CS50 Syntax

Overview

C is a programming language with which you can write programs. Programming languages like C require you to write using a very specific **syntax**: a set of rules that describe how to arrange words and symbols (like brackets and parentheses) in order to write working statements that together can form a complete program. C's syntax might seem complicated at first, but with practice, the syntax of the language will start to become second nature to you.

Key Terms

- C
- syntax
- function
- string
- compile

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("hello, world\n");
6  }
```

Your First C Program

The code to the left is an example of a simple program in C which displays "hello, world" in the terminal window when it runs. Line numbers have been added to the left side of each line for reference, but they shouldn't be included in the code itself.

On line 1, `#include <stdio.h>`, tells your program to access a set of pre-written **functions** stored in a file called `stdio.h`, where a function is a collection of programming statements that performs a particular task.

By including `stdio.h`, your program can now take advantage of code that people have already written in the past: in particular, a function called `printf`, which displays text on your screen.

On line 3, `int main(void)` defines a function which acts as the beginning of your program, serving the equivalent of the "When Green Flag Clicked" button in Scratch. When your C program runs, it will look for the `main` function to know where to start. The curly braces on lines 4 and 6 hold the code of the `main` function together. Anything inside of the curly braces is therefore a part of the `main` function.

In this program, the `main` function has just one programming statement: `printf("hello, world\n")`. `printf` is a function (that was written in `stdio.h`) which displays a **string** (which is just a fancy way of saying "text") on the screen. In C, strings are always surrounded by double quotation marks.

Within the parentheses for `printf`, we've provided `printf` with a string as input, so that `printf` knows what string to display on the screen. In this case, our string is `"hello, world\n"`. The `\n` character tells `printf` to display a new line. The result of displaying `"hello, world\n"`, then, is to print out the words `hello, world` on the screen, followed by a new line. Finally, at the end of line 5 is a semicolon (`;`), which is C's way of defining the end of a programming instruction.

Compile and Run Your Program

Now that you've written your program, it should be saved in a file (typically ending with `.c`). In this case, we might call our file `hello.c`. This is your source code file. However, computers can't understand C code directly: remember that computers can only understand sequences of 0s and 1s. First, we need to **compile** our program: converting it from source code to object code, which is just sequences of 0s and 1s. Once the source code is compiled into object code, it can be executed. Any lines of code that begin with `//` or are enclosed with `/*` and `*/` will be ignored by the compiler: these lines are called comments, and will frequently be used by programmers to document their code so that people reading the code later (including themselves) can understand what's happening in the code.

Several compilers exist, including `clang` and `gcc`. We can also compile our program by typing the command `make hello` at the command line, which uses the `clang` compiler. If the program compiles successfully (without errors), then we can run the program by typing `./hello` at the command line. If all goes well, `hello, world` should be printed to the screen.

CS50 Variables

Overview

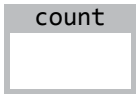
A **variable** is a storage container for data that is capable of holding different values that may change or update as programs execute. Your program can read the contents of a variable, update the contents of a variable, and display the value of a variable on the screen. Computer programs can use variables in order to remember useful information that the programs can then use later in the code.

Key Terms

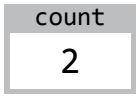
- variables
- declaration
- type
- integer

Declaring and Setting Variables

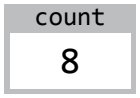
```
1 | int count;
```



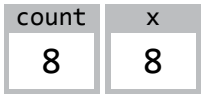

