).

---

\(^2\)The syntax of a language is the set of rules that define what constitutes a legal program; we’ll talk more about C syntax throughout the book, albeit somewhat informally. See other sources for more formal definitions [C99, C11].
Sometimes, we need to define more than one variable at once. In such a case, we could simply write each definition out, one line at a time:

```c
int x;
int y;
int z;
```

However, that can be a lot of redundant typing, and, more importantly, can actually make the code less readable by using up a lot of vertical space. Thus, as a shorthand, you can use a comma as a separator to define more than one variable on a line, as follows:

```c
int x, y, z;
```

### 7.3 Declarations vs. Definitions

In the C code you’ve seen thus far, you can define a variable (an integer), which makes it available for use in your program and, in addition, allocates space for it in memory.

There is another way to make a variable accessible within a piece of code but without making space for it in memory; this presumes, of course, that the some other part of code has already defined the variable and thus made space for it. We call the process of making a variable accessible (thus knowing its name and its type) without making space for it declaration. In fact, defining a variable can be viewed as making space for it in memory and also declaring it (and thus making it usable in your program).

We won’t need this functionality for some time, so we won’t (yet) show how to declare a variable (without defining it). However, if you want to read ahead, there are good sources of information on this topic elsewhere [A17]. Otherwise, be patient; we will get to it!

### 7.4 Legal Names

Programming languages all have rules for what constitute legal names, and C is no exception in this regard. The C rules are pretty simple, as variable names can consist of:

- Upper-case letters such as A, B or even Z
- Lower-case letters such as a, b or even z
- The underscore character _
- Numeric digits 0, 1, . . . , 9  
  (exception: variable names cannot begin with a digit)

Thus, names such as len, num_elements, NumElements, and _secrets are all legal variable names; because C is case sensitive,
LARI
A
B
 is a different name than \texttt{foo}. In contrast, names such as the
\texttt{rarely seen 2.cant.start.with.digits,minusus-not-valid, and NamesWithOtherSymbols*Inside+Them} are not valid.

There are a few other restrictions we need to consider in naming. Specifically, most languages have reserved words that cannot be used as variable names at all. In C, they are: \texttt{auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while}.

### 7.5 Choosing Good Names

There is a famous short story by Ursula K. Le Guin called “The Rule of Names” [L64]. In it, one character says “To speak the name is to control the thing,” thus recognizing the power of knowing the true name of someone or something. There must be some sort of similar corollary in computer systems; minimally, it is clear that good naming is critical to making programs readable and easy to understand. While the computer may not care what you call a variable, a person reading the program definitely will.

Unfortunately, there are no simple rules for creating good names – you have to use your own experience, intelligence, and creativity to do this well. It is certainly clear that poor names greatly decrease code readability. For example, let’s look at the following expression:

\[
\frac{y}{z}
\]

When examining this bit of code, we naturally have no idea what it is doing, as \texttt{y} and \texttt{z} are not descriptive names. On the other hand, let’s look at this same snippet:

\[
\frac{\text{sum}}{\text{count}}
\]

Here you can see that \texttt{sum} is being divided by the \texttt{count}; perhaps the mean is being computed? By adding sensible names to programs, we make it so that humans can understand the code better.

For example, \texttt{TheNumberOfElementsInTheArray} is incredibly descriptive, but almost definitely too long and verbose; it is a pain to type, and will fill up the page of text with needless characters. On the other hand, \texttt{n} may not be very helpful, as it could stand for anything. Perhaps a compromise is in order: something like \texttt{num_elements} or the slightly more concise \texttt{nelements}. See Kernighan and Pike’s book on programming practices for more excellent tips on this topic and many others [KP99].
Tip: Choose Names to Improve Readability

Making code readable – that is, easy to understand by another human, including yourself in the future(!) – should be your foremost goal when programming. Of course you can have other goals, too; for example, writing efficient code is often important. But without readability, you soon end up with an unmanageable, unmaintainable mess. Names are a large part of this, so think carefully when choosing them; with good naming, code can almost become a form of literature [K84].

7.6 Making Mistakes

While we haven’t gotten to the part where you write your own full programs (be patient in this regard, too), we will soon, and thus it’s good to see what happens when you make a mistake. For example, let’s say you tried to define an integer as follows:

```c
int 1a;
```

As we know from the rules above, a variable name cannot start with a digit. When you try to create a program with such a name in it, the programming language will not let you; in fact, it will tell you something like the following:

```c
error: expected identifier or '('
  int 1a;
^ 1 error generated.
```

As you can see, sometimes these types of messages aren’t the easiest to read. But, they give you good information: something is wrong with the “1” in that name. We also see that C expects you to provide an identifier instead of the numeric value “1”; this is just a fancy way of saying it expects a valid name.

Here’s another version of that same error, on a different system:

```c
error: invalid suffix "a" on integer constant
error: expected identifier or ( before numeric constant
```

As you become better at C, you’ll also have to get better at reading these error messages.

Let’s make some more mistakes and see what happens:

```c
int x;
int x;
```

---

4 This output comes from the clang compiler, commonly installed on Apple machines; we’ll learn more about compilers later.

5 This output comes from the gcc compiler on a Linux-based system.
Here, we mistakenly redefine $x$. On one system (Mac OS X), we see this error:

```
error: redefinition of 'x'
  int x;
```

Finally, we try to use a keyword as a variable name. The C language doesn’t like it when you do that:

```
int if;
```

This is what you will see on a Linux-based system:

```
error: expected identifier or '('
  int if;
```

As you move forward in this book, we encourage you to try to make as many mistakes as you can, and see if you can figure out what is causing them. Learning to read error messages is an essential part of becoming a top-notch programmer, so go forth, be unafraid of making mistakes, and see if you can master the art of analyzing error messages and then fixing your code.

### 7.7 Summary

We have introduced one of the most fundamental aspects of computer programs: the variable. The variable is just a named container which serves as a convenience to programmers; instead of having to remember the address of each piece of information we are interested in, we can instead use a human-friendly name. Exactly which names you use makes no difference at all to the computer, but they make a huge difference to humans who have to read (and understand) the code. Thus, choose your names wisely, and make your code as clear, simple, and understandable as you can.
References

[A17] “Declare vs. Define in C and C++”
Alex Allain
http://www.cprogramming.com/declare_vs_define.html
A short useful description of the difference between declaration and definition.

[C17] “The C Programming Language”
https://en.wikipedia.org/wiki/C_(programming_language)
Wikipedia to the rescue, again.

[K84] “Literate Programming”
Donald E. Knuth
Knuth was an early advocate of viewing code as something you read, like literature. Perhaps this view is a bit extreme, but good to consider nonetheless; if you can’t read it, you can’t understand it, and then who will know if the code is even doing what it is supposed to do?

[L64] “The Rule of Names”
Ursula Le Guin
A short story that formed the core of the later Fantasy masterpiece, “The Wizard of Earthsea.”

[KP99] “The Practice of Programming”
Brian W. Kernighan, Rob Pike
Addison-Wesley, 1999.
Kernighan was a co-inventor of the C programming language; Pike a co-inventor of the Go programming language. Needless to say, they probably know a thing or two or one thousand about programming.

[W17] “A rose by any other name would smell as sweet”
https://en.wikipedia.org/wiki/A_rose_by_any_other_name_would_smell_as_sweet
Another one of these amazing wikipedia pages about something so specific; interestingly, as described on said page, the exact lines may have actually been cobbled together by editor Edmond Malone, as the differing surviving texts each don’t say it quite as well. And yet, no one knows his name...