IEOR 4729

Addresses, arrays, pointers, et al

In this brief write-up we review basic memory concepts as they occur in the C language – you should look at the Kernighan-Ritchie book.

We may think of "memory" as a one-dimensional collection of bits (each bit has a zero or a 1). The memory is "aligned," or arranged, into bytes, that is to say, groups of consecutive 8 bits. Suppose that in a C program we have a variable, whose name is k, which is an integer, and that furthermore at some point of the program this variable is given value 5:

```
int k;

…

k = 5;
```

Now, this variable is stored in some location in memory. This location is the "memory address", or simply, the address, of variable k. If we think of memory again as a one-dimensional object, the address of k will be that particular byte of memory where the representation of k starts. This representation occupies 32 consecutive bits (4 bytes), but think of the address of k as being that particular byte where the representation begins.

So, for example, if the address of k were 0x0012ff6c, then if we went to location 0x0012ff6c and looked at it we would see the value 5. More precisely, if we looked at the 4 bytes starting at this address, we would see the value 5 (in its binary representation). Incidentally, the value 0x0012ff6c is simply an integer, in its base-16 (hexadecimal) representation.

Suppose that, for some reason, we wanted to keep track of the address of k. In fact, suppose that we wanted to use a variable to keep track of the address of k. Then we would use the following syntax:

```
int k, *pk;

…
pk = &k;
```

Here, variable pk is of type "address of an integer" or "pointer to an integer". This is what the int * statement does. The statement pk = &k; is simply saying:

> The value of variable pk is the address of k

Now, of course pk is itself a variable. So it is also stored in memory, in some place (namely, the address of pk, or &pk).

Think of a library. A given book is stored somewhere, say it's the third book on the second shelf in the fourth bookcase. This information (the third book etc.) is stored in the index card for the

book. The index card is itself stored somewhere. Think of the book as the variable k; of the position of the book as the address of k; and of the index card as the variable pk.

Now, suppose we have a piece of code of the form:

```
double x, *y;

…
x = *y;
```

What does this do? First, it states that y is the address of a double. The statement: x = *y does the following: you look at the double stored in address y, and assign that double to x. More pedantically, the statement copies the double stored at address y, to the address where x is stored. In other words, the * operator, applied to a memory address, simply produces the value stored at that address. So think of taking an index card and reading what the index card says.

Arrays

An array is a set of contiguous memory all "formatted" to hold a certain data type. For example, an array of 100 doubles is simply a set of 800 bytes (= 8x100 bytes; eight bytes are needed to hold a double). In C, the array name is a variable, whose value is precisely the memory address where the first of those 100 doubles is stored (that is to say, the zeroth element of the array). Suppose we have:

```
double *x;

x = (double *)calloc(100, sizeof(double));
…
x[0] = 10.2;
x[2] = 5.8;
```

This says that we have a variable x, whose value (after the calloc) will be the memory address of the zeroth entry of the array. We want to stress that it is the **value** of the variable x that indicates where the array begins. The variable x is, itself, stored *somewhere* (everything is stored somewhere). Where is x stored? At the memory location &x. So, for example:

- The value of x is 0x00300050. If we look at memory address 0x00300050, what we will see is the value 10.2 (i.e., x[0]).
- The address of x is 0x0012ff34 – this is where x is stored. If we look at memory address 0x0012ff34, what we will see stored there is the value 0x00300050
- *x is the value stored at the memory address whose value is x. That is to say *x is the same as x[0].

Now here is a bit of C trickery. Consider the value x + 1. What is this value? You *might* think that this is the following:

1. take the value of x, namely the integer whose hexadecimal representation is 0x00300050 (in other words, $3 \times 16^5 + 5 \times 16 = 3145808$).
2. To this integer, add the value 1. You would get 0x00300051, right?

Well, this is wrong. Instead, what you get is 0x00300058. What happened is that x was declared as a double *, i.e. the address of a double. Incrementing x by 1, in this case, increments x by the

number of bytes in a double, i.e. 8. This goes to show that the "+" operator does not have the usual arithmetic sense when it is applied to a memory address.

So, what is *(x + 2)? Well, it is the value stored at the memory address obtained by starting at x, and moving 2 doubles (16 bytes) over. In other words, this is x[2] = 5.8.

Functions and addresses

Consider the following code fragment.

```
int i;

i = 35;
increase(i);
printf("i = %d\n", i);
```

where the function increase has the following definition:

```
void increase(int k)
{
  k = k + 1;
}
```

What will happen? After the call to increase, the printf will tell us that i = 35, i.e. the call had no effect. What happened?

In C, function arguments are copied. More precisely, when we enter function increase, the argument is copied into a new memory address. It is that copy that is increased. The original is unchanged. So how would we change the function to make it work? Try the following:

```
void increase(int *pk)
{
  *pk = *pk + 1;
}
```

and the call would be:

```
increase( &i );
```

How does this work? Well, when we call increase, we are passing to it the *address* of i. The function will create a copy of the variable that is passed. So, for example:

- The address of i is 0x0013f012. Before the function call, this address contains the value 35.
- The function call is passed the argument 0x0013f012. A copy of this argument is made, and is stored elsewhere. In the function call, this copy has the name pk. So the value of pk is 0x0013f012, and pk is stored somewhere in memory, say at address 0x0023fa05. Thus, if we were to look at address 0x0023fa05, we would see in there precisely the value 0x0013f012.
- Thus, *pk is: the value that we find if we look at memory location 0x0013f012, namely, 35.

- So the statement *pk = *pk + 1; does the following: add 1 to the value stored at memory location 0x0013f012, and then store that value in memory location 0x0013f012.
- Thus, upon return from the function increase, the value of i will indeed by 36, as desired.