Introduction to Computing
IV - Functions and Structures in C

Jonathan Mascie-Taylor
(Slides originally by Quentin CAUDRON)

Centre for Complexity Science,
University of Warwick

Outline

Functions : a Definition

Functions are short sets of instructions that do something and then generally return some output. You've used some before : `printf` and `scanf` are functions. You can write your own :

Functions : a Definition

Functions are short sets of instructions that do something and then generally return some output. You've used some before : `printf` and `scanf` are functions. You can write your own :

```
1  int biggest(int x, int y)
2  {
3     if(x > y)
4     {
5        return x;
6     }
7
8     else
9     {
10       return y;
11    }
12 }
```

Function Types

Functions always have a *type* in C. Just like variables can be `int`, `double` or other types, because functions return some output, they are declared with a type. Those that simply do something without returning anything also have a type, called `void`.

Function Types

Functions always have a *type* in C. Just like variables can be `int`, `double` or other types, because functions return some output, they are declared with a type. Those that simply do something without returning anything also have a type, called `void`.

```
int biggest(int x, int y)
```

This function is of type `int` - we expect it to do something, and then return an integer. It also takes two `int` arguments, `x` and `y`. It will do something with those variables, and then give us something back. We can *call* the function like this :

Function Types

Functions always have a *type* in C. Just like variables can be `int`, `double` or other types, because functions return some output, they are declared with a type. Those that simply do something without returning anything also have a type, called `void`.

```
int biggest(int x, int y)
```

This function is of type `int` - we expect it to do something, and then return an integer. It also takes two `int` arguments, `x` and `y`. It will do something with those variables, and then give us something back. We can *call* the function like this :

```
int hello = 5;
int goodbye = 8;
int ourResult;
ourResult = biggest(hello, goodbye);
```

Returning a Value

A function of a certain type always returns that type of output. When we calculated π, we noticed that there was no version (*overload*) of the `pow` function that takes in integers. Let's write our own.

Returning a Value

A function of a certain type always returns that type of output. When we calculated π, we noticed that there was no version (*overload*) of the `pow` function that takes in integers. Let's write our own.

```
1  int pow(int base, int exponent)
2  {
3    int i;
4    int result = 1;
5
6    for(i = 0; i < exponent; i++)
7    {
8      result *= base;
9    }
10
11   return result;
12 }
```

Declaring Functions

Functions are best *declared* using a *prototype* under your `#includes` and then *defined* after `main()`.

Declaring Functions

Functions are best *declared* using a *prototype* under your `#includes` and then *defined* after `main()`.

```
int myFunction(int a, int b); //Prototype
```

Declaring Functions

Functions are best *declared* using a *prototype* under your `#includes` and then *defined* after `main()`.

```
int myFunction(int a, int b); //Prototype
```

Alternatively, you can just put the full function definition instead of the prototype, above `main()`, and nothing after `main()`.

Declaring Functions

Functions are best *declared* using a *prototype* under your `#include`s and then *defined* after `main()`.

```
int myFunction(int a, int b); //Prototype
```

Alternatively, you can just put the full function definition instead of the prototype, above `main()`, and nothing after `main()`.

Finally, functions can be declared in a separate file, and that file can then be included, but note the slightly different syntax :

```
#include "myfile.h"
```

Declaring Functions

Functions are best *declared* using a *prototype* under your `#include`s and then *defined* after `main()`.

```
int myFunction(int a, int b); //Prototype
```

Alternatively, you can just put the full function definition instead of the prototype, above `main()`, and nothing after `main()`.

Finally, functions can be declared in a separate file, and that file can then be included, but note the slightly different syntax :

```
#include "myfile.h"
```

A requirement is that your function be *declared* **before** you call it anywhere, and *defined* **somewhere**.

Declaring Functions in a .c File

```c
#include <stdio.h>

double addTwo(double); // Prototype

int main()
{
    double x = 0.63;
    printf("%lf\n", addTwo(x));
    return 0;
}

// Definition
double addTwo(double aNumber)
{
    return aNumber + 2;
}
```

Structures

A structure is a type of variable that you define yourself, and that contains any number of other variables. They provide a way to store a group of variables (of potentially different types) under the same name.

