Testing and Debugging

Programming Tools and Environments

Debugging and testing

- Testing: "running a program with the intent of finding bugs"
- Debugging: once errors are found, "finding the exact nature of a suspected programming error and fixing it"

Testing is difficult

- People by nature assume that what they do is correct
- People by nature overlook minor deficiencies in their own work
- Easy to overlook or ignore bad results
- Easy to choose only test cases that show that the program works
- Good to get someone else’s help on this

95% of the time of debugging is finding the cause of a problem

- syntactic bugs
- design bugs
- logic bugs
- interface bugs
- memory bugs

Defensive programming

Anticipate potential problems and design and code the system so that they are accounted for, or detected as early as possible

- defensive design – Minimise confusion due to complexity
- defensive coding -- take steps to localize problems

Defensive design

- Simplicity of design
- Encapsulation
- Design with error in mind
- Prototype, walk-through
- **Make all assumptions and conditions explicitly**

Encapsulation

- Provide a sufficient interface
  - Hide all data behind an interface
- Minimize coupling (dependency) between classes

Designing with error in mind

- Error handling is often added as an afterthought
- Should be part of the interface
  - Right as you code
- Ask “What if?” often

Design reviews

- Show the design to another group: management, other developers, or outside consultants.
- Writing/presenting a design teaches the designer a lot by making more details explicit in his/her mind.
- Reviewers can provide a new viewpoint on the design different implicit assumptions than the original designer.

Making assumptions explicit: pre- and post-conditions for methods

- A precondition for a method specifies the assumptions that are made on the input parameters and the state of the object when the method is invoked
- A postcondition specifies assumptions made on the output values of the method and the state of the object upon return.

An example

- DeriveExpr parse(istream&);
- Takes an input stream corresponding to one line of the input and returns the expression tree corresponding to that line.

design: how to handle input errors?

- Depends on the application: a simple way would be to return NULL for any invalid input
- Or return a token of subclass DeriveExprError indicating what kind of error and the appropriate error message ("token" is an object – idea is from programming language theory)
  - token can return additional information to the receiver in a data member.

Or throw an exception

new class DeriveParseError

- DeriveExpr parse(istream&) throw (DeriveParseError)

State pre/post conditions in comments

```
DeriveToken nextToken() throw(DeriveParseError);
// Preconditions:
//     The token stream has been initialized with
an //       istream&
//     The token stream may be at the end of the
//     stream.
// Postconditions:
//     Blank space is ignored and the next valid token
//     is returned.  EOS is returned at the end of the
//     stream. DeriveParseError thrown if the next
//     token is invalid.
```

Formal and informal pre/post-conditions

- Pre and post conditions stated in terms of predicate logic are the basis of proofs of correctness and mechanical program verification (Dijkstra, Hoare, Gries & Schneider).
- Pre- and post- conditions stated informally (in words) as documentation should be used only to state non-obvious/important assumptions needed to design and code correctly.

Defensive coding techniques

- Be suspicious
  - public methods should check parameters for proper ranges
  - check that called methods return valid values
  - check for error conditions returned by other routines
  - check values/class fields at intermediate points in a computation when it can be done inexpensively

More defensive coding techniques

- Ensure that variables and fields are initialized to meaningful values
- Keep the *code* simple
- Do a *code* review: double-check code, read it critically

More defensive coding techniques

- Use assertions liberally
- Use exceptions only to indicate and return error conditions
- Use flags to turn off assertions/debug log for release build
- Exceptions should throw types that are defined as“exception classes” (even though C++ lets any type be an exception)
- If a routine can throw an exception declare it to do so

Assertions

- `#include <assert.h>`

  ...

