[THE CODING SURVIVAL GUIDE]

Habits and Pitfalls

Written by Martyr2 – Version 1.1

Table of Contents

Introduction

Learning how to code can be an enjoyable experience and a challenging one. You have to be a person who is willing to sit on a problem for an hour, a day, a week or even a month before the answer finally hits you. It is a profession where there are a lot of problems to solve and you are expected to provide a lot of solutions. But those solutions will reward you with a sense of accomplishment; a sense that you beat the problem and that can be a euphoric experience.

Our only hope, as programmers, is that we learn enough tools and techniques to make the problems we face seem less daunting. Maybe even allow us to get to the more difficult problems which will change the world. This survival guide is meant to provide you with some tips and tricks I have learned over the past 16 years of developing. By learning these tricks I hope you will find programming a little easier and sharpen your programming skills. In other words, I hope these tricks will help you stand on the shoulders of giants.

This guide will be your ticket to learning about the great habits you should embrace and the pitfalls many programmers face during their careers. I have bundled this guide with the popular ebook "The Programmers Idea Book: 200 Software Project Ideas and Tips to Developing them" to provide you some direction while you code your next project. If you find yourself stuck, review these tips and tricks and see if perhaps one will help you with your current situation. Enjoy!

Habits You Should Embrace

Stubbing - Stubbing is the process by which you first identify the functions, methods, classes, structures and files you will need to help solve the problem. You then code the framework of those pieces to fill in later with detail. For example, if you were building a calculator you might need functions to handle each of the arithmetic operations... add, subtract, multiply, divide and maybe a function to print a sum of values. Below are three stubs for add, subtract and a third to print the sum of values in an array. Notice that we have only written the "signature" of these functions and have not actually written their function bodies.

```
int add(int num1, int num2) {

}

int subtract(int num1, int num2) {


}

void printSum(int nums[]) {
```

```
}
```

You would stub out each function that you need for your calculator, and once you discover all the functions and defined their headers, you will have a “code by number” setup where you can easily code the body of each function. This would help you stay focused on each function and reduce the complexity of having to keep track of multiple things going on at the same time.

You could use stubbing to “stub out” whole classes, subsystems or often a whole program. After stubbing out everything you will have a better picture of how each of the pieces might work together. This will give you a better idea of where you might need that extra function or class to complete the project.

Use braces whenever possible - One style I always recommend you do is putting in the braces (also known as braces or brackets) characters for those languages that use them. In languages like C++, C# or Java you can define one line loop bodies, and control flow statements, that don’t require braces. When the compiler encounters this type of code it looks for the braces to denote the body of the loop or statement. If these braces are not found, the compiler will only execute the first line following the condition of the loop or statement. Below is an example...

```
for (int i = 0; i < 10; i++)
     cout << “Hello World”;
```

This for loop has a one line body which prints out the statement “Hello World” ten times. If we were to add another line after the cout statement, the compiler would see that line as an independent line, not part of the loop. For the compiler to see multiple lines in a loop, we would have to add the braces...

```
for (int i = 0; i < 10; i++) {
     cout << “Hello World”;
     cout << “How are you doing?”;
}
```

This tip recommends that you just add the braces even if the body has one line. Why do that? Well, there might be a chance (a pretty good chance in fact) that you or another person may need to add additional commands to the body later. At a future time you would have to add the braces anyways. So you might as well do it now and make it easier for others later. This future person may not be you and they could forget the braces, causing a hidden issue. So add them as a matter of kindness and increased readability. The braces

will make your loops / if conditions easier to read and easier for other programmers to simply add more code to.

Refactoring - Refactoring is the process of taking existing code and rewriting it into a simpler, and often easier to maintain, form. To help guide the user in refactoring, programmers often refer to “patterns” which are recognizable techniques for rewriting good and useful code. Refactoring may or may not optimize the performance of the code, but the point is usually to reduce redundancy and complexity while increasing readability.