```
2 | count = 2;
```




```
3 | count = 8;
```




```
4 | int x = count;
```



The first step to using a variable in C is to let your program know that you want the variable to exist. This step is called the variable's **declaration** (also known as initialization). In C, this is done by first specifying the variable's **type**, which tells the program what kind of information will be stored inside of the variable, and then by specifying the variable's name (followed by a semicolon to end the programming statement).

For instance, in line 1 to the left, we've declared a new variable of type `int` to be named `count`. An `int` is a data type which stores an **integer**, which could be positive whole numbers, negative whole numbers, or zero (but not fractions or decimals). Currently, no value has been assigned to `count`: we've just told the program to create a space within which values can be stored later.

Once a variable has been declared, it can be manipulated in various ways. Line 2 takes the variable `count` and assigns its value to be `2`. Now, the number `2` is stored inside of the variable `count`. Optionally, we could have combined lines 1 and 2 into a single programming statement to declare a variable and set its value at the same time, via a line of code such as: `int count = 2;`.

After a variable has been given a value, its value can be updated. Line 3 updates the value of `count` again, this time to be `8`. Now, `count` forgets the number `2` and remembers the number `8` instead.

The value of a variable can be accessed just by using its name. For instance, line 4 declares a new variable (also of type `int`) this time named `x`, and initially sets its value to be `count`. This tells your program to go to the `count` variable, see what value is inside, and set the value of `x` to be that value. Since the current value of `count` is `8`, the value of `x` is set to also be `8`.

Variables from User Input

In many cases, a program may need to take input from the user and store the input as a variable. CS50 has written several functions (declared in a file called `cs50.h`) that serve this very purpose.

For instance, `get_int("prompt_string")` prompts the user to input an integer. In the program to the right, line 6 uses `get_int()` to take in an integer as input from the user with the prompt "`Integer please:`", and saves that integer in a variable called `i`.

```
1 | #include <cs50.h>
2 | #include <stdio.h>
3 |
4 | int main(void)
5 | {
6 |     int i = get_int("Integer please: ");
7 |     printf("i is %i \n", i);
8 | }
```

Line 7 then displays the value of the variable on the screen. The `%i` in the string is a special syntax which acts as a placeholder for an integer. We tell `printf` what integer to use in that placeholder by passing it an additional argument, where an argument is just a value inside of the parentheses of a function. Inside of the parentheses next to `printf` we've included two arguments: the string `"i is %i"`, and the integer `i`, which will take the place of `%i`. If the user were to enter the number `28` as input on line 6, then line 7 would replace `%i` with the value of `i` (which is `28`) and display the string `"i is 28"` on the screen followed by a new line denoted by the `\n`.

CS50 Data Types

Overview

Unlike many modern programming languages, C is a **statically-typed** language; it requires that every time you declare a variable, that you specify the data type of that variable. Many modern languages are **dynamically-typed**: at runtime, the program figures out the type of all the variables in the program. There are several different primitive (or basic) data types that are built in to C, and several more that are offered by the CS50 Library.

Key Terms

- statically-typed
- dynamically-typed
- native
- qualifiers

Native Types

C's **native** data types are the data types built into the programming language. An `int` is a data type which represents an integer: its value could be a positive or negative whole number, or zero. Numbers like 5, 28, -3, and 0 can be represented as `int`s, but numbers like 2.8, 5.124, and -8.6 cannot. When an `int` is declared, the computer allocates 4 bytes worth of space for it. Since 4 bytes is 32 bits, this means that there are 2^{32} (more than 4 billion) possible integers that can be represented as an `int`: in the range from -2^{31} to $(2^{31} - 1)$.

What if you need to store an integer outside of this range? C also includes **qualifiers**, which are keywords that can be added in front of type names to cause changes to the type. One such qualifier is the `unsigned` qualifier, which designates a type to be not negative. As a result, an `unsigned int`, while still 4 bytes in size, doesn't need to include the negative numbers in its range of possible values. An `unsigned int` can therefore take on a value in the range 0 to $2^{32} - 1$.

Another qualifier is the `long` qualifier, which allocates more bytes to the variable, allowing it to store more values. The long long integer (denoted by the type `long long`) is an integer which uses 8 bytes of storage instead of 4, allowing numbers in the range from -2^{63} to $(2^{63} - 1)$.

In addition to `int`s, C also has several other native data types. A `char` is a data type which represents a character of text. A `char` in C is surrounded by single quotation marks. Examples of possible `char` values include lowercase letters like 'a', uppercase letters like 'Z', symbols like the exclamation point '!', or even the newline character `'\n'`, which counts as a single character.

To store numbers that isn't a whole number, C has a type called `float` (short for floating-point), which uses 4 bytes to store a decimal value like 2.8 or 3.14. C also has a native type called `double`, which also stores decimal values but does so using 8 bytes instead of 4.

CS50 Library Types

The CS50 Library makes other types available to you, so long as you remember to type `#include <cs50.h>` at the start of your program. The `bool` type (short for Boolean) stores one of only two values: `true` or `false`.

The CS50 Library also defines a type called `string`, which stores text.

C doesn't limit users to only using the data types built into the programming language. It also offers additional features which allow the programmer to define their own custom types to use in programs.

Data Type	Native?	Sample Values	Size
int	Yes	5, 28, -3, 0	4 bytes
char	Yes	'a', 'Z', '?', '\n'	1 byte
float	Yes	3.14, 0.0, -28.56	4 bytes
double	Yes	3.14, 0.0, -28.56	8 bytes
long long	Yes	5, 28, -3, 0	8 bytes
bool	No	true, false	1 byte
string	No	"Hi", "This is CS50"	4 or 8 bytes

CS50 Operators

Overview

You're probably familiar with **operators** from math: the + symbol means addition, the - symbol means subtraction, etc. C also has operators, which you can use to modify or combine values. In addition to having operators that perform basic mathematical operations like addition, subtraction, multiplication, and division, C also has operators that perform other functions: like finding the remainder when dividing, or updating the value of a variable.

Key Terms

- operator
- arithmetic operators
- assignment operators