  `assert(x==0);`

  causes program to abort with message (typically, line number of file) if condition is not true when assert line executed.
- `CC –DNDEBUG test.cpp // turns off asserts`

An example of error handling

```
class DeriveParserInfo {
private:
      DeriveTokenStream token_stream;
      DeriveExprFactor expr_factory;
      DeriveToken cur_token;
public:
      DeriveParserInfo(DeriveExprFactory);

      DeriveExpr parse(istream&) throw(DeriveParseError);
      //Preconditions:  cur_token is set to the initial
token;
      //Postconditions:  cur_token is the token after the
expr.
      //    returns a non-NULL expr. or throws
DeriveParseError

      // if the expr is not valid

}
```

Example of error class

```
using namespace std;
#include <exception>

class DeriveParseError : public exception{
private:
      String error_message; // String,ConstText defined
                               // by Reiss
public:
      DeriveParseError(String& msg) {error_message = msg;}

      ConstText message() const     {return
error_message;}

};
```

Evolutionary Programming

- Compile your code often
- Test your code before you implement everything
- Evolve your program by writing modules and using stubs

Debugging techniques

- Use a symbolic debugger (e.g., gdb, or, one provided by your IDE) to locate problems
- If a memory-checking tool is available learn how to use it (to find dangling pointers, memory leaks, etc.)
- THINK
- Keep an error log
- Whenever you discover an error, check for the same error in other parts of the program

Finding errors

- Where is the error? What causes it?
- Use inductive or deductive reasoning

Inductive and deductive reasoning

- Inductive reasoning -- look at the symptoms to determine the cause
  - Once you have a hypothesis, does it fit the facts? Does it explain everything?
  - Don't fix until you are sure
- Deductive reasoning -- think about all the possible causes and think which ones can explain the symptoms
- Both work better with more experience

Inductive reasoning at work: a statistics program

- Evidence
  - Test1: 2 elements (1..2), mean = 1.5, median 1
  - Test 2: 200 elements (1..200), mean = 100.5, median = 100
  - Test 3: 51 elements (0*0..50*50), mean = 842.686, median = 25
  - All values seem correct but test3's median should be 625 rather than 25.

Observations

- One median calculation wrong.
- All means correct.

Hypotheses

- Mean calculation is correct.
- Median calculation is wrong.

between the calculations that worked and the one that didn't?

- Test 1: 2 elements (1..2), mean = 1.5, median 1
- Test 2: 200 elements (1..200), mean = 100.5, median = 100
- Test 3: 51 elements (0*0..50*50), mean = 842.686, median = 25

- Odd number of elements submitted in test 3, others were even
- Geometric sequence used in test 3, others were arithmetic

A bug we find

- The median routine works by sorting the array and returning the index of the middle element, instead of the value of the middle element
- This fits all the symptoms, so it might be the cause of our errors (instead of another bug that’s not the cause).

the code loose among users and get more error reports

- Sometimes the mean is calculated incorrectly
- Rechecking our test cases, we find that
  - Test 3: 51 elements (0*0..50*50), mean = 842.686, median = 625
  - But actually the mean is 841.66667.

Deductive reasoning at work

- What are the possible causes?
  - The data is inaccurate, either the test program set it up wrong, or it is stored wrong.
  - The computation itself is bad, possibly because it used the wrong divisor or summing the sequence incorrectly.
  - The computation is correct but the routine returns the wrong answer.
- Order explanations by probability

Order explanations by probability

- How difficult can it be to input integers? (1st cause unlikely)
- Not all the tests return wrong values. There are no other values around that the routine could mistakenly use to print instead of the right answer. (3rd cause unlikely).
- This leaves us with the second cause -- that the computation itself is bad, as the most likely.

What could cause the computation to be bad?

- The sum is not initialized correctly.
- The sum is not computed correctly (too many, too few, and/or wrong values used)
- The quotient is not computed correctly.

Order explanations by (subjective) probability

- A quick check of the code indicates that the sum is initialized to 0.
- A quick check of the code indicates that the quotient is computed correctly.
- So the iteration used to computed the sum is probably wrong.

Actual cause

- Iterator doesn’t stop in time, goes beyond end of array
- “Extra” array element is usually zero unless the memory has been previously used.

Try keeping an error log

- As you gain more experience, you will get better at both deductive and inductive reasoning -- you will be able to relate it to something you've seen before.
- An explicit error log is a way of increasing the learning effect of making mistakes.