One such example is the extraction method. This pattern tells us that if we have a long winded function, we can take several lines of it and wrap it into its own method. This often makes the original function much easier to understand and read. For example, if we have a function that looks like the one below, we can remove several lines and put it into a new function. We then call that function from the original.

```
double findTheAverage(int nums[], int size) {
     int sum = 0;
     for (int i = 0; i < size; i++) {
           sum += nums[i];
     }
     double average = (double) sum / (double) size;
     return average;
}
```

This function could be refactored into two functions where the second function would be in charge of summing up the values in the array and would then be called from the findTheAverage() function.

```
int getTheSum(int nums[], int size) {
     int sum = 0;
     for (int i = 0; i < size; i++) {
           sum += nums[i];
     }

     return sum;
}


double findTheAverage(int nums[], int size) {
     // Notice here we call the new function we extracted lines to
     int sum = getTheSum(nums, size);
     double average = (double) sum / (double) size;
     return average;
}
```

The lines that were in charge of summing the array have been extracted from the findTheAverage function and placed into a new function which we then call.

This is just one form of refactoring and there are dozens, if not hundreds, of various refactoring patterns out there. There are so many patterns in fact that they warrant their own books. If you are interested in making code sleek and easier to maintain, those types of books might be worth considering as future reads.

Limit parameters to functions - Have you ever found yourself writing a function which takes in a lot of parameters? When I say a lot I mean upwards of 4, 5 or more? Taking in a lot of parameters to a function is usually a classic sign that the function you are writing is going to be a complex one and might in fact be doing too much work. Each function should have a single task. Adding two numbers together while establishing a network connection and looking up a word from a dictionary is more than any one single function should be doing.

If you find yourself in this situation, consider looking at the function and deciding which responsibilities can be delegated to another function. In other words, can you break your complex function down into a series of simpler ones? This might be a perfect situation for using the extraction refactoring method described in the previous trick. You might be looking at a situation similar to something like this (written in PHP)...

```
function passResultToDB($num1, $num2, $tableOfValues, $connection,
$user, $pass) {
     // Complex code trying to add numbers, lookup values
     // and establish a connection
     // All in one function?
}
```

Just by seeing the number of, and types of, parameters this function is taking, we know that this function is doing too much. Try breaking the function down into a series of other functions that you can call in sequence to accomplish the same task. This will reduce the number of parameters each function needs and make things simpler to understand.

```
$result = sum($num1, $num2);
$foundValue = lookupTableValue($result);
$connection = establishConnection($user, $pass);


if ($connection) {
     saveToDatabase($connection, $result);
}
```

Here we have several new functions which have one or two parameters. These functions will tend to be less complex and easier to use. If there is an error, it will be easier to also identify which function is responsible instead of sifting through an entire lengthy function.

Knowing when to use an abstract class versus an interface - This question comes up a lot. When do you create an abstract class versus just implementing an interface? It will always come down to your program's design, but generally if you have some very dissimilar classes that may need to implement one or two similar methods maximum, go with an interface. Especially if those classes already exist in the system and are not easily inheritable. If the structure of the system calls for several classes that are quite similar then go with an abstract class and inherit from it. Abstract classes are great in situations where you know you are going to have future classes based on the classes you are going to create.

For example, if you are working with a banking application that deals with the idea of an "Account" it is better to create that as an abstract class. This is because you know you are going to have various kinds of accounts at some point in the future. You may have some "No limit banking account" type and tying that into a system with abstract classes I find has the maximum flexibility. At some later point if you have a new type of account, all you have to do is inherit and add the extra functionality.

But again there are no hard fast rules and often times using one method over the other is preferable based on the system requirements. Interfaces offer more of a "bolt on" way of adding additional code while abstract classes are more in line with extending existing class hierarchies.

Now let's say you had a banking account class, a mortgage class, a wire transfer class and all of them needed one method to return a balance, then an interface may be your best bet. All these classes are not really the same yet all need to be able to return some form of total balance value. Implementing an interface and adding it to each class will then allow you to add a getBalance() function to each without having to alter the classes too much in order to make them match up.

Incessant testing - I once heard a rule that said that 30% of your time should be coding and design while the other 70% should be for testing. I have seen different takes on this number but the one thing they have all shared was that the testing percentage was much higher than you may expect – typically over 50%. In short, follow the mantra "test test test!"

Testing is not very exciting, most of the time, but it can certainly be the difference between your software being highly polished versus something destined for the nearest dumpster. You should look to do testing on each function, method or class separately (unit testing) and then testing multiple pieces working together (integration testing).

