24 Patterns for Clean Code

Techniques for Faster, Safer Code with Minimal Debugging

ROBERT BEISERT

Techniques for Faster, Safer Code with Minimal Debugging

By Robert Beisert

Published January 2016

Introduction

When I was a child, I thought I could do any number of things. Before I could talk, I fancied that I was really good at running. In kindergarten, I was proud of my penmanship and math skills. In elementary school, I honestly thought I was an excellent video gamer.

However, I was not that good at any of those things.

It’s not like it was exactly my fault. After all, I was still new at these activities. I had not received the training and the merciless beatings that life throws at us to make us better.

The same rule applies to programming.

Early on, I became aware that there *had* to be better ways to do things. My code was unruly and difficult to read, much less debug. I didn’t know how some relatively simple things worked. I made the same mistakes again and again. But I learned.

This book contains some relatively basic approaches for programming in languages such as C. In fact, although I preferentially work in C, these patterns demonstrate how to accomplish most tasks faster and easier in any language.

I strongly suggest that you skim this work once, then go back to study the details. None of these are difficult, but all are essential for optimal programming.

1. Simplifying Your Work

Document as you go

Ask any programmer what the most irritating part of programming is, and they'll either say "debugging" or "documentation". Admittedly, I am one of the rare exceptions (I dislike the "coding" part most, because it's less about solving a problem than it is translating the solution), but I have a standard which removes the most irritating factors of documentation.

The Classic Method

During the programming phase, code authors sit together and draft up diagrams and flowcharts for how their code should function. They then sit down at their machines and, over a course of weeks and countless modifications, produce some quality code.

After they're done writing the code, they go off to do more coding. Because they're programmers, this is their natural state.

At some point, however, they have to sit down and create documentation which details the operations of their functions, the code's basic assumptions, and requirements on the end-users.

The thing is, at this point, **even the authors don't remember what the code does, or what the assumptions are**. Even if they are documenting the code right after cleaning it up, there are functions whose purposes are now essentially erased from memory.

This means that the authors have to sit down, re-read their code, figure out the assumptions and operations again, and remember what the requirements are.

What a long, boring waste of a programmer's time.

Documenting as you go

First, we have to remember a vital documentation rule: **Everything which is related to the code is documentation.** All your comments, graphs, doodles, and idle musings are valuable insights into the mind of the programmer which constructed this code (as opposed to the programmer who is currently reading the code).

I highly recommend that, as you write your code, you place useful comments and tags in there. Something as simple as "THIS BLOCK IS FULL OF VARIABLES WE NEED FOR THIS OPERATION" can save you minutes of inquiry, while longer comments detailing the flowchart for a function (or subfunction) can save you hours. Also, if you use meaningful naming conventions and meaningful cross-referencing, you can skip around your code in

seconds, connecting the spaghetti into a single whole.

Of greatest importance to the end-user, **always** document your function's purpose, return types, and arguments before you even start fleshing out the details. This will spare you confusion (as you never let the complexity of the function spiral beyond the defined purpose), and it will allow you to create your final documentation almost effortlessly.

NATURALLY, IF YOU MAKE FUNDAMENTAL CHANGES TO A FUNCTION, YOU HAVE TO UPDATE YOUR DOCUMENTATION. IT MAY STILL BE WORTH LEAVING SOME OF THE OLD CODE COMMENTED OUT, JUST SO YOU HAVE EVIDENCE OF A USEFUL PATTERN OR IDEA.

Doxygen

I am a tremendous fan of the doxygen program. This program interprets declarations and specially-formatted comments to generate HTML documentation, complete with hyperlinks, graphs, and flow-charts. With this tool, you can rapidly convert useful source-code commentary into powerful documentation.

The format is extremely minimal. All comments surrounded by /** … **/ blocks are interpreted by Doxygen, along with all in-line comments starting with ///. This means that a single-character addition in a regular comment results in a doxygen comment.

The tag system in doxygen is also simple. Every line beginning with a particular tag will produce a section of meaningful output in the final documentation. These tags include (but are not limited to):

- @param – Describes a function input parameter
    - @param[in] – Declares the parameter to be input only
    - @param[out] – Declares the parameter to be output only
    - @param[inout] – Declares that the parameter is read in, transformed, and passed back out
- @return – Describes the expected return value(s)
- @author – Lists a name as the author of a file, function, etc.
- @brief – Declares that the line which follows is a brief description of the function
- @detail – Declares that everything up to the next tag is part of a detailed function description

We also can use ///< to describe enumerations, objects within a structure, etc.

It is a tremendously powerful tool which encourages ample in-source commentary and provides rapid documentation on demand. For more information, check out the **Doxygen user's manual**.

Lesson: Documenting as you go improves the accuracy of your documentation, reduces the time it takes to generate documentation, and allows the programmer to spend more time on more valuable (or interesting) tasks.

Treat your documentation as a contract

When you commit to a set of documentation, you are telling all the users how your code will work and how they should use it. This has a trickle-down effect, such that any changes to the documented features can destroy entire suites of programs.

Violation of the Rule

Suppose you provide documentation for an early-release form of your code. Other teams and other companies start working your function calls and structures into their code. Perhaps these new codes are built into libraries, which are used to build large distributables.

Then you add just one parameter to one major function. It’s mostly harmless, right?

Unfortunately, you just broke everything.

Now the groups that were using your code have to go through the entire design process again. They have to update all of their code to match your new definitions, and then they have to test everything to make sure it still works. Your 15 second change results in tens (or even hundreds) of hours of work across the board.

Adherence to the Rule

There are several ways we can preserve the documentation contract.

1. **Employ modular design principles**. All additions to the documentation are new modules, while the old modules remain intact. This allows your end-users to decide whether to accept your changes or not.
2. **Hide what you’re doing.** If you leave the function interfaces alone, you can make changes to the functionality without necessarily informing the end-users. This only really applies with runtime libraries, though.
3. **Build from wide to narrow.** Just like a pyramid, you start with functions that take all the arguments you can possibly think of. Then, you tighten the focus with each new function until you have the minimal function. This pattern is extremely useful when you have default inputs but you still want to be able to change those defaults if desired.

Lesson: Your Documentation tells the world how to use your code. Make sure that you preserve the documented functionality and interfaces.

Use VI operations to keep .h and .c files consistent

Generally, I don't much care for IDEs. In my experience, IDEs add increasingly specific features to maintain their relevance, resulting in bloated software which takes too many resources to accomplish a relatively simple task.

However, as functions get more complex, it becomes increasingly difficult to make your code match the original prototypes. This is where Vi's "yank" and "mark" features truly shine.

Yank to Mark

We know that the vi equivalent to "copy-and-paste" is "yank-and-put". What many people do not realize is that you can open any number of files in a single vi instance, allowing you to yank from one file and put into another.

The final piece of this puzzle is the "mark" feature. It is possible for us to create a mark on a given line with the following command:

m<letter>

EXAMPLE: ma

We can then return to that line with the command:

‘<letter>

EXAMPLE: ‘a

This allows us to yank large sections of code from one place to another using the command:

y’<letter>

EXAMPLE: y’a

This is the "yank to mark" feature.

Application 1: Code Completion

We can use the yank-to-mark feature to copy entire function prototypes back into our new code. To do this, we first navigate to the spot where our function call will go. We can then go to the header file containing the required function prototype and yank it out. Finally, we "put" the prototype into our new code, then swap the prototype values out for our new values.

