Software Engineering for Systems Hackers

DRAFT: Do not redistribute!

David G. Andersen

Contents

Chapter 1

Introduction

These notes attempt to capture some of the techniques that experienced systems hackers use to make their lives easier, their development faster, and their programs better. In our journey, we will scrupuously avoid metrics, excessive formalism, and the traditional trappings of very large-scale software engineering that are so often taught in undergraduate software engineering courses. While these techniques have their place and value, there is a middle ground of *practical* software engineering techniques that work well for programming in small groups—say, one to five people. We focus our attention on *systems* programming, the often low-level and intricate software development in operating systems, networks, compilers, and the other components that build the foundation of higher layer software systems.

We will explore both practices and tools that facilitate software development. Our view of these tools is that they should be *low-overhead*. They should be easy to start using, provide significant "bang for the buck," and be capable of supporting a project if it grows. The benefit from the practices should be clear and realizable without spending a lot of time on them.

An important question to ask over time is how much time to spend learning new techniques. Our view is to take the middle road. Be willing to keep your knowledge of tools fresh. This process is one of balancing between expertise with your current tool set, avoiding fads, and learning useful new tools that can significantly improve your productivity. Pick your tools carefully; most take some time to learn, and longer to become fluent. The process is much like that of optimizing the performance of a program: identify the small set of things that consume the most time, and optimize those first.

This same guideline applies to the suggestions we make in this text. While we believe that nearly all of them apply to anyone doing systems programming, some of the details may not match your style. Approach the problem from a puzzle-solving standpoint: How can *you* optimize the amount of time you get to spend doing the things that you enjoy?

Chapter 2

Revision Control

We begin our journey into software engineering before we write a single line of code. Revision control systems (RCSes) such as Subversion or CVS are astoundingly useful for single-developer projects, and essential for even the smallest group projects. These systems allow deveopers to obtain a copy of the current source code, work on it, and then "check in" their changes. The systems permit you to go back to any earlier version, and see the changes that were made between revisions. A good revision control system provides several benefits:

- **Super-undo**: You can go back to arbitrary saved versions of your source code.
- **Backups**: Using a revision control system means that you can keep the source code master copy on a safe, well-maintained machine, and edit/compile/develop on an arbitrary machine (such as your laptop).
- **Tracking changes**: The RCS can give you a list of all of the changes that affected a particular file (or project). This can be very useful in figuring out when something broke—and who broke it.
- **Concurrent access**: Many RCSes are designed to allow concurrent, safe access to the source code. More about this later.
- **Snapshots**: Is your code at a particular good working point? (e.g., a release, or in the case of 15-441, ready for a checkpoint.) An RCS should let you "snapshot" the code at that point with a memorable name so that you can easily access it later.
- **Branches**: A branch allows you to commit changes to an "alternate" copy of the code without affecting the main branch. A good example is the optimization contest in the second 15-441 assignment. With a branch, you can create a separate, derived tree for your code where you play around with ideas, knowing that they won't break the main, safe branch. You don't have to use branches, and they're a mildly "advanced feature," but if you're putting serious effort into the optimization contest, you might want to consider using them.

As a random aside, note that revision control is useful beyond just code. I (Dave) use subversion to store papers that my research group is working on, my c.v., my web pages, the configuration files for some of my machines, and so on. I use it exactly as I mentioned above in "backups" to keep all of my documents and projects on a server, while being able to edit them on my laptop.

Original file

```
This is a test file
It started with two lines
```

Modified file

```
This is a test file
It no longer has two lines
it has three
```

The "universal" diff (`diff -u`) between the two files is:

```
--- file        Mon Aug 28 11:31:53 2006
+++ file2       Mon Aug 28 11:32:07 2006
@@ -1,2 +1,3 @@
 This is a test file
-It started with two lines
+It no longer has two lines
+it has three
```

Figure 2.1: Diff of two small files.

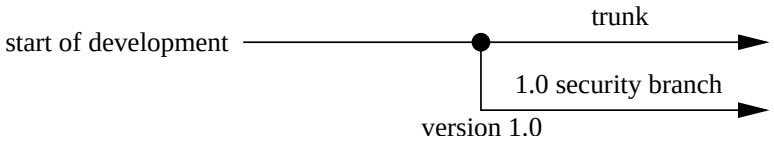

Figure 2.2: Two branches and a tag marking version 1.0.

2.1 Revision Control Concepts

Concept 1: The master copy is stored in a repository. Revision control systems store their data in a "repository" that is separate from the copy you're editing. In many cases, the repository may be a database or a set of binary files. To obtain a copy of the repository, you *check out* a copy. Doing so creates a local copy of the source code tree. The changes you make to files are only saved to the repository when you explicitly choose to do so by performing a *commit* operation. The copy of the code that you're working on is often called a *working copy*.

Concept 2: Every revision is available. When you commit to the repository, the changes you've made since your previous commit (or since check-out) are all saved to the repository. You can later see each of these revisions; perhaps more importantly, you can view a *diff* between arbitrary versions, showing only what changed. Diffs are most commonly expressed in unix `diff` format (Figure 2.1).

Concept 3: Multiple branches of development. Most RC systems support the idea of multiple lines, or "branches" of development. For example, consider a project that has a main line of development (which we will call the "trunk"). At some time, this project makes a public release of version 1.0. Afterwords, it releases only critical security updates to version 1.0, while continuing to add features to the "trunk" in preparation for version 2.0. (Figure 2.2).

Branches are a great idea, but they also add complexity for the developer, because someone must often ensure that desirable changes to one branch get propagated to the other branch, which may have somewhat different code. This process is called *merging*, and is similar to resolving conflicts (see below).

Concept 4: Concurrent development. The final benefit from an RC system is that it lets multiple developers work on the code concurrently. Each checks out his or her own copy of the code, and begins editing. Developers commit their changes independently, and can *update* to receive changes that have been committed to the repository since they checked out the code or last updated.

This works perfectly if the developers are working on different parts of the code, but what if they are editing the same file at the same time? The answer depends. If the diffs are reasonably separate (different parts of the file, for example), then the system will usually be able to *merge* them automatically. If, however, the updates touch similar lines of code, then the updates will *conflict*, and the revision control system will report this to the user. The user must then merge the changes manually.

2.2 A practical introduction to Subversion

In 15-441, you'll use Subversion for source control. This section explains some of the basic subversion commands and how to make your way around the system.

1. **Check out the repository**
   The course staff will have created a repository for you to use. It will initially be empty.

   ```
   svn checkout https://moo.cmcl.cs.cmu.edu/441/441proj1-group1
   ```

   This will create a local directory called 441proj1-group1 (please replace with your own group name as appropriate), and fill it with the latest version of your source code.

2. **Add a file to the repository**
   Create the file you wish to add (or copy it) to the repository. Put it in your checked out source code tree wherever you want. Then

   ```
   svn add <filename>
   ```

   Note that this step will *schedule* the file to be added, but until you commit, it will just be in your local copy.

3. **Add a directory**
   You can add a directory to the repository by either making it using mkdir and adding it, or by using `svn mkdir`. We suggest using the latter, because it means that subversion is immediately aware that the directory exists, and can use it as a target for move operations.

   ```
   svn mkdir bar
   ```

4. **Commit your changes**

   ```
   svn commit
   ```

   You can commit from either the top of your directory tree (in which case all of your outstanding changes will be committed), or from a sub-directory, or just for a particular file by naming it after the commit (e.g., `svn commit foo`). This will send your local changes to the repository. When you commit, svn will prompt you for a log message to describe, for the benefit of you and your fellow developers, the changes you've made. Don't leave this blank—good log messages are very useful for debugging and to help coordinate with your partner.

   If the version of the file you were editing is not the latest one in the repository, `commit` will fail and let you know. At this point, you'll need to update (5) and perhaps resolve any conflicts between your edits and the previous ones (Section 2.2.1 below).

5. **Update to get the latest changes**

   ```
   svn update
   ```

   This will check out the latest changes from the repository. Subversion will print messages indicating what files were changed. For example, if the update added a new README file and changed the Makefile, subversion would indicate this as:

   ```
   A  src/trunk/README
   U  src/trunk/Makefile
   ```

6. **Make a snapshot of your code to hand in**

   Subversion uses the "copy" command to create both tags and branches. (Internally, they're the same thing, but humans treat them differently by not committing to a tag). By convention, subversion tags are stored in a top-level "tags" directory. We'll use this for checking out a copy of your projects so that you can keep developing without clobbering the handed-in version.