Make a priority to test as much as you can in the time you are given. To do this, create test cases where you write out a scenario and identify all the input information to the system. Then come up with the output based on the supplied input and run it against the system. Look for the output to be the same and analyze the system when the output is wrong. Make sure you account for all the edge cases as well. Does your method expect a number? What if you put in -77, 0, 5.4 or 100,000,000,001? Does it handle all cases successfully? Did you change something to fix a bug? Make sure to rerun all your tests again to check if everything still works fine. This is the part where some automated testing tools will prove time saving.

Pick a naming convention and be consistent with it - Naming conventions are always a hot topic in the world of development. Which naming convention is best? Well, there are some that are well recognized and some that are more / less descriptive than others. The trick here is that whatever naming convention you decide to go with, be consistent! Don't change it from one function to the next. If I read Hungarian notation style variable naming in the first function, I better see that same style in the last function as well. Following naming conventions is a great habit to get into and will help you adapt to all kinds of different projects.

Often times when you go and work for a software company they will have instituted their own naming convention style through a coding style guide of some sort. It would be advisable on day one to read that guide and immediately adopt the style... even if you don't 100% agree with it. You will want to be consistent with the other developers and it will also help keep your code readable when compared to other code generated by others in your team.

Reduce the number of calls to a function if it always results in the same answer - This trick is going to help save you some cycles when it comes to executing code, like statements found in loop headers. Many programmers sometimes stick a function call in something that is evaluated over and over again even though the answer is going to be the same. Instead, look to call the function once and save the result, then use the result in your loop.

For example, let's assume we have an array of values that we are going to iterate through. The length of this array is pretty much fixed since we are not adding or subtracting items from it. Whether we call the "length" method on it once or thousand times the answer would be the same.

```
for (int i = 0; i < somearray.length(); i++) {
     // Access the array elements here
}
```

In the code shown above we are calling the length() method each time on the array as we iterate through its elements. If the array's length is 100, we are going to call length() 101

times and that means executing the same code inside of length() 101 times. We are doing this over and over again even though each time it returns 100 as the length. If we save the value first, then we can use the length variable in the loop and save the processing power of having to compute the length for an answer we already know.

```
// Here we are calling length() once
int lengthOfArray = somearray.length();


for (int i = 0; i < lengthOfArray; i++) {
     // Access the array elements here
}
```

This setup will help when it comes to looking up the same value in a table (like a search), running a complex math calculation that hasn't changed its value (like a computed list of interest rates for a given principle over a set period of time), or for functions that return a resource that you may need to access again and again (like a database connection or file handle). In the case of functions that return a resource, we are especially concerned since we might be building and tearing down an object over and over again which could lead to a real performance hit. So this tip certainly has helped me, and many other programmers, increase the speed of their programs and make better use of limited resources.

Use flags where they make sense - Beginner programmers discover the idea of "flags" and usually go on some kind of binge with them, using them everywhere to control the flow of their code. For those who don't know what flags are, they are typically a variable value that you set to determine the state of execution. For instance you may see something like this...

```
boolean flagVar = true;


while (flagVar) {
      flagVar = false;

      // Do some code…

      if (true_condition) {
           flagVar = true;
      }
}
```

Or in the case of debugging, you might have run into a similar idea when it comes to debug mode. If in debug mode, set some guard condition based on the value of a particular variable.

Guard conditions are one example of good use of flags. The use of the flag variable to control the condition of a while loop is also a good use (most of the time). But what you don't want to be doing is overusing flags. Excessive use of flags slows down other coders doing maintenance because they are forced to look up what a flag exactly does, when it is set, what is the current state of a flag at any given time and forces the programmer to constantly go back and see what is setting the value of the flag. An example of bad use of flags would be something like this...

```
boolean flagVar = true;


if (flagVar) {
     // Do some code here
     otherflag = flagVar && otherFlagVar;
}
else {
     otherflag  = !flagVar;
}


if (otherflag) {
     // Do some code here
}
```

Where does otherFlagVar come from? When it is true, what does that mean? What statements modify that flag? Then we got one flag based on another flag. We might have been better off just to test code conditions themselves instead of flags based of flags based off of other flags.

Life should always be done in moderation, so flags should be used in moderation too and with well named variables. They should be used to control one state and try to avoid flags that are based off of other flags values. Complex use of flags leads to programmers having to keep track of a truth table inside their head. Then of course this leads to bugs and often times these types of bugs will sneak their way into production environments.

