Exercise 27. Creative and Defensive Programming

You have now learned most of the basics of C programming and are ready to start becoming a serious programmer. This is where you go from beginner to expert, both with C and hopefully with core computer science concepts. I will be teaching you a few of the core data structures and algorithms that every programmer should know, and then a few very interesting ones I've used in real software for years.

Before I can do that, I have to teach you some basic skills and ideas that will help you make better software. Exercises 27 through 31 will teach you advanced concepts, featuring more talking than coding. After that, you'll apply what you've learned to make a core library of useful data structures.

The first step in getting better at writing C code (and really any language) is to learn a new mind-set called *defensive programming*. Defensive programming assumes that you are going to make many mistakes, and then attempts to prevent them at every possible step. In this exercise, I'm going to teach you how to think about programming defensively.

The Creative Programmer Mind-Set

It's not possible to show you how to be creative in a short exercise like this, but I will tell you that creativity involves taking risks and being open-minded. Fear will quickly kill creativity, so the mind-set I adopt, and many programmers copy, is that accidents are designed to make you unafraid of taking chances and looking like an idiot. Here's my mind-set:

- I can't make a mistake.
- It doesn't matter what people think.
- Whatever my brain comes up with is going to be a great idea.

I only adopt this mind-set temporarily, and even have little tricks to turn it on. By doing this, I can come up with ideas, find creative solutions, open my thoughts to odd connections, and just generally invent weirdness without fear. In this mind-set, I'll typically write a horrible first version of something just to get the idea out.

However, when I've finished my creative prototype, I will throw it out and get serious about making it solid. Where other people make a mistake is carrying the creative mind-set into their implementation phase. This then

leads to a very different, destructive mind-set: the dark side of the creative mind-set:

- It’s possible to write perfect software.
- My brain tells me the truth, and it can’t find any errors: I have therefore written perfect software.
- My code is who I am and people who criticize its perfection are criticizing me.

These are lies. You will frequently run into programmers who feel intense pride about what they’ve created, which is natural, but this pride gets in the way of their ability to objectively improve their craft. Because of this pride and attachment to what they’ve written, they can continue to believe that what they write is perfect. As long as they ignore other people’s criticism of their code, they can protect their fragile egos and never improve.

The trick to being creative *and* making solid software is the ability to adopt a defensive programming mind-set.

The Defensive Programmer Mind-Set

After you have a working, creative prototype and you’re feeling good about the idea, it’s time to switch to being a defensive programmer. The defensive programmer basically hates your code and believes these things:

- Software has errors.
- You aren’t your software, yet you’re responsible for the errors.
- You can never remove the errors, only reduce their probability.

This mind-set lets you be honest about your work and critically analyze it for improvements. Notice that it doesn’t say *you* are full of errors? It says your *code* is full of errors. This is a significant thing to understand because it gives you the power of objectivity for the next implementation.

Just like the creative mind-set, the defensive programming mind-set has a dark side, as well. Defensive programmers are paranoid, and this fear prevents them from ever possibly being wrong or making mistakes. That’s great when you’re trying to be ruthlessly consistent and correct, but it’s murder on creative energy and concentration.

The Eight Defensive Programmer Strategies

Once you’ve adopted this mind-set, you can then rewrite your prototype and follow a set of eight strategies to make your code as solid as possible.

While I work on the real version, I ruthlessly follow these strategies and try to remove as many errors as I can, thinking like someone who wants to break the software.

Never Trust Input Never trust the data you're given and always validate it.

Prevent Errors If an error is possible, no matter how probable, try to prevent it.

Fail Early and Openly Fail early, cleanly, and openly, stating what happened, where, and how to fix it.

Document Assumptions Clearly state the pre-conditions, post-conditions, and invariants.

Prevention over Documentation Don't do with documentation that which can be done with code or avoided completely.

Automate Everything Automate everything, especially testing.

Simplify and Clarify Always simplify the code to the smallest, cleanest form that works without sacrificing safety.

Question Authority Don't blindly follow or reject rules.

These aren't the only strategies, but they're the core things I feel programmers have to focus on when trying to make good, solid code. Notice that I don't really say exactly how to do these. I'll go into each of these in more detail, and some of the exercises will actually cover them extensively.

Applying the Eight Strategies

These ideas are all as great pop-psychology platitudes, but how do you actually apply them to working code? I'm now going to give you a set of things to always do in this book's code that demonstrates each one with a concrete example. The ideas aren't limited to just these examples, so you should use these as a guide to making your own code more solid.

Never Trust Input

Let's look at an example of bad design and better design. I won't say good design because this could be done even better. Take a look at these two functions that both copy a string and a simple `main` to test out the better one.