Error logs

- When was the error made?
- Who made the error?
- What was done incorrectly?
- How could the error have been prevented?
- How was the error found?
- How could the error have been detected earlier and with less work?

Think before repairing errors

- Usually fixing the program doesn't fix the error
  - The symptom is caused by several errors.
  - The fix may be incorrect
    - It doesn't fix the problem
    - It causes other problems somewhere else
- Are you fixing the problem or fixing the symptom?
  - (NULL pointer problem -- change at that point to some non-NULL value, without thinking about which non-NULL value is needed)

Testing

- Static testing
  - Code inspections
  - Walk throughs
- Dynamic testing
  - Module testing
  - Integration testing
  - System testing
    - Regression testing (use the same test cases each time)

Software Testing Myths

- If we were really good programmers, there would be no bugs to catch
- Testing implies an admission of failure
- Testing is a punishment for our errors
- All we need to do is:
  - Concentrate
  - Use OO methods
  - Use a good programming language

Software Testing Reality

- Humans make mistakes, especially when creating complex artifacts
- Even good programs have 1-3 bugs per 100 lines of code
- People who claim that they write bug-free code probably haven’t programmed much

Goals of Testing

- Discover and prevent bugs, **not** show that program works
- The act of designing tests is one of the best bug preventers known
- Even tests are sometimes buggy
- The real goal of testing is to reduce the risk of failure to an acceptable level

Functional vs Structural Testing

- Functional testing (black box):
  - Implementation details are "invisible"
  - Program is subjected to inputs, and its outputs are verified for conformance to specified behavior
- Structural testing (white box):
  - Details are visible
  - Exercise the different control and data structures in the program knowing the implementation details

Myths about bugs

- Benign bug hypothesis: bugs are nice, tame, and logical
- Bug locality hypothesis: a bug discovered within a component affects only that component's behavior
- Control bug dominance: most bugs are in the control structure of programs
- Corrections abide: a corrected bug will remain correct
- Silver bullets: a language, design method, environment grants immunity from bugs

Complete Testing

- Complete testing is NOT possible for non-trivial software both practically and theoretically
- Assuming a program only has one input of 10 characters, it would require 2^{80} tests, which at 1microsecond/test would take more than twice the current estimated age of the universe

Test coverage

- Statement coverage: each statement is executed at least once
- Decision coverage: every branch is taken at least once
- Test for invalid, unexpected conditions
- Test for boundary conditions
- Use varied tests

Regression testing

- Every time new code is introduced/bugs are fixed, all old test cases should still produce the correct output
- Every time a new test case uncovers a bug, add it to your suit of test cases

Mutation testing

- Testing technique that focuses on measuring the adequacy of test cases
- Should be used together with other testing techniques
- Based on the *competent programmer hypothesis:* a programmer will create a program, which if incorrect, is very close to the correct program

Mutation Testing

- Faults are introduced into the program by creating many versions of the program called mutants
- Each mutant contains a single fault
- Test cases are applied to the original program and the mutant
- The goal is to cause the mutant program to fail, thus demonstrating the effectiveness of the test case

Example of program mutation

```
void max(int x, int y)   void max(int x, int y)
{ {
int mx = x;      int mx = x;
if (x>y)  if (x<y)
  mx = x; mx = x;
else else
  mx = y; mx = y;
return mx;      return mx;
} }
```

Categories of mutation operators

- Replace an operand with another operand or constant
- Replace an operator or insert new operator
- Delete the else part of the if-else statement
- Delete the entire if-else statement

Testing maxims

- A successful test case is one that finds a bug.
- Always test your code thoroughly.

Mistakes in testing mean/median code

- Didn’t compare to correct answer in test results
- Didn’t adequately test -- should cover all possible executions
  - Test on all possible inputs?
  - Test so that every statement is executed at least once (statement coverage)
  - Test so that all branches are taken (decision/condition coverage)

How to get adequate condition/decision coverage without exhaustive analysis of code

- Test for invalid or unexpected conditions
- Test for boundary conditions
  - If a program wants x in 1..10, give it 0,1,10 and 11.
- Give varied tests
  - Don’t give data all in ascending order

How to find a problem

- THINK.
- If you reach an impasse, sleep on it.
- If you reach an impasse, describe the problem to someone else.
- Use debugging tools as a second resort.
- Use experimentation as a last resort.