Ask yourself "what if this happened?" - Another great habit to get into is asking yourself "What if this if statement was false? What if this variable had not been set? What would happen if I change this do while loop into a pre-condition while loop? Would it be faster?" In other words, stay curious and don't be afraid to experiment. Oftentimes programmers can find themselves in a rut where they do loops in one certain style or write functions a certain

way just out of repetition without giving it much thought.

You should try to get into the habit of always thinking out of the box and thinking about how you can make your code easier to read, easier to maintain, easier to understand and if perhaps you could write it with a little more efficiency (just don't be incessant about it, read the trick above about incessant optimization). That is simply how you learn and grow.

If you are not sure how something works, give it a try. Run a test on it and take a moment to try it out in a separate project. Being curious has been at the center of some of the greatest discoveries and minds in history. If you don't find yourself asking yourself that question enough, then you probably are not thinking about the problem enough or writing code that might be adequate, but not great. Perhaps you should write "What if this happened?" on a post-it note and put it on your computer monitor to remind yourself. Whatever works.

Use constants - I have to admit that I don't use constants enough myself and I have been trying to make this a habit of mine as well. One way to make sure that your programs can avoid some serious bugs is to make sure that values that shouldn't change DON'T! If you set a variable called "PI" to "3.1415" then you should simply not be allowed to change it to "5.9987". If you don't set it to be a constant value, then you might accidentally change it and that means any functions using it are going to fail or crash a miserable death. Java has the keyword "final" while C++ and C# have "const" and some languages, like Python, don't have your typical constants so be careful.

The idea of constants is also going to move any errors from runtime to compile time and we want that as programmers. We want the compiler to throw out an error saying that a variable's value, which shouldn't change, was trying to be modified by code at compile time. This tells us that something we designed wasn't working right from the start versus sneaking its way into runtime where it may not be discovered until the end user uses the software. Constants are just one of many tricks of defensive programming.

Being resourceful - This habit is often a very hard one to get into doing. What does it mean to be resourceful? It means knowing where to find that one thing you are after and sometimes through creative means.

For example, you are working on a function that is in charge of establishing a networking connection with a POP3 server to fetch email messages. How do you talk to such a service? There has to be a protocol in place. Being resourceful is to know where the answer most likely lies and where it would make sense to find it. "Oh, perhaps it is in a Request for Comment (RFC) document. Where would I find that document? I know that I can find a lot of RFCs at the IETF website." So you go there and find the answer. Can't find the answer there and Google is no help? Have you tried a forum? Not specific enough? Have you tried an IRC chat room?

Maybe you are working on something unique that no one has ever done before. Knowing where to go for help and resources is one of the greatest tricks programmers have in their toolbox. You are not going to know everything, but knowing where to find the stuff you don't

know can certainly help you in your programming career. Perhaps in an interview a potential employer will ask you about a certain feature of a programming language. You don't know, but you know exactly where the answer can be found and this skill is often what employers are looking for. Resourcefulness is practiced by asking yourself "Where would I find this information?" and then finding it. Do it enough and you will build yourself a personal library of resources you can look up or people you know who can give you the answer.

It is also the difference between those programmers who seem to know infinite things and those who seem to know nothing.

Now that we have covered some of the greatest habits you can gather, we will switch gears and talk about some of the potential pitfalls programmers should avoid. Great habits and knowing where pitfalls lie will give you the double edged sword you need to hack your way through great code frontier... and do it with an effortless swagger of a true professional.

Pitfalls to Avoid

Long methods, especially in main() - One of the first rookie mistakes I come across, and everyone has done it at one time or another, is to make a really long method that is doing everything. Those brand new to programming languages like Java or C/C++ tend to put all their code in the main() method. The problem with this approach is that if something goes wrong, you have a ton of code in one method to debug instead of several small bite size code "snippets" that could easily be reviewed and written off as not being the problem.

Another problem you run into with putting all your code in one function is that you lose the ability of reuse. If I am adding values in an array together, it might make sense to put that logic in its own function. From then on you can call that function instead of having to rewrite the same logic code over and over again... making the long winded function even longer.

