CS50 Functions

Overview

Functions are reusable sections of code that serve a particular purpose. Functions can take inputs and outputs, and can be reused across programs. Organizing programs into functions helps to organize and simplify code. This is an example of **abstraction**; after you've written a function, you can use the function without having to worry about the details of how the function is implemented. Because of abstraction, others can use (or "call") the function without knowing its lower-level details as well.

Key Terms

- functions
- abstraction
- return type
- side-effect
- return value
- scope

Function Syntax

```
1  #include <stdio.h>
2
3  void say_hi(void)
4  {
5      printf("Hi!\n");
6  }
7
8  int main(void)
9  {
10     say_hi();
11     say_hi();
12 }
```

All programs you've written in C already have one function: `main`. However, programs in C can have more functions as well. The program on the left defines a new function named `say_hi()`.

The first line of a function requires three parts: first, the function's **return type**, which is the data type of the function's output that is "returned" to where the function was called. If the function does not return a value, the return type is `void`. Second, the function's name; this cannot include spaces and cannot be one of C's existing keywords. Third, in parentheses, the function's parameters, also known as arguments. These are the function's inputs (if there are none, use `void`). After this first line (known as the declaration line), the code defining the function itself is enclosed in curly braces.

In the example above, the `say_hi()` function causes `"Hi\n"` to be printed to the screen. This is called a **side-effect**, which is something a function does outside of its scope that is not returning a value. The `say_hi()` function is then called twice in the `main` function. Functions are called by writing the function's name, followed by any arguments in parentheses, followed by a semicolon. When the program is run, `"Hi\n"` prints to the screen twice.

Inputs and Outputs

The example on the right shows a function, `square`, which takes input and output. `square` takes one input: an integer called `x`. It also returns an `int` back to the where the function is called. Line 5 of the function specifies the function's **return value**, denoted by the word `return`. In this case, the `square` function returns the input value `x` multiplied by itself. When the `return` line is reached, the function is exited and the return value is returned to where the function was initially called.

Now that we've written this function, we can use `square` elsewhere in our program anytime we want to square a number. In the `main` function on the right, the `square` function is called three times: each time, the function is evaluated and returns the appropriate return value in the place of the function. So `printf("%i\n", square(2))` has the equivalent effect of writing `printf("%i\n", 2 * 2)` or `printf("%i\n", 4)`.

```
1  #include <stdio.h>
2
3  int square(int x)
4  {
5      return x * x;
6  }
7
8  int main(void)
9  {
10     printf("%i\n", square(2));
11     printf("%i\n", square(4));
12     printf("%i\n", square(8));
13 }
```

Scope

Variables that are defined inside of functions or in the list of function parameters have local **scope**, meaning those variables only exist inside of the function itself and have no meaning elsewhere. In the example above, if you were to try to reference the variable `x` inside of the `main` function, the compiler would give you an error; the `main` function doesn't know what `x` means, only `square` does. Likewise, any variables defined inside of `main` can't be accessed from inside of `square`.

If variables are defined outside of any functions, they have global scope instead of local scope. This means they can be accessed from any of the functions in the file. However, global variables are more difficult to keep track of and can be changed from any location in the program. Because global variables have global scope, that variable name cannot be reused in other parts of your program.