CSE 1325: Object-Oriented Programming Lecture 04 – Chapters 05, 26

Errors, Exceptions, Testing, and Debugging

Mr. George F. Rice george.rice@uta.edu

Based on material by Bjarne Stroustrup www.stroustrup.com/Programming

ERB 402
Office Hours:
Tuesday Thursday 11 - 12
Or by appointment



Lecture #3 Quick Review

- Dividing big computations into many little ones is called <u>decomposition</u>.
- Providing a higher-level concept that hides detail is called <u>abstraction</u>.
- Why are concise operators (+=) usually preferable to infix operators (= ... + ...)? They are clearer and express our intent better
- Which of the following is a C++ statement? All of them
 - a. An expression terminated with a semicolon
 - b. A declaration
 - c. A "control statement" that determines the flow of control
- Describe the two different "for" loops in C++, and when the use of each is preferable? The 3-term loop is for general counting, input, and other general "loop until condition" constructs. The for-each loop is for iterating over vectors and similar containers.
- A procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute, is called a(n) <u>algorithm</u>.

Lecture #3 Quick Review

Which of these represents a "class" in UML? b does

a.

format:String
formatls()
readItem()
writeItem()

#Dead #NoEmotion +Stalk() Ranges o
Action

Range_1
Range_2
Size_3

• In a UML class diagram, public visibility of a field or method is indicated by a plus (+) while private visibility by a minus (-).

+Haunt()

- A UML <u>association</u> relationship indicates a reference from one class to another.
- The notation "« interface »" (a name inside of guillemets) is called a <u>stereotype</u>. This is one of 3 mechanisms to customize UML.
- A general reusable algorithm solving a common problem is called a software design pattern.
 - How are new ones created? By analyzing a LOT of code, proposing the pattern, and then waiting to see if many other engineers concur ("meritocracy")
- The <u>singleton pattern</u> restricts a class to instancing a single object, typically to coordinate actions across a large system.
 - What is the key to implementing this? A private constructor

Homework #2: Questions?

- Create an OO-based program that accepts a student's name and exam grades, and outputs their semester average.
 - Bonus: Also include homework grades in their semester average.
 - Extreme Bonus: Read the data from a file, and write the same file in the same format but with additional information added.
- As always, details are on Blackboard.
 - Requirements are in a ZIP archive along with test data

"Now is the time for all good persons to come to the aid of their final average."

— Patrick Henry*

```
student
- student_name : string
- exam_sum : double
- exam_num_grades : double
+ Student(name : string)
+ name() : string
+ exam(grade : double)
```

```
+ average() : double
                                      make clean
rm -f *.o main test student
ricegf@pluto:~/dev/cpp/201708/P2/fc$ make
g++ -std=c++11 -o main main.cpp
./main
Enter student's name: Fred
Enter next grade: 100
Enter next grade: 87
Enter next grade: 98
Enter next grade: 93
Enter next grade: 97
Enter next grade: -1
Fred has a 95 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$ ./main
Enter student's name: Ruth
Enter next grade: -1
Ruth has a 100 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$
```

Today's Topics

- Errors
 - Avoiding
 - Detecting
 - Reporting
 - cerr
 - Exceptions
- Testing
- Debugging



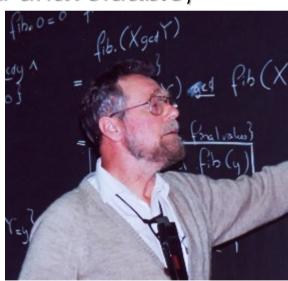
Errors

- " ... 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



"Program testing can be used to show the presence of bugs, but never to show their absence!"

Dr. Edsger Dijkstra

For This Class, Your Program

- Should produce the desired results for all legal inputs
- Should give a reasonable error message for every illegal input
- Need not worry about misbehaving hardware
- Need not worry about misbehaving system software
- Is allowed to terminate with an informative message after finding an error

Sources of errors

- Ambiguous Requirements — Have I mentioned this one yet?
 - "What's this supposed to do?"
- Incomplete programs
 - "I'll get around to that ... tomorrow"
 - Traditionally, pending work is marked with TODO (or #TODO)
- 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
 - "I don't always test my code, when when I do, I test it in production!"