Structures

A structure is a type of variable that you define yourself, and that contains any number of other variables. They provide a way to store a group of variables (of potentially different types) under the same name.

```
1 struct Location
2 {
3     double latitude;
4     double longitude;
5     int elevation;
6 };
```

Structures

A structure is a type of variable that you define yourself, and that contains any number of other variables. They provide a way to store a group of variables (of potentially different types) under the same name.

```
1 struct Location
2 {
3     double latitude;
4     double longitude;
5     int elevation;
6 };
```

To create a variable of type Location, you declare one like this :

```
1 struct Location myLoc;
```

Using Structures

```
1  struct Location myLoc;
2
3  myLoc.latitude = 52.38408;
4  myLoc.longitude = -1.56047;
5  myLoc.elevation = 81; // meters
6
7  printf("At %lf degrees North and %lf"
8  "degrees West, \n the elevation above"
9  "sea level is %d meters or %f feet.",
10 myLoc.latitude,
11 myLoc.longitude,
12 myLoc.elevation,
13 myLoc.elevation * 3.2808);
```

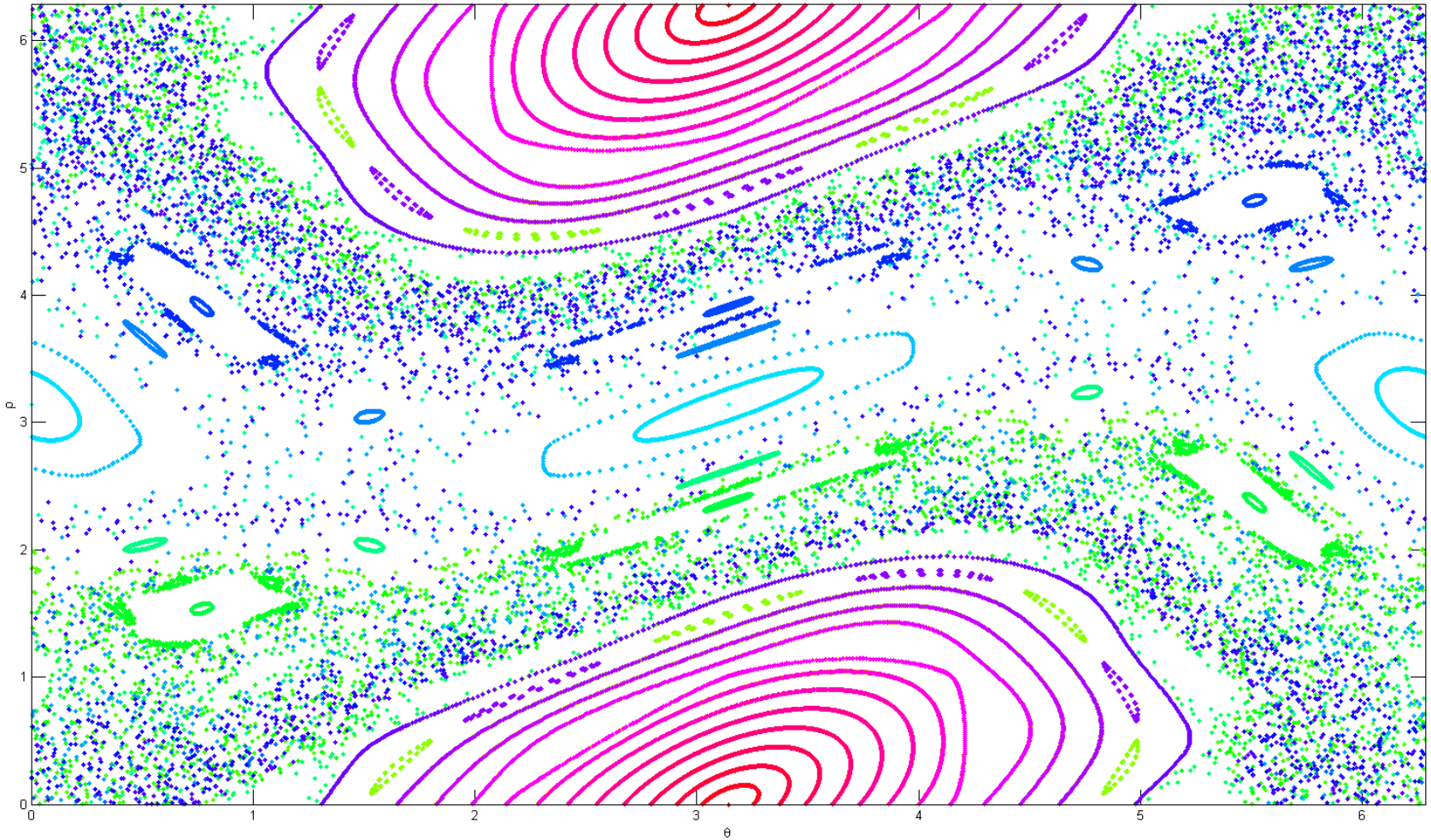
Note the way we use quotes to break up a long line.