For example, suppose we have a prototype that looks something like this in a file FILE.H:

int function(

int Value_one,

char *Value_two,

struct astructure Value_three

)

If we want to call this function in main.c, we can open vi like so:

vi main.c file.h

We then yank everything from “int function(” to “)” using our “yank-to-mark” feature and put it into main.c.

At that point, it is a simple matter to replace the return value and all parameters with the values that we should pass into the function, all while ensuring that the types match.

Application 2: Headers match Code

For many programmers, it is a pain to ensure that your headers and code files match exactly. This is easily achieved with yank, however.

We can copy all the code from the start to the end of the code file using the command:

gg (go to start)

yG (yank to end of file)

We can then put everything from the code file directly into the header.

We can delete all the code inside our function prototypes using the “go-to-matching” command. We can highlight the start of a code block (marked by the { brace) and hit the % key to go directly to its matching closing brace (}). Thus, we can delete all the code using the command:

d%

Then, we can insert a semicolon to complete the function prototype.

Using this feature, we can create header files in seconds which EXACTLY match the code files to which they belong.

NOTE: IF YOU’RE USING DOXYGEN-STYLE COMMENTS, YOU SHOULD WRITE THEM DIRECTLY INTO YOUR CODE FILES. THIS WAY, WHEN YOU UPDATE THE FUNCTION DEFINITIONS IN THE CODE, YOU CAN PERFORM ANOTHER SIMPLE YANK TO REPLACE THE OLD PROTOTYPE WITHOUT HAVING TO ADJUST YOUR COMMENTS AGAIN. IT IS A REAL TIME-SAVER AND IT ALLOWS YOU TO SEE YOUR COMMENTS WHEREVER THE FUNCTION IS DECLARED.

Lesson: IDEs such as VI contain powerful tools while remaining minimally invasive. Learn to leverage these simple tools to speed up development.

Comments - your path back through history

Programmers are not unlike Dory from “Finding Nemo”, in that we have a limited memory. It seems like every time we sit down to code, and we are randomly struck by a lightning bolt of inspiration, we immediately lose it when we notice something shiny.

That’s an extreme example, sure, but programmers do have a problem of forgetfulness. It’s the nature of the job – we have so many tasks to perform and so many paths to track that we can’t possibly hold on to all our thoughts.

Thank God for comments.

Level 0 Comments: In-line

This practice, so common to college-age programmers, is often lost quickly in the “real world”. However, these comments are perhaps the most useful comments we can have.

After all, which is faster: tracing all the calls and variables in a block of code, or reading a short sentence describing the intended function?

While I generally recommend you not write useless comments (“This printf() function prints a line of text”), there are several key things you can do:

- Outline your code BEFORE you start writing, then use that outline to walk your way back through it later
- Explain why you chose one function over another
- Describe the operation of a library-based function, so you don’t have to keep looking it up
- Leave TODO markers in your code (vi will highlight these specifically, so they’re easy to find again)
- Comment out a “bad” line, so that you can see what it did before the fix
- Leave some tag that’s easy to search for later (like TODO, only without the convenient highlighting)
- etc.

All of these comments **improve readability** by restoring some of the mindset you had when you were writing the code in the first place.

Level 1 Comments: Flowerboxes

Less common, but equally important, are flowerbox comments. These comments allow the author of a piece of code to relay more detailed information in a compact, highly-visible

format.

There are a number of uses for flowerboxes:

- **Doxygen comments** – these not only generate HTML documentation, but they also describe a function's purpose, arguments, and return types inside of the code itself
    - I cannot recommend Doxygen-style commentary enough
    - Seriously, if you haven't looked into it before, **LOOK IT UP**
- **Flow descriptions** – these comments describe a higher-level flow for a function or program, allowing the programmer to quickly get a deeper sense of how the program is supposed to work
- **Disclaimers** and **Formalities** – Want the world to know who designed the code, and what it's for? Flowerboxes at the top of the page get it done
- Detail an **event** or **conversation** relevant to the code – Maybe an offhand quote from a fellow programmer inspired the design of the next segment of code. Recording that information helps future programmers understand not just what the code is doing, but why you chose to do it that way

Level 2 Comments: Logs

Some of my more recent work contains fewer comments than I usually employ. This is because, instead of using inline commentary to describe an event, I print out a string detailing what is supposed to come next.

These are still basically comments, because they serve the purpose of a comment while providing information during runtime. It's a win-win.

Level 3 Comments: Code Segments

Sometimes (usually) we decide to replace whole sections of code with new code. However, when we do a delete-and-replace, we run the risk of damaging functionality with no way to roll back the source.

Using **flowerboxes** or **#if** statements, we can make sure that the old code is safely kept away from the final product while allowing us to restore that functionality if the time comes.

Also, it's interesting to see how the code has evolved over time.

Level 4 Comments: Extra Documentation

Strictly speaking, everything that happens during the development of a piece of code should be documented. All conversations, whiteboard diagrams, emails, flowcharts, and other documents should be retained, if only so you can see what decisions were made.

Lesson: We put comment features into our languages for a reason. Use them liberally to spare everyone a lot of time and effort.

Leave the standards alone - avoid changing defaults (aliases and/or variables)

Fun fact: more than a few code conventions are inspired by the fact that everyone's computer is set up differently.

Problem 1: Non-standard tab spaces

The standard UNIX/Linux tab standard is 8 spaces. That means that, to create a similar spacing to the default tab, you would have to hit the space bar eight times. This standard embraces the binary underpinning of computers, and it creates effective spacing comparable to a seven-character variable and a space.

However, for one reason or another, people have been changing that standard value.

I've seen everything from 2 spaces to 7 spaces (although no one seems to go over 8), but rarely do I see anyone employing the default tabs.

This makes it harder to write "pretty" code, because notation that lines up nicely on one machine looks chaotic on another machine.

Of course, you could always destroy your space bar to avoid tabs altogether, but I argue it'll never be worth it. At worst, it makes your code harder to modify without being really obvious, and at best it takes time that you could spend anywhere else.

BESIDES, IF YOU NEED TO REDUCE THE TAB SIZE TO MAKE YOUR CODE FIT ON THE LINE, YOU PROBABLY NEED TO SERIOUSLY REEVALUATE YOUR CODING PRACTICES.

Problem 2: Aliasing

Most beginning programmers are unaware that the UNIX/Linux systems allow you to assign an alias (new name) to a function or set of commands. These are stored in the system instead of in a code file.

The problem here should be obvious: **the other guy's computer lacks your alias**.

This usually trips up programmers when they are helping out a friend, only to find that their aliases don't work.

Generally speaking, don't use aliases unless you can comfortably operate without them.

Problem 3: Other

There are any number of customization options on machines these days, and most of them are harmless. At the end of the day, it usually doesn't matter to me whether the machine has a GNOME, X, or KDE environment. I can even work with VI when someone's fiddled

with all the highlighting options.

However, when you start fussing with anything non-standard (libraries, programs, etc.) , you make it harder for everyone else to replicate what you're doing. In a corporate or open-source sense, that's disastrous.

The Solution: STANDARDS

I'd argue that the best standards are usually those that come with the system, but there are other ways to make it work.