`ex27_1.c`

Click here to view code image

```
#undef NDEBUG
#include "dbg.h"
#include <stdio.h>
#include <assert.h>

/*
 * Naive copy that assumes all inputs are always valid
 * taken from K&R C and cleaned up a bit.
 */
void copy(char to[], char from[])
{
    int i = 0;

    // while loop will not end if from isn't '\0' terminated
    while ((to[i] = from[i]) != '\0') {
        ++i;
    }
}

/*
 * A safer version that checks for many common errors using the
 * length of each string to control the loops and termination.
 */
int safercopy(int from_len, char *from, int to_len, char *to)
{
    assert(from != NULL && to != NULL && "from and to can't be NULL");
    int i = 0;
    int max = from_len > to_len - 1 ? to_len - 1 : from_len;

    // to_len must have at least 1 byte
    if (from_len < 0 || to_len <= 0)
        return -1;

    for (i = 0; i < max; i++) {
        to[i] = from[i];
    }

```

```
 38        to[to_len - 1] = '\0';
 39
 40        return i;
 41    }
 42
 43    int main(int argc, char *argv[])
 44    {
 45        // careful to understand why we can get
these sizes
 46        char from[] = "0123456789";
 47        int from_len = sizeof(from);
 48
 49        // notice that it's 7 chars + \0
 50        char to[] = "0123456";
 51        int to_len = sizeof(to);
 52
 53        debug("Copying '%s':%d to '%s':%d", from,
from_len, to, to_len);
 54
 55        int rc = safercopy(from_len, from, to_len,
to);
 56        check(rc > 0, "Failed to safercopy.");
 57        check(to[to_len - 1] == '\0', "String not
terminated.");
 58
 59        debug("Result is: '%s':%d", to, to_len);
 60
 61        // now try to break it
 62        rc = safercopy(from_len * -1, from, to_len,
to);
 63        check(rc == -1, "safercopy should fail
#1");
 64        check(to[to_len - 1] == '\0', "String not
terminated.");
 65
 66        rc = safercopy(from_len, from, 0, to);
 67        check(rc == -1, "safercopy should fail
#2");
 68        check(to[to_len - 1] == '\0', "String not
terminated.");
 69
 70        return 0;
 71
 72    error:
 73        return 1;
 74    }
```

The `copy` function is typical C code and it's the source of a huge number of buffer overflows. It's flawed because it assumes that it will always receive a valid, terminated C string (with `'\0'`), and just uses a `while-loop` to process it. Problem is, to ensure that is incredibly difficult, and if it's not handled right, it causes the `while-loop` to loop infinitely. *A cornerstone of writing solid code is never writing loops that can possibly loop forever.*

The `safercopy` function tries to solve this by requiring the caller to give the lengths of the two strings it must deal with. By doing this, it can make certain checks about these strings that the `copy` function can't. It can check that the lengths are right, and that the `to` string has enough space, and it will *always* terminate. It's impossible for this function to run on forever like the `copy` function.

This is the idea behind never trusting the inputs you receive. If you assume that your function is going to get a string that's not terminated (which is common), then you can design your function so that it doesn't rely on it to work properly. If you need the arguments to never be `NULL`, then you should check for that, too. If the sizes should be within sane levels, then check that. You simply assume that whoever is calling you got it wrong, and then try to make it difficult for them to give you another bad state.

This extends to software you write that gets input from the external universe. The famous last words of the programmer are, "Nobody's going to do that." I've seen them say that and then the *next day* someone does exactly that, crashing or hacking their application. If you say nobody is going to do that, just throw in the code to make sure they simply can't hack your application. You'll be glad you did.

There is a diminishing return on this, but here's a list of things I try to do in all of the functions I write in C:

- For each parameter, identify what its preconditions are, and whether the precondition should cause a failure or return an error. If you are writing a library, favor errors over failures.
- Add `assert` calls at the beginning that check for each failure precondition using `assert(test && "message");`. This little hack does the test, and when it fails, the OS will typically print the `assert` line for you that includes that message. This is very helpful when you're trying to figure out why that `assert` is there.
- For the other preconditions, return the error code or use my `check` macro to give an error message. I didn't use `check` in this example

since it would confuse the comparison.

- Document *why* these preconditions exist so that when a programmer hits the error, he or she can figure out if they're really necessary or not.
- If you're modifying the inputs, make sure that they are correctly formed when the function exits, or abort if they aren't.
- Always check the error codes of functions you use. For example, people frequently forget to check the return codes from `fopen` or `fread`, which causes them to use the resources the return codes give despite the error. This causes your program to crash or open an avenue for an attack.
- You also need to be returning consistent error codes so that you can do this for all of your functions. Once you get in this habit, you'll then understand why my `check` macros work the way they do.

