17. Introduction to pointers

Herman Kamper

This lecture introduces pointers. They are the thing that almost only C programmers know about, and they are also the point where many people who learnt to program in another language first get lost. If you have written Delphi, Python or Java before, most of C so far will have felt familiar with slightly different punctuation. Pointers are probably the first new idea.

Two things about that before we start. The first is that nothing in this lecture is complicated. Every step is simple, and some of it will feel almost too simple. The second is that everything that follows in the course builds on it, so it is worth reading slowly and making sure each step clicks before moving on. If you understand boxes in memory, you will understand pointers.

Two things that pointers explain, and more

You have already seen two things in C that are slightly odd, and pointers are what explain them.

The first is `scanf`. When you read an integer from the user, you write

```
scanf("%d", &a);
```

with a little `&` in front of the `a`. Why is it there, and what does `scanf` do with it?

The second is something that happens when you pass an array to a function. Look at this program and work out what it prints on each of the two `printf` lines:

```
int main()
{
    int b[] = {1, 2, 3, 4};

    printf("%d\n", b[3]);
    foo(b);
    printf("%d\n", b[3]);

    return 0;
}

void foo(int array[])
{
    array[3] = 20;
}
```

The first line prints 4. The function `foo` then sets element 3 of its array to 20, and when we get back to `main` the second line prints 20. The function reached inside `b` and changed it. That does not happen with other variables: if `foo` took an `int x` and changed `x`, the variable back in `main` would be untouched.

Pointers explain both of these cases. This lecture gives you the pieces, and in the following lectures we will actually anwer the two questions properly using these pieces.

More generally, pointers are what let a C program work with a computer's memory directly, which is a large part of why C is used for systems programming at all. Once pointers make sense to you, so does a great deal of how a computer actually works.

Three kinds of variables

So far we have used two kinds of variables. The first is a scalar:

```
int a;
```

This creates a single box in memory. The box has a type (here an integer, which tells C how large the box is and how to interpret the zeros and ones inside it), it has a name `a`, and it lives at a specific address in memory.

The second is an array:

```
int b[5];
```

This creates five boxes, all of type integer. They share the name `b`: the first is `b[0]`, the second `b[1]`, and so on. Each of these invidiaul boxes also lives at a specific address.

The third kind of variable is new. A pointer is also just a box in memory. What makes it different is what it stores: a pointer stores an address, the memory location of another variable. That sounds strange, so let us make it concrete.

Your first pointer

Here are two ordinary variables:

```
int x, y;

x = 12;
y = 5;
```

The first line creates two boxes somewhere in the computer's memory, one called `x` and one called `y`. Straight after that line they contain rubbish, whatever happened to be in that part of memory before. The next two lines put 12 into the `x` box and 5 into the `y` box.

Each box now has four properties: a type (both are integers), a name (`x` and `y`), a value (12 and 5), and an address. The address is literally where on the memory chip the zeros and ones for that variable are stored. Real addresses are enormous numbers, so let us just make some up: say `x` lives at address 3000 and `y` somewhere else, at 8096.

Now we declare a pointer. I will call it `fooPtr`:

```
int *fooPtr;
```

The asterisk is what makes this a pointer. The line creates one more box in memory, called `fooPtr`, and the `int *` says what type of box it is: a pointer to an integer. It will hold the address of an `int`. Like any freshly declared variable, it contains rubbish to begin with.

Then we put something in it:

```
fooPtr = &x;
```

An assignment with = puts a new value into a box, exactly as before. The value this time is `&x`, and the `&` means "the address of". So `&x` is not 12, it is the address of the box that holds 12, which is 3000. After this line, the `fooPtr` box contains 3000. The whole thing is sketched below, including one more line that we will get to shortly.

