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; and others can use (or "call") the function without having to worry about the lower-level details as well.

Key Terms

- functions
- abstraction
- return type
- return value
- scope

Function Syntax

```
#include <stdio.h>

void sayHi(void)
{
    printf("Hi!\n");
}

int main(void)
{
    sayHi();
    sayHi();
}
```

All programs you've written in C already have one function: `main`. However, programs in C can have more functions as well. The program at left defines a new function, called `sayHi`.

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 the place where the function was called. If the function does not return a value, use `void`. Second, the function's name: which must not include spaces, and cannot be one of C's existing keywords. Third, in parentheses, include the function's parameters, also known as arguments, which are the function's inputs if it has any (if none, use `void`). After this first line (known as the declaration line), the code for the function itself is enclosed in curly braces.

In the example above, the `sayHi` function causes `"Hi\n"` to be printed to the screen. This is called a side-effect, which is an effect that a function has other than returning. Then in the `main` function, the `sayHi` function is called twice. Functions are called by writing the function's name, followed by any arguments in parentheses, followed by a semicolon. The result is that `"Hi\n"` is printed to the screen twice.

Inputs and Outputs

```
 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  }
```

The example at right demonstrates 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. The `return` line should generally be the last line of your function.

Now, we can use the `square` function elsewhere in our program anytime we wish to square a number. In the `main` function at right, the `square` function is called upon three times: each time, the function is evaluated, and its return value is used in place of the function. So `printf("%i\n, square(2))` has the equivalent effect of writing `printf("%i\n", 4)`.

Scope

Variables that are defined inside of functions, or in the list of parameters to a function, have local **scope**: the variables can only be referenced 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 inside of `square`.

If variables are defined outside of any functions, then they have global scope instead of local scope: they can be accessed from any of the functions in the file.