Abstraction via Functions

Functions allow you to:

- separate out "encapsulate" a piece of code serving a single purpose
- test and verify a piece of code
- reuse the code
- shorten code resulting in easier modification and debugging

Functions we already use:

- From stdio.h: `printf()`, `scanf()`
- From stdlib.h: `rand()`

Structure of a Function

1. Return type
2. Function name
3. Parameters (inside brackets, comma separated)
4. Variables (only accessible within function)
5. Return statement

```
int add_numbers(int num1, int num2) { // 1, 2, 3
    int sum; // 4
    sum = num1 + num2;
    return sum; // 5

}
```

Function Prototypes

- Each function has a function prototype.
- It tells the compiler that the function exists, and the structure it has.
- It includes the key information about the function.
- Examples:

```
int add_numbers(int num1, int num2);
void print_asterisks(int numAsterisks);
```

Function Calls

When a function is called:

1. space is allocated for its parameters and local varables.
2. the parameter expressions in the calling function are evaluated and, if necessary, converted to the declared parameter types.
3. C uses "call-by-value" parameter passing
   parameters function works only on its own local copies of the parameters, not the ones in the calling function.
   EXCEPT arrays!
4. a function's (local) variables must be assigned values (initialized) before use
5. function code is executed, until the first **return** statement is reached.

The `return` Statement

6. when a `return` statement is executed, the function terminates:

```
return expression;
```

7. the returned expression will be evaluated and, if necessary, converted to the type expected by the calling function.
8. all local variables and parameters will be thrown away when the function terminates.
9. the calling function is free to use the returned value, or to ignore it.

Functions can be declared as returning `void`, which means that nothing is returned. The `return` statement can still be used to terminate such a function:

```
return;
```

Program Structure

1. Header comment
2. #included files
3. #defines
4. prototypes
5. main function
6. functions

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;
}
```

Calling a Function - Example

```c
// calculate x to the power of n
int main(void) {
    double a, b;
    printf("42 cubed is %lf\n", power(42.0, 3));
    a = 2;
    b = power(a, 8+8);
    printf("2 to the power of 16 is %lf\n", b);
    return 0;
}
```

Function Properties

- function have a type - the type of the value they return
- type **void** for functions that return no value
- **void** also used to indicate function has no parameters
- 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.

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

Function Parameters

- functions 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

```
double 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!

```
double 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 150
```

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

```c
#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.