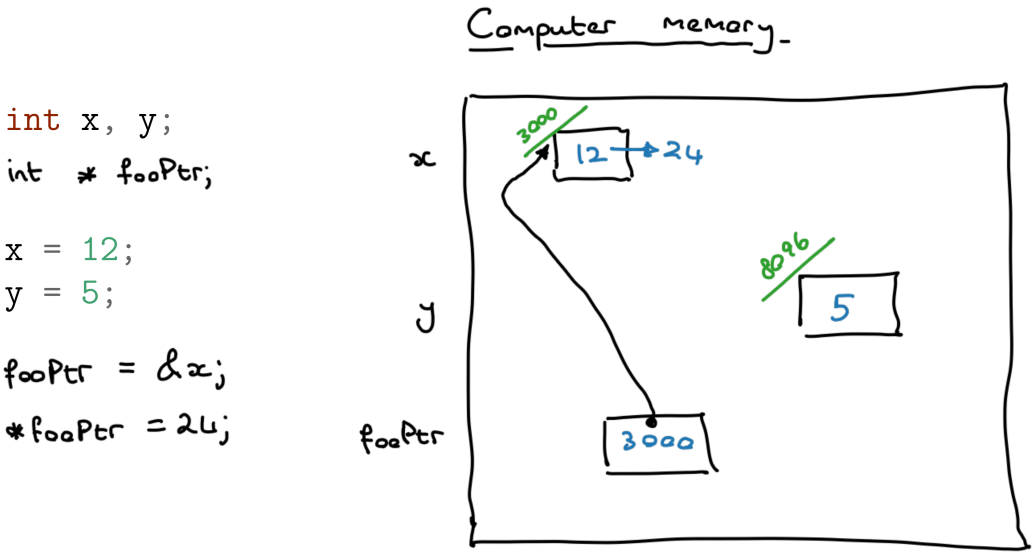

Figure 1: A pointer in memory. The boxes `x` and `y` hold 12 and 5 at the made-up addresses 3000 and 8096. The box `fooPtr` holds 3000, the address of `x`, which is drawn as an arrow pointing at `x`. The 12 is later changed to 24 through the pointer.

Because `fooPtr` holds the address of `x`, we say that `fooPtr` points to `x`, and on diagrams we draw an arrow from the pointer to the box whose address it holds. Just a note: these arrows don't really exist; all that exist is boxes with values inside them. It's just that, in the case of pointers, what is in the box is an address. That is all a pointer is. You do not yet know why this is useful, and that is fine. For now you only need to understand what is in each box.

Declaring pointers

A few more details about declarations. The type in front of the asterisk says what the pointer will point at. So `int *myPtr;` declares a pointer to an `int`, and you can declare pointers to any type in the same way: `double *` is a pointer to a `double`, while `char *` a pointer to a `char`.

You can declare several variables in one line, but each pointer needs its own asterisk:

```
int *myPtr1, *myPtr2, myInt;
```

Here `myPtr1` and `myPtr2` are pointers to integers. `myInt` does not have an asterisk, so it is an ordinary integer. Pointers and ordinary variables can be mixed in one declaration like this, but it is easy to misread, so be careful.

Sometimes you want a pointer that does not point at anything yet. For that you set it to `NULL`:

```
int *myPtr1 = NULL;
```

`NULL` is a special value that means "this is not a valid memory address; I am pointing nowhere". You will also see pointers set to 0 for the same purpose, but `NULL` makes the intent clear and is preferred.

You can even put a specific hardware address into a pointer yourself:

```
int *myPtr2;
myPtr2 = 0x22FF7C;  // hardware address
```

The `0x` means the number is written in hexadecimal, base 16, which is a convenient way of writing out the zeros and ones of an address. It is still just a number. You will rarely do this in an ordinary program, since you almost never know or care exactly where a variable lives; you get addresses from `&` instead. The point is that an address really is only a number sitting in a box.

Indirection: going to the box you point at

There is one more thing you need, and then you have everything for this lecture. If a pointer points at a variable, how do you use the pointer to change that variable?

Go back to `fooPtr`, which holds 3000 and so points at `x`. Now consider this line, which is the last one in the sketch:

```
*fooPtr = 24;
```

This does not put 24 into the `fooPtr` box. The asterisk in front of `fooPtr` means: go to the box that I am pointing at, and change the value there. So the line jumps to the box at address 3000, which is `x`, and replaces its 12 with a 24. That is the arrow from 12 to 24 in the sketch.

It helps to read the line out in full. `fooPtr` is an address, 3000. `*fooPtr` says go to address 3000. `*fooPtr = 24` says go to address 3000 and put 24 in the box you find there. After this line, `x` is 24, even though we never wrote `x` on the left of an equals sign.

