



PRIFYSGOL  
**BANGOR**  
UNIVERSITY

# Imperative Programming in C

---

Laboratory 5: Functions

*Dr. Llyr ap Cenydd*

# Functions

By now you should know how to create variables, conditional statements and loops in C. The next tool we are going to look at is functions. You should already have some idea of how a function looks like in C because you've been creating them every lab. For example here:

```
int main(int argc, char *argv[])
{
    exercise1();
    system("PAUSE");
    return 0;
}
```

we are creating the main function, which is always the first function that runs when we execute our program. Similarly, for each exercise you will have created a new function:

```
exercise1()
{
    //code
}
```

And invoked the function from the main method by typing its name (exercise1(); in this case).

When we create a function, we can define which parameters it accepts as input, and what it returns after it has finished running. If we do not specify any parameters and return types, the compiler will automatically take this as void (which means 'nothing'). For example creating the exercise1 function like above is the equivalent of writing:

```
void exercise1(void)
```

In other words – *“This is a function named exercise1 that accepts **no parameters** and sends **no information** back after it has finished running”*.

You'll notice that the main method has two parameters – the integer argc and a character array argv. This allows us to pass information into the program when we execute it via arguments – for example from the command line.

The main method has a return type of int, and by default it returns the value of 0 on the last line. For main(...), this is so that we can optionally send messages back to whatever invoked the program, usually in the form of an error id. Returning 0 means “program has finished running, no errors to report”.

# Declaring Functions

Let's take a look at an example function:

```
int mult (int x, int y)
{
    return x * y;
}
```

This function simply takes two integers as input (x and y) and returns another integer, the calculation of  $x * y$ .

To call this function, we can write something like this:

```
int main()
{
    int a;
    int b;

    printf( "Please enter two numbers: " );
    scanf( "%d", &a );
    scanf( "%d", &b );
    printf( "The product of the two numbers is %d\n", mult( a, b ) );
    getchar();
}
```

Notice that we are passing `mult(a,b)` in as a parameter into another function here (`printf`). It is very common to 'nest' functions in this way. Alternatively we could have calculated the product and then passed it into the print statement:

```
int result = mult(x,y);
printf( "The product of the two numbers is %d\n", result );
```

There are two ways to pass parameters to a function:

**Pass by Value:** used when we don't want to change the value of the passed parameter. When parameters are passed by value the function creates a copy of the passed variables and does the required processing on these copies.

**Pass by Reference:** used when we want a function to alter the actual variables being passed in, so that the changes are visible to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly on the variables. We will look at pass by reference next week and in future labs when we look at pointers.

## Function Prototypes

One important aspect of functions in C is that we usually need to declare a **function prototype**:

```
int mult ( int x, int y );
```

These are also known as **function declarations** or **function signatures**. A prototype specifies what a function is called, what **arguments** it takes, and what it will **return**.

A function prototype does not have a body, and are usually declared at the top of a C source file, or in a separate header file (.h). In actual fact, when we import libraries we are importing header files that contain function prototypes.

The function prototype above essentially lets the compiler know that somewhere in the code there exists a function called 'mult' that accepts **two integers** as **parameters** (called x and y), and will **return an integer** when it's finished running. Where the definition of the function (i.e. the actual code) is found isn't important, just that the compiler knows it exists. The compiler will read through the source code sequentially – that is, in order from top to bottom (and usually across multiple files).

Without declaring a function in this way, you can get into a situation where a function is requesting another function to run which isn't known to the compiler yet.

Take a look at this example code:

```
int mult ( int x, int y );

int main()
{
    printf( "The product is %d\n", mult( 5, 10 ) );
    getchar();
}

int mult (int x, int y)
{
    return x * y;
}
```

Here the function `mult` is defined below the `main` method. Because its prototype is above `main`, the compiler still recognizes it as being declared, and so the compiler will not give an error about `mult` being undeclared. As long as the prototype is present, a function can be used even if there is no definition. However, the code cannot be run without a definition even though it will compile.

For small programs like this and most exercises in this module, it is enough to declare the functions inside the source code. However, in larger programs these function declarations are placed in header files. Not only does this accommodate splitting your code across multiple source files, but you can also easily import header files into other projects and make full use of the functions they declare.

