9. Functions

Herman Kamper

Everything so far has gone inside `main`. That was fine while the programs were twenty lines long, but it does not scale, and it is not how real C is written. If you open up an actual C file, almost every block of code you see will be a function definition. This lecture is about what that means and how to write your own.

The good news is that you already know what a function is, and you have already used several without thinking about it. We start there, then build one from scratch, then step through what happens in memory when it runs, because that last part is where people get stuck.

What is a function?

If you have not met the word in a programming context before, the closest thing to it is a machine. You put something in at one end, something happens inside, and something comes out at the other end. That really is the whole idea.

You have seen this before in mathematics. Somewhere around grade two, a teacher wrote something like this on the board:

$$f(x) = \begin{cases} x^2 & \text{when } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

You put numbers in and different numbers come out. So if we ask what $y_1 = f(5)$ is, the answer is 25: five is bigger than zero, so we square it. Now what about $y_2 = f(-5)$?

If you answered 25, you are in good company, and you are wrong. Feed -5 into the machine and follow the algorithm honestly: is x bigger than zero? No, it is not. So we take the second branch, and y_2 is 0.

The things we feed in have a name. They are called the arguments of the function. In $f(5)$ the argument is 5; in $f(-5)$ the argument is -5. The function itself is f, and it stays the same each time: only the argument changes.

The right-hand side of the figure is the same idea written the way a program would write it. There is a name, `sin`, and then something in parentheses. That something is the argument. `sin` takes an angle and hands back a number.

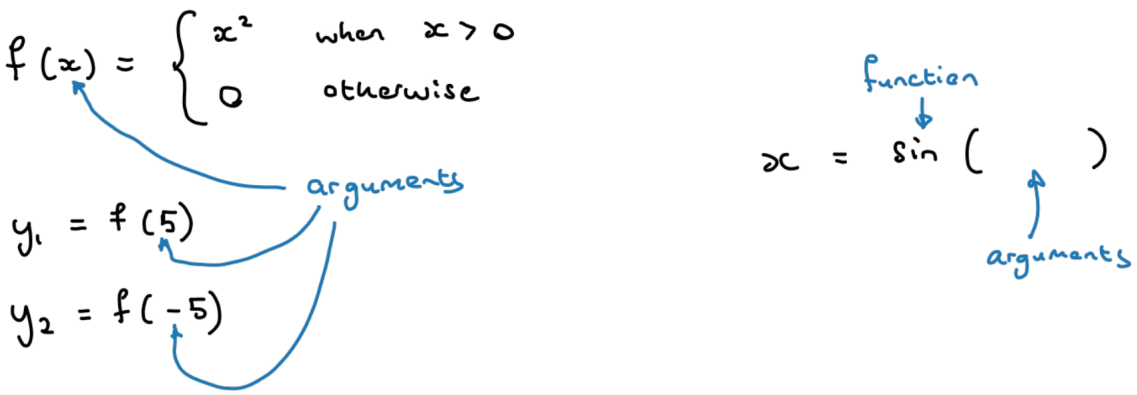

Figure 1: The mathematical function on the left and a function call on the right, with the arguments marked in each case.

Functions on a computer work in almost exactly this way. We pass things in, something happens inside, and normally we get something back out.

Functions you have already used

Now that you know the word, you will recognise that you have been calling functions since your very first program.

`printf` is a function:

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

There is the function name, `printf`, and then a list of arguments in parentheses. It can take a whole list of them, separated by commas. What does it do with them? It puts things on the screen. That is the work that happens inside the machine.

`scanf` is a function too:

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

Same shape: a name, and then the arguments it needs to do its job.

Both of these live in `stdio.h`. That is where those functions are defined. Someone sat down and wrote them, and then packaged them up so that everyone else could use them without having to look inside. You may also have used `pow` from `math.h`:

```
x = pow(5, 3);
```

which calculates 5^3. Here the arguments are two numbers: the first is the base, the second is the power to raise it to.

The last one is the interesting one. In the first five minutes of this course you wrote this:

```
int main()
{
    ...
    return 0;
}
```

That is a function. You have already written one. `main` is special in that it is the first function that runs when a program is executed (it is the entry point into the whole program), but structurally it is an ordinary function like any other, and once you can read the rest of this lecture you will see that every piece of it has a purpose.

So: you have used other people's functions and you have written `main`. What is left is to write functions of your own.

Writing your own functions

Every function you write has the same shape:

```
type functionName(argument1, argument2)
{
    // Declare variables

    // Statements

    return <something>;
}
```

Reading that from the top:

- It starts with a type. This is the type of thing the function will hand back when it is done, so `int` or `float` or `double`. This is called the return type.
- Then comes the function name. This is just a name you choose.
- Then parentheses, holding the list of arguments the function takes in, separated by commas.
- Then curly braces. Everything inside them is the body of the function.

Inside the body you can write any code you like. Typically it goes in three parts. First you declare the variables the function needs. These belong to the function: they only exist while it is running, and outside the function they mean nothing at all. Then come the statements that do the actual work, and these can be anything: `if` statements, loops, a `switch`, whatever the job requires. Finally there is usually a `return` statement, and whatever you put after `return` is what the function hands back to whoever called it.

That is what the `return 0;` at the bottom of `main` has been doing all along. It hands back a zero, which by convention means that the program completed successfully.

A first function: the maximum of three numbers

Descriptions of syntax are not much use on their own, so let us write one. We want a function that takes in three integers and tells us which of the three is the largest.

Here is the program before the function exists. It declares its variables and reads three numbers from the user:

```c
#include <stdio.h>

int main()
{
    // Variables
    int num1, num2, num3;
    int result;

    // User input
    printf("Enter three integers:\n");
    scanf("%d", &num1);
    scanf("%d", &num2);
    scanf("%d", &num3);

    return 0;
}
```

`result` is where we will eventually put the largest of the three. Compiling and running this at least tells us whether we have left out a semicolon somewhere, but it does nothing else: you type three numbers and the program ends.

Now the function. We follow the template exactly. The type first: this function hands back the biggest of three integers, so the type it returns is `int`. Then a name, and `maximum` is a reasonable one. Then the arguments in parentheses. There are three of them and they are all integers, so:

```c
int maximum(int x, int y, int z)
{
    ...
}
```

Note that each argument gets its own type written out in full. You cannot write `int x, y, z` in an argument list and hope C works it out. Every argument is declared separately, type and name, every time.

These three names are how the function refers to whatever it is given. When we call the function later, the values we pass in will land in `x`, `y` and `z`.

Now for the body. Before writing code it is worth having the algorithm straight, and this one is simple enough to say in a sentence: pick one of the numbers and guess that it is the biggest, then check the other two in turn, and whenever you find something bigger, that becomes your new guess. By the time you have looked at all three, your guess is right.

Written out:

```
int maximum(int x, int y, int z)
{
    int max;

    max = x;

    if (y > max)
    {
        max = y;
    }

    if (z > max)
    {
        max = z;
    }

    return max;
}
```

`max` is a new variable that belongs to this function. Inside the function we can use it alongside `x`, `y` and `z` as freely as we like. We start by assuming `x` is the largest. Then we ask whether `y` beats the current maximum, and if it does, `y` becomes the new maximum. Then the same question for `z`. At the end, `max` holds the largest of the three, and `return max;` hands it back.

If you are not completely sure why this gives the right answer, put the book down for a minute and convince yourself with a few sets of three numbers. It is worth being certain about this before the next part.

Now compile and run the program with the function added. Type in 4, 5 and 6, and… nothing happens. The program reads three numbers and exits, exactly as before.

That is not a bug. We have written the function but we have never called it. When a C program runs, it goes to `main`, executes the statements there in order, and stops. Nothing else runs. A function that is never called is just a description of something that could happen, sitting in the file doing nothing.

The prototype

Before we can call it, there is one more line to add, right at the top of the file:

```
int maximum(int x, int y, int z);
```

This is called a prototype. It looks like the first line of the function, but with a semicolon instead of a body. It is a promise: there is a function called `maximum`, it takes three integers, it hands back an integer, and I will define it properly further down. Hold on to your seat.

The reason it is needed is that C reads your file from top to bottom. When it reaches the call

in `main` and the definition is still a hundred lines below, the prototype is what tells it that the function exists and what shape it has.

This is also what `#include <stdio.h>` has been doing all along. That file is full of prototypes (for `printf`, for `scanf`, for everything else in `stdio`), and including it tells C which functions you are allowed to call.

Calling the function

Now we can use it. In `main`, after the three numbers have been read:

```
result = maximum(num1, num2, num3);
```

Run it, type 1, 9 and 3, and once again nothing appears on the screen. This time it genuinely has calculated the answer; it just has not been asked to say anything about it. We need a `printf`:

```
printf("The max is: %d\n", result);
```

Here is the whole thing:

```
/*
 * Calculate the maximum of three numbers using a function.
 * Author: Herman Kamper
 */

#include <stdio.h>

int maximum(int x, int y, int z);


int main()
{
    // Variables
    int num1, num2, num3;
    int result;

    // User input
    printf("Enter three integers:\n");
    scanf("%d", &num1);
    scanf("%d", &num2);
    scanf("%d", &num3);

    // Determine maximum
    result = maximum(num1, num2, num3);
    printf("The max is: %d\n", result);

    return 0;
}
```

```c
int maximum(int x, int y, int z)
{
    int max;

    max = x;

    if (y > max)
    {
        max = y;
    }

    if (z > max)
    {
        max = z;
    }

    return max;
}
```

And it works:

```
Enter three integers:
1
9
3
The max is: 9
```

It is worth trying a case where the answer is not obvious, because it is easy to write a maximum function that quietly gets negative numbers wrong:

```
Enter three integers:
-1
-2
-9
The max is: -1
```

What actually happens in memory

That is the program working, but it is worth going through it very slowly once, because this is the part that people find genuinely confusing and it does not get easier by being skipped.

Run it again and think about the boxes in memory. The first two lines of `main` create four of them: `num1`, `num2`, `num3` and `result`. They all belong to `main`. At this point they are empty: declaring a variable makes a box, it does not put anything in it.

Then the three `scanf` calls run. Type 7 and a 7 goes into the `num1` box. Type 1 and it goes into

num2. Type 4 and it goes into num3. So we arrive at the call with three boxes filled and result still empty.

Now this line runs:

```
result = maximum(num1, num2, num3);
```

Ignore the result = part for a moment and look at what is on the right. It calls a function, and it calls it with the values 7, 1 and 4. It makes no difference to the function where those values came from: as far as maximum is concerned, this is the same as if we had written maximum(7, 1, 4).

C now jumps to the function. Inside it, three new boxes come into existence. They are not called num1, num2 and num3: their names are x, y and z. The values get copied across, 7 into x, 1 into y, 4 into z. Then int max; creates a fourth box belonging to the function.

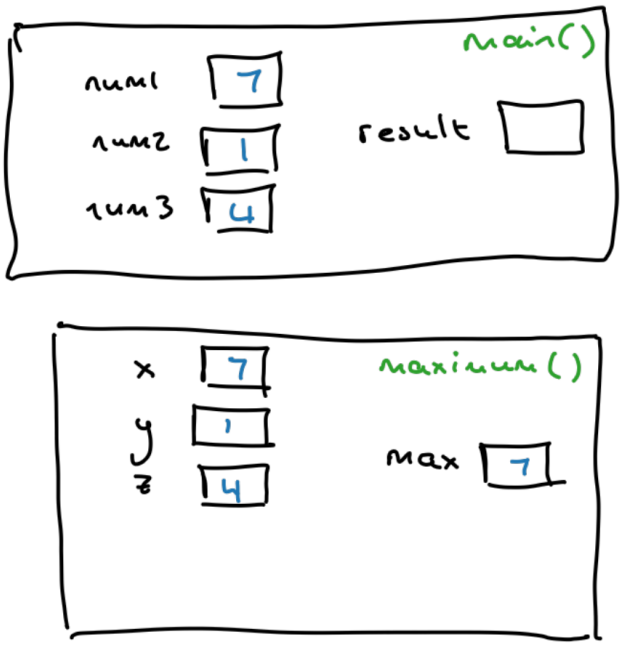

Figure 2: The boxes in memory. The four at the top belong to main; the four at the bottom are created when maximum is called, and the values from num1, num2 and num3 are copied into x, y and z.

From there we just follow the statements. max = x puts 7 in the max box. Is y bigger than max, that is, is 1 bigger than 7? No, so that block is skipped. Is z bigger than max, is 4 bigger than 7? Also no, so that block is skipped too. Had either been true, the value in the max box would have been swapped out for the bigger number.

Then the last line, return max;. The way to think about what happens here is that the entire function call collapses into the value it produced. The function was declared to return an int, and the int it is returning is 7, so the whole of maximum(num1, num2, num3) becomes, simply, 7. Everything else in the function is wiped away. What is left is:

```
result = 7;
```

A 7 goes into the `result` box, the `printf` runs, and 7 appears on the screen.

Two things follow from this picture, and they are worth stating plainly. You cannot use `x`, `y` or `z` anywhere in `main`, because those boxes do not exist there. And you cannot use `num1` inside `maximum` – that box is not one of the function's own. Each function can see only its own boxes.

If this is the bit that loses you

The line that trips people up is almost always this one:

```
result = maximum(num1, num2, num3);
```

Here is another way to look at it. `maximum` is a new command. It is a command that did not exist until we created it, in exactly the same way that `printf` is a command that someone else created before we came along.

Suppose I simply told you: there is a command called `maximum`, you give it three numbers, and it gives you back the biggest of them. I would not tell you how it works and you would not ask. Would that line make sense then? Almost certainly yes. You would read it as "give the three numbers to the command, and put whatever comes back into `result`", and you would be entirely correct.

If so, you are most of the way there, and the only thing left to absorb is that this command is one we wrote ourselves rather than one we were handed. And there is nothing special about the name. We could have called it anything at all: change the name in the prototype, in the definition and at the call, and the program behaves identically. The name is for your benefit, not the compiler's.

Inside that command there are variables of its own, boxes that it uses to work out the answer and that vanish when it is done. That is the only extra idea.

Functions that return nothing

Our `maximum` hands something back. Many functions do not.

`printf` is the obvious case: you call it to put text on the screen, not to obtain a value from it. (It is a slightly awkward example, because both `printf` and `scanf` do hand back a number: how many characters were printed, and how many items were successfully read. You almost never look at those, and neither is the reason you called the function.) There are plenty of functions that genuinely hand back nothing at all.

Those functions get a special return type, `void`, which is C's way of saying that this function returns nothing. A `void` function has no `return` statement at the end; it just runs to the closing brace and stops.

Let us write one. It will be a function that makes absolutely certain you know who this program belongs to:

```
void printCopyright()
{
    printf("(c) Herman Kamper, 2026\n");
}
```

The type is `void`, the name is `printCopyright`, and this one does not even take any arguments, so the parentheses are empty. There is no `return` at the end, because there is nothing to return.

It needs a prototype like any other function:

```
void printCopyright();
```

And now we have a new command that prints the copyright notice, so we can use it wherever we want. To be really sure the message gets through, we can call it twice, once at the start of `main` and once at the end:

```
#include <stdio.h>

int maximum(int x, int y, int z);
void printCopyright();


int main()
{
    // Variables
    int num1, num2, num3;
    int result;

    printCopyright();

    // User input
    printf("Enter three integers:\n");
    scanf("%d", &num1);
    scanf("%d", &num2);
    scanf("%d", &num3);

    // Determine maximum
    result = maximum(num1, num2, num3);
    printf("The max is: %d\n", result);

    printCopyright();

    return 0;
}
```

Running it:

```
(c) Herman Kamper, 2026
Enter three integers:
7
7
7
The max is: 7
(c) Herman Kamper, 2026
```

Follow that through slowly, the same way we did before. `main` starts and creates its four boxes. Before anything goes into them, it hits `printCopyright()`. C jumps down to that function. There are no arguments, so there is nothing to copy across. It runs the statements in the body, which puts the notice on the screen. It reaches the closing brace, and because the function is `void` there is nothing to hand back. Control jumps straight back to the point it was called from, and carries on with the next line.

Then the `scanf` calls run, the maximum is calculated and printed, and `printCopyright` is called a second time, which does the whole jump-run-return dance again. Finally `main` returns 0 and the program ends.

This raises a fair question: if `printCopyright` can be `void`, why is `main` not `void`? It never seems to use that 0. The answer is that `main` is not returning it to your program, it is returning it to the operating system, which checks it to find out whether the program completed successfully. Zero means all is well. If the program had fallen over halfway through, the operating system would have received something else. It is arguably an odd design – Java made the opposite choice – but it is the design we have.

One last small thing. You will sometimes see people write `void` in the argument list as well:

```
int main(void)
```

That just means "this function takes no arguments". It is not wrong. There is no need to be fancy about it.

Why bother with any of this?

We have written two functions now, and it is reasonable to ask what has actually been gained. There are three answers.

The first, and probably the biggest, is that you avoid writing the same code over and over. In real programs you constantly find yourself needing the same few lines again. If everything lives in `main`, the only way to do that is to copy and paste, and every copy is a place where a mistake can creep in, or where you fix a bug in one copy and not the other four. Write it once as a function, check it carefully, and then use it as often as you like. If I need the maximum of three numbers ten times, I do not want ten copies of that logic.

The second is abstraction. Once `maximum` is written and you are confident it works, you are allowed to forget how it works. You just use it. This is exactly the relationship you already have

with `printf`: you have no idea how it puts text on a screen and you have never needed to. Every function you write buys you another thing you no longer have to keep in your head.

The third is what people call divide and conquer. When you have a big problem, you do not solve all of it at once. You notice that solving it requires some smaller sub-problem to be solved first, so you write a function for that, check it works, and then carry on with the larger problem with one less thing to worry about. Programming a large system is mostly this, repeated.

A rough rule of thumb used in industry is that if a function is longer than the screen you are looking at, it is too long and wants breaking up. `main` may be an exception, but for everything else it is a decent guide. If you open a real C file, that is what you will find: a stack of prototypes at the top, and below them a collection of small functions that build up to whatever the program actually does.

There is also a good practical signal to watch for. If you catch yourself thinking "I am sure I have written something like this before", that is almost always a sign that the code in question wants to be a function.

Converting between types

There is one wrinkle you will run into as soon as you start passing real data into functions, so it is worth dealing with now.

Our `maximum` requires three integers. Suppose the values you have are not integers but floats:

```
int mval;
float a, b, c;

a = 10.0;
b = 12.0;
c = 11.0;

mval = maximum(a, b, c);
```

C will complain. The function wants `int` boxes and you are handing it `float` boxes, and C is not clever enough to just deal with it silently. You have to say explicitly that you know these are floats and you want them treated as integers. That is called casting, and it is written by putting the target type in parentheses in front of the value:

```
mval = maximum((int) a, (int) b, (int) c);
```

`(int) a` takes the float 10.0 and converts it to the integer 10.

The important thing to understand about casting downwards like this is that it throws information away, and it does so by truncating rather than rounding. If `a` held 12.3, then `(int) a` is 12: the `.3` is simply discarded. If it held 12.97345, `(int) a` is still 12. Nothing is rounded up, ever.

Casting the other way, from an integer to a float, is safe: there is no information to lose. It is going downwards that needs care. So cast deliberately, and be sure you are happy to lose whatever gets thrown away.

The maths library

`math.h` is a library where clever people have gone to considerable trouble to write a large number of very useful mathematical functions, so that you never have to.

Using it is the same as using `stdio.h`. Put this at the top of your file:

```
#include <math.h>
```

and then you can call anything in it:

```
double x;
x = sqrt(1000.0);
```

One thing to know before you start: the maths functions take `double` values and hand back `double` values. If what you have is an integer, it needs to become a double first.

The table below lists the ones you are most likely to want:

Function	Description	Example
`sqrt(x)`	square root of `x`	`sqrt(900.0)` is `30.0`
`exp(x)`	exponential function e^x	`exp(1.0)` is `2.718282`
`log(x)`	natural logarithm of `x` (base e)	`log(2.718282)` is `1.0`
`log10(x)`	logarithm of `x` (base 10)	`log10(100.0)` is `2.0`
`fabs(x)`	absolute value of `x`	`fabs(-5.0)` is `5.0`
`ceil(x)`	rounds `x` up	`ceil(9.2)` is `10.0`
`floor(x)`	rounds `x` down	`floor(9.2)` is `9.0`
`pow(x, y)`	`x` raised to the power `y`	`pow(2, 7)` is `128.0`
`fmod(x, y)`	remainder of `x/y` for floating point	`fmod(13.657, 2.333)` is `1.992`
`sin(x)`	sine of `x`	`sin(0.0)` is `0.0`
`cos(x)`	cosine of `x`	`cos(0.0)` is `1.0`
`tan(x)`	tangent of `x`	`tan(0.0)` is `0.0`

Two of these deserve a warning. `ceil` and `floor` are rounding, but always in a fixed direction rather than to the nearest: `ceil` goes up and `floor` goes down. With negative numbers this is less intuitive than it sounds, so it is worth checking rather than guessing. `ceil(-9.8)` is -9.0, because -9 is above -9.8, and `floor(-9.8)` is -10.0.

And `fmod` is the modulus operator for floating point numbers, filling the gap left by `%`, which only works on integers.

Finally, `sin`, `cos` and `tan` all work in radians, not degrees. If your angle is in degrees, convert it before you pass it in. This is a mistake everybody makes at least once, and the annoying thing about it is that the program runs perfectly happily and simply gives you the wrong answer.

Looking forward

You can now write functions of your own, call them, get values back from them, and use the enormous collection of functions that other people have already written for you. That is most of what functions are.

The next lecture continues with them. There are two things still missing. The first is how to split a program across more than one file, which is what header files are for; once you understand prototypes, header files are a short step. The second is more fundamental: at the moment, when you pass a variable to a function, its value is copied into a new box, and nothing the function does can affect the original. That is called passing by value, and it is the only thing we have done so far. There is another way, passing by reference, which lets a function reach back and change the caller's variables. It is how `scanf` manages to put a value into your variable, which is also why you have been writing that `&` in front of the variable name all this time without being told why.

Exercises

1. Write a function `square` that takes an integer and returns its square. Call it from `main` for the numbers 1 to 10 and print the results.

2. What does the following print? Work it out on paper before running it.

```
printf("%d\n", (int) 7.9);
printf("%d\n", (int) -7.9);
printf("%f\n", ceil(7.1));
printf("%f\n", floor(-7.1));
```

3. Write a `void` function `printLine` that prints a row of 40 dashes followed by a newline, and use it to separate sections of output in a program of your own.

4. Take the `maximum` program and delete the prototype at the top of the file, leaving everything else as it is. What does the compiler say? Now move the whole definition of `maximum` above `main` instead and try again.

5. Write a program that reads an angle in degrees and prints its sine, cosine and tangent. Remember that the maths library works in radians.

6. Trace through the following on paper, and say what is printed and why:

```
int change(int n)
{
    n = n + 1;
    return n;
}

int main()
{
    int a = 5;
    change(a);
    printf("%d\n", a);
    return 0;
}
```