Lecture 2

CSE 110

1 July 1992

1 A Warning

I think that in order to program effectively, you have to understand how a computer really works, and you also have to understand what the compiler is really doing. Mainstream computer science pedagogues disagree with me about this. Nevertheless, because I am an employee of the university and not a graduate student, I am accountable to practically nobody and so we can learn this the right way instead of the usual way.

However, that means that before we dive into the syntax of the language, we have to spend some time discussing some of the details of how a computer works, and we don't get to dive into the syntax of the language itself for a while. I think that is all right—C is a language for talking about what goes on in computers, and if you don't know what goes on in computers you won't have anything intelligent to say in C.

So if these first lectures are not what you expected, please be a little patient.

2 The Blackboard

2.1 Bytes

Computer memory is like a blackboard divided into squares. The squares are called *bytes*. Each byte can hold one piece of information. Each square has a name, which is called its *address*. Like street addresses, the names are numbers.

Bytes vary in size from computer to computer, but on most modern computers each byte contains eight *bits*. The *bit* is the smallest possible amount of information. It represents the amount of information that one can store in a physical object which can be in one of only two states. For example, one bit of information suffices to describe the state of a light switch: the switch is either on or off.[1] Since there are 8 bits in a byte, each byte is in one of 2^8 states. 2^8 is 256, so we usually imagine that the information in a byte is a number between 0 and 255.

So each byte has exactly two properties: It has a value, which we can think of as a number between 0 and 255, and it has an address which tells the computer where it is on the microchip. The address is the only thing that distinguishes two bytes, so two bytes never have the same address. Of course two different bytes can have the same value.

2.2 Bytes are Enough to Represent All Kinds of Data

Bytes only hold numbers between 0 and 255, which is too small a range to be useful. So when we want to talk about a bigger number, we use an old trick. In our number system, we have only ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When we want to write a number bigger than 9, we write two or more symbols together and say that the symbols mean different things depending on where they are relative to one another. For example, the number 674 means 6 hundreds, 7 tens, and 4 units. Because the 6 contributes most to the value of the numeral 674, we say it is the *most significant digit*. Similarly, the 4 is the *least significant digit*. This is called a *place value* system because the value of each symbol depends on the place it is on the paper. We say it is a *base 10* or *decimal*[2] system because each place is worth 10 times as much as the one to its right.

We do the same thing with the bytes on the blackboard, only we use a base 256 system. If we want to represent a number bigger than 255, we use two or more bytes and let some of them be more significant than others: each byte is worth 256 times as much as the next. For example, to represent the number 674, we put 2 in the most significant byte and 162 in the least significant byte. The 162 in the least significant byte is worth 162. The 2 in the most significant byte is worth 256 times as much as a 2 in the least significant byte would be, so it is worth $256 \cdot 2$ or 512. Thus the total value of this two-byte object is $162 + 512 = 674$.

Clearly if we're going to have multi-byte objects then it's no longer sufficient to only keep track of where a certain number is in the computer's memory: we

[1]Objects that can be in only one state clearly can't be used to store any information. It turns out that the amount of information you can store in an object with n possible states is no more than $log_2 n$ bits. These notions were first clarified by Claude Shannon of Bell Laboratories in the middle 1950's.

[2]*Decimal* is from *Decem*, the Latin word for 'ten'

also need to keep track of how many bytes are being used to store that number. If someone writes a two-byte number into memory and then tells us where it is so that we can look at it or change it, they had better tell us how long it is, or else we might not look at all of it (which would be like someone writing the number 674 on the blackboard and us thinking that it was three numbers, a 6, a 7, and a 4), or else we might look at too much (which would be like someone writing the numbers 674 and 523 on the blackboard very close together and us thinking that they were the single number 674523). Either would be a big disaster.

Mostly the compiler keeps track of the lengths of things and handles them for you. That's one of the big reasons for having a computer language and a compiler in the first place.

Also we want to store other information than just numbers. Say we want to store someone's name—how do we do that with just numbers between 0 and 255? Of course the answer is that we assign a number to each letter of the alphabet, and we store the numbers that correspond to each letter in the name. It would be a big disaster if we forgot that this sequence of numbers was actually supposed to represent a sequence of letters. Another reason we have computer languages is to keep track of which numbers on the blackboard really represent letters and which ones really represent numbers.

2.3 Variables and Types

So suppose we are going to write a program that will need to remember a number–say it's a program that thinks of a secret number between 1 and 10000, and then the user tries to guess what that number is. We need to find a blank spot on the blackboard big enough to store the number. We need at least two bytes to store the number in, because it might be bigger than 255. (Two bytes are enough to store numbers between 0 and 65,535.) We also want to remember that this is a two-byte number and not the first half of a four-byte number, and not the first two letters in someone's name, or anything like that.

Here's what we say in C to accomplish all that:

```
int my_number;
```

`int` is short for 'integer', which is a mathematical word for a whole number. The actual size of an <`int`> depends on what computer and what compiler you are using, but it is never less than two bytes. When the compiler sees this instruction, it finds enough free space on the blackboard to hold an <`int`>, reserves it, and makes a note to itself that from now on, this space is going to hold an <`int`> and that the name `my_number` refers to the space.[3] You don't need to know the details about how it finds this space or where it is; all you need to know is that the space is there and that you can call it `my_number`. The semicolon (`;`) is just punctuation; it tells the compiler where the end of the statement is.

This whole process is called *declaring* a *variable*. The *name* of the variable is `my_number` and the *type* of the variable is <`int`>. A variable has two other properties: Its value (which depends only on the values of the bytes that make it up and on its type), and its address, which is the same as the address of the first byte that makes it up. Unless you say otherwise, <`int`> variables always start with the value 0.

In some languages you can use variables without declaring them; what happens then is that the compiler assumes that they have a certain type. You can't do that in C. If you name a variable you haven't declared, the compiler will signal an error.

2.4 Other Types

C has a few other primitive types:

- <`short int`> and <`long int`> are like <`int`>, but might be shorter or longer. <`short int`> is always at least two bytes long, and it's never longer than `int`. <`long int`> is always at least four bytes, and it's never shorter than `int`. How long <`short int`> and <`long int`> actually are depends on your compiler. <`int`> is supposed to be a size that is convenient for your computer and you should use <`int`> rather than <`long int`> or <`short int`> unless there's a good reason. You use <`short int`> to save blackboard space and <`long int`> when you need to handle bigger numbers.
- <`char`> is short for *character*; you use it when you want to store a character, which could be a letter, a digit, or any other symbol on the keyboard. It's always exactly one byte long.
- <`float`> is short for *floating-point number*. A floating point number is one which is represented internally in scientific notation, with a mantissa and an exponent.[4] This means that a <`float`> can represent a fractional number like $\frac{3}{4}$ or π. A <`float`> variable has a wide range, and can

[3] Actually the compiler doesn't find the space right away; rather, it writes out machine instructions into the executable version of your program that find the space when they are executed.

[4] This means, for example, that a number like 1234567890 is stored as 1.234567890×10^9.

typically hold a number between about 10^{-38} and about 10^{38}, but it loses precision towards the ends of its range. `float`s on typical machines are four bytes long. There's a more precise version of a <`float`> called a <`double`>, which is twice as long and twice as accurate. Similarly, there are `<long double>`s which are even longer than ordinary <`double`>s.

That's all the simple types there are. There are other types, but nearly everything else is built up from these few.

1.234567890 is the *mantissa* and 9 is the *exponent*. If you don't understand this, don't worry; we won't need it for a long time.