# Exercise 1 – Mass Converter

Write a program that converts between different masses:

- Convert ounces to grams
- Convert grams to ounces
- Convert pounds to stones
- Convert stones to pounds

Your program should consists of six functions, the usual `exercise1()` function and five that follow the following prototypes:

```
void printOptions();
```

```
float ouncesToGrams (float ounces);
```

```
float gramsToOunces(float grams);
```

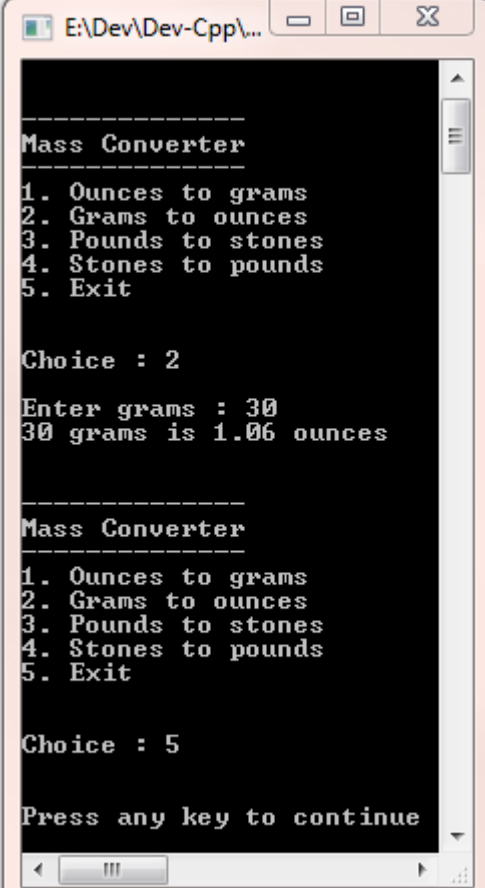
```
float poundsToStones(float pounds);
```

```
float stonesToPounds(float stones);
```

Your program should keep on asking the user for a choice until they pick the option to exit.

## Hints

- Google “grams to ounces” etc. to find the conversion formulae
- Get the menu system working first (see last week’s lab)
- Get one conversion function to work before adding the other three
- You can use the command **`sleep(value)`** to ask the program to sleep for a certain amount of time (in milliseconds, so 1000 = 1 second).
- The function `sleep` can be found in `<time.h>`



```
E:\Dev\Dev-Cpp\...
-----
Mass Converter
-----
1. Ounces to grams
2. Grams to ounces
3. Pounds to stones
4. Stones to pounds
5. Exit

Choice : 2

Enter grams : 30
30 grams is 1.06 ounces

-----
Mass Converter
-----
1. Ounces to grams
2. Grams to ounces
3. Pounds to stones
4. Stones to pounds
5. Exit

Choice : 5

Press any key to continue
```

## Exercise 2 – Half Pyramid

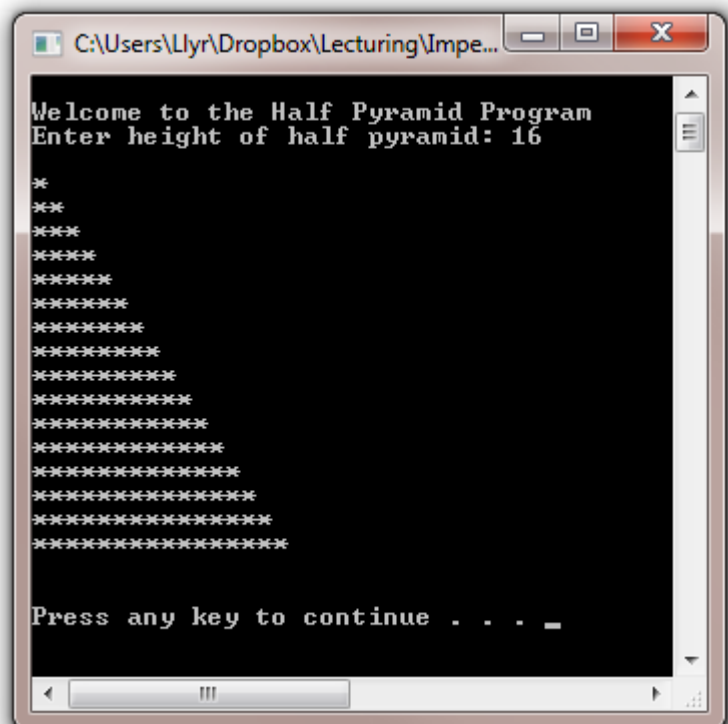
Write a program that asks the user for a height value, and then prints a half pyramid of that height. The code for printing the shape should be its own function, accepting a single parameter as input. The half pyramid should be made of \* characters. The deceleration of the function should look like this:

```
void printHalfPyramid(int height);
```

And the output should look something like this:

## Hints

- You will need two for-loops
- You might find it easier to create a cube first, then figure out how to turn into a triangle



# Array of Strings

In C, a string is represented as an array of characters. In order to represent an array of strings we have to make array-of-an-array of characters. There are two main ways of doing this:

## Using two dimensional arrays:

```
char arr_a[10][50]; // 2D array for storing 10 strings each of length up to 49 characters  
  
arr_a[0] = "abc";  
arr_a[1] = "def";  
  
//etc
```

We can also set the strings during declaration, in which case we don't need to specify the length of characters.

```
char arr_a[4][] = { "abc" , "def" , "ghi" , "jkl" };
```

*Note the use curly braces.*

## An array of character pointers:

```
char *arr_b[10]; // array of 10 character pointers - we will have to allocate memory  
dynamically for each string.  
arr_b[1] = malloc(50 * sizeof(char)); //This allocates a memory of 50 characters. You need to  
allocate memory for each element of the array
```

However just like with two dimensional arrays, if we are using string literals defined at declaration the compiler will figure out how much memory to allocate automatically:

```
char *arr_b[] = { "abc" , "def" , "ghi" , "jkl" };
```

We will look at malloc and dynamic memory allocation in more detail in later labs.



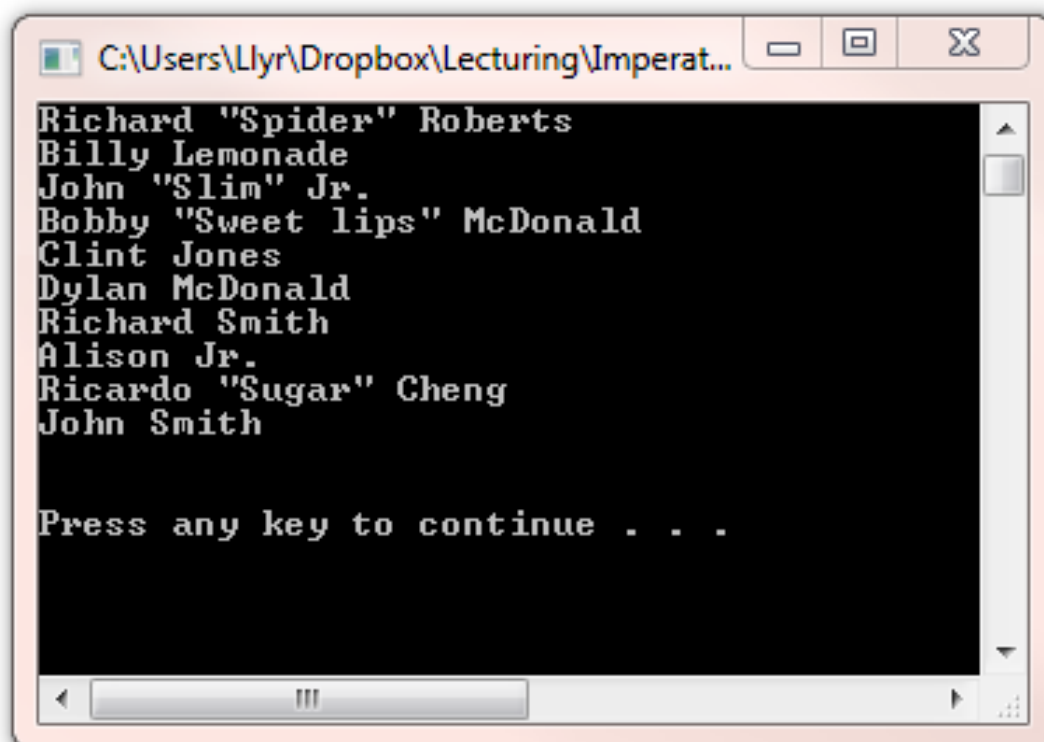
## Exercise 3 – Random Names

Your task for this exercise is to write a random name generator. The generator should work using the following formula:

Forename + (30% chance of nickname) + Surname

Test your program out by printing 10 random names.

Example output:



```
C:\Users\Llyr\Dropbox\Lecturing\Imperat...  
Richard "Spider" Roberts  
Billy Lemonade  
John "Slim" Jr.  
Bobby "Sweet lips" McDonald  
Clint Jones  
Dylan McDonald  
Richard Smith  
Alison Jr.  
Ricardo "Sugar" Cheng  
John Smith  
  
Press any key to continue . . .
```

Hints:

- You will need to create three string arrays to store your potential first, last and nicknames.
- Try to create about 20 options for each type of name
- You should only need 3 calls to `rand()` in your code

## Exercise 4 – Curvy

You can calculate the point on a repeating Sin curve using the following formula:

$$x = \sin(\text{time} * \text{frequency}) * \text{amplitude}$$

Where the sin component will always alternate between -1.0 and 1.0.

Write a program that generates a vertical sin curve on the screen, as shown in the example output.

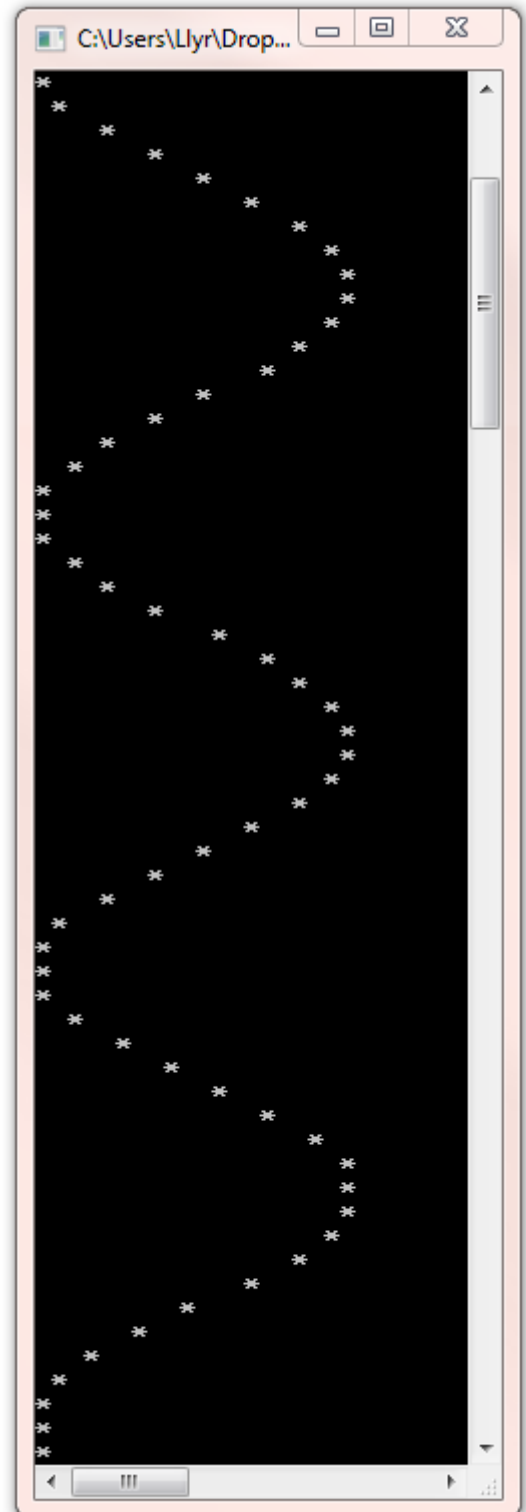
Your program should contain a function called `drawSinCurve()` that accepts three parameters – the **frequency**, **amplitude** and **duration** of the curve. When called, the function should draw a sin curve in the console with a **duration** number of \* points.

Hints

- `<math.h>` contains a [sin function](#)
- Try getting the program to print out a vertical line first, allowing you to adjust where the line is being drawn horizontally
- Make sure your sin curve formula is returning a number between 0 and amplitude.
- You can complete this exercise with two for-loops and an if-else statement
- Use the `sleep()` function of `<time.h>` inside the draw loop to slow down the printout

## Challenge - DNA

Can you get your function to print out a double-helix pattern? Hint - an interleaved version should be possible by changing a single parameter.



## Challenge – Full Pyramid and Temple

Extend your half-pyramid program from Exercise 1 so that it can create a full pyramid by creating a new function called `printFullPyramid()`. The program should only accept odd numbers for height. All other values should result in the program asking again.

Create another function called `printFullTemple` which accepts two parameters – a height, and a cap. The cap represents how pointed a pyramid is – for example, a high value will create a trapezoid temple shape. The function declarations of the programs should resemble the following:

```
void printHalfPyramid(int height);  
void printFullPyramid(int height);  
void printFullTemple(int height, int cap);
```

For a further challenge, create a third parameter called `step`, which adjusts the step reduction at each pyramid level. For example, step 2 would cause the pyramid to taper twice as fast.

```
void printFullTemple(int height, int cap, int step);
```

### Hints

- The full pyramid code can be done in many ways
  - Nesting four for-loops
  - Nesting two for-loops and a while loop
  - Recursive function calls
- Printing spaces is just as important as printing \*
- The extension to temples should be very simple
- The step reduction extension is a bit trickier...

Example output:



# Appendix

[Online C Programming Resources](#)

[Complete C Reference Library](#)

## C Programming IDE's

[Dev-C++](#) (Windows)

[Code::Blocks](#) (Windows, Mac, Linux)

[Visual Studio/C++ Express](#) (Windows)

[Netbeans C/C++](#) (Windows, Mac, Linux)

[Codelite](#) (Windows, Mac, Linux)



PRIFYSGOL  
**BANGOR**  
UNIVERSITY