Errors

Lecture 5

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Course Project

Course Project

- Everybody should have received email with group number to join
  - Everybody must accept project using the link in email
- Milestone 1: Proposal
  - Project portfolio template: in the project repository
- System Requirements
  - Name
  - User stories
  - Core Features, viable features, stretch features
- Management
  - Continuation of operation plan (COOP)
    - Communication, contingency,
  - Project Plan Proposal
- Deadline for this: Friday, February 9, 11.59pm
  - Push edited document to repository

Software Development Notes

Manual Testing

- The most basic form of testing
  - Run the program yourself
  - When an error occurs, write it down
  - Ensure that the error can be reproduced

- Create issue on Github
  - How to run? What input used?
  - What system run on?
  - Everything needed to reproduce
  - What result is expected? What result is seen?

Issues with Manual Testing

- To be effective, bugs must be tracked
  - Bug reproduction steps must be carefully retained
  - Have to re-test after changes
- Expensive and error-prone

Automated Testing

- Manual testing is still necessary for sanity checks
- But we can create automated tests to provide immediate feedback
- Tests are automatically run for each change set (commit)
- We receive a checklist afterwards seeing how many, and which, tests succeeded and failed

Test Framework

- Testing and the Catch2 framework:
  - https://teaching.hkaiser.org/resources/testing.html
- Read this to learn everything!
- Integrates well with VSCode
  - Run tests before pushing, make sure all tests pass
- Integrates well with Github automated testing
  - Each push triggers the tests – immediate feedback
  - This is the basis for grading!

Unit Testing

- The most common kind of automated testing is Unit Testing
- Unit testing is a form of automated testing where you test a single class, module, or method
- A unit is the smallest testable portion of an application:
  - Each unit test tests one thing
  - We can test every function of the unit, and every meaningful case of the function
  - Pay particular attention to testing boundary cases
    - Zero length strings
    - Check at minimum/maximum index bounds

Regressions

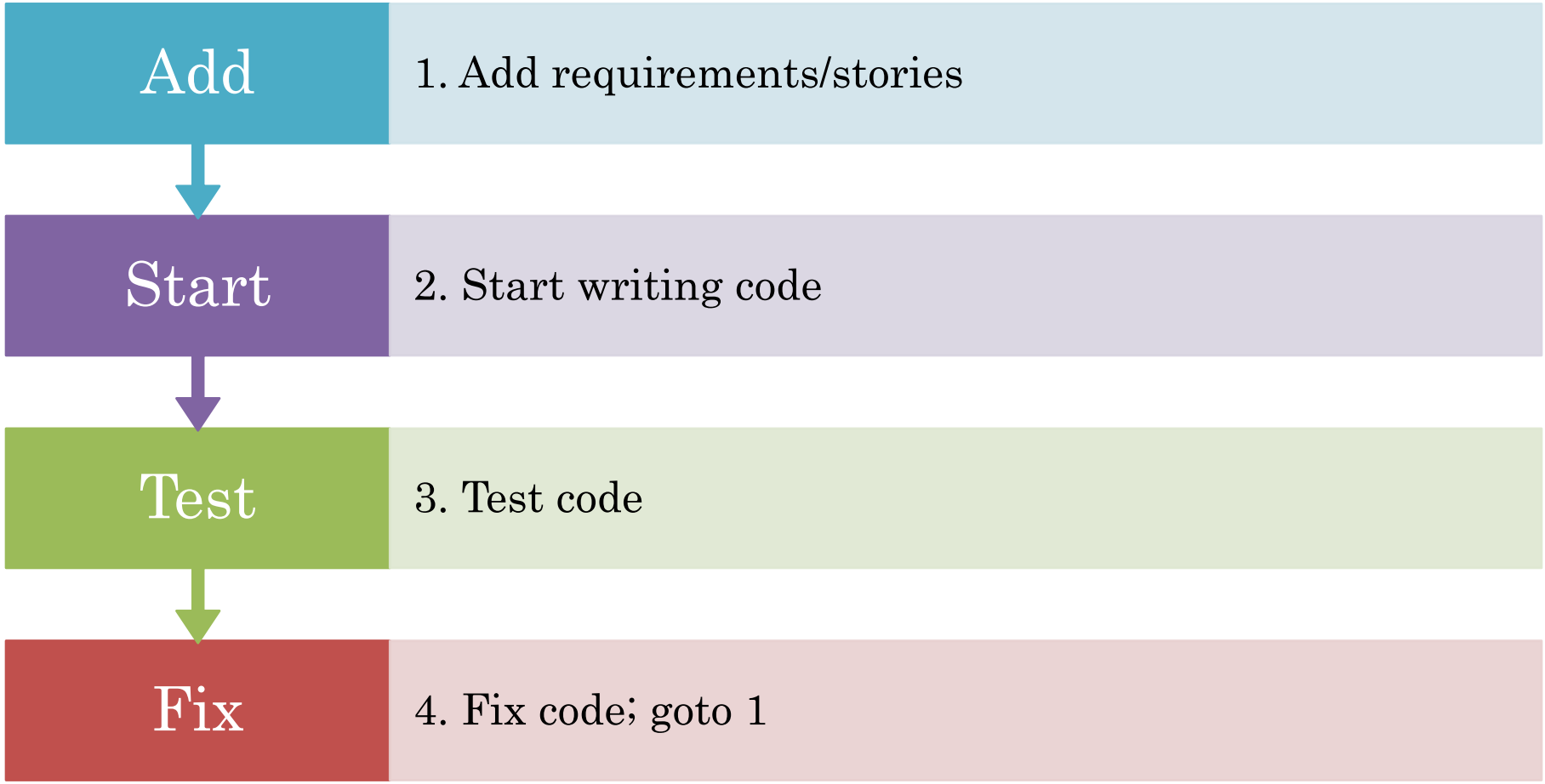
- It is extremely common for a new update to fix old bugs, but introduce new ones
- If a test used to pass, but after applying a change set it fails, this is called a regression
- Automated testing provides a mechanism to quickly discover regressions
- When fixing regressions always add a test verifying it has been fixed

