**The 5 Most Common Problems New Programmers Face--And How You Can Solve Them**

When you're just starting out with programming, it's easy to run into problems that make you wonder how anyone has ever managed to write a computer program. But the fact is, just about everyone else who's learned to code has had that experience and wondered the same thing, when they were starting out. I've helped teach several introductory programming classes, and there are some problems that trip up nearly every student--everything from getting started to dealing with program design.   
  
I'll prepare you to get past these challenges--none of them are insurmountable.

## Getting set up

Learning to program is hard enough, but it's easy to get tripped up before you even begin. First you need to chose a programming language (I recommend C++), then You need a [compiler](http://www.cprogramming.com/compilers.html) and a [programming tutorial](http://www.cprogramming.com/tutorial.html) that covers the language you chose and that works with the compiler that you set up. This is all very complicated, and all before you even start to get to the fun parts.   
  
If you're still struggling with getting the initial setup, then check out our page [on setting up a compiler and development environment (Code::Blocks and MINGW)](http://www.cprogramming.com/code_blocks/) which walks you through setting up a compiler with a lot of screenshots, and gets you up to the point of having an actual running program.

## Thinking Like a Programmer

Have you seen the State Farm commercials where the car wash company returns the cars to customers with the soap suds still on the car? The company washes the car, but it didn't rinse it. This is a perfect metaphor for computer programs. Computers, like that car wash company, are very very literal. They do exactly, and only, what you tell them to do; they do not understand implicit intentions. The level of detail required can be daunting at first because it requires thinking through every single step of the process, making sure that no steps are missing.

This can make programming seem to be a tough slog at first, but don't despair. Not everything must be specified--only what is not something the computer can already do. The header files and libraries that come with your compiler (for example, the iostream header file that allows you to interact with the user) provide a lot of pre-existing functionality. You can use websites like <http://www.cppreference.com> or our own [function reference](http://www.cprogramming.com/function.html) to find information on these pre-existing libraries of functionality. By using these, you can focus on precisely specifying only what is unique about your program. And even once you do that, you will begin to see patterns that can be turned into functions that wrap up a bunch of steps into a single function that you can call from everywhere. Suddenly complex problems will begin to look simple. It's the difference between:

Walk forward ten feet

Move your hand to the wall

Move your hand to the right until you hit an obstacle

...

Press upward on indentation

and

Walk to door

Find light switch

Turn on light

The magic thing about programming is that you can box up a complex behavior into a simple subroutine (often, into a [function](http://www.cprogramming.com/tutorial/lesson4.html)) that you can reuse. Sometimes it's hard to get the subroutine done up just right at first, but once you've got it, you no longer need to worry about it.   
  
You can go here to read more about [how to think about programming](http://www.cprogramming.com/tutorial/thinking.html), written for beginners.

## Compiler Error Messages

This may seem like a small thing, but because most beginners aren't familiar with the strictness of the format of the program (the syntax), beginners tend to run into lots of complaints generated by the compiler. Compiler errors are notoriously cryptic and verbose, and by no means were written with newbies in mind.   
  
That said, there are a few basic principles you can use to navigate the thicket of messages. First, often times a single error causes the compiler to get so confused that it generates dozens of messages--always start with the first error message. Second, the line number is a lie. Well, maybe not a lie, but you can't trust it completely. The compiler complains when it first realizes there is a problem, not at the point where the problem actually occurred. However, the line number does indicate the last possible line where the error could have occurred--the real error may be earlier, but it will never be later.   
  
Finally, have hope! You'll eventually get really really good at figuring out what the compiler actually means. There will be a few error messages that today seem completely cryptic, even once you know what the real problem was, that in a few months time you will know like an old (if confused) friend. I've actually written more about this in the past; if you want more detailed help, check out my article on [deciphering compiler and linker errors](http://www.cprogramming.com/tutorial/compiler_linker_errors.html).

## Debugging

[Debugging](http://www.cprogramming.com/debuggers.html) is a critical skill, but most people aren't born with a mastery of it. Debugging is hard for a few reasons; first, it's frustrating. You just wrote a bunch of code, and it doesn't work even though you're pretty sure it should. Damn! Second, it can be tedious; debugging often requires a lot of effort to narrow in on the problem, and until you have some practice, it can be hard to efficiently narrow it down. One type of problem, [segmentation faults](http://www.cprogramming.com/debugging/segfaults.html), are a particularly good example of this--many programmers try to narrow in on the problem by adding in print statements to show how far the program gets before crashing, even though the debugger can tell them exactly where the problem occurred. Which actually leads to the last problem--debuggers are yet another confused, difficult to set up tool, just like the compiler. If all you want is your program to work, the last thing you want to do is go set up ANOTHER tool just to find out why.   
  
To learn more about debugging techniques, check out this article on [debugging strategies](http://www.cprogramming.com/debugging/debugging_strategy.html).

## Designing a Program

When you're just starting to program, design is a real challenge. Knowing how to think about programming is one piece, but the other piece is knowing how to put programs together in a way that makes it easy to modify them later. Ideas like "[commenting your code](http://www.cprogramming.com/tutorial/comments.html)", "[encapsulation and data hiding](http://www.cprogramming.com/tutorial/lesson12.html)" and "[inheritance](http://www.cprogramming.com/tutorial/lesson19.html)" don't really mean anything when you haven't felt the pain of not having them. The problem is that program design is all about making things easier for your future self--sort of like eating your vegetables. Bad designs make your program inflexible to future changes, or impossible to understand after you've written. Frequently, bad design exposes too many details of how something is implemented, so that every part of the program has to know all the details of each other section of the program.   
  
One great example is writing a checkers game. You need some way to represent the board--so you pick one. A fixed-sized global [array](http://www.cprogramming.com/tutorial/lesson8.html): int checkers\_board[8][8]. Your accesses to the board all go directly through the array: checkers\_board[x][y] = ....; Is there anything wrong with this approach? You betcha. Notice that I wrote your accesses to the **board** all go directly through the **array**. The board is the conceptual entity--the thing you care about. The array happens to be, at this particular moment, how you implement the board. Again, two things: the thing you represent, and the way you represent it. By making all accesses to the board use the array directly, you entangle the two concepts. What happens when you decide to change the way you represent the board? You have an awful lot of code to change. But what's the solution?   
  
If you create a function that performs the types of basic operations you perform on the checkers board (perhaps a get\_piece\_on\_square() method and a set\_piece\_to\_square() method), every access to the board can go through this **interface**. If you change the implementation, the interface is the same. And that's what people mean when they talk about "encapsulation" and "data hiding". Many aspects of program design, such as inheritance, are there to allow you to hide the details of an implementation (the array) of a particular interface or concept (the board).   
  
Now go eat your spinach! :)