Your JOB is to reasonably respond to all possible events without introducing undue burden on the primary use case(s)

Kinds of Errors

- Compile-time errors
 - Syntax errors
 - Type errors
- Link-time errors
 - Missing libraries
 - Missing .o files
- Run-time errors
 - Unreported (crash)
 - Reported (exceptions)
 - Handled (exception handlers)

problem has been detected and Windows has been shut down to prevent damage o vour computer.

RIVER_IRQL_NOT_LESS_OR_EQUAL

f this is the first time you've seen this Stop error screen, estart your computer. If this screen appears again, follow

heck to make sure any new hardware or software is properly installed. f this is a new installation, ask your hardware or software manufacturer or any Windows updates you might need.

f problems continue, disable or remove any newly installed hardware n software. Disable BIOS memory options such as caching or shadowing. f you need to use Safe Mode to remove or disable components, restart our computer, press F8 to select Advanced Startup Options, and then elect Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x000000000,0xF86B5A89)

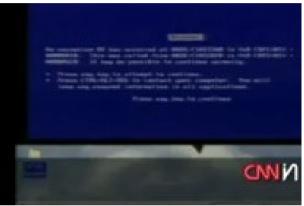
gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory hýsical memory dump complete. ontact your system administrator or technical support group for further:

- Logic errors
 - Detected by automated tests (regression test, specification tests)
 - Detected by programmer (code runs, but produces incorrect output)

Embarrassing BSoDs





Microsoft USB Demo at Comdex

Nine Inch Nails, in concert



Bank of America ATM



2008 Olympics in Beijing

Avoiding Errors Check your inputs

- Before trying to use an input value, check that it meets your expectations/requirements
 - Function arguments
 - Data from input (istream)
- EXAMPLE: SQL Injection Attack (common against poorly coded websites)

```
SELECT * FROM items WHERE owner = 'ricegf' AND itemname = 'book';
```

```
SELECT * FROM items WHERE owner = 'ricegf' AND itemname = 'book' OR 'a'='a';
```

Avoiding Errors Catching Bad Function Arguments

- Why worry?
 - Your code should have a reputation for "robustness"
 - Others will call your functions incorrectly
 - So will you!
 - Be gentle with others and yourself
 - Documentation doesn't help who reads documentation?
 - 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, at least test some of the functions

The mark of professionally written code is excellent regression tests

Avoiding Errors

Catching Bad Function Arguments

- The compiler can help
 - Number and types of arguments must match
- The compiler cannot help:
 - Assumptions about arguments must match

Avoiding (and Reporting) Errors Catching 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
 - Lots of unnecessary overhead
 - The function should check
 - Return an "error value" (not general, problematic but "the C way")
 - 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

Options for Error Reporting Returning an "Error Value"

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; // || means or
   return length*width;
}</pre>
```

So, "let the caller beware"

```
int z = area(x,y);
if (z<0) cerr << "bad area computation" << endl;
// ...</pre>
```

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

Very common!

Options for Error Reporting Setting an "Error Status"

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;
}</pre>
```

So, "let the caller check"

```
int z = area(x,y);
if (errno==7) cerr << "bad area computation" << endl;
// ...</pre>
```

- 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?

Options for Error Reporting Throwing an Exception

- Report an error by throwing an exception
- Handle the error by catching the exception

```
#include <iostream>
                              Include the exception library
#include <exception> ◀
using namespace std;
                                               Define a new exception
class Bad area: public exception { };
                                               (don't sweat the syntax yet)
int area(int length, int width) {
  if (length<=0 || width<=0) throw Bad area{ }; // note the {} - a value
  return length*width;
                                      Create (instance) the exception object
int main() { Begin watching for an exception
  try 🖊
                                        Catch the exception with a matching type
    cout << area(3, 5) << endl;</pre>
                                        (e.g., Bad area) and then execute the code
    cout << area(8, 2) << endl;</pre>
                                        in the new scope (i.e., inside the { })
    cout << area(3,-5) << end1;
    return 0; // success
                                                        ricegf@pluto:~/dev/cpp/04$ ./a.out
  } catch(Bad area e) {
    cerr << "Bad area call - fix program!" << endl;</pre>
                                                        Bad area call - fix program!
    return 1; // failure
                                              Use cerr to send text to STDERR, similar
        Return a non-zero result from main
                                              to using cout to send text to STDOUT.
        to signal an error to the OS
```

Why cerr and Not cout?

```
ricegf@pluto:~/dev/cpp/04$ cat test_exception_cout.cpp
 #include "std_lib_facilities.h"
 class Bad_area { }; // any class can be used as an exception
 int area(int length, int width) {
   if (length<=0 || width<=0) throw Bad_area{}; // note the {} - a value
   return length*width:
 int main() {
   try {
     cout << area(3, 5) << endl;</pre>
     cout << area(8, 2) << endl:
     cout << area(3,-5) << endl;</pre>
   } catch(Bad_area e) {
     cout << "Bad area call - fix program!" << endl;</pre>
     return -1:
   return 0;
ricegf@pluto:~/dev/cpp/04$ g++ -w test_exception_cout.cpp
ricegf@pluto:~/dev/cpp/04$ ./a.out > text
ricegf@pluto:~/dev/cpp/04$ g++ -w test_exception.cpp
ricegf@pluto:~/dev/cpp/04$ ./a.out > text
Bad area call - fix program!
ricegf@pluto:~/dev/cpp/04$
```

Redirecting STDOUT causes errors on **cout** to "disappear"

Redirecting STDOUT does NOT cause errors on **cerr** to "disappear"

```
ricegf@pluto:~/dev/cpp/04$ cat avoid exception.cpp
 #include "std lib facilities.h"
 class Bad_area { }; // any class can be used as an exception
 int area(int length, int width) {
   if (length<=0 || width<=0) throw Bad_area{};</pre>
   return length*width;
 int main() {
   trv {
     cout << area(3, 5) << endl;</pre>
     cout << area(8, 2) << endl;</pre>
 // cout << area(3,-5) << endl;
   } catch(Bad_area e) {
     cerr << "Bad area call - fix program!" << endl:
     return -1;
   return 0;
ricegf@pluto:~/dev/cpp/04$ g++ -w avoid_exception.cpp
ricegf@pluto:~/dev/cpp/04$ ./a.out
15
16
ricegf@pluto:~/dev/cpp/04$ if [ $? -eq 0 ]
 then
> echo Success!
 else
 echo Failure!
> fi
Success!
ricegf@pluto:~/dev/cpp/04$ g++ -w test_exception.cpp
ricegf@pluto:~/dev/cpp/04$ ./a.out
15
Bad area call - fix program!
ricegf@pluto:~/dev/cpp/04$ if [ $? -eq 0 ]
> then
> echo Success!
 else
 echo Failure!
> fi
Failure!
```

ricegf@pluto:~/dev/cpp/04\$||

// note the {} - a Wahy return != 0?

Return -1 on an exception, or 0 on no exception.

Bash responds to success of the program.

Bash responds to failure of the program.

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

Exception – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code.