```
1 | int a = 2 + 8;
2 | int b = 10 - 3;
3 | int c = 4 * 7;
4 | int d = 10 / 2;
5 | int e = 10 / 3;
6 | int f = 13 % 3;
```

a
10

b
7

c
28

d
5

e
3

f
1

Arithmetic Operators

C's **arithmetic operators** perform mathematical functions on numbers. The `+` operator adds two numbers, the `-` operator subtracts one number from another, the `*` operator multiplies two numbers, and the `/` symbol divides one number by another. See lines 1 through 4 of the code to the left to see how such operators work.

When working with `int`s and dividing, it's especially important to be aware that an `int` cannot store non-integer values. For instance, in line 5, we try to store the value of `10 / 3`. C sees a division of two integers, and tries to make the result an integer as well. But since the "real" value of `10 / 3` isn't a whole number, everything after the decimal gets cut off (or "truncated") and `e` is set to just `3`. In order to save the value with the decimal included, we would need to use floating-point numbers, like `float e = 10.0 / 3.0`.

C has another operator, `%`, which is called the modulus operator. The modulus operator gives us the remainder when the number on the left of the operator is divided by the number on the right. Line 6 demonstrates the modulus operator: the remainder when dividing `13` by `3` is `1`, so the value of `f` is set to `1`.

Assignment Operators

C also provides **assignment operators**, which provide a variety of ways to update the value of a variable. The standard assignment operator (`=`) is demonstrated on line 7: it sets the value of `e` to be equal to whatever's on the right side of the equals sign: in this case, the current value of `f` added to `1`.

The variable you're assigning can also be on the right of the equals sign itself. On line 8, the value of `e` is set to the existing value of `e` plus one. While `e = e + 1` might not make logical sense in algebra, it's valid in C. Updating the value of a variable based on its existing value is so common that C has special syntax for it: the operators `+=`, `-=`, `*=`, and `/=` will set a variable to its existing value plus, minus, multiplied by, or divided by some other number.

C also includes special syntax for increasing the value of a variable by one or decreasing the value of a variable by one, by writing the name of the variable followed by `++` or `--`. So a statement like `e++` on line 11 takes the value of `e` and increases it by 1.

