10. Building your own library (and generating random numbers)

Herman Kamper

C programs are really all about functions. In the previous lecture we wrote our first ones, and saw that a function lets you define a new command and then use it without thinking again about how it works inside.

This lecture takes that one step further. Instead of keeping your functions in the same file as `main`, we are going to move them into a file of their own, so that you can pick them up and carry them into the next program you write. That is what a library is, and it is exactly what `stdio.h` has been doing for you all along. In the second half we look at one particular library function that is more interesting than it first appears: `rand`.

The program we are starting from

Here is the program from the previous lecture, extended slightly. It reads three numbers, then reports both the largest and the smallest:

```c
#include <stdio.h>

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

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

    printCopyright();

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

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

    // Determine the minimum
    printf("The minimum is: %d\n", minimum(num1, num2, num3));

    return 0;
}
```

The three functions themselves are further down the file, and there is nothing new in them: `maximum` is what we wrote before, `minimum` is the same thing with the comparisons turned round, and `printCopyright` is a `void` function that puts a line on the screen and returns nothing.

Quickly, to make sure the vocabulary is settled. A function is a block of code that defines your own instruction, in the same way that `printf` and `scanf` are instructions. When the program runs, C goes into `main`; when it meets `printCopyright()` it jumps to that function, runs the line inside, and jumps back. `printCopyright` is a `void` function, and we know that because it says `void`, which means it returns nothing. `maximum` is different: it returns an `int` and it takes three `int` values as its arguments. The three lines at the top of the file, the ones ending in semicolons, are the function prototypes, which tell C that these functions exist and will be defined somewhere further down.

Running it does what you would expect:

```
(c) Herman Kamper, 2026
Enter three integers:
5
1
4
The maximum is: 5
The minimum is: 1
```

Why you would want a library

Now imagine that `maximum`, `minimum` and `printCopyright` turn out to be functions you use again and again, across completely different projects. Every time you start something new, you want them available.

You could copy and paste them into each new program. But that is exactly the kind of duplication that functions were supposed to save you from, just moved up a level: now instead of five copies of the same lines inside one program, you have five copies of the same functions scattered across five programs, and when you find a bug you have to remember to fix all of them.

What you actually want is a file. Put the functions in it once, and when you start a new project, copy that one file across and you have them all.

This is not a trick I am inventing for the sake of the example. It is precisely what `stdio.h` is. Somebody wrote `printf` and `scanf`, packaged them up, and now nobody has to write them again. The mechanism that makes that possible is called a header file, and you have been using one since your first program. `stdio.h` is a header file that says which functions inside `stdio` you are allowed to use. We are now going to write our own.

The three files

The plan is to stop putting the whole program in one `main.c` and to break it into smaller files instead. There will be three, and it is worth seeing the shape of the whole thing before we build any of it.

The first file is the header, and we will call it `mylibrary.h`. You can call it whatever you like: `mymath.h`, `apples.h`, anything. This file contains the function prototypes, and nothing else.

The second is `mylibrary.c`, which contains the function definitions, the actual code. It starts with a line that pulls in the header:

```
#include "mylibrary.h"
```

What that line does is take everything in `mylibrary.h` and dump it into this file at that point, which means the prototypes are now available here.

The third is `main.c`, which is what you have always had: the `#include <stdio.h>`, the `int main()`, all of it. The only new thing is that it also includes the header:

```
#include "mylibrary.h"
```

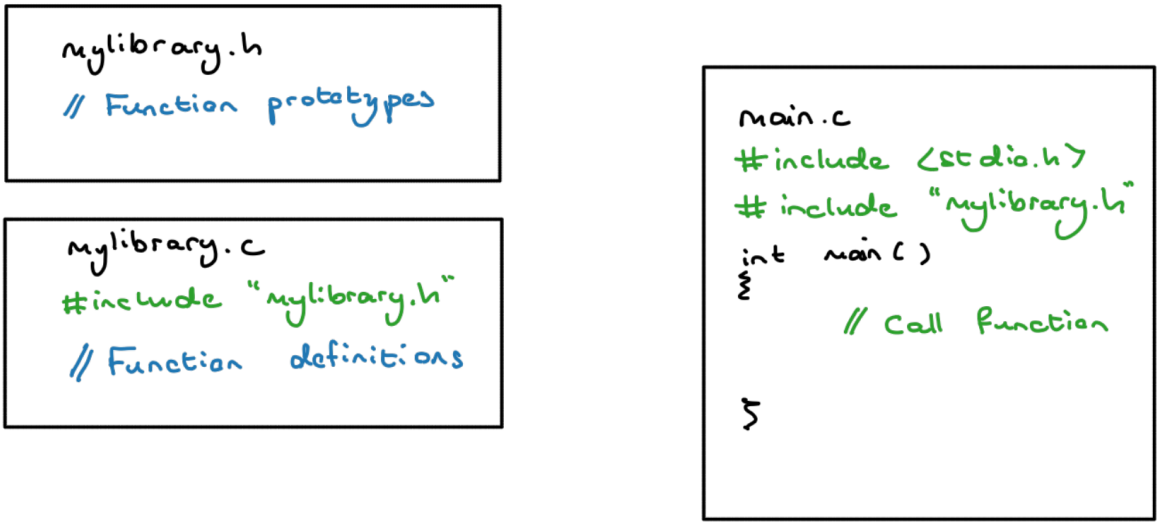

Figure 1: The three files that make up a program with its own library. The header holds the prototypes, the source file holds the definitions, and both the library and the main program include the header.

which tells the compiler that this file is going to use some functions declared in that header, somewhere inside `main`.

That is the whole idea. Three files, and once they exist, starting a new project that needs these functions means copying `mylibrary.h` and `mylibrary.c` into the new folder and adding those two lines.

The header file

Here is `mylibrary.h` in full:

```
#ifndef MYLIBRARY_H_INCLUDED
#define MYLIBRARY_H_INCLUDED

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

#endif // MYLIBRARY_H_INCLUDED
```

The three lines in the middle are the ones we care about. They are the same prototypes that used to sit at the top of `main.c`, moved here.

The lines around them are something your editor will usually write for you when you create a new header file, so for now you can treat them as boilerplate that comes with the territory. What they do, briefly, is stop the file being pulled in twice. The first time the header is included, `MYLIBRARY_H_INCLUDED` has not been defined, so the contents are used and that name gets defined. If something tries to include the same header again, the name is now defined and

everything between the `#ifndef` and the `#endif` is skipped. In a program with several files that include each other, this saves you from a pile of confusing errors about things being declared twice.

The source file

Then `mylibrary.c`, which is where the functions actually live:

```
#include "mylibrary.h"
#include <stdio.h>

int maximum(int x, int y, int z)
{
    int max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}


int minimum(int x, int y, int z)
{
    int min = x;
    if (y < min)
        min = y;
    if (z < min)
        min = z;
    return min;
}


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

Two lines at the top rather than one. The first pulls in our own header, which gives this file the prototypes. The second pulls in `stdio.h`, and that one is easy to forget: `printCopyright` calls `printf`, so this file needs `stdio.h` just as much as `main.c` does. Each source file has to include what it uses. It does not inherit anything from the file that will eventually be linked alongside it.

The main program

And finally `main.c`, which is the program we started with, minus the prototypes and minus the function definitions:

```c
#include <stdio.h>

#include "mylibrary.h"


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

    printCopyright();

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

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

    // Determine the minimum
    printf("The minimum is: %d\n", minimum(num1, num2, num3));

    return 0;
}
```

Build it and run it, and it behaves exactly as it did before. Nothing about what the program does has changed. What has changed is that `maximum`, `minimum` and `printCopyright` now live somewhere you can reuse them.

One practical note. If you are using an IDE, creating the files through the interface is not quite the same as making them in a folder: the project has to know that `mylibrary.c` is one of its source files, otherwise it will never get compiled. When you create the file there is usually a checkbox for adding it to the build, and for an existing file you can right-click the project and add it. If you copy the two files into a new project later, remember to add `mylibrary.c` to that project as well.

Angle brackets or quotation marks?

You will have noticed that the two kinds of include look different:

```
#include <stdio.h>
#include "mylibrary.h"
```

The distinction is about where the compiler goes looking. Angle brackets mean this is part of the standard C infrastructure, so go and find it wherever the system keeps such things. Quotation marks mean this is something I wrote myself, so look here, in my project.

That is the whole rule. `stdio.h` is obviously not sitting in your project folder, and `mylibrary.h` obviously is.

How it all gets built

It is worth knowing what happens when you press build, because now there is more than one `.c` file and something has to bring them back together.

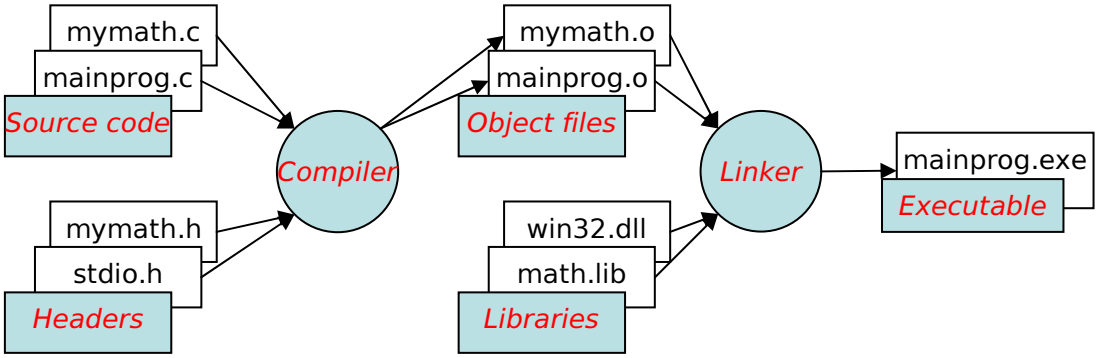

Figure 2: Source files are compiled individually into object files, and the linker then joins those together with the pre-built standard libraries to produce the executable.

The compiler works through your `.c` files one at a time. For each one it produces a chunk of machine code, called an object file. The headers come into this because the compiler needs to know the shape of any function that is called but not defined in the file it is currently looking at, and the header is what tells it.

At this point you have several separate pieces of machine code that know nothing about each other. The linker is the step that takes all of them, along with the already-compiled standard libraries, and glues them into a single executable that you can run.

This also explains why a header on its own is not enough. The header tells the compiler that `maximum` exists and what it looks like, which is enough to compile `main.c` without complaint. But if `mylibrary.c` never gets compiled, there is no machine code for `maximum` anywhere, and it is the linker that will complain about it rather than the compiler. An error that mentions an undefined reference, at the very end of a build that seemed to be going fine, usually means exactly this: a file that should be part of the project is not.

What else comes with C

The standard library is large, and it is worth at least knowing what is in it so that you do not write something that already exists. Every one of these is included the same way, with angle

brackets.

Header	What is in it
`<assert.h>`	Macros and information for adding diagnostics that aid debugging
`<ctype.h>`	Functions that test characters for certain properties, and convert between upper and lower case
`<errno.h>`	Macros useful for reporting error conditions
`<float.h>`	The floating point size limits of the system
`<limits.h>`	The integer size limits of the system
`<locale.h>`	Information letting a program adapt to local conventions for dates, times and currency
`<math.h>`	The maths library functions
`<setjmp.h>`	Functions that allow bypassing of the usual function call and return sequence
`<signal.h>`	Functions and macros to handle conditions arising during execution
`<stdarg.h>`	Macros for handling a list of arguments whose number and types are unknown
`<stddef.h>`	Common definitions of types used by C for certain calculations
`<stdio.h>`	The standard input and output functions
`<stdlib.h>`	Number and text conversions, memory allocation, random numbers and other utilities
`<string.h>`	String processing functions
`<time.h>`	Functions and types for manipulating the time and date

Being honest about it, most of that list is not something you will reach for. In practice `stdio.h` is in everything, `math.h` comes up constantly, `stdlib.h` is next in this lecture, `time.h` appears alongside it, and `string.h` becomes important later in the course when we deal with strings, which are a genuine pain in C. Several of the others I have never had occasion to use. The point of the table is recognition rather than memorisation: when you see one of these at the top of somebody else's file, you will know roughly what it is doing there.

Random numbers

Now for something different, though still about functions. There is one in `stdlib.h` called `rand`, and it does something no function we have written could do: it produces a number you cannot predict.

```
#include <stdlib.h>

int i;

i = rand();
```

Calling `rand` gives you a whole number somewhere between 0 and a constant called `RAND_MAX`. How big `RAND_MAX` is depends on your system. The C standard only promises it will be at least 32767, and you will see that figure quoted often, but on essentially any modern machine it is much larger: 2147483647, which is $2^{31} - 1$. If you want to know what it is on the machine in front of you, print it.

Either way, what you get out of `rand` is an unpredictable number from an enormous range, and the immediate question is how to turn that into a number in a range you actually want.

Getting a number in the range you want

Suppose you have built a robot. It can do five things: move forward, turn left, turn right, reverse, or stand still. Number them 0 through 4. You want the robot to pick an action at random, so you need a random number between 0 and 4 inclusive.

The tool for this is the modulus operator, `%`, which you have met before. It gives you the remainder after division. So `7 % 5` means divide 7 by 5 and tell me what is left over. Seven divided by five is one remainder two, and `%` gives you the two:

$$\frac{7}{5} = \frac{5}{5} + \frac{2}{5} = 1 + \frac{2}{5}$$

The whole part is 1, the bit left over is 2, and `7 % 5` is 2. Similarly `9 % 5` is 4, because nine is five plus four left over.

Here is the useful thing. Look at what happens for a spread of different numbers, all divided by 5:

$$\begin{aligned} 7 \,\%\, 5 &= 2 \\ 105 \,\%\, 5 &= 0 \\ 108 \,\%\, 5 &= 3 \\ 1109 \,\%\, 5 &= 4 \\ 1110 \,\%\, 5 &= 0 \\ 1114 \,\%\, 5 &= 4 \\ 32764 \,\%\, 5 &= 4 \end{aligned}$$

Every single answer is between 0 and 4. It has to be: if the remainder after dividing by 5 were 5 or more, you could have divided one more time. So no matter how enormous the number `rand` hands back, taking it modulo 5 squeezes it into exactly the range we want:

```
rand() % 5      // between 0 and 4
```

Note that it is `% 5` and not `% 4`. This catches almost everybody. You want five possible outcomes, so you divide by 5, and the remainders you can get are 0, 1, 2, 3 and 4. Dividing by 4 would give you four outcomes, 0 through 3, and your robot would never reverse.

Shifting the range is easier still. If `rand() % 5` gives you 0 to 4, then adding one to it gives you 1 to 5:

```
rand() % 5 + 1    // between 1 and 5
```

So the general recipe is that to get a random number from a to b inclusive, you take the size of the range, which is $b - a + 1$, use that as the modulus, and add a:

```
rand() % (b - a + 1) + a
```

For a number from 9 to 25, for instance, that is 17 possible values starting at 9, so `rand() % 17 + 9`. For an ordinary six-sided dice it is `rand() % 6 + 1`.

A note on fairness (advanced)

There is a subtlety hiding here, and it is a good question to ask: does this actually give every outcome an equal chance?

Strictly, not quite. Suppose `rand` could only return the values 0 through 9, and you took the result modulo 6. Then 0 and 6 both give 0, 1 and 7 both give 1, 2 and 8 give 2, 3 and 9 give 3, but 4 and 5 can only be reached one way each. Four of the six outcomes are twice as likely as the other two. That is a badly biased dice.

What rescues us in practice is the sheer size of `RAND_MAX`. When there are over two billion possible values being folded into five buckets, the buckets differ by at most one value out of four hundred million, and no experiment you run will ever detect the difference. The bias is real but it is not worth thinking about at this scale. It is only when the range you are folding from is comparable to the range you are folding into that it matters.

The same numbers every time

Write the loop, run it, and you get something like this:

```
A random number: 4
A random number: 2
A random number: 3
A random number: 1
A random number: 4
A random number: 1
```

Run it again, and you get 4, 2, 3, 1, 4, 1. Run it a third time, and you get 4, 2, 3, 1, 4, 1. The same supposedly random numbers, in the same order, every single time.

The reason is that generating genuinely random numbers is impossible for a computer. What `rand` actually runs is a complicated but entirely deterministic algorithm, and like any algorithm, the same input gives the same output. The input in this case is a starting value called the seed. Give it the same seed and you get the same sequence, and if you never set a seed at all, C quietly uses the same default one on every run.

You set it with `srand`, which means seed the random number generator:

```
srand(42);
```

Do that and you will get a different sequence from the default one, but it will still be the same sequence on every run, because 42 is the same number every time.

What you want is a seed that differs each time the program starts, and there is a neat trick for it. The `time` function, from `time.h`, gives you the current clock reading, and the clock is never the same twice:

```
#include <stdlib.h>
#include <time.h>

srand(time(NULL));
```

Now every run starts from a different point in the sequence, and the numbers really do differ:

```
A random number: 5
A random number: 1
A random number: 4
A random number: 2
A random number: 4
A random number: 1
```

and running it again:

```
A random number: 2
A random number: 2
A random number: 4
A random number: 4
A random number: 1
A random number: 4
```

Three things to get right here. `srand` is called once, at the start of the program, not before every call to `rand`. It does not generate anything itself: it only sets the starting point, and `rand` is still the function that hands you a number. And the `NULL` being passed to `time` is, for now, just something you write. It is a little like `void` in that it stands for nothing at all, and we will come back to what it really means later in the course.

Putting it together, here is a program that prints six random numbers between 1 and 5:

```c
/*
 * Random number generation.
 * Author: Herman Kamper
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int main()
{
    int i;
    int count;

    srand(time(NULL));

    for (count = 1; count <= 6; count++)
    {
        i = rand() % 5 + 1;
        printf("A random number: %d\n", i);
    }

    return 0;
}
```

One warning about the sample outputs above: the actual numbers depend on which system you are on, because different C implementations use different algorithms inside `rand`. The unseeded sequence being identical on every run is universal. Which particular numbers it is are not.

Why any of this matters

It would be reasonable to think that random numbers are a novelty, useful for dice games and not much else. They are not. Randomness is one of the more powerful ideas in computing, and it sits underneath a lot of what is interesting at the moment.

Take a robot that has to learn to get around a world it knows nothing about. It can see where it is, it can move in four directions, and somewhere out there is a reward worth having and a pit it should avoid.

The robot does not begin with a map or a set of instructions. So it does things at random, and it notices that some of the random things it did worked out better than others, and it starts favouring those. Sometimes a random choice sends it into the pit, and it learns from that too. Doing something at random in order to find out what happens is one of the central ideas in modern artificial intelligence, and it is not a metaphor: there really is a call to a random number generator in there, deciding what to try next.

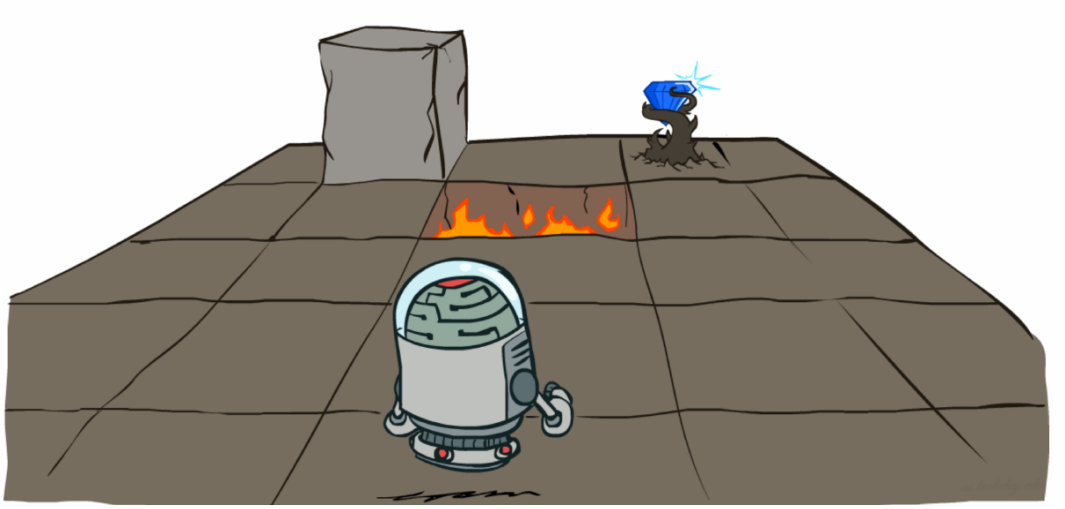

Figure 3: A robot that has to find its way to the reward without falling into the pit. Not knowing anything about the world to begin with, it has to try things at random to find out what works. Image from UC Berkeley CS188.

You have probably seen a version of this yourself. When you press regenerate on a large language model and get a different answer to the same question, nothing about the model has changed. It is choosing its next word from a set of possibilities using a random number, so a different starting point produces a different path through the same set of choices.

There is a much smaller reason to care as well, which is that random numbers make programs fun to write. The first program I ever wrote was in a language called BASIC, and it was a guessing game: the computer picks a number, you guess, and it tells you whether the real number is higher or lower, over and over until you get it. Everything needed to write that is now in your hands. `rand` picks the number, a `do...while` keeps asking, and an `if` tells the player higher or lower. If you want a project to test yourself on, that is a good one.

Looking forward

You can now split a program across several files, write a header to go with each one, understand what the compiler and the linker are each doing with them, and use the standard library rather than reinventing it. That is the last structural thing about functions.

The next lecture puts random numbers to work properly, building a complete game of chance rather than a program that prints six numbers. Along the way it deals with a question we have brushed past several times: exactly where does a variable live, who can see it, and how long does it last? Those are the storage class and scope rules, and they are what makes the difference between a variable that vanishes when a function returns and one that survives until the program ends.

Exercises

1. Take a program you have already written that has more than one function in it, and split it into a header, a source file and a main program. Check that it still runs.
2. Add a `range` function to `mylibrary`, which takes three integers and returns the difference between the largest and the smallest. Write the prototype in the header and the definition in the source file, and use `maximum` and `minimum` inside it rather than starting from scratch.
3. Deliberately break the build: remove `mylibrary.c` from the project while leaving `#include "mylibrary.h"` in place. Look carefully at the error you get, and note that it comes from the linker rather than the compiler. This is worth seeing once so that you recognise it later.
4. Write down the expression that produces a random integer in each of these ranges: 0 to 9, 1 to 6, 1 to 100, 9 to 25, and -5 to 5.
5. Write a program that rolls a six-sided dice 60 times and counts how many times each face comes up, using an array or six separate counters. Are the counts roughly equal? Run it a few times.
6. Run a program using `rand` without calling `srand`, three times in a row, and confirm you get identical output. Then add `srand(time(NULL))` and confirm you do not. Then try `srand(42)` and predict what will happen before you run it.
7. Write the guessing game. The program picks a random number from 1 to 100 and repeatedly asks the user to guess, replying "higher" or "lower" until the guess is right, then reports how many guesses it took.