Reliable programming

How to write programs that work

- Think about reliability during design and implementation
- Test systematically
- When things break, fix them—correctly
- Make sure everything stays fixed

A reliable design is...

- Modular: You can break it into pieces and verify each piece separately
- Robust: Even nonsensical input will cause it to fail in a predictable way
- Deterministic: If it fails, you can easily make it fail the same way again
- Testable: You can look inside and see why it's doing what it does

Modular design

- If you break a system into pieces, then you can
  - examine the communication between the pieces
  - capture communication samples for examination and replay
  - test each piece separately

Testing pieces separately

- Stable interfaces are the key; changing an interface may
  - force a change in test strategy
  - require coordinated changes on both ends of the changing interface
  - create contradictions among versions
- Interface changes are much harder to test than implementation changes

Robust design

- The best bet is for a component to handle all possible inputs gracefully
- Second best (and often good enough): Fail in a clear way on bad input
- Worse: Do something random on bad input (garbage in, garbage out)
- Worst of all: Don't know what input is good and what isn't

Programming without limits

- One important form of robustness is avoiding fixed limits in programs
  - Library algorithms are a good start
  - "Who needs more than two digits in a year, anyway?"
- If you absolutely must have a fixed limit, document and check for it

Deterministic design

- The hardest programs to test are those that give different output when run twice on the same input
  - interactive programs
  - programs that depend on system state
  - programs that use random numbers
- Unless you isolate indeterminacy, you have a hard struggle ahead of you

Isolating indeterminacy

- Make the indeterminate parts of your program as small as you possibly can
- Define interfaces to the rest of the system
- Capture sample data flow across those interfaces; use them for testing
- Example: Capture keystrokes, mouse events, etc. in a GUI

Testable design

- It is often much easier to determine whether a result is correct than it is to compute it
  - Checking if an array is sorted after the fact
  - Sanity checks on output
  - Data structure audits
- Such tests can often reveal problems before they affect other components

Assertions

- If you write

```
#include <assert.h>
      assert(expr);
```

- If `expr` is zero (false), the program will halt at that point with a diagnostic message (unless preprocessor variable NDEBUG is set).

Logging

- It can often be useful for programs to leave a trail of bread crumbs while they're running
  - Write significant events into a log file
  - Examine the log file afterwards to see if everything worked OK
- You can turn off logging later if it turns out to cost too much

A reliable implementation is…

- Well specified
- Easy to understand
- Easy to explain
- Careful about edge cases

Well specified implementation

- Careful specification is important
  - doesn't have to be formal
  - has to exist—probably in writing
- How can you know if a program is doing what it's supposed to do if you don't know what it's supposed to do?
- A specification gives you a test standard

Clean implementation

- If it's obvious how a program works, it's likely that it does work…
- …and if it doesn't work, the reason is likely to be obvious
- Messy implementation is often a symptom of not understanding the problem thoroughly

Explaining programs

- The best way to be sure you understand something—or to find out where you don't—is to try to explain it to someone else
- This fact is sometimes formalized in inspections, code reviews, etc.
- If you don't understand why something doesn't work, try to prove that it works

Edge cases

- Lots of programs deal with collections
- When asking if such a program works, consider looking at
  - the smallest possible input (usually null)
  - the next smallest input (a single element)
  - the largest possible input
  - one less than the largest possible
  - one more than the largest possible

Proving programs

- Sometimes it is possible to prove that a program works. For example:

```
double power(double x0, double n0) {
  double x = x0;
  unsigned n = n0;
  double r = 1;
  while (n != 0) {
    r *= x;
    --n;
  }
  return r;
}
```

Loop invariant: $r = x^{(n0-n)}$

Systematic testing

- If a component has
  - a clear specification, and
  - input and output that can be captured,

  then it is easy to generate test cases
- You (or—better yet—someone else) can write test cases together with the code
- Be sure you can run them automatically

What is most important to test?

- The most important parts of the program—those on which many other parts depend
- The most difficult parts of the program
- Parts whose performance is critical to the performance of the whole system

Unit testing

- You should have
  - a collection of tests for each component
  - additional tests for the whole system
  - programs to run tests automatically
- If you are sure that each component works before you put the system together, it will save lots of work in testing the whole system

Debugging

- A necessary nuisance
- Easy to get wrong
- Requires a special state of mind
- Many tools available—some of them even work

A necessary nuisance

- Thinking carefully in advance can avoid many bugs
- Looking carefully at your programs and trying to prove them can avoid others
- Type checking and other linguistic tools can avoid still more
- Nevertheless, sometimes things just don't work right

How to avoid making things worse

- Be sure that you're running the program that you're trying to fix
- Be sure it's broken before you fix it
- Be sure that what you're trying to fix is actually the part that's broken
- Be sure that you understand completely why what you're doing will fix the bug that you found

A special state of mind

- Your program is broken because you misunderstood something…
- …but if you knew what you had misunderstood, you wouldn't have misunderstood it
- Therefore, you have to assume that much of what you know might be wrong

How not to do it

- Find a piece of code that looks like it might be related to the problem
- Change it at random and try it out
- If it looks like it worked, you're done
- Otherwise, go back to the beginning

A more productive approach

- Find a piece of code that looks relevant to the problem
- Form a hypothesis about why the code is wrong
- Write a test case that makes it fail
- Fix it and verify that it now works
- Save that test case—you'll need it later

Simplifying the bug hunt

- If you can break it, you're almost done
- What did you change since it worked?
- If the input is right and the output is wrong, what's the first point at which the program is misbehaving?
- This is where the ability to capture inter-module communication comes in handy

Once you think you've found it

- Do you understand how the bug you found accounts for the behavior you saw? All of it?
- Does fixing the bug correct the behavior in the way you expected?
- If not, did you remember to recompile the program before running it again?
- Did fixing it break anything else?

Debugging tools

- Useful tools: stack traces, breakpoints, ability to print variables, etc.
- Less useful for interactive or distributed programs
- Sometimes have their own bugs
- Supplement, not substitute, for careful testing, source code control, etc.

Did fixing it break anything else?

- 100 little bugs in the code,
  100 bugs in the code;
  Fix a bug, compile it again,
  101 little bugs in the code...

Ensuring that bugs stay fixed

- Remember that test case I said you'd need later? Try to build
  - a library of test cases that correspond to bugs that you've found
  - a mechanism for running all those test cases automatically every time you change anything significant
- A good library of test cases can be more valuable than the system itself

Example: compiler test library (1988)

- For every bug report, we created a program that reproduces the bug
- Every time we built an internal version of the compiler (at least weekly), we ran every test case and reported the ones that now failed that used to work

When does a compiler test work?

- If the program is expected to run without diagnostics, we designed it to print pairs of identical output lines
- If the program is expected to produce a message, we put a special comment in the source code; a little program compared the compiler output with the comments to verify the messages

Running batches of tests

- Every test gets a number
- A little program recompiles the compiler and runs every test
- The result is a list of tests passed and failed, which we compare with last week's results
- We really care about tests that used to pass and now fail

Summary

- It is much harder to make a system reliable as an afterthought than to design it that way from the start
- Reliability demands constant vigilance