```
7  | e = f + 1;
8  | e = e + 1;
9  | e += 1;
10 | e *= 7;
11 | e++;
```

e
2

e
3

e
4

e
28

e
29

CS50 Boolean Expressions

Overview

Conditions are how programmers can make decisions in programs, by allowing some parts of the code to only run under certain circumstances. Conditions will generally work by evaluating a **boolean expression**, which is an expression that will have a value of either `true` or `false`. Programmers can set conditions such that different code will run depending on what the value of the boolean expression is.

Key Terms

- condition
- boolean expression
- if statement
- switch statement
- ternary operator

```
1 | bool a = 3 < 5;
2 | bool b = 2 >= 8;
3 | bool c = a && b;
4 | bool d = a || b;
5 | bool e = !d;
```

a	b	c	d	e
true	false	false	true	false

Boolean Operators

Boolean operators are used to create boolean expressions that evaluate to `true` or `false`. Common boolean operators include the comparison operators: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to), and `!=` (not equal to). For instance, in line 1 to the left, `a` is set to `true` because the expression `3 < 5` is true (because 3 is in fact less than 5). In line 2, `b` is set to `false` because the expression `2 >= 8` is not true.

Logical operators can also be used to combine boolean expressions. `&&` is the logical AND operator: it will evaluate to `true` if both expressions on either side of it are true. `||` is the logical OR operator: it evaluates to `true` if at least one of the two expressions on either side is true. And `!`, the logical NOT operator, evaluates to the opposite of whatever the expression immediately after it is.

Conditions

Conditional branching refers to the idea that different parts of code should execute under different circumstances. The most common type of conditional is the **if statement**: where a certain block of code (enclosed in brackets) will only run if the condition (whatever is in the parentheses after the word `if`) evaluates to `true`.

Optionally, C also allows you to include an `else` block after an `if` statement, which defines which code should run if the `if` condition evaluates to `false`. C will also allow you to include one or multiple `else if` statement after an `if` statement, to add additional conditions that could run different blocks of code. The if statement to the right (lines 1-12) will print `"positive\n"` if the value of `x` is greater than `0`, `"negative\n"` if the value of `x` is less than `0`, and `"zero\n"` if the value of `x` is equal to `0`.

C also has other ways of expressing conditionals. The **switch statement**, shown to the right (lines 15-25), takes one variable, and defines what code should run based on which `case` the variable matches. In the example at right, if `x` is equal to `1`, `"A\n"` is printed; if x is equal to `2`, `"B\n"` is printed, and in all other cases (the `default` case), `"C\n"` is printed. Code within cases should end with `break` so that the program knows to stop executing code and go to the end of the `switch` statement.

The ternary operator is a third type of condition. The **ternary operator** takes an expression, and evaluates to one value if the expression is true, and another value if it is false. In the example on line 28, if `x > 3`, `y` is set to 2, and `1` otherwise.

```
1  if (x > 0)
2  {
3      printf("positive\n");
4  }
5  else if (x < 0)
6  {
7      printf("negative\n");
8  }
9  else
10 {
11     printf("zero\n");
12 }
13
14
15 switch (x)
16 {
17     case 1:
18         printf("A\n");
19         break;
20     case 2:
21         printf("B\n");
22         break;
23     default:
24         printf("C\n");
25 }
26
27
28 int y = (x > 3) ? 2 : 1;
```

CS50 Loops

Overview

Loops are a way for a program to execute the same code multiple times. Instead of copying and pasting the same lines back-to-back, loops allow for code to be repeated. The resulting code is better designed: if you need to change the code that gets repeated, you only need to change it once. C has multiple different types of loops: all of which can accomplish the same things, though some may be preferable to others depending on the circumstances.

Key Terms

- loops
- for loop
- while loop
- infinite loop
- do while loop

For Loops

