Translating Algorithm Components

Once you have written the function declaration and written the algorithm steps as comments, you are ready to translate each step of the algorithm to code, line by line. If you have written good (*i.e.*, clear and precise) steps in Step 3, this translation should be fairly straight forward—most steps you will want to implement naturally translate into the syntax we have already learned:

Repetition

Whenever you have discovered repetition while generalizing your algorithm, it translates into a loop. Typically, if your repetition involves counting, you will use a ***for*** loop. Otherwise, if you are sure you always want to do the body at least once, a ***do-while*** is the most appropriate type. In other cases (which typically align with steps like "as long as (something)..." ***while*** loops are generally your best bet. If your algorithm calls for you to "stop repeating things" or "stop counting" you will want to translate that idea into a ***break*** statement. Meanwhile, if your algorithm calls for you to skip the rest of the steps in the current repetition, and go back the start of the loop, that translates into a ***continue*** statement.

Decision Making

Whenever your algorithm calls for you to make a decision, that will translate into either ***if/else*** or ***switch/case***. You will typically only want ***switch/case*** when you are making a decision based on many possible numerical values of one expression. Otherwise, you will want ***if/else.***

Math

Generally, when your algorithm calls for mathematical computations, these translate directly into expressions in your program which compute that math.

Names

When your algorithm names a value and manipulates it, that translates into a variable in your program. You need to think about what type the variable has, and declare it before you use it. Be sure to initialize your variable by assigning to it before you use it—which your algorithm should do anyways (if not, what value did you use when testing it in Step 4?).

Altering Values

Whenever your algorithm manipulates the values that it works with, these translate into assignment statements—you are changing the value of the corresponding variable.

The answer is...

When your algorithm knows the answer and has no more work to do, you should write a ***return*** statement, which returns the answer that you have computed.

Complicated Steps

Whenever you have a complex line in your algorithm—something that you cannot translate directly into a few lines of code—you should call another function to perform the work of that step. In some cases, this function will already exist—either because you (or some member of your programming team) has already written it, or because it exists in the standard C library (or another library you are using). In this case, you can call the existing function (possibly reading its documentation to find its exact arguments), and move on to translating the next line of your algorithm.

In other cases, there will not already be a function to do what you need. In these cases, you should decide what arguments the function takes, what its exact behavior is, and what you want to call it. Write this information down (either on paper, or in comments elsewhere in your source code), but do not worry about defining the function yet. Instead, just call the function you *will write in the future* and move on to translating the next line of your algorithm. When you finish writing the code for this algorithm, you will go implement the function you just called—this is a programming problem all of its own, so you will go through all of the Steps for it.

Abstracting code out into a separate function has another advantage—you can reuse that function to solve other problems later. As you write other code, you may find that you need to perform the same tasks that you already did in earlier programming problems. If you pulled the code for these tasks into their own functions, you can simply call those functions. Copy/pasting code is generally a terrible idea—whenever you find yourself inclined to do so, you should instead find a logical way to abstract it out into a function and call that function from the places where you need that functionality.

With a clearly defined algorithm, the translation to code should proceed in a fairly straightforward manner. Initially, you may need to look up the syntax of various statements (you did make that quick reference sheet we previously recommended, right?), but you should quickly become familiar with them. If you find yourself struggling with this translation, it likely either means that your description of your algorithm is too vague (in which case, you need to go back to it, think about what precisely you meant, and refine it), or that the pieces of your algorithm are complex and you are getting hung up on them, rather than calling a function (as described above) to do that piece, which you will write afterwards.

✓

Completed