Just doing these simple things will improve your resource handling and prevent quite a few errors.

Prevent Errors

In response to the previous example, you might hear people say, "Well, it's not very likely someone will use `copy` wrong." Despite the mountain of attacks made against this very kind of function, some people still believe that the probability of this error is very low. Probability is a funny thing because people are incredibly bad at guessing the probability of any event. People are, however, much better at determining if something is *possible*. They might say the error in `copy` is not `probable`, but they can't deny that it's `possible`.

The key reason is that for something to be probable, it first has to be possible. Determining the possibility is easy, since we can all imagine something happening. What's not so easy is determining its probability after that. Is the chance that someone might use `copy` wrong 20%, 10%, or 1%? Who knows? You'd need to gather evidence, look at rates of failure in many software packages, and probably survey real programmers about how they use the function.

This means, if you're going to prevent errors, you still need to try to prevent what's possible but first focus your energies on what's most probable. It may not be feasible to handle all of the possible ways your software can be broken, but you have to attempt it. But at the same time, if you don't constrain your efforts to the most probable events, then you'll be

wasting time on irrelevant attacks.

Here's a process for determining what to prevent in your software:

- List all the possible errors that can happen, no matter how probable (within reason, of course). No point listing "aliens sucking your memories out to steal your passwords."
- Give each possible error a probability that's a percentage of the operations that can be vulnerable. If you are handling requests from the Internet, then it's the percentage of requests that can cause the error. If they are function calls, then it's what percentage of function calls can cause the error.
- Calculate the effort in number of hours or amount of code to prevent it. You could also just give an easy or hard metric, or any metric that prevents you from working on the impossible when there are easier things to fix still on the list.
- Rank them by effort (lowest to highest), and probability (highest to lowest). This is now your task list.
- Prevent all of the errors you can in this list, aiming for removing the possibility, then reducing the probability if you can't make it impossible.
- If there are errors you can't fix, then document them so someone else can fix them.

This little process will give you a nice list of things to do, but more importantly, keep you from working on useless things when there are other more important things to work on. You can also be more or less formal with this process. If you're doing a full security audit, this will be better done with a whole team and a nice spreadsheet. If you're just writing a function, then simply review the code and scratch these out into some comments. What's important is that you stop assuming that errors don't happen, and you work on removing them when you can without wasting effort.

Fail Early and Openly

If you encounter an error in C you have two choices:

- Return an error code.
- Abort the process.

This is just how it is, so what you need to do is make sure the failures happen quickly, are clearly documented, give an error message, and are

easy for the programmer to avoid. This is why the `check` macros I've given you work the way they do. For every error you find, it prints a message, the file and line number where it happened, and forces a return code. If you just use my macros, you'll end up doing the right thing anyway.

I tend to prefer returning an error code to aborting the program. If it's catastrophic, then I will, but very few errors are truly catastrophic. A good example of when I'll abort a program is if I'm given an invalid pointer, as I did in `safercopy`. Instead of having the programmer experience a segmentation fault explosion somewhere, I catch it right away and abort. However, if it's common to pass in a NULL, then I'll probably change that to a `check` instead so that the caller can adapt and keep running.

In libraries, however, I try my hardest to *never* abort. The software using my library can decide if it should abort, and I'll typically abort only if the library is very badly used.

Finally, a big part of being open about errors is not using the same message or error code for more than one possible error. You typically see this with errors in external resources. A library will receive an error on a socket, and then simply report "bad socket." What they should do is return the error on the socket so that it can be properly debugged and fixed. When designing your error reporting, make sure you give a different error message for the different possible errors.

Document Assumptions

If you're following along and using this advice, then what you're doing is building a contract of how your functions expect the world to be. You've created preconditions for each argument, you've handled possible errors, and you're failing elegantly. The next step is to complete the contract and add invariants and postconditions.

An invariant is a condition that must be held true in some state while the function runs. This isn't very common in simple functions, but when you're dealing with complex structures, it becomes more necessary. A good example of an invariant is a condition where a structure is always initialized properly while it's being used. Another example would be that a sorted data structure is always sorted during processing.

A postcondition is a guarantee on the exit value or result of a function running. This can blend together with invariants, but this is something as simple as "function always returns 0 or -1 on error." Usually these are

documented, but if your function returns an allocated resource, you can add a postcondition that checks to make sure it's returning something, and not NULL. Or, you can use NULL to indicate an error, so that your postcondition checks that the resource is deallocated on any errors.

In C programming, invariants and postconditions are usually used more in documentation than actual code or assertions. The best way to handle them is to add `assert` calls for the ones you can, then document the rest. If you do that, when people hit an error they can see what assumptions you made when writing the function.

Prevention over Documentation

A common problem when programmers write code is that they will document a common bug rather than simply fix it. My favorite is when the Ruby on Rails system simply assumed that all months had 30 days. Calendars are hard, so rather than fix it, programmers threw a tiny little comment somewhere that said this was on purpose, and then they refused to fix it for years. Every time someone would complain, they would bluster and yell, "But it's documented!"

Documentation doesn't matter if you can actually fix the problem, and if the function has a fatal flaw, then just don't include it until you can fix it. In the case of Ruby on Rails, not having date functions would have been better than including purposefully broken ones that nobody could use.

As you go through your defensive programming cleanups, try to fix everything you can. If you find yourself documenting more and more problems you can't fix, then consider redesigning the feature or simply removing it. If you *really* have to keep this horribly broken feature, then I suggest you write it, document it, and then find a new job before you are blamed for it.

Automate Everything

You are a programmer, and that means your job is putting other people out of jobs with automation. The pinnacle of this is putting yourself out of a job with your own automation. Obviously, you won't completely eliminate what you do, but if you're spending your whole day rerunning manual tests in your terminal, then your job isn't programming. You are doing QA, and you should automate yourself out of this QA job that you probably don't really want anyway.

The easiest way to do this is to write automated tests, or unit tests. In this book I'm going to get into how to do this easily, but I'll avoid most of the

dogma about when you should write tests. I'll focus on how to write them, what to test, and how to be efficient at the testing.

Here are common things programmers fail to automate when they should:

- Testing and validation
- Build processes
- Deployment of software
- System administration
- Error reporting

Try to devote some of your time to automating this and you'll have more time to work on the fun stuff. Or, if this is fun to you, then maybe you should work on software that makes automating these things easier.

Simplify and Clarify

The concept of simplicity is a slippery one to many people, especially smart people. They generally confuse comprehension with simplicity. If they understand it, clearly it's simple. The actual test of simplicity is comparing something with something else that could be simpler. But you'll see people who write code go running to the most complex, obtuse structures possible because they think the simpler version of the same thing is dirty. A love affair with complexity is a programming sickness.

You can fight this disease by first telling yourself, "Simple and clear is not dirty, no matter what everyone else is doing." If everyone else is writing insane visitor patterns involving 19 classes over 12 interfaces, and you can do it with two string operations, then you win. They are wrong, no matter how elegant they think their complex monstrosity is.

Here's the simplest test of which function is better:

- Make sure both functions have no errors. It doesn't matter how fast or simple a function is if it has errors.
- If you can't fix one, then pick the other.
- Do they produce the same result? If not, then pick the one that has the result you need.
- If they produce the same result, then pick the one that either has fewer features, fewer branches, or you just think is simpler.
- Make sure you're not just picking the one that is most impressive. Simple and dirty beats complex and clean any day.

You'll notice that I mostly give up at the end and tell you to use your

judgment. Simplicity is ironically a very complex thing, so using your taste as a guide is the best way to go. Just make sure that you adjust your view of what's "good" as you grow and gain more experience.

Question Authority

The final strategy is the most important because it breaks you out of the defensive programming mind-set and lets you transition into the creative mind-set. Defensive programming is authoritarian and can be cruel. The job of this mind-set is to make you follow rules, because without them you'll miss something or get distracted.

This authoritarian attitude has the disadvantage of disabling independent creative thought. Rules are necessary for getting things done, but being a slave to them will kill your creativity.

This final strategy means you should periodically question the rules you follow and assume that they could be wrong, just like the software you are reviewing. What I will typically do is go take a nonprogramming break and let the rules go after a session of defensive programming. Then I'll be ready to do some creative work or more defensive coding if I need to.

Order Is Not Important

The final thing I'll say on this philosophy is that I'm not telling you to do this in a strict order of "CREATE! DEFEND! CREATE! DEFEND!" At first you might want to do that, but I'd actually do either in varying amounts depending on what I wanted to do, and I might even meld them together with no defined boundary.

I also don't think one mind-set is better than another, or that there's a strict separation between them. You need both creativity and strictness to do programming well, so work on both if you want to improve.

Extra Credit

- The code in the book up to this point (and for the rest of it) potentially violates these rules. Go back and apply what you've learned to one exercise to see if you can improve it or find bugs.
- Find an open source project and give some of the files a similar code review. Submit a patch that fixes a bug.