Accessing a variable through a pointer like this is called indirection, and the `*` used this way is called the indirection or dereferencing operator. It works for reading as well as writing:

```
int x = 5;             // assign 5 to x (direct reference)
int *xPtr;             // declare an integer pointer
xPtr = &x;             // assign the address of x to xPtr
printf("%d", *xPtr);   // indirect reference to x, prints 5
*xPtr = 7;             // indirect reference, same effect as x = 7
```

Using `x` by name is a direct reference. Using `*xPtr` is an indirect one: you reach the same box, but by way of its address.

One warning. When you dereference a pointer, the pointer has to contain a valid memory address. A pointer that has just been declared contains rubbish, and `*` on rubbish means "go to some random address and change what is there". Always give a pointer an address before you dereference it.

Three meanings of the asterisk

You have now seen the asterisk used in three completely different ways in C:

1. Multiplication: `a * b`
2. Declaring a pointer: `int *fooPtr;` means create a box called `fooPtr` that will hold addresses of integers.
3. Indirection: `*fooPtr = 24;` means go to the box that `fooPtr` points at.

The second and third look alike but are not the same thing at all. The declaration creates the pointer box. The indirection changes what is inside a different box (the one being pointed at) and leaves the pointer itself alone.

You have met this kind of double meaning before with arrays. In `int b[5];` the 5 is a size: create five boxes. In `b[4] = 3;` the 4 is an index: put 3 in the last of those boxes. The square brackets mean different things in a declaration and in a statement, and the asterisk is the same. Whenever a line with an asterisk confuses you, first ask whether it is a declaration.

Back to `scanf`

We now have enough to understand part of the question at the start of the lecture: what does `&a` mean in `scanf("%d", &a)`? It means the address of `a`.

So `scanf` is not given the value inside `a`; it is given the location of the box. That is exactly what it needs, because its job is to put a number into that box. It takes the address, goes to the box at that address, and stores what the user typed there. This is how `scanf` manages to change `a` even though it does not return the number to you. How a function you write yourself can do the same thing is the subject of the next lecture.

A longer example: two pointers

Let us now step through a longer example, drawing memory as a column of boxes with the addresses on the left and the variable names on the right. Start with three integers:

```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
```

The integers get the values 1, 5 and 8, at the addresses 3000, 3004 and 3008. The last line declares two pointers, which get boxes of their own at 3012 and 3016. Nothing has been put in those two boxes yet, so they hold rubbish, shown in grey below.

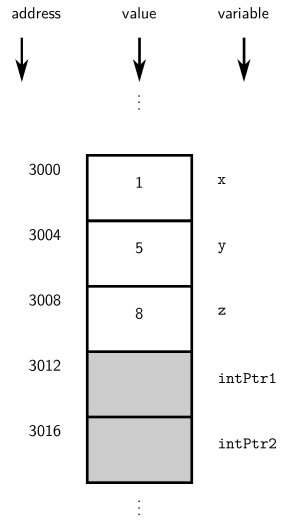

Figure 2: Memory after declaring three integers and two pointers. The pointer boxes have not been given values yet.

As an aside, the pointers here have `Ptr` in their names, but nothing requires that. A pointer is just a box, and it can be called anything. `int *apples;` declares a perfectly good pointer to an integer, and that pointer is called `apples`. Putting `Ptr` in the name is only a convention that helps you remember what a variable holds.

Now we run three lines, one after the other:

```
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
```

The first line puts the address of `x`, which is 3000, into the `intPtr1` box, so `intPtr1` points to `x`.

The second line is the first one that can look confusing, but you only need to open boxes. An assignment copies the value on the right into the box on the left. The value in `intPtr1` is 3000. So 3000 is copied into `intPtr2`, and now both pointers point to `x`. Nothing is done to `x` itself, and the two pointers are still two separate boxes that happen to hold the same number.

The third line puts the address of `y`, 3004, into `intPtr1`. That changes where `intPtr1` points, but `intPtr2` still holds 3000 and still points to `x`. The memory after each of these three lines is shown below.