   ```
   svn copy trunk tags/checkpoint1
   ```

7. **See what changed in a file**

   To see the changes you've made since your last checked out or updated a file, use:

   ```
   svn diff file
   ```

   You can also see the difference between arbitrary revisions of the file. For example, to see the difference between revision 1 and revision 2 of a file:

   ```
   svn diff -r 1:2 file
   ```

You can also use `svn log` to see what changes have been recorded to a file.
Table 2.1 lists other commands you may find useful:

`svn remove`	Remove a file or directory
`svn move`	Move or rename a file or directory
`svn status`	Show what files have been added, removed, or changed
`svn log`	Show the log of changes to a file (the human-entered comments)
`svn blame`	Show each line of a file together with who last edited it (also `svn annotate`).

Table 2.1: Other useful Subversion comments

2.2.1 Resolving conflicts

Eventually, you'll have conflicting edits to a file. Fortunately, resolving them in subversion isn't too hard. Let's say that you and your partner both edit the file `testfile`. Your partner commits her changes first. When you update, you'll see that your edits conflict with the version in the repository:

```
~/conflict-example> svn up
C    testfile
Updated to revision 17.
```

If you look in the directory, you'll see that subversion has put a few copies of the file there for you to look at:

```
~/conflict > ls
testfile        testfile.mine   testfile.r16    testfile.r17
```

The file `testfile` has the "conflict" listed. The text from the two revisions is separated by seven "<" "=" and ">" markers that show which text came from which revision, such as:

```
This is a test file

It has a few paragraphs,
to which we'll be making some
conflicting edits later in this
<<<<<<< .mine
This line was added on machine 1
=======
This line was added on machine 2 - hah, I got it committed first!
>>>>>>> .r17
exercise.

Yes, a few paragraphs it has.
Wooo, that's cool.  I like paragraphs!
```

You can see that one version has a line that says "This line was added on machine 1" and the other version has a line that says "This line was added on machine 2 - hah, I got it committed first!" You have a few options for resolving the conflict:

1. **Throw out your changes**. If you just want your partner's changes, you can copy her file on top of yours. Subversion conveniently supplies you with a copy of the latest version of the file; in this case, it's `testfile.r17`.

   `cp testfile.r17 testfile`.

   Then tell subversion you're done merging:

   `svn resolved testfile`

   and commit.

2. **Overwrite your partner's changes**. Just like the previous example, but copy `testfile.mine` onto testfile instead.

3. **Merging by hand**. Open `testfile` in an editor and search for the conflict markers. Then select which of the versions (if either) you want to preserve, and update the text to reflect that. When you're done, `svn resolved` and commit.

2.3 Thoughts on using revision control

A selection of thoughts from our experience using revision control.

- **Update, make, test, and then commit**. It's good to test your changes with the full repository before you commit them, in case they broke something.

- **Merge relatively often.** We come from the "merge often" school. See the discussion in the next section about breaking down your changes into manageable chunks.

- **Commit formatting changes separately.** Why? So that you can more accurately identify the source of code and particular changes. A commit that bundles new features with formatting changes makes it difficult to examine exactly what was changed to add the features, etc.

- **Check `svn diff` before committing.** It's a nice way to check that you're changing what you meant to. We've often discovered that we left in weird debugging changes or outright typos and have avoided committing them by a quick check.

- **Try not to break the checked in copy.** It's convenient for you and your partner if the current version of the source code always at least compiles. We suggest breaking your changes down into manageable chunks and committing them as you complete them.

  For example, in Project 1, you may start out by first creating a server that listens for connections and closes them immediately. This would be a nice sized chunk for a commit. Next, you might modify it to echo back all text it receives. Another commit. Then create your data structure to hold connection information, and a unit test case to make sure the data structure works. Commit. And so on.

  If you're going to make more invasive changes, you may want to think about using a branch. A good example of branches in 15-441 is the optimization contest for the second project: You may want to experiment with techniques that could break your basic operation, but you don't want to risk failing test cases just to make your program faster. Ergo, the "optimization"

branch. You can make changes in this branch (and save your work and get all of the benefits of source control) without affecting your main branch. Of course, if you create a branch and like the ideas from it, you'll have to merge those changes back later. You can either use `svn merge`, or you can `svn diff` and then patch the files manually. The nice thing about using `svn merge` is that it records which change you're propagating.

- **Use meaningful log messages**. Even for yourself, it's great to be able to go back and say, "Ah-ha! That's why I made that change." Reading a diff is harder than reading a *good* log message that briefly describes the changes and the reason for them.
- **Avoid conflicts by good decomposition and out-of-band coordination**. Revision control is great, but conflicts can be a bit of a pain. To keep them to a minimum:
  - Make your program modular. If one person can work independently on, say, the user IRC message parsing code while the other works on the routing code, it will reduce the chances of conflicts—and it's good programming practice that will reduce your headaches in lots of other ways. Have this modularity reflected in the way you put what code in what file.
  - Coordinate out of band with your partner. Don't just sit down and start working on "whatever"—let your partner know what you're working on.

2.4 Other source control systems

Table 2.2 lists a number of other popular source control systems that you might encounter after this class.

2.5 Trying it out on your own

Subversion is installed on the Andrew linux machines. To create a repository, try:

```
mkdir svn
svnadmin create svn/test
svn checkout file:///afs/andrew.cmu.edu/YOUR/PATH/svn/test
```

(Of course, replace "YOUR/PATH" with that to your repository!)

You'll be able to access this repository from any machine with AFS access. You can also access it via SSH by specifying the URL as:

```
svn checkout  \
 svn+ssh://you@linux.andrew.cmu.edu/afs/andrew....../svn/test
```

2.6 Recommended Reading

The best subversion book is "Version Control with Subversion." It's available in book form and for free online [1].

RCS	An early source control system. Allows files to be locked and unlocked; does not address concurrent use and conflict resolution. Sometimes used for web pages and configuration files where changes occur slowly but revision control is useful.
CVS	The Concurrent Version System. CVS is built atop RCS and uses its underlying mechanisms to version single files. Very popular. Major users include the FreeBSD project and many other open source systems.
Subversion	Subversion is designed to fix many of the flaws in CVS while retaining a familiar interface. Adds a number of capabilities to CVS (e.g., the ability to rename files and directories) without breaking the basics. Quickly gaining popularity among open source projects.
Bitkeeper	A commercial distributed source control system. BitKeeper used to be used for the Linux kernel.
Git	A distributed source control system used for the Linux kernel. Does not have one central repository; each participant merges changes into their own tree.
Visual SourceSafe	Microsoft's source control system.
Perforce	A popular, heavy-weight commercial revision control system.

Table 2.2: Popular revision control systems.

Bibliography

[1] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly and Associates. ISBN 0-596-00448-6. Available online at http://svnbook.red-bean.com/.

Chapter 3

Build Tools and Automation

"Why write a makefile? My project is only one .c file!"

```
gcc foo.c -o foo      (16 characters)
make                  (4 characters)
```

repeated over the lifetime of even a small project, the 12 character difference will add up to a *qualitative* improvement in your enjoyment of writing your software. Easier to build means easier to test and revise; easier to test and revise makes for happier programs and happier programmers. And few projects—particularly not those you're going to write in this class—stay that simple forever.

Start out with a makefile from minute one.

Don't develop bad habits with a project, because they're hard to break. Don't develop ad-hoc fixes (shell aliases, etc.) that will just break later on. Make is a tool that any serious programmer should learn to use well. It will repay your investment in much saved time, *even just over the course of this class.* Just as an example, I (Dave) use makefiles for building code, running latex and bibtex on papers, and creating Web pages and pushing them to my server. The notes you're reading are created by a 44 line makefile and a handful of smaller makefiles for the examples.

3.0.1 Which make?

The answer to this question usually trades portability for power. bsd make, gnu make, imake, and friends all add power to the basic commands supported by "make" at the cost of adding a dependence on having another tool installed.