- Issue everyone the same distribution, with standardized libraries and all
- Make everyone work off of virtual machines, which are synched on a daily basis
- All agree to a standard
- Work in terminals without access to the variables required to change the standards

Personally, I'm not a fan. Just make sure that your scripts clearly dictate what non-standard programs and libraries they require, and we should all be fine.

And don't change the tab space. Eight is great: four is poor.

Lesson: Standards allow everyone to stay on the same page.

2. Debugging and Maintenance

C “classes” using naming conventions

Encapsulation is a key aspect of the Object Oriented Programming model, and for good reason. We have an easier time grasping units which do one thing than hundreds of independent operations and objects that can do a number of things. While C does not contain the “class” descriptor found in C++ or Java, we can accomplish a similar task with naming conventions.

Principles of Classes

Before we consider a structure for naming classes, we have to consider the meaning of “class”.

The highest-level observation about classes is that they **act as a unit**. Specifically, a class consists of a structure and a set of functions that operate on that structure. Of course, the structure is the key root of the class, because all the functions address that structure in one way or another.

Second, we can observe that classes in Java are relegated to their OWN separate code files. This is due to a shortcut by the Java developers (who decided it was easier to prevent collisions by relegating classes to specifically named files), but it is applicable to all coding conventions. In compliance with the unitary nature of classes, it makes sense to keep all the functionality required for a class in a single file.

Finally, we observe that each class has its own functions (or “methods”) that work explicitly on that class object. These methods may not have globally unique names in the source code (which is a cause for complaint by those of us who employ Linux utilities to accelerate our programming), but the symbol table produced always contains globally unique names by prepending a host of unique identifying values, which may include class name, namespace, and (potentially) file name from which the function is derived.

These observations allow us to construct classes in C, only with much simpler dynamics.

Naming Convention

Let’s approach the naming convention using the pattern of our observations.

First, we note that classes are built around a structure. This structure should have a unique name (because all names in C must be unique). Furthermore, all functions which we would call “class methods” in an OOP language should be visually tied to that structure.

First: All functions related to a structure (“class”) must contain the name of that

structure.

Second, we note that all classes in Java are relegated to a single file. This is a reasonable practice.

Second: All class structures and functions must be contained in the same file.

Third, we observe that each symbol produced by the OOP compilers and interpreters is globally unique, regardless of the original name's uniqueness. We can apply that to our code by prepending an aspect of the file name directly to everything related to the class.

Third: All class structures and functions must begin with the name of the file in which they are contained (or a logical subset thereof, which is known to all users).

This results in a naming convention that looks something like this:

FileName_Structure

FileName_Structure_Function

If you code using typedef-ed pointers (which is useful for creating "private" data elements - you conceal the structures by omitting them from the .h file, restricting access only to defined accessor functions), you can use a format like this to immediately differentiate between pointers and structures:

typedef struct FileName_Structure FILENAME_STRUCTURE

Finally, if you're concerned about confusing variables which are passed-in with those employed or created internally, you can designate a naming convention like this to keep them safe:

function_variable

Lesson: We can use naming conventions to logically group functions and structures, establishing meaningful names while preserving global uniqueness.

Employ Meaningful Return Types

I don’t know how many hundreds of functions I’ve dealt with with either a **void** or **boolean** return type. While a boolean return type at least tells us whether the function ever completed properly, a void return carries absolutely no meaning. Where’s the debugging or error-handling value in that?

Constructors

Constructor functions take a number of arguments and return a pointer to an allocated space. Even so, there is a simple rule for meaningful returns:

Constructors return either a newly allocated pointer or NULL

Of course, if you work with a slightly more complex constructor, you can return the pointer in a parameter. In these cases, you should still make the constructor return something meaningful.

Personally, in these cases, I use a tri-state return type. If the function is successful, I’ll return the size of the allocated space (in bytes). If it has an error, I’ll return a negative value correlating to the type or location of the failure. However, if the function works EXCEPT that the malloc does not successfully allocate any space, I’ll return 0.

Simple Access Functions

A simple access function returns some meaningful data value from a structure. In these cases, we return the **value that we retrieved**.

Anything that can possibly fail (even when it can’t)

If it’s possible for a function to fail, we return that value inside of a parameter and use the basic rule:

Return 0 on success and non-zero on error

This basic rule applies generally, across the board. Even when the operation has no chance of failure (a state which is less common as I construct better code), we return the 0 value.

Debugging and Logging errors

As systems grow in complexity, the number of locations where an operation could fail increases.

Further, as the size of a function increases, the number of ways it could fail increases.

When we employ meaningful return types, we create a path directly to the area where the

problem occurred. So long as we know that function **panda** failed with an error value of -5, we know where the error was triggered (even if the system is thousands or millions of lines long). Even better, if we designed our return codes around specific tests, we know exactly **how** the function failed.

This means that, without ever touching the debugger, we have identified the location of our failure and can immediately begin determining the sequence of events that led to failure.

As Martha Stewart would say, “It’s a good thing”.

> **Lesson: We can use return values to give us insight into a function (specifically, how it fails). It also standardizes your interfaces somewhat.**

Use Meaningful Names

While it's usually less of a problem in C, in my Java days I saw any number of functions with names like "solve" or "act". These functions were usually overloaded, so that "solve" meant one thing for integers and a wholly different thing for strings.

Tell me, if you were reading a piece of code and you see a line like this:

```
solve(a, b, c)
```

How would you know what that function is doing?

Meaningful names

We've discussed defining function and object names which employ a CLASS_OBJECT_function structure. The problem is, there's little difference between the following Java and C:

```
Class.function
```

```
Class_object_function
```

In both of these cases, if the "function" name is just "solve", there's not a lot for us to go on.

Now, if we replaced the function name with something like "RSA_encrypt" or "DES_rotate", we have something meaningful. We now know that the function is supposed to (in the examples) encrypt some input using RSA or perform the rotate operation defined in the DES standard.

Now we know exactly what the function is supposed to do, so we don't have to go very far to determine whether it's working properly or not.

This goes further than simple object and function naming conventions. When you instantiate a variable, you run into the same problems.

Suppose you have a set of variables with names like these:

```
int thing;
```

```
char *stuff;
```

```
uint8_t num;
```

While these may be fine in small functions (because there's no room for confusion – you see it instantiated and employed in the same screen), they're problematic in larger and higher-level functions.

If your main function has variables like that, you don't know what they're supposed to represent. However, if they have names like these:

```
uint64_t RSA_256_key_pub[4];
```

```
int object_expected_size;
char *error_message;
```

Suddenly, we have an idea of what these variables do.

Admittedly, this increases the amount of time you spend typing (and my heart bleeds for you), but it reduces confusion while you're writing the code and makes maintenance hundreds of times easier.

Lesson: Well-named objects and functions make reading and designing code easier.

Always compile with -g

99 little bugs in the code

99 little bugs in the code

Take one down, patch it around

117 little bugs in the code

-Alex Shchepetilnikov, TWITTER Jul 2013

Experienced programmers have embraced the truth that, as the size of the code increases, the number of bugs increases exponentially. While we do our best to eliminate the most critical bugs (through patterns and careful debugging), we never catch them all.

Anyone who has ever played a modern AAA game can attest to this fact.

Debug tags during development

If you don't use the debug tag during development, your debugger is useless. Sure, you can still run down the biggest errors by watching the warnings and running the code with varied tests, but you handicap yourself.

It takes all of fifteen seconds to add -g to your makefiles' default compiler tags. The value you gain by preserving the symbol tables is too great to ignore.

Shipped code

So, you shipped your code. If you're a giant AAA game company like EA, you might be able to get away with ignoring the vast majority of error reports (because you're busy raking in money from your annual franchises), but the rest of us take some pride in our release code. More importantly, corporate programs need to work or the company takes on work, risk, and financial liability.

Have you ever tried to run a debugger over code without debug tags? There are no variable names, the registers hold meaningless values, and you will spend more time trying to read the log outputs than you do fixing the problem.

Worse still, if you try to recompile with debug tags, you might eliminate or shift the problem. This means that you will not successfully debug the problem, and you may well introduce new bugs into the code.

This is why it's valuable to release code with the debug tags intact.

When an error report comes back with all of your (highly detailed) logs, you can quickly recreate the issue in your debugger. Even better, you can set effective breakpoints and manipulate the variables such that you are now able to determine the source of the error and correct it, distributing a quick and (relatively) painless patch to eliminate the issue.

THERE ARE EXCEPTIONS. IF WE’RE DEALING WITH SECURE CODE, WE DO NOT WANT IT TO BE EASY FOR JUST ANYONE TO ACCESS THE INTERNALS AND MODIFY THEM. IN THESE CASES, WE CAN’T EMPLOY THIS PATTERN (WHICH MEANS WE HAVE TO DOUBLE-DOWN ON OUR LOG FILES TO MAKE UP FOR IT).

Lesson: Debugging does not end on release. Maintenance is easier when we leave debug tags available.

Log Messages - the more content, the better

Has this ever happened to you: a program crashes, and all you get is a black screen or a set of meaningless numbers?

I know it happens to me.

Even the better programs designed by, say, Microsoft, we get an error report containing numbers that mean absolutely nothing to human beings – it's designed for computers only.

And yet, Microsoft has basically the right idea.

Logs tell us what's happening

At its most basic, a **log** is a message that a program prints out for later reference. These messages can be as simple or complex as the programmer desires, but the programmer has to decide when these logs are printed and what they'll say.

Commonly (assuming you don't have your own log utilities), we use formatted text strings passed out to the standard error stream. These logs look a bit like:

```
fprintf(stderr, "%s:%d\tERROR HAPPENED HERE\n", __FILE__, __LINE__);
```

This log message consists of a few key parts:

- fprintf – a function that allows us to print to any stream we choose (files, output or input streams, etc)
- stderr – the standard error output stream
- A formatted string detailing the error information
    - __FILE__ is a macro that returns the name of the file in which this error is thrown
    - __LINE__ is a macro that returns the line number on which this error is written

With logs like these, we can rapidly pinpoint where errors have occurred and what kind of error has occurred.

Logs: For good times and bad

Most programmers only think of log messages when something goes wrong, but I argue that logs are essential in both paths of the tree:

- If it fails, you want to know where, why, and how it failed
- If it didn't fail, you want to know that it worked

- If it failed where you didn’t expect, you want to know that the “it worked” message didn’t happen

These logs allow the programmer to determine how much information he wants to see, and where in the code these messages should appear. As a result, the flow of information out of the program is determined by the sort of programmer who is likely to be maintaining the code in the future.

Lesson: Logs can provide a wealth of debugging and operations information, so long as the programmer designs that functionality into the code.

Build around Errors, not Successes

In 99.9% of cases, programmers spend most of their planning and initial coding creating the program that they want. They then spend twice as much time adding error-handling code to the original program, because they forgot that things can (and often do) go wrong.

This is “happy path” programming, or **success oriented code**.

Error orientation

We notice that, in happy path programming, there is exactly one path from start to finish. Because there are so few branches by default, **most programmers write code in a way that makes error handling more complicated.** The patterns they tend to employ include:

- Wedging all the functionality into one (main) function
- Assuming all functions terminate with perfect results
- Failing to create log outputs to track flow through the program

The problem with these patterns is that they’re unrealistic. Wedging all the functionality into the main function eliminates modularity, which reduces the efficacy of error checking. Assuming all functions are successful is a fool’s dream. And if you don’t create logs when things go right, you’ll never figure out how they went wrong.

The solution to all of this is to **design with failures in mind**.

Enforce modularity, because it limits the range of things that could go wrong at one time.

Check every input to every function, to make sure they aren’t nonsense (like writing 500 bytes to a single character pointer).

Use the if-else programming pattern to make functions internally modular as well as externally modular.

Create lots of logs. You want to keep records of everything that’s supposed to happen and everything that’s not supposed to happen. For every possible failure, attach a specific failure message that references the location of the failure in your code. Between these logs, you’ll know exactly what’s happening inside your program at all times.

If you create habits that revolve around all the things that can go wrong instead of what you expect to go right, you’ll vastly reduce your maintenance and debugging workload.

Lesson: Design your code with failures in mind. It saves you lots of time in the long run.

Employ the if-else chain

In languages like Java, we have a standardized error-handling paradigm in the try-catch expression. Effectively, this hands all error-handling off to the computer, which monitors all code in the try loop for any and every kind of possible error. While we are able to restrict the range of errors in many cases, the fact is that this bloats the program in both memory footprint and time required to operate.

In C, we have no such paradigm, so we have to use alternative methods to handle errors.

Common Practices

There are any number of ways to deal with errors outside of a rigidly defined paradigm.

Some (like those who operate predominantly in C++) may define supervisor functions that simulate the try-catch expression. This is less than common, because in defining the supervisor function you usually begin to appreciate the range of all possibilities. When you start trying to handle every possible error with one megafunction, you start to appreciate the simplicity of catching errors manually.

The most common practice is to test the output of functions and related operations. Whenever you call an allocation function like malloc() or calloc(), you test the output to ensure that space was properly allocated. When you pass-by-reference into a function, you test the inputs to ensure that they make sense in the context of your program. Methods like these allow us to manually handle both the flow and the error-handling of our code.

However, in most cases we have a “multiple-breakout” pattern of tests. These patterns look something like this:

```
char * blueberry = (char * ) malloc(50*sizeof(char))
if(blueberry == NULL)
  return -1;
int pancake;
do_thing(blueberry, “there is stuff”, pancake);
if(pancake < 0 || pancake > 534)
  return -2;
do_other_thing(pancake, time() );
if(pancake < 65536)
  return -3;
…
```

```
return 0;
```

This pattern runs the risk of terminating before memory is properly freed and parameters are properly reset. The only ways to avoid this terrible condition are to manually plug the cleanup into every error response (**terrible**) or to use goto (**not terrible, but not strictly kosher**).

If-Else Chain

There is one method for handling errors that is consistent with another pattern we cover (error-orientation): we build a great if-else chain of tests.