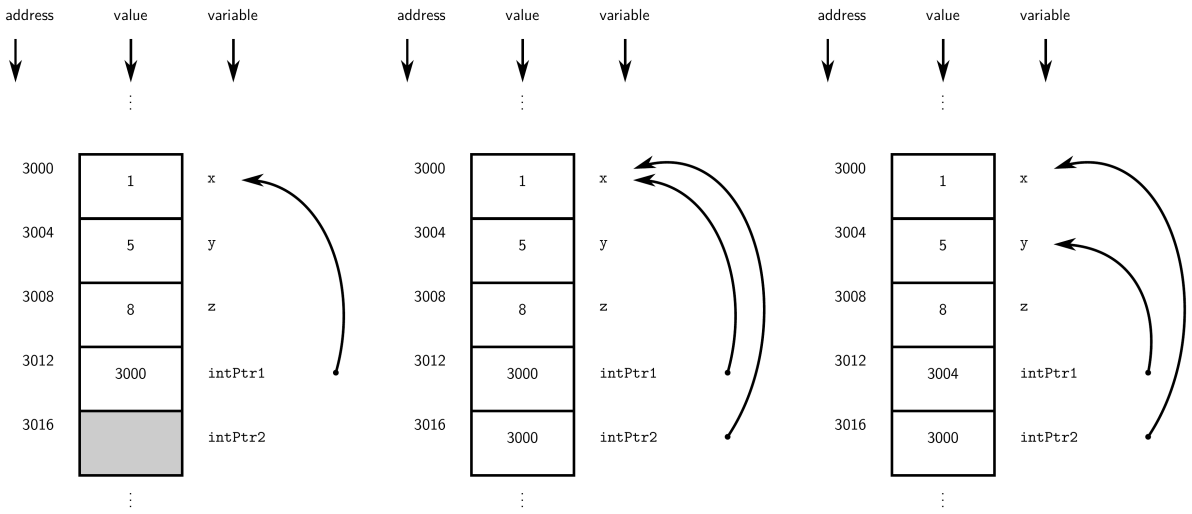

Figure 3: Memory after `intPtr1 = &x;` (left), after `intPtr2 = intPtr1;` (middle), and after `intPtr1 = &y;` (right).

Now three more lines:

```
*intPtr2 = 10;
*intPtr1 = z;
z = (*intPtr1) * (*intPtr2);
```

Before reading on, work out which variable each of them changes. It is very easy to look at the wrong pointer here.

In the first line, `intPtr2` holds 3000. The asterisk says go to the box at address 3000 and put 10 in it. That box is `x`, so `x` becomes 10.

In the second line, `intPtr1` holds 3004, so we go to the box at 3004, which is `y`. What goes into it is the value in the box called `z`, which is 8. So `y` becomes 8.

In the last line, we again go through the pointers one at a time. `intPtr1` holds 3004, so `*intPtr1` is the value at address 3004, which is now 8. `intPtr2` holds 3000, so `*intPtr2` is the value at 3000, which is now 10. Multiplying gives 80, and that is put into `z`. The memory after each of these three lines is shown below (top of next page).

You do not actually need the parentheses in that last line. Writing

```
z = *intPtr1 * *intPtr2;
```

gives exactly the same result, because the dereferencing `*` has higher precedence than the multiplication `*`. C first resolves the pointers, gets the values out of the boxes they point at, multiplies those, and only then stores the result. The parentheses just make it easier on the eye, and I would keep them.

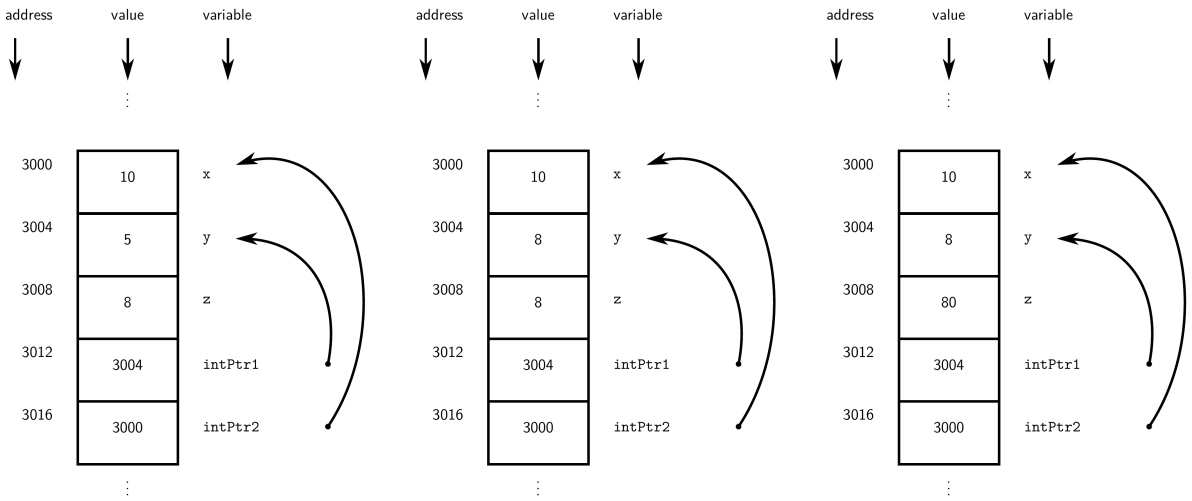

Figure 4: Memory after `*intPtr2 = 10;` (left), after `*intPtr1 = z;` (middle), and after `z = (*intPtr1) * (*intPtr2);` (right). At the end `x` is 10, `y` is 8 and `z` is 80, while the pointers still point at `y` and `x`.

Let's add one more line. What would the following do?

```
intPtr1 = &intPtr2;
```

You can answer it without any new rules by just going through the motions. The `&` means "the address of". `intPtr2` is a box like any other, and in our picture it lives at 3016. So `&intPtr2` is 3016, and that number is put into `intPtr1`, which now points at the `intPtr2` box. A pointer is a variable, so it has an address of its own, just like `x` does. (And, if we wanted to, we could point to that address.)

There is a catch, though. `intPtr1` was declared as a pointer to an `int`, and `intPtr2` is not an `int`; it is a pointer. The address of a pointer is a different type from the address of an integer, and if you compile that line your compiler will warn you about an incompatible pointer type. C does let you have pointers to pointers, but they are declared differently, and we will see that a bit later.

Another full example (and printing addresses)

Let us write an actual program. Here I have a pointer called `apples`, just to make the point that the name can be anything:

```c
/*
 * A first example using pointers.
 * Author: Herman Kamper
 */

#include <stdio.h>

int main()
{
    int a;        // a is an integer
    int *apples;  // apples is a pointer to an integer

    a = 7;
    apples = &a;  // apples is set to the address of a

    printf("The address of a is:     %p\n", &a);
    printf("The value of apples is:  %p\n\n", apples);

    printf("The value of a is:       %d\n", a);
    printf("The value of *apples is: %d\n\n", *apples);

    printf("* and & are complements:\n");
    printf("The value of &*apples:   %p\n", &*apples);
    printf("The value of *&apples:   %p\n", *&apples);

    return 0;
}
```

The format specifier `%p` is the one meant for printing addresses, and it prints them in hexadecimal. When I ran the program, the first two lines gave:

```
The address of a is:     0x7ffc25cb598c
The value of apples is:  0x7ffc25cb598c
```

Your numbers will be different, and if you run the program again they will probably change, because the operating system decides where in memory to put `a` each time the program starts. What will not change is that the two lines agree. The box `apples` contains the address of `a`, so printing the value of `apples` prints that address.

You may also see addresses printed with `%d`, as if they were ordinary integers. That compiles, with a warning, and gives a normal-looking decimal number, but it can come out negative. An address on a modern computer is usually 64 bits long, while `%d` expects a 32-bit integer, so only part of the address is printed and the top bit gets interpreted as a sign. Use `%p` for addresses.

It is worth trying one thing with this program. If you print `apples` before the line `apples = &a;`, you get some unrelated number. That is the rubbish that was in the box before we put anything in it.

The next two lines print 7 twice. `a` is the box itself, so it prints 7. `apples` on its own would print the address again, but `*apples` says don't print what is in the `apples` box; go to the box it points at and print what is in there, which is 7. The memory for this program is sketched below, with the enormous addresses written only as their first few digits (in green).

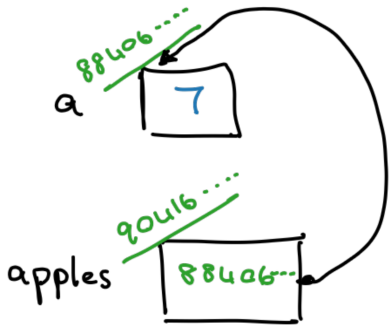

Figure 5: The program's memory. The box `a` holds 7. The box `apples` holds the address of `a`, and so points at it. The pointer has an address of its own too.

`*` and `&` cancel out

The last two `printf` lines in the program above are the hard ones. Take `&*apples` first. When an expression like this breaks your brain, read it from the inside out.

- `apples` is a box that holds an address, the address of `a`.
- `*apples` is the thing that `apples` points at. That is the box `a`. Printing it on its own would give 7.
- `&*apples` puts "the address of" in front of that. The address of the box `a` is just the address of `a`.

So `&*apples` prints the address of `a`, the same number as the first two lines.

Now the other order, `*&apples`, again from the inside out.

- `apples` is the pointer box.
- `&apples` is the address of that box. That is a new number, the address where the pointer itself lives (we haven't printed this out).
- `*&apples` says go to that address and give me what is in the box there. That box is `apples`, and what is in it is the address of `a`.

So `*&apples` also prints the address of `a`. The two operators undo each other. `&` says "give me the address of a box", and `*` says "go to the box at an address", so doing one and then the other gets you back where you started, in either order. The operators are said to be *complements of each other*.

You would not write this in a real program. The point is that if you know exactly what `&` and `*` each do on their own, then even an expression that looks crazy can be worked out one step at a time.

The arrow does not exist

It is worth being clear about one thing. When we draw a pointer, we draw an arrow from the pointer to the box it points at. That arrow is only a picture. In memory there is no arrow; there is a box, and inside the box is a number, and that number happens to be the address of another box. When you dereference a pointer, C takes that number and goes to that address.

Everything you can do with an ordinary box, you can do with a pointer box. You can copy its value into another box, as we did with `intPtr2 = intPtr1`. You can take its address, as we did with `&intPtr2`. You can print it. The only thing special about a pointer is what you intend the number inside it to mean.

The pointer box also has a size, like every box. Since it holds an address, it has to be big enough to hold any address the machine can use. On a 64-bit computer that is normally 8 bytes, no matter whether the pointer points to a `char`, an `int` or a `double`.

Looking forward

This lecture has been about the pieces: what a pointer is, `&` for the address of a variable, and `*` for going to the box a pointer points at. You should now be able to draw the boxes for any short piece of pointer code and say what is in each one afterwards.

What we have not done is use pointers for anything you could not already do. We will now start to use it from the next lecture onwards. We also haven't answered the question of how these relate to arrays (the motivation I started the lecture with); we will also get into the relationship between pointers and arrays in a following lecture.

Exercises

1. Draw the boxes for the following code, with made-up addresses, and write down the final values of `a` and `b`. Then run it and check.

```
int a = 3, b = 4;
int *p, *q;

p = &a;
q = &b;
*p = *q + 1;
q = p;
*q = *q * 2;

printf("a = %d, b = %d\n", a, b);
```

2. Which of the variables in each declaration below are pointers, and which are ordinary variables?

```
int *p, q;
double *r, *s, t;
char c, *d;
```

3. What is wrong with the following, and why might it crash or not crash when you run it? Fix it so that it puts 5 into an integer variable called `n` through the pointer.

```
int *p;
*p = 5;
```

4. Compile the line `intPtr1 = &intPtr2;` from the two-pointer example and read the warning carefully. In your own words, why does the compiler consider `&intPtr2` to be the wrong type for `intPtr1`?

5. Work out what each of the following prints, reading from the inside out, before running it. Assume `int a = 7;` and `int *apples = &a;`.

```
printf("%d\n", *apples);
printf("%d\n", *&a);
printf("%p\n", &*apples);
printf("%d\n", *&*apples);
```

6. Suppose `x` is 3 and `y` is 9, and `int *p = &x, *q = &y;`. Using only `p`, `q` and one extra `int` variable called `temp`, and never writing `x` or `y` again, swap the values of `x` and `y`. Print both variables afterwards to check.