Be Careful

- The natural instinct is to rely on automated testing
- It is NOT a replacement for manual testing
- And manual testing is not a replacement for code review

- In studies, code review catches more difficult-to-detect bugs than testing!
  - Juristo, N., & Vegas, S. (2003). Functional testing, structural testing and code reading: What fault type do they each detect?. In Empirical Methods and Studies in Software Engineering (pp. 208-232). Springer, Berlin, Heidelberg.

Normal Process



Guidelines

Use

Use the CHECK methods instead of plain assert for nicer error messages.

- Allows to see why the test failed

Don't cram

Don't cram too much into one test

- Test one thing at a time

Don't mix

Don't mix tests for different classes in unit tests

- This is a different kind of testing called "integration testing" which we'll talk about later.

Avoid

Avoid making tests depend on one another.
Don't call tests from tests.
Factor out common code into methods and call those.

Errors and Error Handling

Abstract

- When we program, we have to deal with errors. Our most basic aim is correctness, but we must deal with incomplete problem specifications, incomplete programs, and our own errors. Here, we'll concentrate on a key area: how to deal with unexpected function arguments. We'll also discuss techniques for finding errors in programs: debugging and testing.

Overview

- Kinds of errors
- Argument checking
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Testing

Errors

- " … I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  - Maurice Wilkes, 1949
- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors.
  - Eliminate most of the errors we made anyway.
    - Debugging
    - Testing
  - Make sure the remaining errors are not serious.
- My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
  - You can do much better for small programs.
    - or worse, if you're sloppy

Your Program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

- 3, 4, and 5 are true for beginner's code; often, we have to worry about those in real software.

Sources of Errors

- Poor specification
  - "What's this supposed to do?"
- Incomplete programs
  - "but I'll not get around to doing that until tomorrow"
- Unexpected arguments
  - "but sqrt() isn't supposed to be called with -1 as its argument"
- Unexpected input
  - "but the user was supposed to input an integer"
- Code that simply doesn't do what it was supposed to do
  - "so fix it!"

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
  - Undefined symbols
  - Non-existing libraries
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Check your Inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (istream)
- Make sure the expectations of the algorithm (function) are met (preconditions)
  - Argument to sqrt is non-negative
  - Index used to access element of array is in range
  - Iterator is not equal to end iterator
  - Sequence passed to unique is actually sorted
  - Etc.

Bad Function Arguments

- The compiler helps:
  - Number and types of arguments must match

```
int area(int length, int width)
{
    return length*width;
}

int x1 = area(7);            // error: wrong number of arguments
int x2 = area("seven", 2);   // error: 1st argument has a wrong type
int x3 = area(7, 10);        // ok
int x5 = area(7.5, 10);      // ok, but dangerous: 7.5 truncated to 7;
                             //      most compilers will warn you

int x = area(10, -7);        // this is a difficult case:
                             // the types are correct,
                             // but the values make no sense
```

Bad Function Arguments

- So, how about int x = area(10, -7);
- Alternatives
  - Just don't do that
    - Rarely a satisfactory answer
  - The caller should check
    - Hard to do systematically
  - The function should check
    - Return an "error value" (not general, problematic)
    - Set an error status indicator (not general, problematic – don't do this)
    - Throw an exception
- Note: sometimes we can't change a function that handles errors in a way we do not like
  - Someone else wrote it and we can't or don't want to change their code

