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:

- `printf()`, `scanf()`

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
}
```

The `return` Statement

- when a `return` statement is executed, the function terminates.
- the returned expression will be evaluated and, if necessary, converted to the type expected by the calling function.
- all local variables and parameters will be thrown away when the function terminates.
- 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. A **`return`** without a value statement can still be used to terminate such a function.

Defining a Function - Example

```
// calculate x to the power of 3
double cube(double x) {
    double result;
    result = x * x * x;
    return result;
}
```

Calling a Function - Example

```
int main(void) {
    double a, b;
    printf("42 cubed is %lf\n", cube(42.03));
    a = 2;
    b = cube(a);
    printf("2 cubed 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_sign(int b) {
    if (b < 0) {
        printf("negative");
    } else if (b == 0) {
        printf("zero");
    } else {
        printf("positive");
    }
}
```

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
```

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_sign(int b);
```

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;
}
```

Library functions

- Over 700 function are defined in the C standard library.
- You'll need to use less than 20 of these in COMP1511
- The C compiler needs to see a prototype for these fucntion before you use them.
- You do this indirectly with **#include** line
- For example **stdio.h** contains prototypes for **printf** and **scanf** so:

```
#include <stdio.h>

int main(void) {
    printf("Andrew Rocks!\n");
}
```

structs and functions

A structure can be passed as a parameter to a function:

```
void printStudent(Student s) {
    printf("%s z%d\n", s.name, s.zid);
}
```

Unlike arrays, a copy will be made of the entire structure, and only this copy will be passed to the function.
Unlike arrays, a function can return a struct:

```
Student readStudentFromFile(char filename[]) {
    ....
}
```