For purposes of 15-441, we suggest sticking with the basic make commands. If you really insist, judiciously use some of the gnu extensions. Don't add weird things to your build environment. In the long run, it's often better to add an additional, higher-level build tool such as "autoconf" and "automake" (or "autoproject" if you really want to go all-out) that takes care of more of the cross-platform issues and provides additional high level support for project management. But for now, let's stick with plain old Make. It's more useful than you might think.

3.0.2 Automate some tests

Don't just have some unit tests and other goodness, *automate them!* Nothing beats typing "make test" and immediately seeing whether that change you just made broke something else.

```
# This is a comment
CFLAGS=-Wall -g

prog: prog.o
        ${CC} prog.o -o $@
```

Figure 3.1: A simple Makefile

In fairness, we note that test automation in systems programming can be *hard.* We don't suggest going overboard in 15-441 because the limited lifetime of the project won't reward a completely rich, amazing test environment. However, having a basic level of automated testing that can start two instances of your programs, see if they can talk nicely with each other, etc., will be well worth your effort. At an even more basic level, automatically running the unit tests for bits and pieces of your system is easy to implement and pays big dividends.

3.1 Writing Makefiles

Consider the simple Makefile shown in Figure 3.1. When the user types make, it will build the program "prog" from the source file "prog.c". Simple as it is, this makefile demonstrates several important features in Make:

- **First target default**: The first target defined in the makefile is the default. In this case, "prog."
- **Implicit compilation rules**: Make defines implicit rules for compiling, e.g., .c files into .o files. By convention, that implicit rule uses the variable `CFLAGS` to define the C compilation flags.
- **Variables**: The makefile assigns a new value to the variable `CFLAGS` so that we get compiler warnings. It uses the already-defined variable `CC` to represent the C compiler.
- **Special Variables**: The makefile uses the special variable `$@`. This special variable means "the target" (in this case, "prog"). Special variables can let you avoid needless typing, and make your rules a little more easy to change and reuse.

3.2 A more complete example

Let's examine a Makefile and a more complete set of related files for a small project, involving one "main" program and two libraries:

- **prog.c**: The main program file
- **lib.c**: Library #1's .c file
- **lib.h**: Library #1's header file
- **lib_test.c**: A quick set of tests for Library #1
- **lib2.c**: Library #2's .c file

```
# This is a comment
CFLAGS=-Wall -g
LIBS=lib.o lib2.o
HEADERS=lib.h lib2.h
BINS=prog lib_test lib2_test

all:  ${BINS}

prog: prog.o ${LIBS} ${HEADERS}
        ${CC} ${LDFLAGS} prog.o ${LIBS} -o $@

lib_test: lib.o lib_test.o lib.h tests.o tests.h
        ${CC} ${LDFLAGS} lib.o lib_test.o tests.o -o $@

lib2_test: lib2.o lib2_test.o lib2.h tests.o tests.h
        ${CC} ${LDFLAGS} lib2.o lib2_test.o tests.o -o $@

test:
        ./lib_test
        ./lib2_test

clean:
        /bin/rm -rf ${BINS} *.o core *.core
```

Figure 3.2: The more complete Makefile

- **lib2.h**: Library #2's header file
- **lib2_test.c**: A quick set of tests for Library #2
- **tests.h**: A header file for some common test functions
- **tests.c**: A library with some useful test routines
- **Makefile**

Examine the Makefile in figure 3.2. Note that it adds three new targets that are common to most Makefiles:

- **all**: Compile all binaries
- **test**: Run any automated tests
- **clean**: Clean up binaries and object files

Chapter 4

Tools

4.1 Tools for Code Checking

4.1.1 Compiler Checks

The compiler can already perform many checks for you, if you let it:

Always, always, always use gcc -Wall

Don't accept warnings in your code. Some of `gcc`'s warnings are silly and you know that they're not bugs—but ignoring them means that you won't be able to see the *important* warnings! Better to accept a little bit of grief from gcc and initialize variables when they don't need it, etc., then to miss out on all of the potential hassle that gcc can save you.

Along the same lines about fixing bugs early that we discuss in Chapter 11.2.2, eliminate warnings early. It's easier to re-tool the code you just worked on than the stuff you haven't touched in two weeks.

4.1.2 Catching Errors Early with -D_FORTIFY_SOURCE

On recent versions of Linux, compiling your program with `-D_FORTIFY_SOURCE=2` will include additional checks for buffer alignment on system calls such as printf. These checks determine if you pass a too-small buffer to functions such as sprintf, etc. Using this directive is a cheap way to find some additional bugs.

4.1.3 Catching Errors Early with glibc variables

On Linux, you can set the environment variable `MALLOC_CHECK_` to identify more malloc-related problems. Setting this variable to 1 logs errors; setting it to 2 causes an immediate abort.

Chapter 5

Debugging

Some errors are easily deduced from a debugger backtrace: Using the wrong variable in an index, etc. Others, particularly when the program's behavior is unexpected but does not result in a crash, is easier to debug with application-level debugging output.

5.1 A Debugging Mindset

Debugging is a process of problem solving. It's similar to the process that physicians use to diagnose illness: a combination of experience, reasoning, and tools.

Before you start debugging intensively, check the really easy stuff:

- `make clean`—sometimes you may have failed to recompile a particular bit of code. It happens more than you'd think.
- *check compiler warnings*—they're there to let you be lazy. Let them help you. *Always* use `-Wall`. Consider also using `-Wextra, -Wshadow, -Wunreachable-code`.

Next, avoid the "it's not my bug" syndrome. You'll see bugs that "can't happen!" (But it did—you have a core dump or error to prove it.) You may be tempted to say "the compiler's buggy!" (the OS, the course staff, etc.). It's possible, but it's very unlikely. The bug is most likely in your code, and it's even more likely to be in the new code you've just added.

As a result, some debugging is easy—you can identify it quickly by looking at the output of the program, examining the debugger output (a stack trace is very useful), and take a minute to think about what could have caused the problem. Oftentimes, this is sufficient to find the bug, or narrow it down to a few possibilities. This is particularly true in the fairly common case when the bug is in a small amount of recently added code, so look there first. The `svn diff` command can come in very handy for reminding yourself what's changed! Sometimes you may overlook a "simple" change, or have forgotten something done at 3am the night before.

Some very useful steps in debugging, many stolen shamelessly from Kernighan & Pike:

- **Use a stack trace.** The stack trace might tell you exactly where the bug is, if you're lucky.
- **Examine the most recent change.** Bugs usually show up there.
- **Read the code carefully.** Before you start tweaking, read the code and *think* about it for a bit.

- **Explain your code to someone else.** Oftentimes when you read your own code, you'll read what you *expect* to see, not what's really there. Explain the code to anyone, even a pet rock. It helps.

Harder debugging is often approached as a process of hypothesis elimination:

- **Make the bug reproducible.**
- Think about the symptoms (crashing, incorrect result, etc.) of the bug. Look at (or remember) the code involved. Identify the possible and likely causes of these symptoms.
- Think about the correct behavior that you expected the program to exhibit, and identify your reasoning / assumptions about what state or logic flow in the program would make it actually do so.
- Identify an experiment that you can perform that will most effectively narrow down the universe of possible causes of the bug and reasons that the program didn't behave as expected.
- Repeat.

Experiments you can perform include things like:

- Add consistency checks to find out where your assumptions about the program state went wrong. Is the list *really* sorted? Did the buffer *really* contain a full packet?
- Add debugging output to show the state of particular bits of the program, or to identify which parts of the program were being reached (correctly or incorrectly) before the crash.
- Remove parts of the code that may be contributing to the bug/crash/etc., to see if they're really responsible. Note that steps like this are a lot safer if you've recently committed your code, perhaps to a development branch.

Some of this depends on experience: over time, you'll have seen more common bugs and can identify the patterns by which they manifest themselves. But think about it like a binary search process: is the bug caused by one of these causes, or one of those? Can you perform an experiment to cut the field in half?

Making the bug easily reproducible is a very important first step. Imagine that your IRC server crashed when there were 20 clients connected and they'd each sent 1000s of lines of text. The situation that caused the bug tells you little about the reason the program failed. Does the bug still happen if you have only 10 clients? 5? If they send only 10 lines of code? Simplify the cause as much as possible; in its simplest verison, it may directly point out the bug!