How to fix a problem

- Where there is one bug, there is likely to be another.
- Fix the error <u>and</u> the symptoms.
- The probability of the fix being correct is not 100% and drops as the program gets bigger.
- Beware of a fix that creates new errors.
- Error repair is a <u>design</u> process.

Regression testing

- If someone gives you input which produces the bug, make the input part of your test suite after you fix the error.
- Ensures that you don't reintroduce the error in subsequent changes and bug fixes (no going backwards).

Testing guidelines

- A necessary part of a test case is the expected output.
- Avoid attempting to test your own programs.
- Thoroughly inspect the results of each test.
- Test cases must include the invalid and unexpected.
- Check that the program does not do what it is not supposed to do.

More testing guidelines

- Avoid throw-away test cases unless the program is a throw-away program
- Plan testing with the assumption that errors will be found.
- The probability of one or more errors in a section of code is proportion to the number of errors already found in that section.
- Testing is an extremely creative and intellectual challenging task.

Why use a debugger?

- No one writes perfect code first time, every time
- Desk checking code can be tedious and error-prone
- Putting print statements in the code requires re-compilation and a guess as to the source of the problem
- Debuggers are powerful and flexible

Common debugger functions

- Run program
- Stop program at breakpoints
- Execute one line at a time
- Display values of variables
- Show sequence of function calls

The GNU debugger (gdb)

- A debugger is closely tied to the compiler.
  - gcc – gdb, cxx – ladebug, cc - dbx
- Command line debugger for gnu's compilers (gcc, g++)
  - gdb
- The most common way to invoke:
  - gdb *executable*

Invoking gdb

- Start debugging an executable
  - gdb *executable**
- Load a corefile
  - gdb *executable* [-c] *corefile*
- Attach to a running process
  - gdb *executable pid*
    - as long as *pid* is not a file in the current directory

*To look at source code, symbols, etc., must be c

Inspecting a corefile

- You can look at any program that has crashed (and produced a corefile) to see any of its state at the time of the crash
- Load executable and corefile into the debugger
- Use GDB's `backtrace` (`bt`) command to see the call stack

Running a program in GDB

- You can run programs in the debugger
  - see value of variables and expressions
  - look at source code as it's executed
  - change the value of variables
  - move the execution pointer
  - many other things

Compile for debugging

- When compiling your program, add the –g flag to the command line:
  - `gcc -g -o prog prog.c`
- This adds extra symbol information, so the debugger knows how you called the variables in your source, can show you the source code and which line will be executed next

Looking at your source

- `list` or `l` (list code)
  - `list`
  - `list main`
  - `list 56`
  - `list 53,77`

Breakpoints

- A place where execution pauses, waits for a user command
- Can break at a function, a line number, or on a certain condition
  - `break` or `b` (set a breakpoint)
    - `break main`
    - `break 10`
  - `watch` *`expr`*

Execution commands

- `run` or `r` (run program from beginning)
  - `run`
  - `run` *`argList`*
- Or, you can set arguments to be passed to the program this way:
  - `set args` *`arglist`*
- `start`
  - starts debugging, breaks at main
- `kill`
  - stops debugging

More Execution commands

- `next` or `n`
  - execute next line, stepping over function calls
- `step` or `s`
  - execute next line, stepping *into* function calls
- `continue` or `cont`
  - resume execution, until next breakpoint

Examining data

- `print` or p (print value)
  - `print x`
  - `print x*y`
  - `print function(x)`
- `printf`
- `display` (continuously display value)
- `undisplay` (remove displayed value)
- `where` (show current function stack)
- `set` (change a value)
  - `set n=3`

Miscellaneous commands

- `help` or h (display help text)
  - `help`
  - `help step`
  - `help breakpoints`
- `quit` or q (quit gdb)