Functions
Functions

- So far, all the programs we’ve been writing in the course have been written inside of `main()`.

- That hasn’t been a problem yet, but it could be if our programs start to become unwieldy.

- C and nearly all languages developed since allow us to write **functions**, sometimes also known as **procedures**, **methods**, or **subroutines**.

- Let’s see what functions are all about.
Functions

- What is a function?
  - A *black box* with a set of 0+ inputs and 1 output.
**Functions**

- What is a function?
  - *A black box* with a set of 0+ inputs and 1 output.
Functions

- What is a function?
  - A *black box* with a set of 0+ inputs and 1 output.

![Diagram showing the function add(a, b, c) with inputs 3, 6, and 7 leading to an output of 16]
Functions

- What is a function?
  - A black box with a set of 0+ inputs and 1 output.

```
mult(a, b)
```

```
4       5
```

```
\downarrow
```

```
\text{20}
```

```
mult(a, b)
```
Functions

- Why call it a *black box*?
  - If we aren’t writing the functions ourselves, we don’t need to know the underlying implementation.

```
mult(a, b):
    output a * b
```
Functions

- Why call it a *black box*?
  - If we aren’t writing the functions ourselves, we don’t need to know the underlying implementation.

```python
mult(a, b):
    set counter to 0
    repeat b times
        add a to counter
    output counter
```
Functions

- Why call it a *black box*?
  - If we aren’t writing the functions ourselves, we don’t need to know the underlying implementation.

- That’s part of the contract of using functions. The behavior is typically predictable based on that name. That’s why most functions have clear, obvious(ish) names, and are well-documented.
Functions

- Why use functions?
  - Organization
    - Functions help break up a complicated problem into more manageable subparts.
  - Simplification
    - Smaller components tend to be easier to design, implement, and debug.
  - Reusability
    - Functions can be recycled; you only need to write them once, but can use them as often as you need!
Functions

- Function Declarations
  - The first step to creating a function is to declare it. This gives the compiler a heads-up that a user-written function appears in the code.

  - Function declarations should always go atop your code, before you begin writing `main()`.

  - There is a standard form that every function declaration follows.
Functions

- Function Declarations

```c
return-type name(argument-list);
```

- The `return-type` is what kind of variable the function will output.
- The `name` is what you want to call your function.
- The `argument-list` is the comma-separated set of inputs to your function, each of which has a type and a name.
Functions

- A function to add two integers.

```c
int add_two_ints(int a, int b);
```

- The sum of two integers is going to be an integer as well.
- Given what this function does, make sure to give it an appropriate name.
- There are two inputs to this function, and we need to give a name to each of them for purposes of the function. There’s nothing important about these inputs as far as we know, so giving them simple names is okay.
Functions

- A function to multiply two floating point numbers.
Functions

- A function to multiply two floating point numbers.

\[
\text{float } \text{mult\_two\_reals}(\text{float } x, \text{float } y);
\]

- The product of two floating point numbers is also a floating point number.
- Let’s be sure to give this a relevant name.
- Again, the names of these particular inputs don’t seem to be important, so we can call them anything simple.
Functions

- A function to multiply two floating point numbers.

```c
double mult_two_reals(double x, double y);
```

- The product of two floating point numbers is also a floating point number.
- Let’s be sure to give this a relevant name.
- Again, the names of these particular inputs don’t seem to be important, so we can call them anything simple.
Functions

- **Function Definitions**
  - The second step to creating a function is to define it. This allows for predictable behavior when the function is called with inputs.

- Let’s try to define `mult_two_reals()`, from a moment ago.
Functions

- A function definition looks **almost** identical to a function declaration, with a small change.

  ```
  float mult_two_reals(float x, float y);
  
  float mult_two_reals(float x, float y) {
    float product = x * y;
    return product;
  }
  ```

- How would you fill in this black box?
Functions

- A function definition looks **almost** identical to a function declaration, with a small change.

```c
float mult_two_reals(float x, float y);

float mult_two_reals(float x, float y) {
    return x * y;
}
```

- How would you fill in this black box?
Functions

- Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```c
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
}
```
Functions

- Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```c
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
    int sum;       // declare variable
    sum = a + b;   // calculate the sum
    return sum;    // give result back
}
```
Functions

- Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```c
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
    int sum = a + b;  // calc variable
    return sum;       // give result back
}
```
Now, take a moment and try to define `add_two_ints()`, from a moment ago.

```c
int add_two_ints(int a, int b);

int add_two_ints(int a, int b)
{
    int sum = 0;
    if(a > 0)
        for(int i = 0; i < a; sum++, i++);
    else
        for(int i = a; i < 0; sum--, i++);
    if(b > 0)
        for(int i = 0; i < b; sum++, i++);
    else
        for(int i = b; i < 0; sum--, i++);
    return sum;
}
```
Functions

- Function Calls
  - Now that you’ve created a function, time to use it!
  - To call a function, simply pass it appropriate arguments and assign its return value to something of the correct type.
  - To illustrate this, let’s have a look at adder-1.c
Functions

- Function Miscellany
  - Recall from our discussion of data types that functions can sometimes take no inputs. In that case, we declare the function as having a `void` argument list.
  - Recall also that functions sometimes do not have an output. In that case, we declare the function as having a `void` return type.
Functions

- Practice Problem
  - Declare a function called `valid_triangle` that takes three real numbers representing the lengths of the three sides of a triangle as its arguments, and outputs either `true` or `false`, depending on whether those three lengths are capable of making a triangle.

- Note the following rules about triangles:
  - A triangle may only have sides with positive length.
  - The sum of the lengths of any two sides of the triangle must be greater than the length of the third side.
Functions

bool valid_triangle(float x, float y, float z);

bool valid_triangle(float x, float y, float z) {
    // check for all positive sides
    if (x <= 0 || y <= 0 || z <= 0) {
        return false;
    }

    // check that sum of any two sides greater than third
    if ((x + y <= z) || (x + z <= y) || (y + z <= x)) {
        return false;
    }

    // if we passed both tests, we’re good!
    return true;
}