Writing a log file is a great way to help debug. We'll talk about this a little more in Section 5.2. Being able to `grep` through the logfile or analyze it can make the debugging process much easier.

Using tools such as electric fence or system call tracers (below) can help identify particular bugs rapidly.

Finally, if a bug is really persistent, start writing things down. You'll save yourself repeating tests needlessly and will be more likely to cover the space of possibilities.

Once you've found a bug, think about two things:

1. Have I made this bug elsewhere in the code? Bugs that result from misunderstanding interfaces, etc., are likely to show up in multiple places. Do a quick check to proactively eliminate other bugs.

2. How can I avoid making this mistake in the future? You might be able to add test cases to automatically find them, add assertions to the code to detect if the bug happens again, use compiler warnings to automatically detect them, or change the way you write the code to make it impossible to make the mistake.

5.2 Good old printf done better: Debug macros

Instead of randomly throwing in and removing `printfs` while trying to track down bugs, consider using a bit more structure: debug macros that let you add debugging printfs and not remove them. A good set of debug macros will let you selectively enable and disable different levels of debug verbosity, so that you can immediately go from normal (silent) operation to "full debug" mode just by changing the command line arguments.

There are generally two combinable approaches to specifying what debugging information should be output: Some macros provide a tunable "verbosity level" (e.g., 0-9) and other macros allow you to specify which functionality should be debugged (e.g., socket operations, processes, etc.). More complex systems often allow the user to specify verbosity on a per-component or per-functionality basis ("verbosity 9 process debugging but verbosity 1 socket debugging").

For purposes of 15-441, we suggest using a relatively straightforward set of debug macros that provide one or the other verbosity knobs, but not both. Figure 5.1 shows a simplified version of a debug macro header, debug.h that allows the user to specify the binary-OR of different debug facilities to selectively enable debugging for different components of the program.

A version of these debug macros is available from the 15-441 course website.

5.3 Debugging tools

5.3.1 gdb

`gdb` can run in three ways:

- Starting a new process running under the debugger:

  ```
  gdb binary
  ```

- Examining a `core` file from a crashed process

  ```
  gdb binary core
  ```

- Attaching to a running process

  ```
  gdb binary PID
  ```

For the last form, you can get the PID by using the `ps` command. Often useful is

```
ps auxww | grep your_program_name
```

```
#ifndef _DEBUG_H_
#define _DEBUG_H_

#include "err.h"

#include <stdio.h>  /* for perror */
#ifdef DEBUG
extern unsigned int debug;
#define DPRINTF(level, fmt, args...) \
    do { if ((debug) & (level)) fprintf(stderr, fmt , ##args ); } while(0)
#define DEBUG_PERROR(errmsg)                                          \
    do { if ((debug) & DEBUG_ERRS) perror(errmsg); } while(0)
#define DEBUGDO(level, args) do { if ((debug) & (level)) { args } } while(0)
#else
#define DPRINTF(args...)
#define DEBUG_PERROR(args...)
#define DEBUGDO(args...)
#endif

/*
 * Add some explanatory text if you add a debugging value.
 * This text will show up in -d list
 */

#define DEBUG_NONE      0x00        // DBTEXT:  No debugging
#define DEBUG_ERRS      0x01        // DBTEXT:  Verbose error reporting
#define DEBUG_INIT      0x02        // DBTEXT:  Debug initialization
#define DEBUG_SOCKETS   0x04    // DBTEXT:  Debug socket operations
#define DEBUG_PROCESSES 0x08        // DBTEXT:  Debug processes (fork/reap/e

#define DEBUG_ALL  0xffffffff

#ifdef __cplusplus
extern "C" {
#endif
int set_debug(char *arg);  /* Returns 0 on success, -1 on failure */
#ifdef __cplusplus
}
#endif

#endif /* _DEBUG_H_ */
```

Figure 5.1: An example set of debug macros included via the debug.h file.

Command	Function
	Executing Code
run	Begin executing the program
c	Continue running after breaking
s	Step into next source line, entering functions
n	Run until next source line, stepping over functions
	Getting Information
bt	Display a backtrace of all stack frames
p *expr*	Print the value of *expr*
	e.g., `p (5*5)` or `p *f`
b *n*	Set a breakpoint at line *n*
b *func*	Set a breakpoint at function *func*
list	List the source code at the current point
list *func*	List the source code for function *func*
up	Go up to the previous stack frame
down	Go down to the next stack frame
info locals	Show the local variables for the stack frame

Table 5.1: Common gdb functions.

Frequently Encountered Problems with GDB

1. Why don't I have a core file?

The default system limits may be the culprit here. Type the command `limit` (using csh) or `ulimit -a` (using sh or bash), and see what it says. You should get output that looks like:

```
767 moo:~/systems-se> limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    unlimited    <--- note this line
memoryuse       unlimited
vmemoryuse      unlimited
descriptors     1024
memorylocked    unlimited
maxproc         7168
openfiles       1024
```

Your value of `coredumpsize` may be set to zero, in which case core dumps will be disabled. To fix this, type `unlimit coredumpsize` (using csh), or `ulimit -c unlimited` (using sh or bash). Sometimes, you don't *want* core files (because they take time and space to save). You can prevent them by typing `limit coredumpsize 0` (csh) or `ulimit -c 0` (sh/bash).

Different operating systems have different conventions for naming core dump files. Linux may name them simply "core" or "core.PID" (e.g., "core.7337"). BSD names them "program.core". On Mac OS X, the system stores all core dump files in the directory `/cores`.

Many systems prevent core dumps from processes that are setuid or setgid. If you want to get a core from such executables, you'll need to execute them as their owner.

2. GDB doesn't work. When I run it, I get a shell prompt! The most likely cause of this is that you're `exec`'ing a different shell as part of your `.cshrc` or `.bashrc` file. The best way to change your shell is to use the `chsh` command to change it globally. Barring that, you should restrict the `exec` to only change the shell for interactive logins. The easiest way to do so is to do the exec only in your `.login` file.

5.3.2 Using GDB to track down common problems

XXX-TODO

Examples: fewer compile/run cycles by figuring out if a program got to a particular spot w/breakpoints; understanding the path of execution by examining predicates; etc.

5.3.3 System call tracing: ktrace, strace, and friends

Another good way to debug processes (even if you may not have the source code handy) is to use a system call tracing facility such as `ktrace` (BSD) or `strace` (Linux).

To run a program using strace, run:

```
strace <executable> [args]
```

Strace produces output listing each system call, its parameters, and results:

```
516 unix35:~> strace ./syscalls
execve("./syscalls", ["./syscalls"], [/* 34 vars */]) = 0
brk(0)                                  = 0x601000
...
(other output for linking libraries and program startup)
...
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
bind(3, {sa_family=AF_UNSPEC, sa_data="\0\0...\0\0\0"}, 16) = 0
exit_group(0)                           = ?
Process 24787 detached
```

Strace runs quickly and provides a dynamic trace of the program's execution at the system call level. From the example above, the `socket()` and `bind()` calls are listed, along with their return codes. Even from the trace, it's clear that the bind call was being made without properly initialized variables.

Strace output can be very useful to observe the last few system calls made before a program crashed.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>

int
main()
{
    int s;
    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(sin));

    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    /* Deliberately uninitialized sockaddr */
    bind(s, (struct sockaddr *)&sin, sizeof(sin));
    exit(0);
}
```

Figure 5.2: A test program that performs two system calls.

5.3.4 Memory Debugging with Valgrind

Valgrind is a powerful dynamic analysis tool (it tests your code while your code is running). It provides tools that can check for invalid or uninitialized memory use, writing memory after it was freed, and memory leaks. Because it runs programs in a virtual machine (all code executes inside valgrind, not on the normal host processor), programs run about 10x more slowly under Valgrind, but that shouldn't be a problem for code in this class.

To run your code under valgrind, run it as:

```
valgrind <executable> [args]
```

For example, if you were to type in and run the program in Figure 5.3 without valgrind, it would (typically) appear to run properly:

```
508 unix35:~> ./uninit
Buf[0] contains 0 (deliberate uninitialized access)
```

Running it under valgrind produces copious output. Examining some of that shows:

```
507 unix35:~> valgrind ./uninit
...
==23689== Use of uninitialised value of size 8
==23689==    at 0x4E691CB: (within /lib/libc-2.6.1.so)
==23689==    by 0x4E6BB3E: vfprintf (in /lib/libc-2.6.1.so)
```

```
#include <stdlib.h>
#include <stdio.h>

int
main()
{
    int *buf;
    buf = (int *)malloc(10 * sizeof(int));
    printf("Buf[0] contains %d (deliberate uninitialized access)\n", buf[0])
    exit(0);
}
```

Figure 5.3: A test program that accesses uninitialized memory.