Bad Function Arguments

- Why worry?
  - You want your programs to be correct
  - Typically the writer of a function has no control over how it is called
    - Writing "do it this way" in the manual (or in comments) is no solution – many people don't read manuals
  - The beginning of a function is often a good place to check
    - Before the computation gets complicated
- When to worry?
  - If it doesn't make sense to test every function, test some

How to Report an Error

- Return an "error value" (not general, problematic)

```
// return a negative value for bad input
int area(int length, int width)
{
    if (length <= 0 || width <= 0) return -1;
    return length*width;
}
```

- So, "let the caller beware"

```
int z = area(x,y);
if (z < 0) error("bad area computation");
   // …
```

- Problems
  - What if I forget to check that return value?
  - For some functions there isn't a "bad value" to return (e.g. max())

How to Report an Error

- Set an error status indicator (not general, problematic, don't!)

```
int errno = 0;      // used to indicate errors
int area(int length, int width)
{
    if (length <= 0 || width <= 0) errno = 7;
    return length*width;
}
```

- So, "let the caller check"

```
int z = area(x, y);
if (errno == 7) error("bad area computation");
   // …
```

- Problems
  - What if I forget to check errno?
  - How do I pick a value for errno that's different from all others?
  - How do I deal with that error?

How to Report an Error

- Report an error by throwing an exception

```
class bad_area : std::exception {};     // a class is a user defined type
                                        // bad_area is a type to be used as an exception

int area(int length, int width)
{
    if (length <= 0 || width <= 0) throw bad_area();   // note the ()
    return length * width;
}
```

- Catch and deal with the error (e.g., in main())

```
try {
    int z = area(x,y);  //  if area() doesn't throw an exception
}                       // make the assignment and proceed
catch (bad_area const&) {      // if area() throws bad_area(), respond
    cout << "oops! Bad area calculation – fix program\n";
}
```

How to Report an Error

```
int main()
{
    std::cout << "Testing exceptions." << std::endl;
    try {
        throw runtime_error("Doh! Something went wrong!");
        std::cout << "Here we are, still alive and kicking!" << std::endl;
    }
    catch (runtime_error e) {
        std::cout << e.what() << std::endl;
    }
    std::cout << "Done testing exceptions." << std::endl;
    return 0;
}
```

```
Testing exceptions.
Doh! Something went wrong!
Done testing exceptions.
```

Exceptions

- Exception handling is general
  - You can't forget about an exception: the program will terminate if someone doesn't handle it (using a try … catch)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is never really simple

Out of range

- Try this

```
vector<int> v(10); // a vector of 10 ints, each
                   // initialized to the default value,
                   // 0, referred to as v[0] .. v[9]

for (int i = 0; i != v.size(); ++i)
    v[i] = i;  // set values

for (int i = 0; i <= 10; ++i)     // print 10 values (???)
    cout << "v[" << i << "] == " << v.at(i) << endl;
```

- vector's operator[ ] (subscript operator) reports a bad index (its argument) by throwing a std::range_error if you compile in 'Debug' mode (MSVC only)
  - The default behavior can differ, use at() to guarantee checking

Exceptions – for now

- For now, just use exceptions to terminate programs gracefully, like this

```
int main()
{
    try {
        // …
    }
    catch (std::out_of_range const&) {  // out_of_range exceptions
        std::cout << "oops – some vector index out of range\n";
    }
    catch (...) {                       // all other exceptions
        std::cout << "oops – some exception\n";
    }
}
```

Getting an Integer from an Input Stream

- Read one integer from cin:

```
int get_int(std::string const& prompt)
{
    std::cout << prompt;
    int n = 0;
    while (true)
    {
        if (std::cin >> n)
            return n;
        std::cout << "sorry, this was not a number, try again." << std::endl;
        skip_to_int();
    }
}
```

Getting an Integer from an Input Stream

- Read one integer from cin, skip to next valid input character:

```
void skip_to_int()
{
    if (!std::cin)
    {
        std::cin.clear();
        char ch;
        while (std::cin >> ch)
        {
            // throw away non-digits
            if (std::isdigit(ch) || ch == '-')
            {
                std::cin.unget(ch);        // put the digit back
                return;
            }
        }
    }
    throw domain_error("no input");
}
```

How to Look for Errors

- When you have written (drafted?) a program, it'll have errors (commonly called "bugs")
  - It'll do something, but not what you expected
  - How do you find out what it actually does?
  - How do you correct it?
  - This process is usually called "debugging"

Debugging

- How not to do it

```
// pseudo code
while (program doesn't appear to work) {
    Randomly look at the program for something
    that "looks odd".
    Change it to "look better"
}
```

- Key question:
  - How would I know if the program actually worked correctly?