This pattern is confusing to many for two reasons:

- If fundamentally reorients the code away from the "**happy-case**" paradigm (in which all error-handling is a branch off the main path) to a "**failure case**" paradigm (in which the happy-case is the result of every test in the chain failing)
- All our happy-path code finds itself inside of an if() statement – nothing can be permitted to break the chain

It's a bit hard to describe this pattern without an example, so bear with me:

```
int copy(const char * const input, int size, char * output)
{
   int code = 0;
   if( input == NULL )
   {
    code = -1;
   }
   else if ( output = (char *) malloc (size * sizeof(char) ) , output == NULL )
   {
    code = -2;
   }
   else if ( strncpy(output, input, size), 0 )
   {
    //impossible due to comma-spliced 0
   }
```

```
    else if (strncmp(output, input))
    {
     code = -3;
    }
    else if ( printf("%s\n", output) < 0 )
    {
     //printf returns the number of printed characters
     //Will only be less than 0 if write error occurs
     code = -4;
    }
    else
    {
     //could do something on successful case, but can't think of what that would
be
    }
    //Normally we would pass output back, but let's just free him here for fun
    //This is where we do all our cleanup
    if(output != NULL)
    free(output);
    return code;
}
```

As we can see, each step in the error-handling function is treated as a possible error case, each with its own possible outcome. The only way to complete the function successfully is to have every error test fail.

Oh, and because this code is fundamentally modular, it is very easy to add and remove code by adding another else-if statement or removing one.

Lesson: Switching to an if-else chain can improve error awareness and accelerate your programs in operation, without requiring much additional time to design and code.

Check the boundaries (including nonsense)

A wise quote: “When it goes without saying, someone should probably say it.”

This is one of the better known patterns out there, but it still bears repeating.

The boundaries for integers

When you’re working with a range of integers (for example, human age range could be between 0 and 135), we have two obvious boundary points to look at: 0 and 135. However, if you really want to cover most of your gaps, you should appreciate that you have **six** points to consider:

- 0
- 0-1
- 0+1
- 135
- 135+1
- 135-1

We know that, ideally, everything between 0 and 135 (inclusive) should work properly, but on the off chance that you have an off-by-one somewhere in your code, it’s best to check the insides carefully. The same rule applies for everything outside of the range; you want to ensure that you catch an off-by-one early in the process by testing the points just outside of the expected range.

The boundary for characters

When we move on from numbers things get a bit more complicated, but generally speaking you want to ensure that your characters are inside the range of printable characters. This means that your code should be able to handle everything from 0x20 to 0x7E. However, just as with integers, this range provides us six boundaries to test:

- 0x19
- 0x20
- 0x21
- 0x7D
- 0x7E
- 0x7F

If you’re trying to print values, just ensure that the middle 4 conditions print properly and

the outer two do not. Everything else usually takes care of itself.

Extrapolating to Strings

The boundary rule says that we should check just inside and outside of the boundaries that show themselves, but how does that translate to complex values like strings?

With strings, we are usually either handling them character-by-character or as a unit. In the first case, you can use the character boundary rules, but in the second case the biggest thing we worry about is size. Here, we rely on the integer rule.

For example, if you have allocated a buffer of 25 characters, you should test that buffer with the following values:

- 24
- 25
- 26

Ideally, you have no problems with 24 or 25, but there is a chance that you forgot to clear the buffer, leaving an unexpected value inside your buffer on the next use. In this case, testing for 24 will ensure that you see and eliminate the issue.

For 26, we know that we're pushing beyond the boundary of the string. In this case, we want to ensure that we get some form of error handling response, assuring us that we've eliminated buffer overflows in this section of the code.

And it goes on

We can use the same rule for address ranges (which are dependent on your platform, but follow similar rules to string buffers), or for structure values, or for just about anything we ever do on a machine. It's a basic principle.

We should also note that, when we're dealing with robust tests, we want to test every possible boundary state. For example, for a simple line like this…

```
1 if ( i < 0 && j > 36)
```

…we should test every combination of the boundaries around i=0 and j=36. This ensures that we've covered all our bases, and nothing strange goes wrong.

Lesson: When testing our code, we should test successes and failures on either side of each defined boundary point. This ensures that we prevent some of the most common programming errors.

3. Code Conventions

Create in order, Destroy in Reverse Order, Mutex in same order every time

Virtually 100% of all memory leaks are preventable, assuming the programmer knows where to look. In many cases, the leaks occur because a programmer has failed to destroy what he has created, but it can become more complicated than that.

Creation Stack: The Order Matters

In many cases, we allocate memory as we use it, and we keep it at a simple layer. I've nearly lost track of how many calls to calloc() or malloc() I've hard-coded in the past few months alone.

This simple allocation is great, because it lets us dynamically generate variables and store them. These variables can be simple (like char or int), or they can be structures of varied complexity.

The most complex structures we allocate require us to allocate within the struct. Consider the following structure:

```
struct potato
{
struct potato *next;
char * string;
int size;
}
```

This structure contains not one, but two pointers, and we need these pointers to point to something. Generally speaking, we would want to create a simple constructor that does something like this:

```
struct potato * make_potato()
{
struct potato * hi = calloc(sizeof(potato));
return hi;
}
```

This would initialize a potato with null pointers and a size of 0. However, it's usually more

useful for us to fill in some values on construction of a structure like this:

```
struct potato * make_potato(int input_size, const char * something)
{
struct potato * hi = calloc(sizeof(potato));
hi->size = input_size;
hi->string = calloc(size * sizeof(char));
strncpy( hi->string, something, size);
return hi;
}
```

In this case, we not only allocate a structure, but we allocate within the structure. This can get messy very quickly.

Destruction: Follow it up the stack

The rule to prevent memory leaks here is fairly simple: every destructor should be the inverse of its constructor.

If the constructor is simple, the destructor is equally simple:

```
int eat_potato(struct potato * hi)
{
free(hi);
return 0;
}
```

However, when the constructor gets more complex, we have to treat it like a stack. That means that we should run every destructor in the inverse order of construction, so that we make sure to get everything freed before all pointers to it are lost.

```
int eat_potato(struct potato * hi)
{
free(hi->string);
free(hi);
return hi;
}
```

Lesson: You build a building from the ground up, and you tear it down from the top. The same goes for structures.

Yin and Yang - For every alloc, there is a free (and for every {, a })

How many programmers whine that C is dangerous because of memory leaks? How many programmers rely on IDEs because they can’t keep track of how many layers of brackets they’re using?

All of this stress is easily defeated with a little bit of Eastern Philosophy.

Yin: Beginnings

In the binary unifying force of the cosmos, there are the forces of yin and yang. These elements are equal complements to one another – where one exists, the other necessarily exists.

The element of Yin is the feminine element, and it is associated with beginnings and creation. For our purposes, the Yin is all things which begin a course of action.

These are the opening brackets and braces.

These are the allocation functions.

These are the function prototypes and “happy path” logic.

All of these things, which define and begin courses of action, are Yin.

Yang: Endings

Yang, on the other hand, is a masculine element associated with action, completion, and death. For our purposes, the Yang is everything that closes off a course or path of action.

These are the closing brackets and braces.

These are the free or destruction functions.

These are function definitions and error-handling paths.

All of these things are Yang.

Yin and Yang are One

In the Eastern philosophy of the Tao (literally, the “way”), yin and yang are balanced forces that continue in a circle for eternity. The elements pursue one another – destruction follows creation, and a new creation is born out of the old destruction. The masculine and feminine forces of the cosmos require one another for total balance and perfect harmony in the universe.

So, too, it is in our code.

When we create an opening brace, we must immediately define the closing brace, so that the balance is preserved.

When we allocate a resource, we must immediately define its time and place of destruction, so that balance is preserved.

When we prototype a function, we must immediately define its features, so that balance is preserved.

By keeping the two halves of our operations united, we ensure that we never have to chase down the imbalances between them.

Lesson: Creation and destruction are tied together, so we should define them at the same time.