Your functions should be written in such a way that each function does one, and only one, task. If you are putting more than roughly 20 or 30 lines of code in a function, then that is a red flag that you might be doing too much. It may seem to be a bit overkill when you first start out, but avoiding this pitfall is going to help you tremendously later.

Not commenting enough, commenting too much or too vague of comments - Another pitfall to avoid is commenting too much or too little. You can find tons of articles on the Internet these days that talk about how much or how little you should be commenting. But the reality is, you should be commenting only enough to give you, and future programmers, the gist of what is going on. If you find yourself commenting a lot, it means your code is too complex to understand and you are compensating for it by trying to explain the code. Let's take a look at some comments and see which would work and which would not...

```
// Add a + b
c = a + b
```

This comment is reiterating what can obviously be seen in the code itself. We know you are adding a + b, we don't need to be told that we are adding. Besides commenting on the obvious, this is also probably too little commenting. Adding a bit more description would help the situation...

```
// Add a (number of apples) to b (number of bananas)
c = a + b
```

At least now this is telling the future you, or other programmers what "a" and "b" represent. But this type of comment could have been simply replaced with better variable names (which is another way to make your code more readable and negate the need to make a comment).

```
c = apples + bananas
```

Here we see we are adding apples and bananas together. Great! There is nothing else worth mentioning here so we can move along. Below is an example of a long winded comment...

```
/* Even though apples are red and bananas are yellow we are going
to add these together. We felt that fruits should be added
together. After all, fruits are healthy and delicious so they
should always be part of your daily meals. */

c = a + b
```

Here we have way too much commenting. It rambles, doesn't really tell us anything about the actual code and probably tells us too much information that we don't really need to know. Is the real reason you are adding them because they should be part of your daily meal or just because you find them healthy and delicious? Can apples even be added to bananas? Not only that, what about apples that are yellow or green?

Comments should be short, quick, relevant descriptions to give you, or others, the idea of why something is being done. Nothing more and nothing less.

```
// We are adding the quantity of apples and bananas together to get
a count for our fruit basket

c = a + b
```

With a comment like this we are seeing that we are adding their quantities to use in our fruit basket product on our e-commerce site. Even though we could use better variable names here, we know the number of apples and the number of bananas is being added to get a count which is “c”.

But as mentioned before, if you find yourself having to do a lot of explaining for a simple a + b statement, it might be because the code is too complex or the motive is unclear. Review the process to see if it can be rewritten in simpler and more readable ways.

Tight coupling - You may have heard of the term “coupling” and or “cohesion” when it comes to software design. However, those new to programming often end up making classes that are heavily dependent on another class, function or file to work. This is tight coupling. If you have written a class and find that you cannot simply pull it out of the project, save it and reuse it in a completely new program, it means it is tightly coupled to something in the original program.

The reason we want loose coupling is to provide maximum reuse. You worked hard to create that masterpiece of a class; you should be able to reuse it over and over again in all your projects. It should be part of your class library and given to others for them to use without having to carry along other code that might be linked to it. In the case of classes, your class should be self contained. It shouldn’t have a bunch of other classes, or outside functions, that come along for the ride.

You can avoid this pitfall by making sure that all classes you write stand on their own. They can be given information through constructors or methods, but the class itself should not be saying that it needs all this other stuff to do its job. It should be “modular” and ready to be dropped into a project on a moment’s notice.

One exception I want to add to this is when it comes to things like namespaces and header files. Obviously some classes are going to need access to libraries at a project level. When I say they should not be tightly coupled I am referring to other classes, functions or resources that you specifically have defined in a project to help support the class and its functionality.

Let’s run through a simple (and contrived) example first and see how we might fix it.

```
public class Car {
     private int wheels;


     public Car() {
          wheels = WheelClass.getNumberOfWheels();
     }
```

```
	public int getWheelCount() {
		return wheels;
	}
}
```

In this class above we see that in order to set the number of wheels, we are counting on the existence of a WheelClass and that the class has a getNumberOfWheels() method. This means we have to bring along the WheelClass and its functionality if we want this Car class to be reused in another project.

Upon closer inspection perhaps we realize that a Car is always going to have four wheels. In this case we could get rid of the call to the WheelClass and thus break the relationship between the two classes.

```
public class Car {
	private int wheels;


	public Car() {
		wheels = 4;
	}


	public int getWheelCount() {
		return wheels;
	}
}
```

Now we can bring this Car class with us and throw it into other projects without having to bring along the WheelClass.