```
1  for (int i = 0; i < 10; i++)
2  {
3      printf("hello!\n");
4  }
5  for (int j = 0; j < 10; j++)
6  {
7      printf("%i\n", j);
8  }
```

The first type of loop in C is the **for loop**. Defining a `for` loop requires three parts (included in parentheses after the word `for`, and separated by semicolons), demonstrated at left (lines 1-4).

The first part is the initialization: we create a variable `i` initially set to `0`. This variable keeps track of which iteration the for loop is currently on. Second is the condition: as long as the condition `i < 10` is `true`, everything within the curly braces will keep running. As soon as the condition is `false`, then the loop ends. The third part is the loop modification: this code is executed at the end of every loop. In this case, we modify our loop by increasing the value of `i` by `1`.

Thus, each time the loop finishes, `i` will increase in value by `1`. As soon as `i` is no longer less than `10`, the condition fails and the loop will end. The end result is that `"hello\n"` is displayed 10 times.

By taking advantage of loop modification, you can also get a loop to do something slightly different each time the loop iterates. In the second `for` loop example (lines 5-8 above), `j` is initially `0`, and so `0` is printed. Then `j` increments to `1`, and `1` is printed in the next loop iteration. This continues until `j` is no longer less than `10`. The result is that each number from `0` to `9` is printed on its own line.

While Loops

C also includes a type of loop called a **while loop**. A `while` loop checks the condition it is given: if it is true, it executes the code within the braces, and then checks the condition again. This process repeats until the condition is false. The example at right (lines 9-14) does exactly the same thing as our second `for` loop (lines 5-8): printing out the numbers from `0` to `9`.

If the `while` loop is given a condition that is always `true` (like the boolean value `true` itself), then the loop will never stop running. The example at right (lines 15-18) is an example of an **infinite loop**: since the condition will never be false, the loop will continue running indefinitely. `While` loops are particular useful when you don't know in advance how many times a loop should run.

```
 9  int k = 0;
10  while (k < 10)
11  {
12      printf("%i\n", k);
13      k++;
14  }
15  while (true)
16  {
17      printf("hello!\n");
18  }
```

Do-While Loops