Program structure

- Make the program easy to read so that you have a chance of spotting the bugs
  - Comment
    - Explain design ideas
  - Use meaningful names
  - Indentation and formatting
    - Use a consistent layout
    - Your IDE tries to help (but it can't do everything)
      - You are the one responsible
  - Break code into small functions
    - Try to avoid functions longer than a page
  - Avoid complicated code sequences
    - Try to avoid nested loops, nested if-statements, etc.
      - (But, obviously, you sometimes need those)
  - Use library facilities
  - Use consistent error handling

First Get the Program to Compile

- Is every string literal terminated?

```
cout << "Hello, << name << '\n';   // oops!
```

- Is every character literal terminated?

```
cout << "Hello, "  << name << '\n; // oops!
```

- Is every block terminated?

```
if (a > 0) { /* do something */
else { /* do something else */ }   // oops!
```

- Is every set of parentheses matched?

```
if (a                                   // oops!
        x = f(y);
```

- The compiler generally reports this kind of error "late"
  - It doesn't know you didn't mean to close "it" later

First Get the Program to Compile

- Is every name declared?
  - Did you include needed headers? (e.g., iostream, etc.)

- Is every name declared before it's used?
  - Did you spell all names correctly?

```
int count;  /* … */ ++Count;     // oops!
char ch;    /* … */ Cin >> c;    // double oops!
```

- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y) + 2   // oops!
z = x + 3;
```

Debugging

- Carefully follow the program through the specified sequence of steps
  - Pretend you're the computer executing the program
  - Does the output match your expectations?
  - If there isn't enough output to help, add a few debug output statements

```
cout << "x == " << x << ", y == " << y << '\n';
```

- Be very careful
  - See what the program specifies, not what you think it should say
    - That's much harder to do than it sounds

```
for (int i = 0; 0 != month.size(); ++i) {    // oops!
for (int i = 0; i <= max; ++j) {       // oops! (twice)
```

Debugging

- When you write the program, insert some checks ("sanity checks") that variables have "reasonable values"
  - Function argument checks are prominent examples of this

```
if (number_of_elements < 0)
    throw runtime_error("impossible: negative number of elements");

if (number_of_elements > largest_reasonable)
    throw runtime_error("unexpectedly large number of elements");

if (x < y)
    throw runtime_error("impossible: x < y");
```

- Design these checks so that some can be left in the program even after you believe it to be correct
  - It's almost always better for a program to stop than to give wrong results

Debugging

- Pay special attention to “end cases” (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
  - Did you open your files correctly?
    - more on this later
  - Did you actually read that input?
    - Write that output?

Debugging

- "If you can't see the bug, you're looking in the wrong place"
  - It's easy to be convinced that you know what the problem is and stubbornly keep looking in the wrong place
  - Don't just guess, be guided by output
    - Work forward through the code from a place you know is right
      - so what happens next? Why?
    - Work backwards from some bad output
      - how could that possibly happen?
- Once you have found "the bug" carefully consider if fixing it solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I found the last bug"
  - is a programmer's joke

Note

- Error handling is fundamentally more difficult and messy than "ordinary code"
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- The more people use a program, the better the error handling must be
  - If you break your own code, that's your own problem
    - And you'll learn the hard way
  - If your code is used by your friends, uncaught errors can cause you to lose friends
  - If your code is used by strangers, uncaught errors can cause serious grief
    - And they may not have a way of recovering

Pre-conditions

- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
// calculate area of a rectangle
int area(int length, int width)
{
    // length and width must be positive
    if (length <= 0 || width <= 0)
        throw bad_area();
    return length * width;
}
```

Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
// calculate area of a rectangle
int area(int length, int width)
{
    // length and width must be positive
    if (length <= 0 || width <= 0)
        throw bad_area();

    // the result must be a positive int that is the area
    int result = length * width;

    if (result <= 0)        // how could this happen anyways???
        throw bad_area();

    return result;
}
```

Pre- and post-conditions

- Always think about them
- If nothing else write them as comments
- Check them “where reasonable”
- Check a lot when you are looking for a bug
- This can be tricky
  - How could the post-condition for area() fail after the pre-condition succeeded (held)?

Testing

- How do we test a program?
  - Be systematic
    - "pecking at the keyboard" is okay for very small programs and for very initial tests, but is insufficient for real systems
  - Think of testing and correctness from the very start
    - When possible, test parts of a program in isolation
      - E.g., when you write a complicated function write a little program that simply calls it with a lot of arguments to see how it behaves in isolation before putting it into the real program
  - We'll return to this question later

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP