18. Pointers and functions: call-by-reference

Herman Kamper

The previous lecture introduced pointers: boxes in memory that hold the addresses of other boxes. At the end of it you could say exactly what was in every box, but you might reasonably have wondered what the point was. Nothing we did with pointers in the last lecture was something you could not already do more simply by using the variable's name.

This lecture gives pointers their first real job. We are going to use them with functions, so that a function can change a variable that belongs to a different function. This is called *call-by-reference*, and you have actually already seen it twice without the name: once every time you wrote `scanf("%d", &a)`, and once when a function changed the elements of an array that was passed to it. (We'll look more at the realationship between pointers and arrays in a future lecture.)

To understand call-by-reference, we first need to be completely clear about what happens to a variable when it is passed to an ordinary function. That turns out to be the part that trips people up, so I am going to take it slowly. Let us first recap pointers, then recap how ordinary functions work, then consider this new call-by-reference idea.

Pointer recap

First, pointers. There are really only two things you need to know about pointers, and if you know them you can work out anything else. Here they are in a short program:

```c
#include <stdio.h>

int main()
{
    int x = 12;
    int *myPtr;  // declare pointer

    myPtr = &x;  // assign the address of x to myPtr

    printf("The value of x: %d\n", x);
    printf("The address of x: %p\n", myPtr);
    printf("The value of *myPtr: %d\n", *myPtr);

    *myPtr = (*myPtr) * (*myPtr);

    printf("The value of x: %d\n", x);

    return 0;
}
```

The first line in `main` creates a box called `x` and puts 12 in it. That box lives at some address; let us say 90964. The next line creates a second box called `myPtr`. The `int *` says that what goes inside this box is not an integer but the address of an integer. For now it holds rubbish.

The line `myPtr = &x;` is the first of the two things that you should know about pointers: the `&` means "the address of". So `&x` is 90964, and that number is put into the `myPtr` box. The box `myPtr` itself lives at an address too, say 1024. All of this is sketched below.

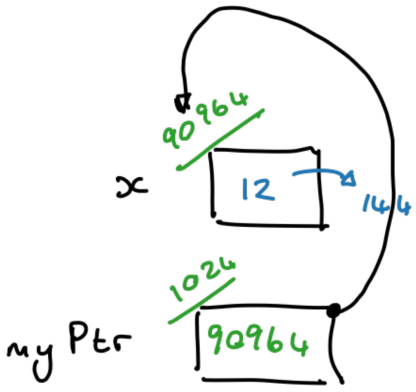

Figure 1: The pointer recap in memory. The box `x` holds 12 at address 90964. The box `myPtr`, at address 1024, holds 90964 and so points at `x`. The line `*myPtr = (*myPtr) * (*myPtr);` then changes the 12 to 144.

The first `printf` prints 12, and the second prints the address that is in `myPtr`, some long hexadecimal number.

The third `printf` uses the second thing you need to know about pointers: a `*` in front of a pointer, anywhere other than in its declaration, means "go to the box that I am pointing at". `myPtr` on its own is 90964. `*myPtr` is the box at 90964, which is `x`, so this prints 12.

Now work out what the line `*myPtr = (*myPtr) * (*myPtr);` does before reading on. The right-hand side is evaluated first. Each `*myPtr` means "go to the box `myPtr` points at and take out what is there". So the right-hand side is 12 times 12, which is 144. Then the left-hand side, `*myPtr =`. This says "go to the box that `myPtr` points at" and put 144 in it. So `x` becomes 144, and the last `printf` confirms it.

So, the two things that you need to know about pointers:

- `&` means "the address of a box". `&x` is the address of `x`.
- `*` in front of a pointer means "the box that I am pointing at". `*myPtr` is the box whose address is in `myPtr`.

One more reminder from last time: the arrow in the sketch does not exist. It is not somewhere on the memory chip. What exists is a box with a number in it, and the only thing special about a pointer's box is that we intend that number to be an address. I draw the arrow so that I remember where to go – it is only for illustrative purposes.

A normal function: call-by-value

Before we use pointers with functions at all, let us look very carefully at what happens when you call the kind of function you have been writing up to know. Here is a function that squares a number:

```
#include <stdio.h>

int square(int a);

int main()
{
    int x;
    int result;

    x = 9;

    result = square(x);

    printf("The value of result: %d\n", result);
    printf("The value of x: %d\n", x);

    return 0;
}
```

```
int square(int a)
{
    return a * a;
}
```

This prints 81 for `result` and 9 for `x`. Nothing surprising. But let us step through it in terms of boxes, because the details here are exactly what makes call-by-reference make sense later.

`int x;` and `int result;` create two boxes in `main`, and `x = 9;` puts 9 in the first. These boxes belong to `main`. They are created when `main` starts and deleted when `main` ends.

Then we reach `result = square(x);`. C jumps to `square`, and `square` gets its own chunk of memory, separate from `main`'s. The first thing that happens there concerns the parameter `int a`.

Look at `int a` inside the brackets of the function definition. It looks like a declaration, and it is one: it creates a box called `a` that belongs to `square`, exactly like a variable declared inside the function body would. What makes it different is that you never write `a = something` yourself. The function call itself (in `main`) does that. When C sees `square(x)` in `main`, it takes the value in `x`, which is 9, and puts a copy of it into the new `a` box. We say the value is *passed-by-value* (or, meaning the same thing, that this is a *call-by-value* function).

So a function argument is a variable that is declared and initialised in one go, at the moment the function is called. The call `square(x)` has the same effect as if the first line of `square` were

```
int a = x;
```

or, since `x` is 9,

```
int a = 9;
```

To make that concrete, here is another way of writing the same function, with a second local variable:

```
int myOtherSquareFunction(int a)
{
    int result;

    result = a * a;

    return result;
}
```

This function has two boxes of its own, `a` and `result`, and they are very similar. `result` is declared in the body and then given a value on the next line. `a` is declared in the brackets and given its value by the function call. That is the only difference between them. Neither of them has anything to do with the `result` or the `x` in `main`, even though one of them happens to share a name.

The rest of `square` runs as normal. `a * a` is 81, and `return` sends 81 back to `main`. Then `square` ends, and its box `a` is deleted; that memory can now be used for something else. Back in

`main`, the 81 that came back is stored in the `result` box.

Two consequences follow from this. The first is that `a` is a copy. If `square` changed `a`, say by writing `a = 100;`, that would change `square`'s box and leave `x` in `main` sitting at 9. The second is that `square` cannot touch `x` at all. If you write `x = 5;` inside `square`, the program does not compile:

```
error: 'x' undeclared (first use in this function)
```

As far as `square` is concerned, there is no `x`. That box belongs to `main`.

Again, this is called call-by-value, or pass-by-value; the two terms mean the same thing. The value is what gets passed: a copy of what is in `x` goes into `a`, and from then on the two boxes have nothing to do with each other. It is what every function in this course has done so far.

If you struggle to see all of this, draw out the memory diagram for yourself, and then step through the example again. Do this before continuing.

Call-by-reference

Now suppose we want something different. We want a function that squares `x` in place: after calling it, `x` in `main` should be 81, without `main` having to catch a returned value and store it. In other words, we want a function that changes a variable belonging to a different function. We have just seen that call-by-value cannot do that. Here is how pointers can:

```c
/*
 * Square a value in place.
 * Author: Herman Kamper
 */

#include <stdio.h>

void square(int *input);

int main()
{
    int x;

    x = 9;

    printf("The value of x: %d\n", x);

    square(&x);

    printf("The value of x now is: %d\n", x);

    return 0;
}
```

```
void square(int *input)
{
    *input = (*input) * (*input);
}
```

The function is `void`, so it returns nothing. And yet when you run it, it prints

```
The value of x: 9
The value of x now is: 81
```

Let us step through it with boxes, in exactly the same way as before.

`int x;` creates a box in `main`'s memory, and `x = 9;` puts 9 in it. Say the box lives at address 80641. The first `printf` prints 9.

Now we reach `square(&x);`. The argument is no longer `x`; it is `&x`, the address of `x`, which is 80641. C jumps to `square`, which gets its own memory, and the parameter is created.

This is the same rule as before. The parameter `int *input` is a declaration: it creates a box called `input` that belongs to the memory-space of the function `square`. The type of this box is `int *`, so what goes inside it is the address of an integer. And it is initialised by the call: whatever was passed is copied into it. What was passed is 80641. So the call `square(&x)` has the same effect as if the first line of `square` were

```
int *input = &x;
```

(That line won't actually work since `x` isn't in `square`, but you get the point.) After the call to `square`, the box `input` contains 80641, so `input` points at `x`. The two functions and their boxes are sketched below. Here is the crucial thing: even though `x` is not accessible in `square`, we can now still reach it because we know where it lives in memory.

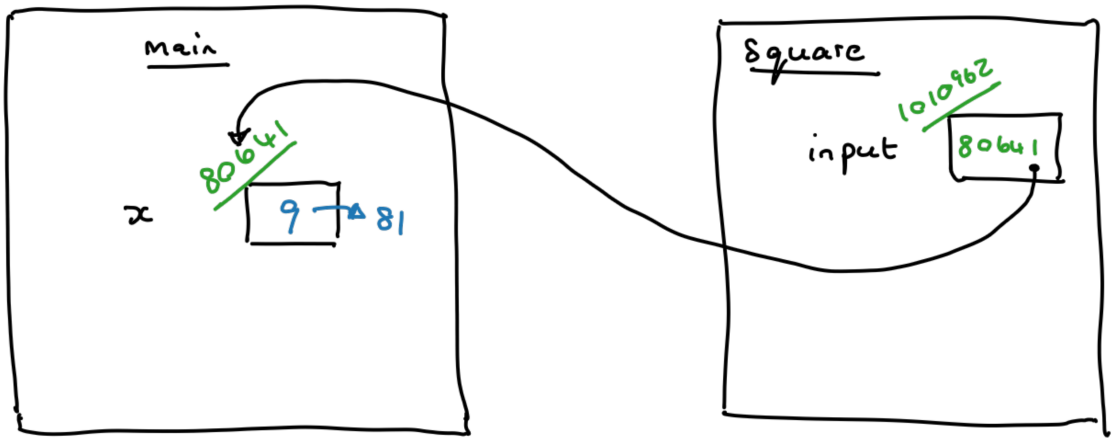

Figure 2: Call-by-reference. The box `x` belongs to `main` and lives at address 80641. When `square(&x)` is called, `square` gets its own box `input`, here at address 1010962, and 80641 is copied into it. Going to the box that `input` points at changes the 9 in `main` to 81.

Now the body of the function runs:

```
*input = (*input) * (*input);
```

On the right-hand side, `*input` means go to the box that `input` points at. `input` holds 80641; the box at 80641 holds 9. So the right-hand side is 9 times 9, which is 81. On the left-hand side, `*input =` means go to the box that `input` points at and put the result there. So 81 goes into the box at 80641, which is `x`.

Then the `square` function ends. Its box `input` is deleted, along with the address that was in it. But `x` is not part of `square`'s memory, so it is untouched by that. It still holds 81, and the second `printf` prints 81.

This is *call-by-reference* (or *pass-by-reference*). Instead of a copy of the value, the function receives the address of the variable, a reference to where it lives. It uses `*` to go to that address and change what is there.

There are two questions that nearly everyone has at this point, so let us answer them directly.

The first is: the box `input` is deleted when `square` ends, so how can it have changed anything in `main`? The answer is that nothing was ever changed in `input`. The value in `input` stayed 80641 the whole time. What changed was the box at address 80641, and that box belongs to `main`. The function was, in a sense, cheating: it was told where one of `main`'s boxes lives, and it went there. That is the whole point of call-by-reference (that we do via pointers).

The second question is: in the previous lecture we first declared a pointer and then set it with a separate line like `myPtr = &x;`. Why is there no line like `input = &x;` here? Because, as with `int a` earlier when we looked at normal call-by-value functions, the parameter is declared and initialised in one go. The `*` in `void square(int *input)` is the declaration version of `*`: it says create a box that holds addresses. The call `square(&x)` provides the initial value. The `*` inside the body, in `*input = ...`, is the other version of `*`: it says "go to the box I am pointing at". Those are two completely different uses of the same symbol, just as they were in the previous lecture: one is declaration, the other is dereferencing.