```
==23689==    by 0x4E73319: printf (in /lib/libc-2.6.1.so)
==23689==    by 0x400562: main (in /home/dga/uninit)
```

Like Electric Fence, below, valgrind also detects other memory errors. It's a great tool to use when confronted with tricky memory leaks or hard-to-reproduce "heisenbugs."

For the most part, Valgrind is only available on Linux.

5.3.5 Memory Debugging with Electric Fence

Electric Fence is a utility that helps debug buffer overruns. It monitors access to malloc-generated buffers to detect if your code writes past the end (or start) of the buffer. Running your code under electric fence will be slower, but it's a good way to detect weird, hard to track down errors involving buffer problems. Electric Fence is faster than Valgrind, but doesn't detect as many kinds of errors.

To run your code under electric fence, run it as:

```
ef <executable> [args]
```

You can also compile your code directly against Electric Fence by linking it with `-lefence`.

Using electricfence, an attempt to access memory outside of the allocated chunk will cause the program to seg fault instead of causing strange, unpredictable behavior.

For example, if you were to type in and run the program in Figure 5.4 without electric fence, it would (typically) run to completion:

```
546 unix35:~> ./overwrite
accessing start of buf
accessing end of buf
accessing past end of buf
done with overwrite test
```

However, if you run it under electric fence, it will crash at exactly the instruction that writes outside the malloc buffer:

```
#include <stdlib.h>
#include <stdio.h>

int
main()
{
    char *buf;
    buf = malloc(64);
    fprintf(stderr, "accessing start of buf\n");
    buf[0] = '\0';
    fprintf(stderr, "accessing end of buf\n");
    buf[63] = '\0';
    fprintf(stderr, "accessing past end of buf\n");
    buf[64] = '\0';
    fprintf(stderr, "done with overwrite test\n");
    exit(0);
}
```

Figure 5.4: A test program that overwrites a malloc buffer.

```
547 unix35:~> ef ./overwrite

  Electric Fence 2.2.0 Copyright (C) 1987-1999
  Bruce Perens <bruce@perens.com>
accessing start of buf
accessing end of buf
accessing past end of buf
/usr/bin/ef: line 20: 11049 Segmentation fault (core dumped)
   ( export LD_PRELOAD=libefence.so.0.0; exec $* )
```

The error message from ElectricFence isn't particularly useful (though it does tell you the process ID that died, which is handy if you're on a system like the Andrew Linux machines that name core dump based on PID), but the resulting core dump file is very valuable:

```
548 unix35:~> gdb ./overwrite ./core.11049
GNU gdb Red Hat Linux (6.1post-1.20040607.43.0.1rh)
...
#0  0x0804847b in main () at overwrite.c:14
14          buf[64] = '\0';
```

5.3.6 Tracing packets with tcpdump and Wireshark

For network projects and debugging distributed systems, packet sniffers such as `tcpdump` and Wireshark (formerly Ethereal) can be a life-saver. These utilities can record packets as they go in and out of a machine, and can decode various protocols. `tcpdump` is a more low-level interface

to raw packets; it's easy to use from the command line and produces useful output. Wireshark understands many more protocols, has a more powerful set of functions (such as reassembling TCP streams), and has a handy GUI.

XXX-TODO: Add a tcpdump or wireshark example!

5.3.7 Examining output

Being able to do a "diff" of your output is often helpful. In particular, when debugging problems with the second project (file transfers), you may find it very useful to know "What's the difference between the file I tried to send and the file that actually arrived?"

- **Send text-only data.** It may seem obvious, but if you make your data contain only ASCII text, you'll be able to examine it more easily with conventional UNIX command-line tools such as `wc, diff` and `grep`.
- **Use the `cmp` command to compare them.** `cmp` will tell you the byte and line at which the two files first differ.
- **Use the `od` command to view binary data.** `od` will convert your data into a hex or octal dump format. You can then diff the output of this.
- **Open it in an editor.** Emacs and vi will display binary data in a way that will allow you to visually scan for differences. It's not perfect, but as a quick hack, it may help you get over a hump and show you some obvious difference in the files.
- **Examine more powerful diff tools.** Students in the past have had luck using a visual binary diff tool:

  `http://home.comcast.net/~chris-madsen/vbindiff/`

These techniques are not specific to project 2: they're useful in dealing with any system that outputs data.

5.4 Debugging other people's code

Debugging a pile of code—or a binary someone handed you—is harder than debugging your own code. Tools like gdb and system call tracing start to shine here, because they show you what the code was actually doing, instead of forcing you to dig through the code (or an object dump!) to figure out what's up.

Chapter 6

Documentation and Style

Finding the sweet spot in documenting your code can be difficult. On one hand, a camp of developers argues that *the code is the documentation.* Their rationale is that if code can change independently of documentation, then documentation can and will get out of date. On the other hand, a new user of the code or libraries developed by people in the first camp may disagree—strongly.

We don't know the perfect answer to this debate, but we do have a few suggestions:

Don't document the obvious, and make your code obvious.

```
for (i = 0;  i < a.max; i++) {
  printf("%d\n", a.contents[i]);
}
```

Comments in code like the above add unnecessary clutter to the obvious. Why is it obvious? We have several hints:

- The use of a standard index variable name, *i*.
- A "typical" for loop for iterating through an array
- The "max" element is clearly named.

Note the example in Figure 6.1. This code snippet from a previous 15-441 project shows two bugs. First, the project uses a magic consant for LSA announcements (how should we remember that type "1" is an LSA announcement?) Ironically, the constant `INCOMING_ADVERTISEMENT` is defined earlier in the file; the programmers just forgot to use it in this case. Imagine the fun tracking

```
/* Send an ACK if this is an LSA annoucement */
if (getMsgType(buffer) == 1) {
  sendACK(&from, length, getMsgSeqNumber(buffer));
}