Use const to prevent functions from changing the inputs

One of the key worries I have heard from those ill-informed OOP programmers is that C cannot protect inputs you pass into functions. They use private fields and retrieval functions to ensure that the stored value is protected from unwanted modification.

However, this concern is addressed by C's **const** keyword.

Taking a const value

There are times when we want to guarantee to all end users that our function will not modify particular values. Perhaps we are reading a stored key value in order to provide an access, but we want to ensure that nothing we do modifies that key.

If our function prototype and related documentation employ pass-by-reference (a common practice for large values) but do not employ the const keyword, the end user has no guarantee that the stored value will be the same after we're done.

The difference between…

```
int func ( char * key)
```

…and…

```
int func( const char * const key )
```

…is that the first interface has full control of the key, while the second promises not to change either the pointer or the values at that pointer.

Creating a const value

Often enough, we want to create a value that never changes throughout operation. Perhaps it's a static variable used to allocate arrays, or a static message we want to print out a number of times. In these cases, we use the **const** keyword to protect an initialized value from future changes. For example, we can create a constant integer value like this:

```
const int index = 256;
```

We can create a constant error message in the same way, but we usually make sure that we preserve both the pointer and the value stored therein:

```
const char * const error_message = "ERROR: Something done goofed\n";
```

NOTE: WE HAVE TO ASSIGN THE VARIABLE WHEN WE DECLARE IT, BECAUSE IT'S A CONSTANT VALUE THAT WE CAN NEVER CHANGE AGAIN.

Keyword const rules

The const keyword prevents anyone from modifying a value. However, when dealing with pointers we actually have **two** values to preserve: the value of the pointer, and the value at that pointer.

The keyword **const** can be seen to preserve whatever comes right after it. That means that the statement…

```
const char *
```

…protects the pointer itself, while the statement…

```
char * const
```

…protects the value at that pointer. To preserve both, we use…

```
const char * const
```

Dangerous, but useful: Casting with const

We can actually protect our values by casting them to a **const** value. For example, if we know we don't want a function to change something it can technically change (no const in the prototype), we can say something like this:

```
int func( char * key ) {}


char * value = “penny”;


int i = func( (const char * const) value );
```

However, we can also cast away the **const** (which is where it gets dangerous). That means that the program can act as though the value is naturally unprotected. That looks something like this:

```
int func( const char * const key )
{
```

```
  (char *) key = (char *) calloc( 50, sizeof(char)
);
}
```

Generally speaking, these actions are both considered unsafe. They can be extremely useful (for example, to free a const value inside of an allocated structure), but exercise extreme caution.

Lesson: The const keyword creates a contract in your code ensuring that the compiler protects the integrity of certain values.

Don’t mangle your iterators (iterators are used only to iterate)

Some patterns are more obvious than others. This is one of the more obvious patterns, but it’s violated often enough to deserve enunciation.

Iterator Abuse

Just about every program out there uses an iterator for one reason or another. Without them, we can’t build for loops, and more than a few while loops require iterators as well.

They let us perform simple operations on buffers and arrays.

They let us control flow of information.

Fundamentally, **an iterator is a control line**. It is essential that we maintain the integrity of our controls, because without that control we exponentially increase the complexity of the problem.

Iterator Abuse is the act of violating the integrity of an iterator, which destroys the line of control and makes the program act in complicated or unexpected ways. This abuse is performed in a number of ways:

- Reusing the iterator before its control function has been completed
- Modifying the iterator inside of the loop (usually by a non-standardized unit)
- Passing the iterator into a function without protection
- Using your only copy of a pointer as an iterator
- etc.

What can you do?

Iterator abuse is one of the more easily preventable issues in our programs. We have to adhere to a few simple rules:

1. Only use specially-marked values as iterators (the classic name is “i” or “j”)
2. Only use iterators for iteration
3. Iterate each value **only ONCE** per cycle (NO I++; I+=J)
4. In cases where we want to modify the iterator by a non-standardized unit (for example, by a number of bits equivalent to a structure), use

extreme caution

5. If iterating inside of a for loop, **never** modify the iterator
6. If iterating over a pointer, iterate over a copy of that pointer instead of the original
7. Either avoid non-standard loop termination (**break**, etc.) or avoid referencing the iterator outside of the loop
8. Don’t pass iterators into functions which might change the iterator value

Lesson: Iterators are oft-abused values. Remember that their purpose is to establish simple loops, and don’t go crazy with them.

Headers should not create object code

When we work in C, we have two basic file types: code and headers. Guess which one is supposed to contain our code?

Header Files

A header file is essentially an interface between a piece of source code and another piece of source code. We can think of it as a library index; it tells us where to find functions and what we need to do to reference them.

However, there is a big problem in the header world (especially for those who come over from C++): some programmers put code in the header file.

This is a problem for a couple of key reasons:

- It's logically inconsistent - headers and code are separate for a reason
- It eliminates library safety - we don't compile header files to object code, so we can't include that code in a library
- It extends the number of files we have to check - a problem can now occur in all headers (in addition to all code)
- It's just plain ugly

Generally speaking, we only put structure definitions, function prototypes, and **#define** macros in a header file. These things are 100% reference - they can't actually produce any code by themselves.

When we do this, we enforce an essentially modular design. The compiler uses headers to create "plug-ins" where library functions should go, and the linker plugs those functions in when the time is right. Everything does its job, and all is right with the universe.

Analogue: #include "*.c"

Very few programmers have ever compiled a source with a #include'd code file.

I have.

It's a dumb thing to do, because it complicates compilation and doesn't make linking any easier.

It's also just bad juju.

Lesson: Code goes in the code file. Headers only contain references to things in the code file.

Prototypes and Optimization - only convert when ready

The hardest interview I ever had: someone told me to go up to a whiteboard and solve a programming problem, in code, optimally, on the first try.

It's a basic fact of our field that we iterate toward the final product. Not unlike a sculptor bringing the DAVID out of a piece of marble, we go from a rough piece of code toward a refined piece of code in a series of stages.

Rough Outline

We usually start with a piece of pseudocode that generally describes what we want a function to do. This can be written in basically any format (text, outline, pseudocode, diagram, etc.), but it has to detail the basic elements we expect to find in our function.

Specifically, our outline serves these two purposes:

- Expose functionality for better breakdown
- Reveal the interface elements that we might want or need

If we skip this step entirely, we tend to spend a lot of time reworking our basic code as we discover features we wanted to include.

Exposed Interface with Dynamic Allocation

Next, we write the first draft of our code. At this stage, we are basically limited to the information provided by the rough outline.

Because we don't necessarily know what size our final data will be, we dynamically allocate space using malloc() or calloc(). This allows us to quickly modify the size and range of our data as we determine what elements we will include in the final product. This serves the secondary purpose of preserving our outline, as we have not yet optimized away extraneous variables required for simple reading.

Furthermore, because we don't yet know how many of our parameters will be standardized (through precompiler definitions or static variables), we want to take as many of them as we can into account. At this stage, our interfaces can be incredibly daunting, because they allow us to tweak EVERY parameter through the function call. We also want to pass pointers into these functions, but we're not sure whether to make them const, so there's a risk of silly errors.

NOTE: At this stage, we might start defining structures to carry these parameters, just to reduce the size of the interfaces.

Refined Interface with Dynamic Allocation

As we move on, we begin to determine the size of our parameters and the scope of our data. While we're not yet ready to jump to static allocation, we are becoming aware of what that statically allocated data might look like.

We are also ready to restrict the interface, as we have determined which parameters we intend to manipulate and which we will set to default.

There are two approaches to entering this stage:

- Rewrite your functions to restrict the interface to the core function
- Write wrapper functions to restrict the interface that the user deals with

Generally speaking, it's safer and easier to go with the latter.

Refined Interface with Static Allocation

Now we've reached the point where we basically know what our data will look like. We've made the tough calls, determined the final flow of the code, and settled on the size and types of data we'll employ at each level.

Now we begin replacing dynamic allocation with static allocation. If you've structured your interfaces properly, this is about as simple as replacing all -> with . and removing all allocation and free operations.

NOTE: Don't forget to clear the values in your static variables, especially if you were relying on calloc() to do that.

Minimal Interface with Static Allocation

Now we perform the true optimization stage.

Because we've finally settled exactly how our code will work, we can begin to restrict the inputs to our functions with **const** and similar keywords. This restricts our interfaces to the smallest and most restricted they can be, resulting in code that is reliable and easy for the end user to work with.

We also start to clean up our headers, removing any of the "old" function interfaces which are wrapped by the cleaner interfaces. This helps restrict the interfaces and prevents the end user from doing dangerous things.

We also start working with our variables. Depending on the level of optimization required, we start to do things like move things to global scope or reorganize code to reuse static variables (which I generally do not recommend if it makes the code harder to read).

This stage should produce a final set of code that you're happy to run and maintain, but there is another possible layer of optimization you can employ…

Optional Library Optimization

This is the sort of thing programmers had to do in the early days, to maximize utility of tiny systems.

Here we start reusing variables on the register level, optimizing out log code (and other valuable information), and generally render the code impossible to maintain.

COMPILING WITH OPTIMIZATION DOES MUCH OF THIS NATURALLY.

Generally speaking, this is only recommended for code you are virtually 100% certain you will never have to touch again.

Lesson: Change and uncertainty is law in the earlier stages of program development. Leave your early code very open for changes, and optimize as the project reaches final definition.

4. Thought Patterns

One Sentence

There is a law of problem solving which has been passed around since time immemorial. To Einstein, the law looked like this:

If you can’t explain it simply, you don’t understand it well enough.

However, this law has seen many refinements, and it came down to me in this form:

If you can’t explain it in one simple sentence, you don’t understand the problem.

While this appears to be an excessively strict rule to the outside observer, to the engineer it is the height of wisdom.

After all, every major program can be described in one sentence, which is the focus of that program:

- Excel – We need a program that automates database management.
- Word – We want a word processor that simulates an entire publishing house.
- Photoshop – We want a graphics program with all the tools available to the best painters.

So, too, can we describe every major function in a program with one simple sentence. After all, we ultimately want these functions to do one logical operation (in compliance with the unit principle). Thus, if you can’t explain a function in one simple sentence, you probably need to break it down some more.

The one-sentence rule is a rule for **clear, purposeful inquiry**. The simplicity of the structure requires you to boil your thoughts down, eliminating the unnecessary details and side-paths to reveal the heart of the matter. If you boil down a catastrophic error to something like:

“This function gives the same wrong answer every 8 or so passes”

you have the information you need to run down the bug. After all, you know what function is failing, you know that the failure is consistent, and you know that the bug occurs sporadically. This usually implies that you forgot to check something in the function, so you know basically how to find and fix your bug from that.

Lesson: Learn to boil your thoughts down to one simple, clear sentence.

Note: If you're trying to write a book, learn to reiterate that same sentence in a number of ways. If I was better at reiteration, my books would look more like Dummies guides or modern programming instruction manuals.

Employ the 3-10 rule

Most programmers are, at least to some degree, familiar with Top-Down design principles. The 3-10 rule applies human nature to the top-down design rules, creating a substantially improved method for project generation.

Top-Down Design Theory

Joke time!

Question: What's the best way to eat an elephant?

Answer: One bite at a time

I love these lateral-thinking jokes, because they defy the methods by which we usually approach problems. Generally, we think of eating an elephant as a lump sum, but this joke encourages us to look at the trivial problem before us: eat one bite.

Parents use the same technique to make kids eat their vegetables. Kids don't want to eat the whole portion, but if you make them take it one bite at a time they'll keep eating.

This is the principle of top-down design: **Break the complex problems down into a set of trivial problems.**

For example, if you wanted to write an email client from scratch, you would attempt to break it into parts. Each protocol would require its own code. The GUI needs to be its own code. Database management requires its own code. Thus, we reduce the impossible to a set of possible tasks.

7 +/- 3

Those who study psychology (whether professionally or as a hobby) notice that people cannot hold too many thoughts in active memory at once. If you've ever crammed for a test, you know that by the time you get halfway through the terms, you've "put away" the meanings of the first few terms. Meanwhile, you can instantly recall the definition for the last term you studied.

Hypnotists (who actively manipulate the conscious and unconscious mind) have a rule: people can only hold between 4 and 10 thoughts in their minds at once. There are several powerful hypnotic techniques that rely on this truth (specifically the "hypnotic blitz", which barrages the recipient with an idea until it sticks).

It's important for every engineer, designer, and programmer to remember this truth: **You can only generally hold 7 +/- 3 thoughts in your mind at once.**

Niklaus Wurth's Rule

I CANNOT CONFIRM THAT NIKLAUS WURTH EXPRESSED IT THIS WAY, BUT IT WAS HANDED DOWN TO ME WITH HIS NAME ATTACHED. I CAN CONFIRM THAT HE WAS A GENIUS, AND HIS WRITINGS ARE WORTH STUDYING.

How can we combine the top-down, stepwise refinement approach to design with our knowledge of human minds?

The Niklaus Wurth rule is actually an algorithm:

For each “lowest level” task on the stack:

If task is trivial (can be done with a short piece of simple code), ignore

If task is not trivial, break that task into 3-10 smaller tasks

Mark the smaller tasks as “lowest level” tasks

Recurse

Why 3-10 pieces, we ask?

If a task has only two trivial sub-components, we can consider the task trivial.

If a task has only two non-trivial sub-components, we are probably not thinking through the problem completely.

If we go above 10 pieces, we won’t be able to hold all those pieces in mind at once. That’s a great way to cause redundancies, accidentally forget portions, and substantially increase complexity.

Lesson: We can only handle so much complexity at once. We need to break the complex down in a way that we can understand, reducing the complex to the trivial.

Logical Unity of Form and Function

It’s amazing how often programmers forget the simplest rule of programming: 1=1.

This is the principle of logical unity (or modularity – they’re largely synonymous).

Unity of Function

If we adhere to the principle of top-down modular design, all our functions should be relatively small. At the bottom of the tree, every function serves one purpose and accomplishes one basic thing. As we climb up, each layer of functions simply combines the basic unit functions to perform a less-basic unit function.

As an example, consider the design of a simple server. Some programmers would design the server as a single block, where every operation is written directly into the main() loop. However, under the unity principle, the code should look more like this:

- main
    - Lookup
    - Initial connection
        - Open socket
        - handshake
            - exchange certificates
                - send certificate
                - receive certificate
            - check remote certificate
    - receive request
        - check buffer
        - deserialize
        - test
        - reply on failure
            - Serialize message
            - Attach header
            - Attach trailer
    - transmit
        - Retrieve data

        - Serialize message
        - Attach header
        - Attach trailer
    - disconnect
        - protocol based – close socket

Unity of Form

Unity of form describes the modularity of objects. Each object should be designed to serve one logical purpose.

For example, the OpenSSL protocol defines an SSL socket distinct from a basic socket. The SSL socket is composed of two units:

- A socket file descriptor
- A context structure containing information relevant to the SSL protocol

Even though the context structure can contain a substantial amount of varied data, at the top level it looks like a distinct unit. It has one form – a container for SSL data.

It’s a bit more difficult to describe the unity of form, but the basic rule is that you should break out what logically makes sense into its own structure.

Lesson: Each structure serves one logical purpose, and each function does one logical task. In this way, we keep things simple for programmers to understand and manipulate.

Design: Everything You Need, and Only What You Need

How many times have I seen this pattern in object oriented code?

- Public Class
- All members of the class are private
- All members can be accessed with accessor functions
- All members can be changed with modifier functions

I hate to break it to y'all, but that's just a **struct**. The only thing you've done is add a hundred lines to a simple process.

Access Functions: Only What You Need

The purpose of an access function is simple: to **provide access to a portion of the data**, while hiding the structure of the class.

As such, it only makes sense to provide access functions in two cases:

- When it's important to obscure how a structure works (due to complex operations, typedef abstraction, etc.)
- When it's important to restrict how much information you reveal

In either of those cases, you only need enough functions to suit the need. If you're exposing the entire structure to the world, there was no reason to make anything private in the first place.

Modifier Functions: Only what you want to change

If your structure or class contains constants you set only once, why would you create explicit modifier functions for it?

Or, if you always modify a set of values simultaneously, why would you create individual modifier functions?

Think about your design before coding up modifier functions.

Constructors: Two Types

There are two reasons to create an explicit constructor function:

- You have to allocate values inside of the new structure (a char *, for

example)

- You want to fill the new structure with default values of some kind (perhaps by input)

In the first case, you usually call either another constructor or an allocation function inside of the constructor, and don't worry so much about anything else.

In the second case, you have two options:

- Take no arguments, and fill in the variables with a general default value
- Take some arguments, and fill in some variables with specific values

Note that I don't list among the options "Fill every variable at once". While not inherently bad, this pattern shows up with alarming frequency in poorly-designed code. I recommend that you consider exactly how many variables you need to fill with a non-default value on initialization.

Destructors: Only One Type

Destructor functions should always follow the same pattern:

- Destroy any allocated values inside the structure
- Destroy the structure

If your destructor does anything else (besides, perhaps, print something), you should reevaluate your design.

Lesson: Think before you do. Lots of programmers over-code, resulting in functions they neither want nor need.

Object Oriented C

Most modern languages are designed around the object oriented design principles. They contain syntactic elements that codify and require these principles for code implementation.

Unfortunately, as is common with modern minds, we leave everything up to the computer. This results in a gigantic block of generalities, the result of which is slow code that is difficult to maintain. For most modern programmers, the choice is between efficient code and object oriented code.

However, we can apply the Object Oriented Design Principles to C code relatively easily. After all, the principles were originally theorized for use in languages like C.

Encapsulation

We've already discussed one aspect of encapsulation (that is, the unitary nature of objects), but the principle also includes "data hiding". In "proper" encapsulation, the structures are perfectly opaque to the end user, so that the only way to access the data is through carefully-constructed methods.

The techniques for constructing methods are already well understood, but how does one create hidden data in C?

Fun fact: If it's not in the header file, the end-user can't touch it.

Fun fact #2: We can use **typedef** to create pointers to objects in code files which do not themselves appear in the header.

Fun fact #3: While the computer can translate between the typedef pointer and a pointer to the original object at link time, the user's code cannot.

Thus, if we define **struct panda** in a code file, but the header only contains methods and the following:

typedef struct panda *PANDA

the end user can only access the structure by passing PANDA to the access methods.

Abstraction and Polymorphism

We perform abstraction in only two ways:

- Function pointers - used to look up objects
- Hiding data with "typedef"

Function pointers are too complicated to go into now, but basically we can write functions that accept pointers to functions, the results of which produce data that the top-level function can work with. This allows us to create essentially polymorphic functions.

Because abstraction is relatively complicated in C, it encourages programmers to rely on patterns and techniques instead of function-overloading and “let the computer figure it out” mental patterns. That means we don’t employ **template** classes, **interface** classes, or **abstract** classes (which I would argue ultimately make programming MUCH harder), but we can still create functional polymorphism and abstraction if we so choose.

NOTE: WE DO CREATE “TEMPLATES” IN C. THESE ARE NOT OOP TEMPLATES, BUT RATHER CODE WHICH IS GENERALLY APPLICABLE TO A NUMBER OF TASKS. WE CAN SIMPLY USE THESE FRAMEWORK TEMPLATES TO PRODUCE NEW CLASSES QUICKLY.

Inheritance

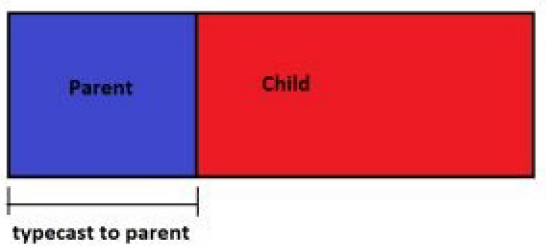
What exactly does inheritance mean?

Functionally, inheritance ensures that one class contains as its first (and thus, most easily addressed) member another class. All the aspects of the parent class are carried into the child class, so that the child is merely an extended version of the parent. As such, all the parent functions should be able to work with the child class.

We can do the exact same thing in C using **type casting**.

We know that we can tell a C program to treat data elements as some other kind of data element. When we do this, C “maps out” the space that the casted element should occupy in the original element and treats that space as the cast element.

It looks something like this:



Sure enough, so long as we ensure that the “parent” structure is the first element in the “child” structure, we can use type casting to perform inheritance. It’s really that simple.

Lesson: Object Oriented Design principles were designed during the C era for use with languages such as C. All of the principles (at least, the ones that don’t cause performance issues) are possible in C.

Conclusion

There are any number of ways to write code which will compile and run, and I've probably seen the majority of them in the past year alone. We tend to get extremely hung up on stylistic choices like brackets (which should always be aligned to one another to make it easier to see), indentation (single tab, default size), or capitalization (variables should never be capitalized), but these things don't really affect the code in any appreciable way. Ultimately, no matter which convention you use, it'll work if you can understand it.

However, there are a number of patterns in thought and practice which observably accelerate programming, debugging, and maintenance. These patterns should be common knowledge, and every programmer should be expected to employ them.

Think about the patterns detailed in this book. Work them into your new code and observe the change in your efficiency.

As the old saying goes, "A stitch in time, saves nine."

About the Author

Robert Beisert is a graduate of the University of Texas at Dallas, where he studied computer engineering. In that time, he was exposed to some of the brightest programmers in the world, and some of the absolute worst. Hopefully, he learned to tell the difference…

If you would like to see more of his content, you can visit his programming blog at fortcollinsprogram.robert-beisert.com

You can also contact him at **robert@robert-beisert.com** with any revisions or further observations concerning the material in this book. The blog is also networked with Disqus.