Sometimes it is not always possible to make a class or function fully modular like this and some instances of coupling may be required, but keep in mind that to use a simple class or function should not force you to bring over massive amounts of extraneous code. Perhaps in our other project we wouldn't even use the WheelClass except to support this Car class. That is just a lot of code bloat that is not needed.

The pitfall to avoid here is using code that needs a lot of supporting code just to keep it running. You can think of it as "code baggage". Travel light and you will have a more pleasant trip.

Variable letter abbreviations - How many times have you seen some code and not really understood what it does because you don't know what "a", "b" or "c" variables represent? This is a pitfall that is relevant to new programmers, but more common to those with a

couple years of experience under their belt. They get to the point where they are experimenting with code design and somehow they think that by using obscure variable names, that are often abbreviations, they are somehow describing the problem in a more brief and concise form.

In short, they are doing the opposite. By shortening variable names to the point of being vague or misleading they make the reader spend even more time trying to figure out what they are doing rather than being able to easily read the code and see the steps.

Obfuscating code in this way is counterproductive and can lead to code that contains missed bugs and adds confusion. Try to come up with names that are easy enough to read and contain enough characters to adequately describe what their purposes are. The variable “qtyOfApples” is more descriptive than “a” or “apples”. This is an easy pitfall to avoid; it just takes a little practice.

Too much error handling - When it comes to languages which feature error handling mechanisms, like try/catching or the dreaded “goto” statement, it can be easy to fall into the trap of providing a large amount of error handling. It is good to catch your errors, but sometimes you can go overboard and provide so much error handling that it makes your code hard to follow. Primarily from the point of view of where the error actually gets caught.

This becomes particularly cumbersome when you find yourself throwing an exception from the catch/final clauses of a try catch. You could easily end up with code that starts looking like this...

```
public void DoSomething() {
      try {
            DoSomethingElse(4);
      }
      catch (Exception ex) {
            // Handle exception from DoSomethingElse
      }
}


public void DoSomethingElse(int parameter) throws Exception {
      try {
            // Code here
      }
      catch (IOException ex) {
            // Do some code for IO problems
      }
      catch (IllegalAccess ex) {
            // Do other code here for access problems
      }
      catch (Exception ex) {
```

```
            // Never do this here, this is swallowing up the original
exception
            throw new Exception();
      }
}
```

Now just imagine if you had some other kind of try catching inside the try clause of that try catch structure. To avoid problems like these, try to see if you can group similar code statements that all have the potential to throw the same type of error. In some cases, you can just declare the function as throwing an exception and don't handle it locally. Let it bubble up to the caller and handle it there. In addition to these points, keep in mind that throwing an error is often an expensive operation, so use moderation.

Exception handling is a good thing and should be used, just avoid the pitfall of going overboard with it and try catching everything imaginable. Once you find yourself nesting your try catching deeply, or creating long catch clause lists, then it is time to review and see if perhaps you could organize things to reduce the amount of error handling.

Complex conditionals - This pitfall is a classic for rookies. You have a value that you want to run through a gauntlet of tests and if it passes, then do something with it. Often times you will be tempted to write all those tests inside one complex condition statement that look like this...

```
int x = 1;


if ((x > 0) && (x < 5) || ((x % 2) == 1) && (isprime(x) && (abs(x)
== 1))) {
        // Do stuff with x
}
```

These things are often a nightmare to sift through and keep straight. When new people come to me and ask me what is wrong with their code, I see things like this all the time. II don't even bother trying to "decipher" it anymore. I often break it apart and rewrite it. How to rewrite it? There are a couple methods you could do.

1. Try to take a few of the tests and put them in their own "descriptive" function and call that function from the condition.
    a. Put something like (x > 0) and (x < 5) and into a function called "checkRange()"
    b. Call checkRange and pass it the value of x which it then checks
    c. Returns true or false if it meets the criteria
    d. Put the call to checkRange in the original if statement... `if (checkRange(x) || ((x % 2) == 1) && (isprime(x) && (abs(x) == 1))) {`

Sometimes you can just break the if statements down into a series of smaller if statements like so...

```
if ((x > 0) && (x < 5) || ((x % 2) == 1)) {
      if (isprime(x)) {
            if (abs(x) == 1) {
                  // Do something
            }
      }
}
```

Of course you can mix both methods 1 and 2 for even greater effect. In the example below you have a couple nested if statements which call well described functions. This allows you to take a bunch of cryptic complex condition statements and get something that is much more readable...

```
if (checkRange(x) || isOdd(x)) {
      if (isPrime(x)) {
            if (abs(x) == 1) {
                  // Do something
            }
      }
}
```

Obviously this method really can't do much unless the value of "x" is 1, but you get the idea of how we turned a cryptic string of complex conditions into conditions that are easier to read and simpler to keep track of in your head.

Conditionals that test the opposite of another conditional - Another pitfall with conditionals that I see newbies make is testing a variable for one value and then testing it again for its opposite value in an else-if. If you test "qtyOfApples" to see if it is greater than or equal to 10, in the else-if that follows, you don't need to test if qtyOfApples is less than 10. We logically know that if the first if statement is false, then it has to be less than 10.

That seems pretty straight forward to some, but then they go off and write a test like this...

```
if (($leftPosition >= 5) && ($canSize === true)) {
      $leftPostion--;
      $biggerSize = $size + 5;
}
else if (($leftPosition >= 5) && ($canSize  === false)) {
```

```
     $leftPostion--;
}
else if (($leftPosition < 5) && ($canSize  === true)) {
     $rightPosition++;
     $biggerSize = $size + 5;
}
```

You can spot this problem primarily when you have a bunch of if statements grouped together and they appear to be testing the same things in different ways. That code above could have been summed up better as...

```
if ($canSize === true) { $biggerSize = $size + 5; }


if ($leftPosition >= 5) {
     $leftPosition--;
}
else {
     $rightPosition++;
}
```

We adjusted this code by simply “boiling down” the tests. Obviously if $canSize is true, we are going to adjust $biggerSize no matter what $leftPosition’s value ends up being. Then if $leftPosition is greater than or equal to 5 we adjust $leftPosition, otherwise we know that $leftPosition is less than 5 so we adjust $rightPosition. We didn’t have to build a test to see if it was less than 5, we knew it was because it failed the first test.

Programs where I often see this pitfall the most are when the programmer is trying to test boundaries of x/y coordinates or the true/false value of a statement. Sometimes it is also accompanied by the ! (NOT) operator...

```
if (isOdd(x)) {
     // x is even
}
else if (!isOdd(x)) {
     // We know that if it is not odd, it was even,
     // why call isOdd to only negate it?
}
```

This is a pattern you can easily learn to spot. Look for complex conditionals that appear to be testing the same variable or return value, which can only have one of two outcomes, but purposely tests the other outcome if the first test fails. If there are two outcomes and it is not

the first one, it is the second one. Simple.

Premature or incessant optimization - Almost all programmers fall into this one at some point in their life... premature or incessant optimization. This is the process of trying to optimize your code as you write it and then continue refining it over and over again before you even finish writing the piece of code you are working on. Sometimes refining your code so often becomes a form of its own Obsessive-Compulsive Disorder (OCD).

The reason this is a pitfall is because it kills your productivity and can even stunt a bit of your creativity. I hear you asking "How can it be stunting my creativity when I am actually thinking of more and more creative ways to be efficient?" Well, I am talking about creativity on a larger scale. If you are fussing with a few lines of code to make it the best, are you really looking at the bigger picture and being creative on how everything works together? You should certainly optimize when you have the chance to make things more efficient but don't sidetrack yourself or get obsessive about it.

You can combat premature optimization by first telling yourself that you are just going to get down all the pieces you need to write first and then go back to it later. Then after you write the code start doing your optimizations. If you do not have to work to some extreme performance requirement, try limiting your time reworking it. No one said you can't come back to the code time and time again later. Just get into the practice of moving on.

One major problem with this pitfall is that you never end up finishing any projects because you are too much of a perfectionist. Sure you throw out some great code in the end, but if it never makes it to the light of day, what good is it to anyone?

Reinventing the wheel - This classic pitfall is when a programmer creates something that has already been created before. We should all re-create something from time to time to help us learn how it works, but in the scope of larger projects and meeting budgets/deadlines re-creating what has already been done before (and probably better / battle tested) is a waste of time.