/* process the incoming LSA message */
processLSA(buffer, ntohs(from.sin_port));
```

Figure 6.1: A commenting and style counter-example from 15-441.

down a bug if they later changed the value used to indicate LSA announcements in the #define and only half the code changed? Second,the code is almost completely self-documenting, in a good way. The comments are extraneous.

6.0.1 Style

Two spaces? Four? A tab? Hungarian notation? Gnu style?

Within bounds, it doesn't matter.

What matters is that you pick a style, and *use it consistently*. The style should do a few things for you:

- Make the code readable (one space indentation is *not* readable).
- Distinguish important things, like class names in C++, variables, globals, etc.
- Not be excessively long and painful. Naming all of your variables bgfooBarThisIsADog will only cause you grief and slow you down. Use naming that is appropriate to the circumstance. The variable "i" in the above code fragment is clear—it's used immediately after it's declared (presumably), it's a "standard" variable name, etc. On the other hand, a global variable named "config.maxLineLen" (or "config.max_line_len") is probably more appropriate than calling it "c.mll," which may leave subsequent readers bitter.

Using the style consistently is much more important than the details. That said, over time a few conventions have popped up that can greatly help make code more readable:

- Name functions with active verbs. `isValid()` is a better name than `checkValid()`, because the semantics are more clear.
- Introduce functions with a brief comment explaning their semantics and anything the caller really needs to know about using it.

6.1 Read other code!

A nice way to get a feel for coding style and tricks is to read some other code. I suggest bits and pieces of the system utilities from the BSD source code. By and large, it's well-organized, consistent, and has stood the test of time. A few ideas:

- The source code to the 'primes' game/utility. Remember that Sieve of Eratosthenes that you probably wrote a few times in earlier CS classes? This program has a really nice implementation of it that doesn't bother with a lot of micro-optimizations (like storing things in bitmaps) but instead takes a bunch of macro-optimizations and ends up faster than a kitten in the dryer. It's about 300 lines of code. `http://www.cs.cmu.edu/~dga/systems-se/samples/primes.tar.gz`
- The testing code for the *ed* utility is nice.
- Lots of the kernel code is interesting, but *dense*.

6.2 Communication

At this point, you may well ask: What in the world is a chapter about *writing* and *graphics* doing in a book about *hacking*? The answer is that programs do not stand in a vacuum, and the similarities between good code and good writing are startling.

In the words of E.B. White, "Eliminate Unnecessary Words!" Good technical communication is concise and precise. Use the right words to describe exactly what you mean, no more and no less. Strive for clarity in all that you write. You'll note that this is much like good programming. A well-written program contains a minumum of unnecessary, repetitive cruft, is easy to understand, and does exactly what it's supposed to do and nothing more. So too does a well-written technical document.

6.3 Further Reading

Srunk & White's "The Elements of Style" [**?**] is one of the best references on English writing. It's short, sweet, and effective, and the lessons therein can be translated to numerous other domains (programming, presenting data graphically, and so on).

The Practice of Programming [1] has a great discussion on programming style with numerous examples.

Bibliography

[1] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley. ISBN 0-201-61586-X.

Chapter 7

Useful Software Components

7.1 Command line parsers

The `getopt` routines are the standard command line parsing functions for C and C++ programs. Their use is common enough that when I start writing a new program, the first thing I do is copy over my `main_skeleton.c` file, which starts out with a template for calling `getopt()`. It's easy to use—don't write your own without reason, just use getopt.

The `getopt_long` commands can provide GNU-style `--long_options=` parsing. If you're writing a new program from scratch, it's probably more user-friendly to start out using the `getopt_long` functions.

7.2 Hints for Configuration Files

Chapter 8

Scripting

Your time is valuable.
Your wrists do not like repetitive tasks.
Your brain does not like pointless work.
Listen to your wallet, wrists, and brain: Automate the frequent boring stuff.
The best way to automate the boring stuff is with scripting languages such as Perl, Python, and Ruby.[1]

8.0.1 Which Language?

I (Dave) know several scripting languages. I learned Perl first, and these days use Ruby. Many of my friends followed a similar path from Perl to Python. *Any of them is much, much better than nothing.* Because I have perl embedded more deeply in my fingers, I tend to use it for three-line programs. Because Ruby (or Python) produces better, more maintanable, and frequently shorter code, I use them for anything over about 10 lines. I don't suggest starting out to master three or even two scripting languages. At this point, I'd pick Ruby or Python, learn it, and be happy. It's pretty easy to learn the others afterwords, though you'll spend time with a syntax reference open on your lap.

8.0.2 One-liners

Most scripting languages have some kind of support for "one-liners" – quick bits of code that you may even type in each time at the shell to accomplish a particular task. A good example of a one-liner is changing every occurrance of "foo" to "bar" in a group of files:

```
perl:    perl -pi.bak -e "s/foo/bar/g"
ruby:    ruby -pi.bak -e "gsub('foo', 'bar')"
python:  This is a bit ugly in Python
```

[1]There are as many scripting languages as there are interpreter hackers. The ones I listed are some of the most popular, and justifiably so: they have large support communities, are up to date and maintained, and work well. If you already have a favorite, you're probably set, don't switch.

8.0.3 The Shell

Just because you've learned a high level scripting language, don't neglect the basics of the shell. It's a tool you'll interact with every time you run a command, move files, list directories, etc. As a tool you use so often, learn some of its time-saving features!

Things anyone should be able to do with the shell:

- Basic wildcards: *, {foo,bar}, [12345].
- For loops
- Redirection
- Create aliases that persist beyond logout for common operations

8.0.4 Other useful tools

The m4 macro language is occasionally useful, and understanding m4 is useful when dealing with tools like automake and autoconf. I've recently found myself leaning more towards embedded scripting languages (such as "erb" – embedded ruby), since it's one less tool to learn, and m4 can be a bit, er, opaque and painful.

Command line, pipe-oriented tools like "awk" and "sed" are useful for one-liners, but the common scripting languages can be used there as well. If you find yourself scrounging around in old shell script programs, you may want to invest a bit of time to learn the very basics of awk and sed. I wouldn't become an "awk" expert unless you have a really compelling need. Do understand the most common idiom of sed usage, which is akin to the earlier example in perl:

```
sed 's/search/replace/'
```

which will take all text input to it (though a pipe, etc.) and replace the first occurrence of "search" with "replace". Append '/g' if you want it to replace all occurrences.

Learn the basics of grep. I use some handy aliases to rapidly grep through collections of files recursively from the current directory: These aliases use the 'find' command to traverse all files of particular types and pass them to 'grep'. In csh aliases format, the aliases are:

```
rgrep "find . -type f -print0 | xargs -0 -n 20 grep \!*"
rgrep-c "find . -type f -name '*.c' -print0 | xargs -0 -n 20 grep -H \!*"
rgrep-cc "find . -type f -name '*.cc' -print0 | xargs -0 -n 20 grep -H \!*"
rgrep-h "find . -type f -name '*.[hH]' -print0 | xargs -0 -n 20 grep -H \!*"
rgrep-ch "find . -type f \( -name '*.[cChHly]' -or -name '*.cc' -or \
          -name '*.cpp' -or -name '*.hpp' \) -print0 | xargs -0 -n 20 \
          grep -H \!*"
cgrep "grep -H \!* *.{C,c,cc,cpp,h,H,hh,hpp,l,y,java}"
rgrep-j "find . -type f -name '*.[Jj]ava' -print0 | xargs -0 -n 20 grep -H \
rgrep-html "find . -type f -name '*.ht{m,ml}' -print0 | xargs -0 -n 20 grep
```

8.0.5 "Scripting" languages go well beyond

A nice thing about many of the so-called "scripting" languages is that they go well beyond simple automation. For many small to medium sized (or sometimes larger) programs, Ruby or Python is an ideal implementation environment. In our research, we use Ruby for complex data analysis (with appropriate C-based libraries to do fast math, etc.). Perl, Python, and Ruby all form the basis of innumerable complex, fully featured and powerful Web sites, and the pain to develop these sites is a small fraction of what it would be in a lower-level language such as C.

Time invested in learning a high level, interpreted language will pay itself back *fast.*

- Lots of powerful, high level libraries
- Fast development and prototyping

8.0.6 A Ruby Primer

Programming Ruby [1] is a nice starting reference.

XXX: Add python and perl refs to biblio.

Bibliography

[1] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, second edition, October 2004. ISBN 978-0-9745140-5-5.

Chapter 9

Program Design

How do you design a program? What is involved in its design? There are two critical aspects of the design that you'll need to address:

- **Data structures**: What data does the program operate on? How does it store it?
- **Modules**: How is the program divided into components? How do those components interact?

Some programs are defined primarily by the way they operate on data. For example, suppose you were writing a program to predict the weather. A great deal of your program's design would center around the way you represent huge volumes of weather sensor data, models of the atmosphere, and so on.

Other programs are defined more by what they do or how they interact with the rest of the world. The IRC server project falls into this category: it must send and receive data from users, process their commands, and figure out which other users should receive messages. But beneath this surface description still lingers the issue of data structures: How should your IRC server remember things like:

- What channel is a user in?
- What sockets do I write to to send a message to a user/channel?
- What state is a socket in? Has the user logged in yet?

The second question addresses how you decompose the functionality in your program into individual pieces of source code. There are a nearly arbitrary number of ways you *could* break this down, but in practice, there are ways that make a lot more sense than others.

1. **Design principle 1: Don't Repeat Yourself**

   DRY is a great rule of thumb for deciding what code should be pulled into a module. If you find yourself repeating the same or nearly the same code over and over—or, worse, copy and pasting it!—it's time to consider separating it into a nice, clean module of its own that you can use elsewhere.

2. **Design principle 2: Hide Unnecessary Implementation Details**

   Doing so will make it easier to change those details later without changing all the code that makes use of a module. For example, consider the difference between two functions:

   ```
   int send_message_to_user(struct user *u, char *message)
   ```

   ```
   int send_message_to_user(int user_num, int user_sock, char *message)
   ```

   The first encapsulates the details of the user into a struct. If those details change, the caller functions don't have to do anything different. In contrast, the second exposes more internal details about how the function is going to get data to the user.

Putting these two principles together, suppose that you had a snippet of code in your IRC server like this:

```
int
send_to_user(char *username, char *message)
{
    ....
    struct user *u;
    for (u = userlist; u != NULL; u = u->next) {
        if (!strcmp(u->username, user)) {
            ...
        }
    }
}
```

and you found yourself using that same linked list search again to perform some other operation (say, "kick this user out of a channel"). This would be a good time to sit back and think, "ah-ha! I could really go for a `find_user(username)` function:

```
struct user *
find_user(username)
{
    struct user *u;
    for (u = userlist; u != NULL; u = u->next) {
        if (!strcmp(u->username, user)) {
            return u;
        }
    }
    return NULL;
}
```

This deconstruction has several advantages:

- If the linked list user search became a bottleneck when your system became more popular than Google and had 100,000 users, you could change the search to a hash table lookup without changing any of the callers.

- You've replaced several lines of code with one simple function call that's clearly named and describes exactly what it's doing. The code is more readable and your partner is happier.

Corolary: Write "shy" code: Don't reveal your details to others, and don't interact with too many other modules. If you do need to modify the way a particular module works, you'll need to change all of the places that modify it. Keeping your code relatively "shy" (not interacting with too many other places, when possible) is a good way to help keep the size of the module's interface small and focused. In general, you'll probably find that there are some modules that can be extremely independent: utility functions, lists, hashes, etc.; other modules are more like integrators that make use of these other modules. The important point is that while it's very reasonable for, say, your connection management routines to depend on linked lists, your linked lists shouldn't depend on your connection management routines!

Design principle 3: Keep it simple!

- Don't prematurely optimize
- Don't add unnecessary features

Design some things for reuse: lists, hashes, etc. Put a bit of effort into making them reusable, add good tests, make them a bit general. You'll use them in both project 1 and 2, and beyond.

9.1 Design for Incremental Happiness

It's extremely unlikely that you're going to sit down and, in one uninterrupted spree, type out an entire, working networks project, compile it, and hand it in. It's even more unlikely that you'll do so in bigger projects.

Instead, before you start coding, sit down and think about the incremental steps that you can take along the way. When developing, target these incremental steps so that you break the problem down into small, manageable chunks. Your goal should be that once you've developed a particular "chunk", you can trust it to keep doing its job thereafter. We suggest looking at two different ways of breaking your programs down:

- Identify individual building blocks (lists, hashes, algorithms, etc.)
- Identify "feature" milestones

The first step yields fairly classical modules that implement clearly defined functions. Once you have a general idea of the way your program will work, you should be able to implement a few of these modules.

The second step says to pare down your program to its basic essence before you start to add on the details. In a word processing program, it's probably best to allow the user to enter text before you start working on the spell checking; in your IRC server, you'd probably be better served by accepting connections and buffering lines before you start implementing message routing.

Note that this tip is not just "design incrementally", but "design for incremental *happiness*." Try to pick milestones that:

- Are incrementally testable (see next section);

- Have clearly defined, useful behavior

The first suggestion means that you want to be able to test whether the functions you've implemented work, and to reduce the amount of work you do, you'd like to ensure that any tests you write for the incremental version also work for the final version.

The second suggestion is more for your own motivation: getting a particular bit of your code working for the first time is a great way to stay excited and constantly see real progress in your code, as opposed to feeling like you're hitting a big brick wall that you have to leap over in a single bound.

9.2 Design for Testability

Testability applies at all levels of your code. Getting back to our theme of reducing end-of-project pain and uncertainty, the first goal of testing *during design and coding* is to ensure that once you've built a particular component or module, you can depend on that component to work and keep working. Later, when you have a bug in your program, you can then focus on the newer code and not have to worry *as much* that you're tickling a bug in one of the modules that code uses. This reduces the complexity (and therefore time) of the debugging and later coding, making you more likely to hand your code in on time and working well.

Unit tests are the most basic element of testing. They're particularly appropriate for the kind of well-defined modules you identified earlier when breaking your code down into steps—utility routines, containers, etc. For each of these modules, we suggest writing a set of tests specific to that module. For some modules, you might even consider *writing the tests before you write the code.* Not always, but sometimes, writing the test cases can make you think harder about the interfaces you're exposing and the behavior you've got to implement.

System tests test the bigger picture, such as the actual operation of your IRC server. Note that even here, you can start writing useful tests from the beginning of the project:

- **Connection test**: Can a client connect to the port that your server is listening on?
- **Alternate listen port test**: Can you specify an alternate port on which to have your server listen?

and so on. In general, for these tests, you may be better off writing the tests in a scripting language. For the class, we write many of our tests in Ruby, and many of them using the "expect" package **XXX-have we eliminated expect yet?**

9.2.1 An Adversarial Mindset

When designing tests, put yourself in the mindset of really and truly trying to break the code. It's much cheaper to break (and fix!) the code immediately after you've written it than to have to remember your way through the code (or your partner's) a month later. Don't just test the easy cases, test the boundary cases. For example, suppose you're creating a hash table. Consider testing:

- Insert an item and retrieve it
- Insert two items and retrieve them both

- Insert an item, delete it, and then try to retrieve it
- Delete a non-existent item
- Insert the same item multiple times
- Delete every item in the hash table and then insert more
- Fill the hash table completely, delete, and insert more

Note the order in which those tests are expressed: simple ones first and complex ones later. It's likely that the first time you try a simple test, you'll find some bugs. Once you've fixed those, move on to more complex tests to turn up harder bugs, etc., etc. Much like you would in debugging, try to keep the tests as simple as possible while tickling the corner cases you want to explore, so that when you have to debug the problems they find, you can do so easily.

9.3 Test Automation

As we've discussed, you want to automate your tests. By doing so, you'll run them more often and they'll be all the more useful to you. Ideally, you should be able to type `make test` and have all of your tests run.

9.4 Recommended Reading

Kernighan and Pike's "The Practice of Programming" covers these issues well [1].

Bibliography

[1] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley. ISBN 0-201-61586-X.

Chapter 10

Coding Tricks

- Do tests against constants with the constant first: if (NULL == x). If you leave out an =, it can't be mistaken for assignment, because you're assigning to a constant.

Chapter 11

Human Factors

11.1 Time Management

Students in our class often begin working on their projects late and find themselves scrambling to complete the assignments as the deadline looms ever-nearer. As much as we'd like to, we don't think we're going to be able to fundamentally change this, but we'd like to help you make the process a little more smooth. If there's a single goal of this chapter, it's to eliminate the (very dreadful and unfortunately common) experience of having an "almost complete, almost working" assignment at the time of the deadline that, because of last minute bugs or integration problems, doesn't pass *any* of the tests, and therefore receives a dismal grade that doesn't fairly reflect the effort that its authors put into it.

Note how time mgmt and modularity interact

11.1.1 Planning and Estimating

Estimating development time is *tough*. Real-world, serious programmers get it wrong all the time. In the following section, we outline a few suggestions that can help make the time that your projects require a bit more predictable.

11.2 An Attitude for Software Development

11.2.1 Program Deliberately!

Have you ever found yourself pseudo-randomly modifying statements, throwing in additional API calls, etc., to "see if it works?" (Be honest. I have.) This practice is what Hunt and Thomas refer to as "Programming by Coincidence" [**?**]. *Avoid it like the plague!*

- **Always understand *why* your code works!**

If you don't, it's much more likely that your code doesn't work robustly. It may rely on strange side-effects; it may fail under unexpected boundary conditions; it may break when you add more code. Instead, spend the time to understand how to make the code work properly:

- You'll learn something. A new API, a new language feature, a deeper understanding of the spec, etc.

- You'll be able to debug your code.

And think about it: You could spend twenty frustrating, pointless minutes trying random things, and get code that maybe works sometimes and fails some of the tricky test cases that the evil TAs, professors, and real-world users feed it. Or you could spend the same amount of time (or a bit longer) really learning how to make the code work, and come away a better programmer. Frustration, or making yourself more skilled and valuable? Tough choice. :)

This kind of programming often creeps up when using complex APIs in which operations must be performed in particular orders; examples include things like GUIs or operating system device drivers. ("Maybe if just set that register to zero first...") However, we've seen the same thing creep up in 15-441 when debugging. It's most easily identified by watching the process you take while programming. As soon as you find yourself thinking "I'll just try X," watch out. You're walking very close to the cliff of coincidence.[1]

11.2.2 Fix bugs early

To understand the importance of this point, consider two questions:

1. How much time would it take you to write in C a program that took a (smallish) list of integers on the command line and output them in sorted order? Assume you don't have to be too rigorous about input checking, etc.

2. How much time will it take you to track down the bug that's causing your IRC server to crash after it's been running for a few minutes?

Starting from a blank editor and not using any existing code, it took me about six minutes to write the first program to the point where it worked, and just under five more minutes to add basic error handling. The program is 41 lines long including whitespace. So a reasonable rough answer for question one would be "10 to 30 minutes" or less if a programmer happened to remember the error-handling syntax of the `strtoul` function, which I didn't.[2]

Question two is hard just to answer! With a tricky bug, it could easily take a half an hour or longer just to find a way to reproduce the problem. Or an hour. Or a day, depending on the subtelty of the bug.

As both a programmer and a student in this class, you would be much better off having to implement a bit of new functionality 60 minutes before the deadline than having to debug something that causes your entire assignment to fail. **Fix bugs early!** It will help smooth out the unpredictable time requirements you run into when working on the rest of the assignment.

[1] Note that we don't mind "trying" in a design sense. "I'll try using the system `strstr()` function and see if it's fast enough" is a perfectly valid approach. "I'll try calling `frobzwiggle()` twice to see if it makes the code work" is not.

[2] As an interesting comparison, the same program, *sans* error handling, took two lines and less than one minute to write in Ruby. One line was the path to the Ruby interpreter...

11.3 Team Dynamics

For many of you, this class is the first time you've had to program with a partner. Getting the most out of a team can take a little effort to make sure things work well, but here again, a bit of effort invested is worth it—your team *can* complete the project with each person doing about 51% as much work as it would take to complete it solo. You can approach the problem from three angles (and we recommend tackling all three!):

- Structuring your code so that it can be worked on independently.
- Structuring your development environment to facilitate working as a team.
- Structuring your personal interaction so that it's productive.

11.3.1 Structuring code to work as a team

This gets back to basic issues of design and modularity, except that in a team, it's even more important. Decompose your code into the right modules (and files) so that you can each work on chunks of the code separately. It's even more useful to have easy to use test cases when working as a team, because you will depend on the correctness of code your partner wrote (and which you may not necessarily be familiar with at first). The more the code is easy to test, the easier it is to relax and focus on your own part of the coding.

11.3.2 Structuring your development environment for team work

Don't rely on out-of-build tools and settings - it's too easy to not check these in, etc.

Have a README

Use subversion commit emails so that you each know what's going on. This can also help preserve project momentum.

11.3.3 Structuring your personal interaction

Pair programming is a popular and effective practice.

Deadlines and getting things done earlier.

Dealing with partner problems

It sounds trite, but the two most important things are:

- Communication
- Dealing with problems early

Chapter 12

Editors

Knowing your editor and knowing it well can save you countless keystrokes and small bits of time. In aggregate, these savings add up rapidly.

The `vi` vs `emacs` debates have raged and will continue to rage about people's favorite editor. In many cases, it doesn't matter. There are outstanding, efficient systems hackers who use either. This doesn't mean that all choices are equivalent—writing code with "notepad" or "cat" isn't going to cut it. An advantage to either of these tools is that they are cross-platform. We suggest picking one, and becoming fluent in using it. The skills will serve you well for years.

There are a number of features an editor can offer you (not all do, and that doesn't make them bad), but some things to look for include:

- **Syntax highlighting**: Shading operators, function calls, comments, etc., differently can make it much easier to scan through code for the thing you're looking for.

- **Tags**: Several editors and integrated development environments (IDEs) offer single click or keystroke commands to let you quickly bounce to and from the file in which a function is declared or implemented.

- **Autocompletion**: Editors offer different types of autocompletion, from parenthesis matching, to function name autocompletion, etc..

- **Documentation cross-references**: Some editors and IDEs can pop up context-sensitive help on function names or partial function names. Can be quite handy when using a complex API.

12.0.4 Emacs Tips

This section is strongly informed by a writeup by Adrian Perrig.

How shall we use emacs? Return to our principle of laziness. In this section, we seek to understand our editor well enough to find:

- Minimum keystrokes for maximum return

- Minimum memorization for maximum return

Conventions

Commands and typed text is in `Mono-spaced font`. The keystroke `C-e` means control-e. The keystroke `M-C-e` means Meta-Control-e. In Emacs, Meta is accomplished either by holding down the "Alt" key or by first typing "Escape."

Basic Movement

In keeping with our principles, we seek movement keys that require minimum hand motion and provide us with the most bang for the buck. The following commands seem to accomplish that as a good starting point:

Command	Effect
C-f	Next character
C-b	Previous character
C-p	Previous line
C-n	Next line
M-f	Next word
M-b	Previous word
C-e	End of line
C-a	Beginning of line
C-v	Next page (down)
M-v	Previous page (up)

Command	Effect
C-s and C-r	Incremental search forward or backward.

Command	Effect
M-C-n	Jump forward to the matching parenthesis
M-C-p	Jump backward to the previous parenthesis

Multiple Windows

Command	Effect
C-x o	Switch to other window
C-x b	Exchange the current buffer with a different one
C-x 4 b	Exchange the other half screen buffer with a different one, creating that buffer if it didn't exist.

Auto-completion

Command	Effect
M-/	Find closest word, dynamic abbreviation. Looks backwards and expands to the closest word that starts with the same letters already typed. Searches forward if it couldn't find anything back.
M-TAB	Expand tab: Tries to complete the word to a function or constant name. May open a separate buffer for disambiguation.

Cutting and Pasting

Emacs stores all deleted text (other than backspaced characters) in a "kill ring." If you kill multiple things in a row without intervening commands, it appends these to the same buffer. Otherwise, it starts a new buffer in the kill ring.

C-y will "yank" text from the kill ring. If you hit M-y *after* yanking some text, emacs will change the yanked text to the next older entry in the kill ring, and so on. Give it a try.

Macros

An amazingly useful feature in emacs is the quick construction of small macros.

Command	Effect
C-x (	Start recording a keyboard macro
C-x)	End recording
C-x e	Invoke the last created macro
M-x name-last-kbd-macro	Assigns a name to the last created macro.
M-x insert-kbd-macro	Inserts a definition of the keyboard macro into the current buffer. You can use this to "capture" your recorded macro and save it in your .emacs file for later use. Consider binding

12.0.5 Tags

Tags are a handy, powerful feature in good editors that give the editor a rudimentry knowledge of the symbols, function names, etc., in your programs. Using tags, you can quickly jump to the definition or prototype for a function or use auto-completion for longer symbol and function names.

Emacs uses a TAGS file created by the program `etags`. Creaing the file is simple: run `etags *.[ch]`. Actually, I suggest either creating a makefile target or an alias for it that's a little more complete:

```
etags *.{c,C,cc,cpp,h,hh,hpp,cpp,java} Makefile
```

to cover most of the bases. Once the TAGS file exists, using it is a piece of cake:

Command	Effect
M-.	Visit tags table.
C-u M-.	Finds the next tag
M-0 M-.	Synonym for find-next tag. Easier to type.
C-u -M-.	Find previous tag (negative modifier)
M-tab	Tag complete symbol. Type the first few characters of a function or variable and hit M-tab to autocomplete it.)
M-,	Search for next tab. Find tag is a strict match; this finds sub expressions that contain what you're looking for. Very handy.
Tab	In some prompts from emacs, the tab key will also complete on tag values.

12.0.6 Integrated Development Environments

Unless you have a compelling reason to learn a particular IDE (existing job, project, etc.) we don't suggest a particular IDE. If you do choose one, consider one that works with a variety of underlying languages and operating systems, such as eclipse. Note, however, that the integration and "do it

for you" features in some IDEs can work against you in a systems programming context, when debugging involves multiple processes or machines or operating system kernel code.

Chapter 13

Recommended Resources

List of our favorite *practical* software engineering texts.

The Practice of Programming - tpop.awl.com
The Pragmatic Programmer
Writing Solid Code (Macguire)
Code Complete
The Mythical Man-Month
C++ Programming Style
Debugging the Development Process (Macguire)
The C Programming Language
Things to look at, Dave:
Programming with Gnu Software