Exercise

Write some code that calculates the area of a box.

- Define your box as a structure which consists of three doubles - the width, height and depth.
- Create a function that takes one argument - the box structure.
- Use the `return` keyword inside the function to return the area.

The Standard Map



The Standard Map System

The Standard Map is a chaotic map that describes the motion of a stick, pinned at one tip, and periodically kicked at the other tip. The 2D system describes its angular momentum (ρ) and angle (θ):

$$\rho_{n+1} = \rho_n + K \sin \theta_n \in [0, 2\pi]$$
$$\theta_{n+1} = \theta_n + \rho_{n+1} \in [0, 2\pi]$$

The Standard Map System

The Standard Map is a chaotic map that describes the motion of a stick, pinned at one tip, and periodically kicked at the other tip. The 2D system describes its angular momentum (ρ) and angle (θ):

$$\rho_{n+1} = \rho_n + K \sin \theta_n \in [0, 2\pi]$$
$$\theta_{n+1} = \theta_n + \rho_{n+1} \in [0, 2\pi]$$

Trajectories in phase space can be calculated easily in **C**. However, in order to capture the true behaviour of the system, we need to constrain it to the torus - that is, give it periodic boundaries in both dimensions.

Exercise : A Modulo Function

Write a function of a variable `x` that returns `x modulo` 2π.

Exercise : A Modulo Function

Write a function of a variable `x` that returns `x modulo` 2π.

- Write a function of type `double`
- Define it to take one argument of type `double`
- Use a while loop to subtract 2π repeatedly until it is less than 2π
- Use another while loop to add 2π until it is positive
- Return the variable you modified
- Use `#include <math.h>` and `M_PI` for π.

Test your function by passing it different values.

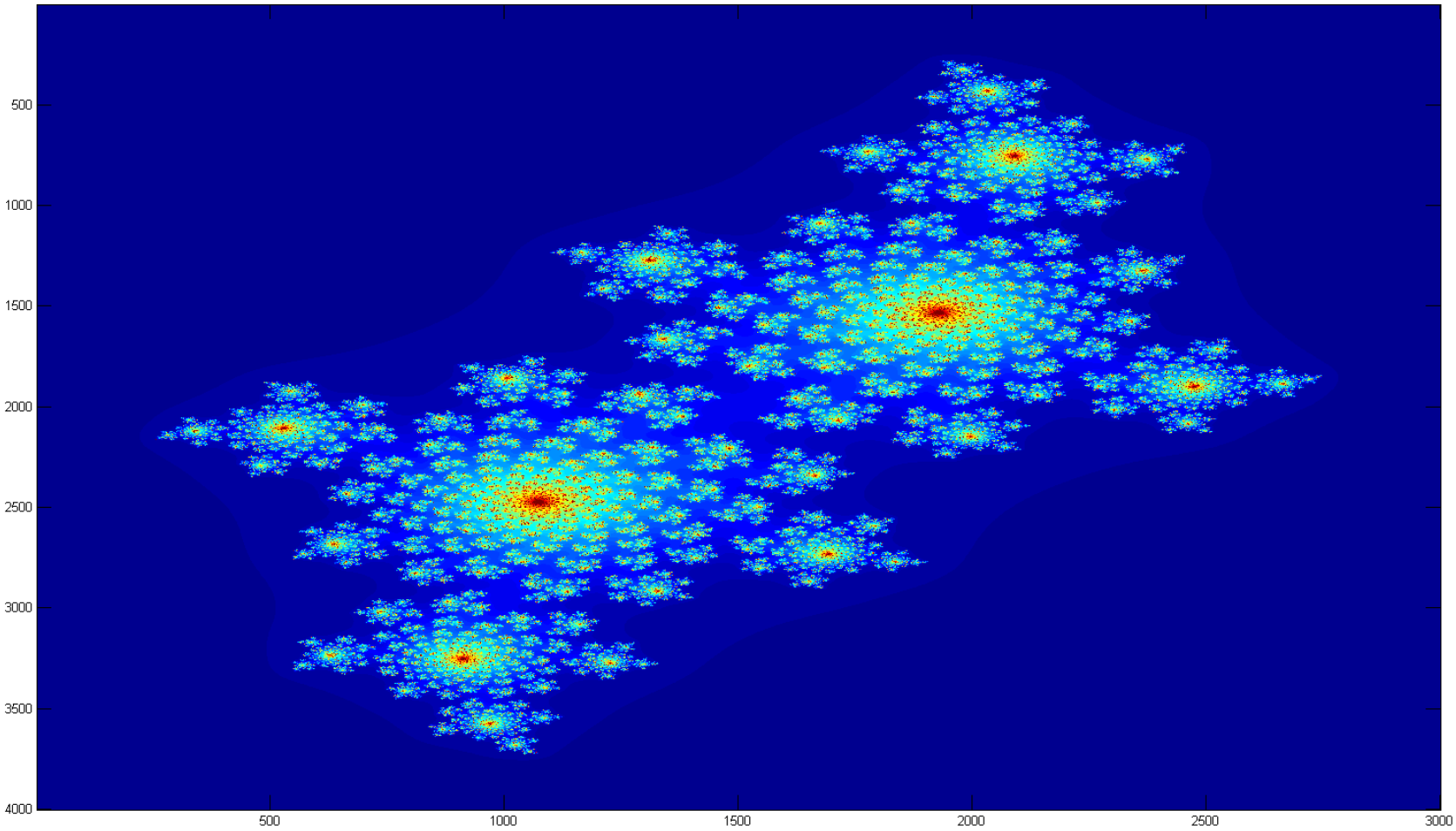
Can you think of a better way to implement this function?

Coding the Standard Map

$$\rho_{n+1} = \rho_n + K \sin\theta_n \in [0, 2\pi)$$
$$\theta_{n+1} = \theta_n + \rho_{n+1} \in [0, 2\pi)$$

- Declare some variables for momentum, angle and kick strength
- Open a file to write results to
- Write a `for` loop to generate 25 trajectories :
  - Set your momentum to some value $\in (0, 2\pi)$, either random or otherwise; set your angle to π
  - Generate a thousand steps on each trajectory using another `for` loop, using your modulo function to constrain the values each time
  - Write momentum and angle to file each step, separated by a space
- Close your file
- Plot in MATLAB using `plot( myData(:, 1), myData(:,2) );`

The Julia Set



The Julia Set

The Julia Set is connected to the Mandelbrot set :

$$z_{n+1} = z_n^2 + c$$

Whereas in the Mandelbrot set, $z_0 = 0$ and c is a point on the plain, the Julia set uses z_0 as the starting point on the plane, and c is a constant. There are therefore an infinite number of Julia sets, one for each c.

The Julia Set

The Julia Set is connected to the Mandelbrot set :

$$z_{n+1} = z_n^2 + c$$

Whereas in the Mandelbrot set, $z_0 = 0$ and c is a point on the plain, the Julia set uses z_0 as the starting point on the plane, and c is a constant. There are therefore an infinite number of Julia sets, one for each c.

This all occurs in the complex plane, but we're going to run the computations with real values. Given that $z = a + ib$, then $z^2 = a^2 + 2\,abi - b^2$, with real components $\mathrm{Re}(z^2) = a^2 - b^2$ and imaginary components $\mathrm{Im}(z^2) = 2\,abi$.

Exercise : Updating Values

Write two functions, to calculate the real and the imaginary parts of a point in the complex plane, according to the rules of the Julia set.

Exercise : Updating Values

Write two functions, to calculate the real and the imaginary parts of a point in the complex plane, according to the rules of the Julia set.

- Write a function of type `double` to calculate the real component of a point
- Define it to take three arguments of type `double`, for the real and imaginary values of the point as well as the real part of c
- Return the real component according to our arithmetic
- Write a similar function for the imaginary component, passing it the real and imaginary values of the point as well as the imaginary part of c

Test your function by passing it different values. Try $z = \pi + \frac{i}{2}$ and $c = -0.7 + 0.27i$. You should get $z^2 + c = 8.92 + 3.41i$.

Coding the Julia Set

- Declare some parameters : resolution (500), $\mathrm{Re}(c) = -0.7$, $\mathrm{Im}(c) = 0.27015$
- Declare your simulation variables for real and imaginary parts of a point, as well as a temporary variable to enable you to simultaneously update values
- Open a file to write your results to
- Iterate over your resolution in 2D (`for` loop on `i` and `j`)
- Set your real and imaginary variables to some point in space
- Use a while loop over two conditions (use `&&` for *logical and*) to run while the magnitude of the point is less than 1000 and you apply less than 255 applications of the Julia function :
  - Set the temporary variable to the current value of the real part of your point
  - Calculate a new value for the real part, using your function
  - Calculate a new value for the imaginary part, using your function, and passing it the *temporary* value of the real part (before its update)
  - Increment a counter for the number of applications
- Print to file the number of applications of the Julia function
- Reset your applications counter
- Between the loops in `i` and `j`, remember to print a new line to file
- Close your file

main() as a Function

```
1 #include <stdio.h>
2
3 int main(int argc, char * argv[])
4 {
5   printf("%s, %s", argv[1], argv[2]);
6   return 0;
7 }
```

`main()` as a Function

```
1 #include <stdio.h>
2
3 int main(int argc, char * argv[])
4 {
5   printf("%s, %s", argv[1], argv[2]);
6   return 0;
7 }
```

Remember that in **C**, arrays index from 0. Notice that `argv[]` is a character array, and that we are printing `argv[1]` and `argv[2]`. This is because `argv[0]` is always reserved for the name of the program itself.

`main()` as a Function

```c
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("%s, %s", argv[1], argv[2]);
  return 0;
}
```

Remember that in **C**, arrays index from 0. Notice that `argv[]` is a character array, and that we are printing `argv[1]` and `argv[2]`. This is because `argv[0]` is always reserved for the name of the program itself.

The array is of type `char *`, and so, we pass `%s` to `printf`. This has a downside - we can't naturally use the values as numerical values.

More Flexibility in Parameters

Modify your code so that `main(int argc, char * argv[])` takes arguments.

- Standard Map : change your value of K to
  `double K = atof(argv[1]);`
- Julia Set : change your value of $\mathrm{Im}(c)$ to
  `double cIm = atof(argv[1]);`

Because the argument is a string, we need to use `atof(myString)` (alphanumeric to float) to convert user input into something we can store in a float or double.

Calling the Program with Arguments

Windows Users :

- Open a command prompt (Start → "cmd")
- Browse to your directory (using `cd` and `dir`)
- `myprogram.exe 1.265`

Mac / Linux Users :

- Open a terminal
- Browse to your directory (using `cd` and `ls`)
- `./myprogram 0.28`

Recursion

We can also create functions which repeatedly call themselves. This is called recursion.

An example of this is Euclid's algorithm for the greatest common divisor of two numbers. Eculid's Algorithm is defined as

- $\gcd(a, 0) = a$
- $\gcd(a, b) = \gcd(b, a \bmod b)$.

Exercise: Code up a version of Euclid's algorithm that works from the command line. (For example, `gcd.exe 10 5` should print 5.)