To avoid this pitfall most people turn to libraries or code snippets that have already been created and tested in full production systems. When we code such libraries or snippets, it is this pitfall we are trying to avoid and why we make our code generic enough for anyone to use. Reuse is one of the cornerstones of modern software development and we don't want to be wasting precious time creating something that ends up being just as good (or perhaps even worse) than what has already been done before.

If what you need is too specialized for existing code out there, then by all means create a new form of the wheel. Just don't code up things that could have been done for you already.

Using code you don't understand - One of the greatest pitfalls programmers can find themselves falling into is using code they simply don't understand. I think every now and then we grab a piece of code we see on the Internet and throw it into a project we are

working on to get us past that really tough problem. However, once you see that it works and your problem is solved, take a minute to go back and see WHY it works.

Never blindly use code without knowing at least the general principles involved. The reason you should never do this is three fold:

1. First, what if something in it doesn't work right and you are expected to fix it with thousands of angry users or millions of dollars on the line? It is better to understand it now than in a pressure cooker situation.
2. Perhaps it is buggy and you wouldn't know that if you didn't read the code through. Maybe the function was for calculating the distance between two points on a Cartesian plane and you are calculating distance on the Earth. This results in you being off on all your calculations.
3. This is the most important thing... YOU NEVER LEARN ANYTHING! If you just use a ton of code that you didn't write and you don't go through it to understand it, you are not going to learn why something works and that means you are missing chances to grow and be better.

Yes it takes a lot of effort to go back and read someone else's code and learn what it does, but this is time well spent. If you are asked to fix or enhance it, no problem, you know what it does. If you are asked to fix it NOW you are in a better position to fix it fast. By reading it you can also catch bugs that would have just gone unnoticed or spot when some code may not exactly work the way you want it to. You would rather have done this instead of at release time when suddenly you are off by a few hundred miles on your new mapping project calculations. Now you experience all these problems just because you used the wrong distance formula. To avoid this pitfall, just invest the time to study. It is as simple as that.

Accepting all the input a user gives "as is" - If you are an experienced programmer you are going to immediately know where I am going to go with this pitfall. Code injection. You should never be satisfied with input supplied by the user or even another computer system that you don't control. Users can be malicious; they could also be stupid and supply all sorts of crazy, highly sophisticated or harmful input to your system.

To avoid this pitfall, always validate the input coming into the system. Check the type of input it is (is the system expecting a number and a letter was given), the range if applicable (you are expecting a number between 1 and 100 and they gave you -3) and if it logically makes sense (you expected a sentence with normal letters and punctuation yet they are giving you something that contains a ton of symbols).

Many new programmers simply accept whatever the user gives the system because they are either unfamiliar with the types of data they might receive, don't know the limitations on input, or simply collect the input incorrectly. When practicing your skills, and your given programming language, take a minute to learn the correct ways to collect and validate input. By investing a little effort into this part of your learning you will solve many of the problems that plague software. It will help you avoid SQL injection to your database, it will avoid

hacks from damaging your software and it could prevent system dumps of passwords as well as other hazards.

Thinking you are done when you are never really done - As the last pitfall of this document I wanted to cover one that many early programmers fall into. Thinking they are done when really the job of a programmer is never done. Sure you may not work on a piece of code for years, but if you work for a company the chances are you will end up revisiting that code one or many times later. Write your code so that when you do revisit it, you can make as much sense of it now as the day you wrote it.

Software development is an iterative process and we are asked to maintain code that we have written and to add / remove features. Software grows and changes throughout time and that means we are never really done tinkering with it. Even if you leave the company, someone is probably going to end up picking up the torch and carrying it on into the future. Make it easy for them to do that. Never fall into the trap of thinking that you will never see your code again!

Conclusion

I hope you found this short guide useful in covering some of the great habits you should take with you as you embark on your future career. We also covered some of the pitfalls that programmers of all skill levels fall into. By knowing both the great habits you should adopt, and the pitfalls to avoid, you can quickly gain the skill set needed to take your coding and programs to the next level. You will have learned how to adapt quickly, avoid the rookie mistakes in front of your boss and walk the path to a glorious and accomplished future.

Be sure to refer to these habits and pitfalls as you work your way through the Programmers Idea Book that accompanies this guide. Knowing these from the start will give you the jump on what took me 16+ years of experience to find out. Lucky you.