```
19  int j;
20  do
21  {
22      j = get_int("Positive Number: ");
23  }
24  while (j <= 0);
```

The **do-while loop** is similar to a while loop in the sense that it repeats a loop until a condition is false. However, a `do-while` loop, unlike a `while` loop, will always execute at least once, regardless of the condition. This is often valuable in cases where user input is required: the program should definitely ask for input once, and may or may not need to ask for input more times if the input is invalid.

In the example at left, the user will be prompted to enter an integer, and will be re-prompted continuously until a positive one is provided.

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.

CS50 Libraries

Overview

Libraries are shared collections of code that programmers can use to work with one another. Libraries usually include functions that may be commonly used among programmers. For example, the C library string.h includes many useful premade functions to manipulate strings (see below). By allowing us to use functions that others have already written, libraries enable us to build off of the work of others and use their functions in our own programs, instead of reinventing those functions ourselves.

Key Terms

- library
- header file

Using Libraries

To use a functions from a library in C, remember to `#include` the **header file** (such as with `#include <math.h>`), which defines the library's functions, at the top of your source code file. When compiling your code, you'll also need to link the library so that the resulting object code knows how to execute the functions.

Some Common C Library Functions

In `ctype.h`:

`isalnum()` takes a `char` as input, and returns `true` if the character is alphanumeric and `false` otherwise
`isalpha()` takes a `char` as input, and returns `true` if the character is alphabetic and `false` otherwise
`islower()` takes a `char` as input, and returns `true` if the character is lowercase and `false` otherwise
`isupper()` takes a `char` as input, and returns `true` if the character is uppercase and `false` otherwise
`tolower()` takes a `char` as input, and returns the character converted to lowercase if possible. If it's not possible, it returns the original character unchanged.
`toupper()` takes a `char` as input, and returns the character converted to uppercase if possible. If it's not possible, it returns the original character unchanged.

In `math.h`:

`ceil()` takes a `double` as input, and returns the smallest integer that is not less than the input, as a `double`
`cos()`, `sin()`, and `tan()` each take a `double` as input, and return the cosine, sine, or tangent of the input
`floor()` takes a `double` as input, and returns the largest integer that is not greater than the input, as a `double`
`pow()` takes two `double`s as input, and returns the value of the first input raised to the second value power
`lround()` takes a `double` as input, and returns a `long int` representing a rounded version of the number
`log()`, `log10()`, and `log2()` take a `double` as input, and return the logarithm of the number (base e, base 10, and base 2, respectively)

In `stdio.h`:

`printf()` takes a `string` as input, and prints it to standard output, displaying it on the screen

In `stdlib.h`:

`atoi()` takes a `string` as input, and converts the `string` to an `int` if possible, returning the `int`
`rand()` returns a pseudorandom integer, and will usually be seeded with `srand()` first

In `string.h`:

`strlen()` takes a `string` as input, and returns the length of the string, not including the null terminator
`strcmp()` takes two `string`s as input, and returns `0` if they are equal, less than `0` if the first `string` comes before the second, and greater than `0` if the first string comes after the second one
`strstr()` takes two `string`s as input, and finds the first occurrence of the second `string` in the first.

There are many more functions defined in these libraries and other libraries, and it is often a good idea to explore the existing C libraries to see what functions are available to you so that you don't re-create code that you could use library functions for instead. Check out reference.cs50.net for more information on C Library Functions.

CS50 Principles of Good Design

Overview

Design is a very important aspect of programming and product development. Good design differentiates programs that work from programs that work well. Programs with robust, consistent, and nonrepetitive code are generally considered to be well-designed. Other measures of design are program **efficiency** and modularization. In order produce portable, scalable, and reusable code, we must keep design in mind while programming.

Key Terms

- efficiency
- magic numbers
- tradeoffs

Loops and Conditionals

Loops are very powerful and often used in programming. However, as they are somewhat costly, we should make sure we use them efficiently. We can check that we are doing so by asking ourselves the following questions: Are each of my loops essential? Can I combine any loops? And am I taking advantage of every iteration of my loops?

Along similar lines, it's important to use conditionals (if, else if, and else) efficiently. Consider a program that takes in a birth month and outputs a corresponding birthstone. We could implement it by checking if the user inputted `"january"`, then checking if the user inputted `"february"`, and so on until we reach `"december"`. But, if we already know that a user inputted `"january"`, why bother checking any of the other months? In this case, we could improve our program's design by using else if statements or switch statements instead of all if statements.

Constants

Magic numbers are hard-coded constants in code. We consider using them to be bad design since they reduce the scalability and readability of code. Furthermore, making changes to hard-coded values must be done manually. Using variables instead can facilitate making such changes. Additionally, we can use `#define` to define constants that will not change, like the number of letters in the alphabet (26) or the value of a nickel in cents (5). We do this at the beginning of our code with `#define` after our header files and outside of our main function.

Functions

It's typically good design to break code out into functions when needed. For instance, if we were performing the same set of mathematical operations to multiple different values, it might make sense to write the set of operations as a function and simply call that function multiple times. Similarly, it's also a good idea to break really long code into different files, linking between them so they can all work together smoothly. In these ways, we can make code that could otherwise be very tedious and complicated to get through be easier to make sense of.

Tradeoffs

Design is subjective and debatable. What one programmer may think is better design, another might fundamentally disagree with. For instance, someone could write code using an uncommon function that makes the program shorter and more concise. However, a person that had never seen the function before and had to look up its documentation could very well argue that the program was not written clearly.

At right are two different implementations for an algorithm that takes in a number of t-shirts and tells us how many boxes we need to store them, if our options are boxes that fit 12 shirts, 10 shirts, 3 shirts, or 1 shirt. Which is best designed? Well, different programmers could argue in favor of either one, since each come with their own set of **tradeoffs**. What do you think?

```
num_boxes += (shirts_left / 12);
shirts_left %= 12;
num_boxes += (shirts_left / 10);
shirts_left %= 10;
num_boxes += (shirts_left / 3);
shirts_left %= 3;
num_boxes += shirts_left;
```

```
int boxes[] = {12, 10, 3, 1};
for (int i = 0; i < 3; i++)
{
   num_boxes += (shirts_left / boxes[i]);
   shirts_left %= boxes[i];
}
```