A slower look: cubing a number

Let us do this once more with a second example, stepping through the memory one frame at a time. This time the function cubes a number, and we will write it both ways (call-by-value and call-by-reference). If the call-by-value version is completely clear to you, the call-by-reference version will be too. Here is the call-by-value one:

```c
#include <stdio.h>

int cubeByValue(int n);

int main()
{
    int number = 5;

    printf("The original value of number is %d\n", number);

    // Pass number by value to cubeByValue
    number = cubeByValue(number);

    printf("The new value of number is %d\n", number);

    return 0;
}

// Calculate and return cube of integer argument
int cubeByValue(int n)
{
    return n * n * n;
}
```

It prints 5 and then 125.

The line `int number = 5;` creates a box in `main` and puts 5 in it straight away. At this point `cubeByValue` has not been called, so its box `n` does not exist yet; it is undefined. Then comes `number = cubeByValue(number);`. The memory just after the call, and then while the function body runs, is shown below. The red outlines mark what is happening at each step.

On the left, the call creates the box `n` and copies the value of `number` into it. That is the declaration and initialisation in one go: it is as if `cubeByValue` started with `int n = 5;`. On the right, `n * n * n` is calculated as 125. The next two steps are shown below.

On the left, 125 is returned to `main`, and `cubeByValue` ends, so `n` is undefined again. On the right, the 125 that came back is stored in `number`.

Notice that `n` and `number` never shared a box. `cubeByValue` never touched `number`; `main` did that itself, with the assignment.

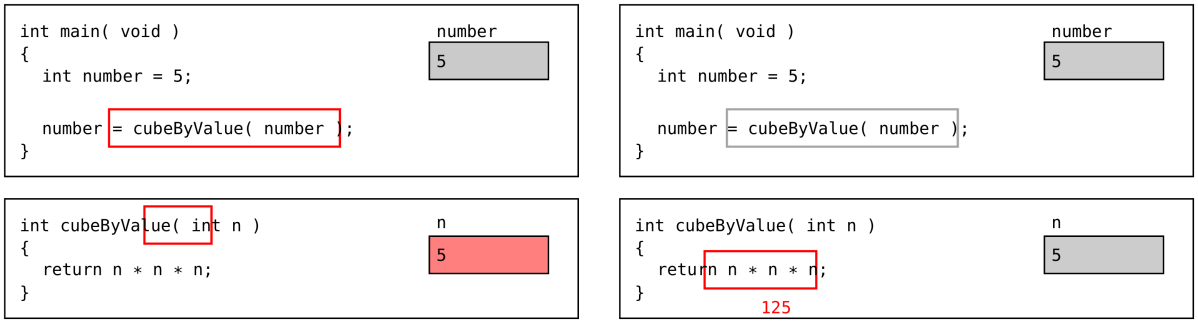

Figure 3: Call-by-value, steps 1 and 2. Left: calling `cubeByValue(number)` creates the box `n` and copies 5 into it. Right: `n * n * n` is 125.

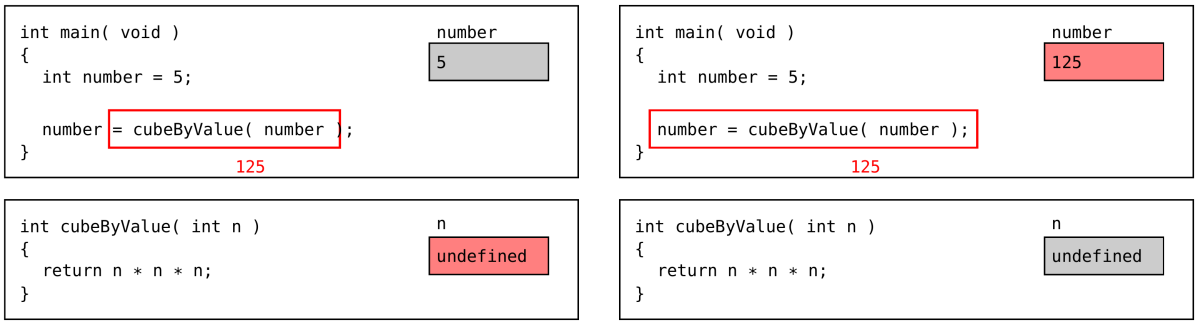

Figure 4: Call-by-value, steps 3 and 4. Left: 125 is returned and `n` is deleted. Right: 125 is stored in `number`.

Now the call-by-reference version:

```c
#include <stdio.h>

void cubeByReference(int *nPtr);

int main()
{
    int number = 5;

    printf("The original value of number is %d\n", number);

    // Pass address of number to cubeByReference
    cubeByReference(&number);

    printf("The new value of number is %d\n", number);

    return 0;
}

// Calculate cube of *nPtr; modifies variable number in main
void cubeByReference(int *nPtr)
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

It also prints 5 and then 125. Let's run through it. `main` starts the same way, with a box `number` holding 5. The function is `void` and takes one parameter, `int *nPtr`, a box that holds the address of an integer. The memory just after the call, and then while the right-hand side is worked out, is shown below.

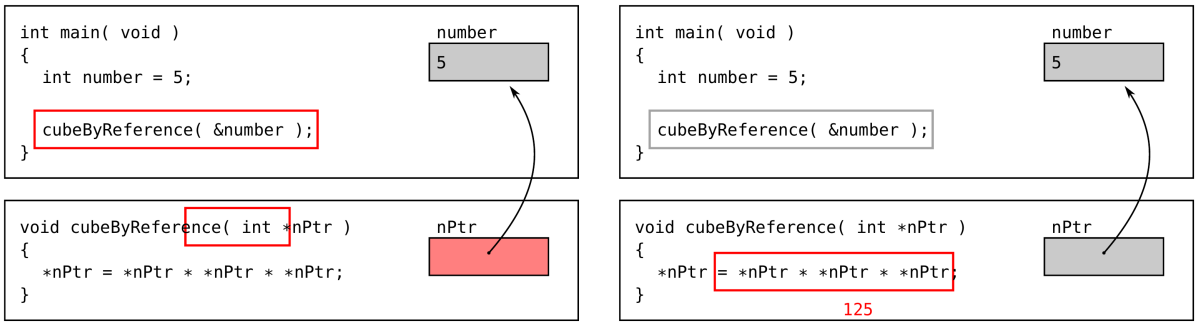

Figure 5: Call-by-reference, steps 1 and 2. Left: calling `cubeByReference(&number)` creates the box `nPtr` and copies the address of `number` into it, so `nPtr` points at `number`. Right: `*nPtr * *nPtr * *nPtr` is 125.

On the left, the call creates the box `nPtr` and puts `&number` in it. Say `number` lives at 90247. Then the call has the same effect as `int *nPtr = &number;`, or `int *nPtr = 90247;`. We

have declared a pointer and set its value in one go, and it now points at `number`. On the right, each `*nPtr` means "go to the box that `nPtr` points at", which holds 5, so the right-hand side is 5 times 5 times 5, which is 125. The next two steps are shown below.

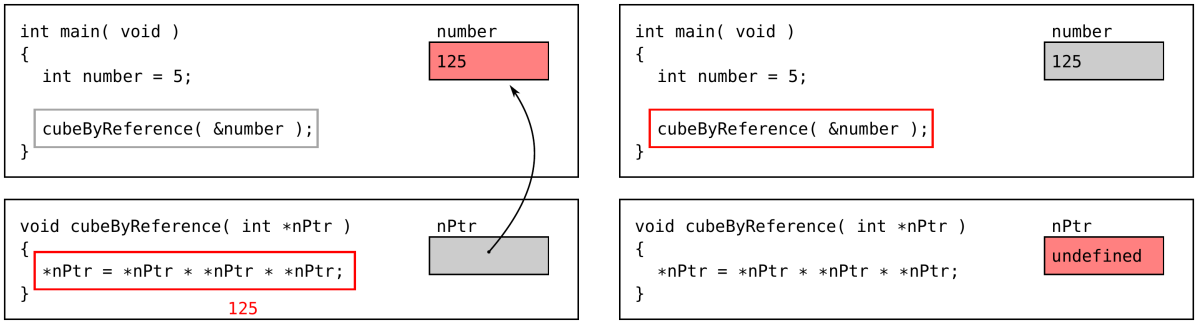

Figure 6: Call-by-reference, steps 3 and 4. Left: 125 is stored in the box that `nPtr` points at, which is `number`. Right: the function ends and `nPtr` is deleted, but `number` stays 125.

On the left, `*nPtr =` says "go to the box that `nPtr` points at and put 125 in it", so `number` changes to 125. On the right, the function ends, `nPtr` is deleted, and `number` still holds 125.

Both programs end with `number` at 125, but they get there in very different ways. The table below puts them side by side.

	Call-by-value	Call-by-reference
The call	`number = cubeByValue(number);`	`cubeByReference(&number);`
The argument	`int n`, a copy of the value	`int *nPtr`, a copy of the address
As if the function began with	`int n = number;`	`int *nPtr = &number;`
What changes `number`	`main`, by storing the returned value	the function, by going to its address

When would you use this?

So far, both versions have done the same job. Why would you ever use call-by-reference? Partly, it is simply a very good way of making sure you understand exactly what pointers and functions do. But it also lets you do things that call-by-value cannot. Let us look at some examples.

The first is something you have been using since the start of the course. When you write

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

you are passing `scanf` the address of `a`. `scanf` does not give you the number it read as its return value. Instead, it goes to the address you gave it and puts the number in the box there. That is call-by-reference, and now you know why the `&` has to be there: without it, `scanf` would get a copy of whatever rubbish is in `a`, and would have no way of reaching `a` itself.

The second example use-case is returning more than one thing. A C function can only return a single value. Suppose you want one function that finds both the smallest and the largest number in a list. You cannot return both. But you can pass in the addresses of two variables, `min` and `max`, and have the function put its answers in those boxes.

There is a well-known animation that illustrates the difference with a cup of coffee (you can watch it here on YouTube or at this website). Pass the cup by reference, and filling it inside the function fills the original cup. Pass it by value, and the function receives its own copy of the empty cup; filling that leaves the original cup empty.

A last example: Swapping two values

Here is one last example, and it is a task that is genuinely awkward without pointers. We want a function that takes two variables and swaps their values: whatever was in the first should end up in the second, and the other way round. With call-by-value this cannot work, because the function only ever sees copies. Swapping the copies inside the function leaves the originals exactly as they were. With call-by-reference it is short:

```c
/*
 * Swap two values in place.
 * Author: Herman Kamper
 */

#include <stdio.h>

void swap(int *a, int *b);

int main()
{
    int x = 7, y = -2;

    printf("x = %d, y = %d\n", x, y);

    swap(&x, &y);

    printf("x = %d, y = %d\n", x, y);

    return 0;
}

void swap(int *a, int *b)
{
    int hold = 0;

    hold = *a;
    *a = *b;
    *b = hold;
}
```

It prints

```
x = 7, y = -2
x = -2, y = 7
```

Before reading the step-by-step explanation, stare at the function and see if you can explain to yourself why it works.

In `main`, `x` holds 7 and `y` holds -2. Say they live at addresses 84083 and 1024. The call `swap(&x,`

`&y);` passes those two addresses. `swap` gets its own memory, with two parameter boxes, `a` and `b`, and they are initialised in one go. So `a` holds 84083 and points at `x`, and `b` holds 1024 and points at `y`. The next line creates a third box in `swap`, called `hold`, with 0 in it. The whole thing is sketched below.

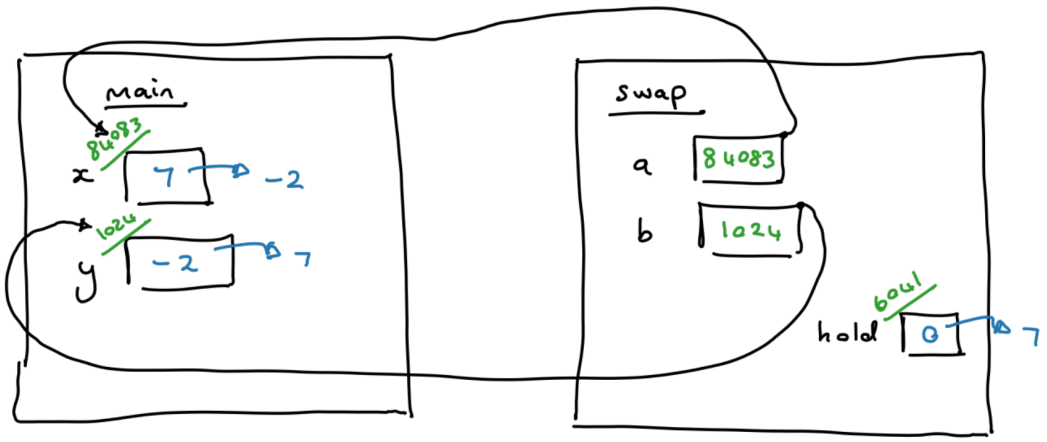

Figure 7: Swapping with pointers. `swap` gets the boxes `a` and `b`, holding the addresses of `x` and `y` in `main`, and its own box `hold`. By the end, `hold` is 7, `x` has become −2 and `y` has become 7.

Now the three lines inside `swap`:

- `hold = *a;` The right-hand side is the box that `a` points at, which is `x`, holding 7. So `hold` becomes 7.
- `*a = *b;` Right-hand side first: the box that `b` points at is `y`, holding −2. Then the left-hand side says go to the box that `a` points at, `x`, and put −2 there. So `x` becomes −2.
- `*b = hold;` `hold` is 7. Go to the box that `b` points at, `y`, and put 7 there. So `y` becomes 7.

The function returns nothing, but by the time it ends it has swapped the values of two variables that belong to `main`.

You might wonder whether `hold` is really needed. Try the swap without it:

```
void swap(int *a, int *b)
{
    *a = *b;
    *b = *a;
}
```

This prints `x = -2, y = -2`. The first line puts −2 into `x`, which wipes out the 7. When the second line then goes to `x` to copy its value into `y`, the value it finds there is already −2. The 7 is gone. That is why the original value has to be held somewhere before it is overwritten.

Looking forward

We know from before that function arguments are declared and initialised in one go when the function is called. If an argument is an ordinary variable, it receives a copy of a value. If it is a pointer, it receives a copy of an address, and with `*` the function can go to that address and change what is there. Everything else follows from that together with the two things from the recap: `&` for the address of a box, and `*` for the box a pointer points at.

It is worth saying that you will very probably never write your own sorting or searching function outside a course like this one; those come ready-made in libraries. Pointers are different. If you write C, and particularly if you work anywhere near hardware, you will use them all the time, and you will occasionally curse them. With these basics, though, they come down to boxes and addresses.

The next lecture looks at more operations on pointers.

Exercises

1. For the call-by-value `square` program, draw the boxes of `main` and `square` just after `square(x)` has been called and just after it returns. Then add the line `a = 100;` just before the `return` in `square`. Predict what the program prints, and run it to check.
2. What does the following print? Draw the boxes before you run it.

```c
void addOne(int n)
{
    n = n + 1;
}

void addOneByReference(int *nPtr)
{
    *nPtr = *nPtr + 1;
}

int main()
{
    int count = 3;

    addOne(count);
    printf("%d\n", count);

    addOneByReference(&count);
    printf("%d\n", count);

    return 0;
}
```

3. In the call-by-reference `square` program, remove the `&` from the call so that it reads `square(x);`. Compile it and read the warning. Explain in terms of boxes what the function would try to do if you ran it anyway.

4. Write a function `void minMax(int array[], int size, int *min, int *max)` that finds the smallest and the largest element in an array and stores them through `min` and `max`. Call it from `main` with an array of your choice and print both results.

5. Write a function `void sortTwo(int *a, int *b)` that makes sure that, after the call, the smaller of the two values is in the variable `a` points at and the larger in the one `b` points at. Use your `swap` function inside it.

6. Try to write a call-by-value swap, `void swap(int a, int b)`, that swaps the values of `x` and `y` in `main`. Run it, and explain in terms of boxes why it cannot work.

7. Write a function `void split(double value, int *wholePart, double *fractionPart)` that splits a number such as 3.75 into 3 and 0.75. Test it with a few values, including a negative one, and decide what you think the right answer for -3.75 should be.