Why Functions

- Any programs can be written without functions
  (some are hard to write without functions)
- Functions allow code reuse.
  Define a function once, call it many times.
  Makes program much easier to change!
- Functions allow code modularisation.
  Interaction with the rest of program is explicit and limited
  We can consider the code in isolation
  Much easier to read, test and debug!
- Breaking up your code into function is essential to reducing the complexity of your programs

Defining a Function - Example

```
// calculate x to the power of n
double power(double x, int n) {
    double result = 1;
    int i = 0;
    while (i < n) {
        result = result * x;
        i = i + 1;
    }
    return result;
}
```

Function Properties

- function have a type - the type of the value they return
- type **void** for functions that return no value
- function can not return arrays
- function have their own variables created when function called and destroyed when function returns
- function's variables are not accessible outside the function
- **return** statement stops execution of a function
- **return** statement specifies value to return
  unless function is of type **void**
- run-time error if end of non-**void** function reached
  without **return**

Functions with No Return Value

- Some functions do not to compute a value.
- They are useful for "**side-effects**" such as output.

```
    int i = 0;
    while (i < n) {
        printf("*");
        i = i + 1;
    }
    printf("\n");
}
```

Function Parameters

- function take 0 or more parameters
- parameters are variables created each time function called and destroyed when function returns
- C functions are *call-by-value* (but beware arrays)
- parameters initialized with the value supplied by the caller
- if parameters variables changed in the function has no effect outside the function

```
void f(x) {
    x = 42;
}
...
y = 13;
f(y);
printf("%d\n", y); // prints 13
```

Arrays as Function Parameters

- arrays function parameters different to other types
- the array is not copied
  changes to array elements visible outside function
- full explanation will have to wait until we cover pointers

```
void g(int x[]) {
    x[1] = 42;
}
...
int y[3] = {10,20,30};
g(y);
printf("%d\n", y[1]); // prints 42
```

Arrays as Function Parameters

- array type must be specified
- length of array function parameter can be left unspecified
- can write C functions that handle arrays of any length
- no way to determine length of array parameter

```
void sum_array(double array[], int length) {
    double sum = 0;
    int i = 0;
    while (i < length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
...
int y[3] = {10,20,30};
printf("%d\n", sum_array(y, 3)); // prints 60
```

Arrays as Function Parameters

- sometimes inconvenient to pass array length as separate parameter
- a stategy to avoid this is put special value at end of array
- function stops when array element with special value reached
- breaks if caller doesn't put special value in array!

```
void sum_array1(double array[]) {
    double sum = 0;
    int i = 0;
    while (array[i] != -1) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
...
int y[5] = {10,20,30,40,50,-1};
printf("%d\n", sum_array1(y, 5)); // prints 60
```

Function Prototypes

- Function prototypes allow function to be called before it is defined.
- Specifies key information about the function:
  - function return type
  - function name
  - number and type of function parameters
- Allows top-down order of functions in file
  More readable!
- Allows us to have function definition in separate file.
  Crucial to share code and for larger programs
- Example prototypes:

```
double power(double x, int n);
void print_asterisks(int n);
void sum_array(double array[], int length)
```

Example: Prototype allowing Function use before Definition

```
#include <stdio.h>

int answer(double x);

int main(void) {
    printf("answer(2) = %d\n", answer(2));
    return 0;
}

int answer(double x) {
    return x * 42;
}
```

Multi-file C Programs

- Large C programs spread across many C files
  e.g. Linux operating system has 50,000+ **.c** files.
- By convention **.h** files used to share information between files.
- **.h** files contain:
  - function prototypes
  - type definitions
- **.h** files should not contain code (function definitions)
- #include used to incorporate **.h** file
  put #include at top of **.h** file

Example: Include File

answer.h

```
int answer(double x);
```

answer.c

```
#include "answer.h"
int answer(double x) {
    return x * 21;
}
```

main.c

```
#include <stdio.h>
#include "answer.h"
int answer(double x);
int main(void) {
    printf("answer(2) = %d\n", answer(1));
    return 0;
}
```

Multi-file Compilation

```
$ dcc main.c answer.c -o answer
$ ./answer
42
```

Can also compile file separately creating bf .o files which contain machine code for one file.

```
$ dcc -c main.c
$ dcc -c answer.c
$ dcc main.o answer.o -o answer
$ ./answer
42
```

Useful with huge programs because faster to re-compile only part changed since last compilation.