Exception Handling Outline

- Include the exception library (#include <exception>)
- Declare an exception (class Bad area : public exception();)
 - Optional you can (and often should) use a pre-defined exception
 - runtime_error("Bad dates") is a popular choice (#include <stdexcept>)
- Throw an exception when an error occurs (throw Bad_area{ };)
 - Optional many library methods already throw exceptions
- Define a scope in which to watch for an exception (try { })
 - The code inside the curly braces is monitored for exceptions
- Immediately after the try scope, define one or more exception handling scopes (catch (Bad_area e) { })
 - Add code inside the curly braces to handle each exception
- If exiting, return a non-zero result to report the error to the OS
 - For console apps, but generally not for graphical apps

Exceptions to Exceptions

Try this

v[10] == 132033

ricegf@pluto:~/dev/cpp/201701/04\$

```
ricegf@pluto:~/dev/cpp/201701/04$ g++ -std=c++11 out_of_range.cpp
ricegf@pluto:~/dev/cpp/201701/04$ ./a.out

v[0] == 0
v[1] == 1
v[2] == 2
v[3] == 3
v[4] == 4
v[5] == 5
v[6] == 6
v[7] == 7
v[8] == 8
v[9] == 9

Wait — no exception???
C++ generally assumes that you know what you are doing.
This is often a bad assumption even for experienced programmers.

Caveat scriptor — Let the programmer beware!
```

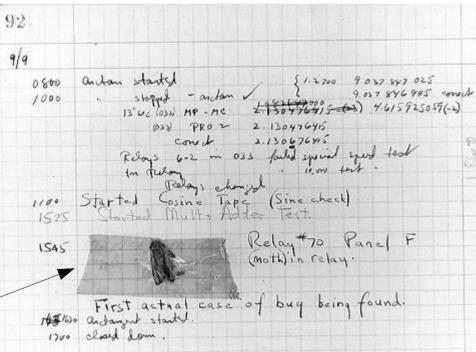
After the Exception is Caught A Note on Error Handling

- Error handling is fundamentally more difficult and messy than "ordinary code"
 - There is basically just one way things can work right
 - There are <u>many</u> ways that things can go wrong
- The more people use a program, the better the error handling must be (and eventually will be if you're paying attention!)
 - 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





- Every program that you write <u>will</u> have errors (commonly called "bugs")
 - It won't do what you want
 - It will do what you don't want
 - It will exit with an exception or a core dump
 - It will lock up the machine (!)
- Fixing a program is called "debugging"



First bug ever found in a program

Bugs Meany image copyright 1963 by Donald Sobol and Leonard Shortall. Fair use for educational purposes is asserted.

First bug image Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. U.S. Naval Historical Center Online Library Photograph NH 96566-KN

Step 0 – Before You Test Minimize Bugs First

- Clearly written code helps to avoid bugs
 - Comments
 - Explain design ideas, NOT how C++ works
 - Use meaningful names, e.g., student_name not n
 - Indent
 - Use spaces, not tabs! Editor tab settings vary
 - Use a consistent visual layout whitespace is important
 - A good code-sensitive editor will help
 - Break code into small functions and methods
 - Try to avoid methods longer than a page
 - Avoid complicated code sequences
 - Try to avoid deeply nested loops, nested if-statements, etc.
 (But, obviously, you sometimes need those)
 - Use library facilities
 - The most reliable code you will ever write is a library call

Step 0 – Before You Test Detecting Bugs in Advance

- Include assertions ("sanity checks") that variables have "reasonable values"
 - Function argument checks are prominent examples of this

```
if (number_of_elements < 0)
    cerr << "impossible: negative number of elements" << endl;
if (number_of_elements > largest_reasonable)
    cerr << "unexpectedly large number of elements" << endl;
if (x<y) cerr << "impossible: x < y" << endl;</pre>
```

 Design these checks so that some can be left in the program even after you believe it to be correct

Better for a program to stop than to give wrong results

More on Assertions 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

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

Challenge

How could the result for area() fail even if the pre-condition succeeded (held)?



More on Assertions

Edge Cases and Post-conditions

```
#include <iostream>
#include <exception>
#include <climits>
using namespace std;
class Bad area: public exception { };
int area(int length, int width) { // calculate area of a rectangle
   // length and width must be positive
   if (length<=0 || width <=0) throw Bad area{};
   return length*width;
                                       ricegf@pluto:~/dev/cpp/201701/04$ g++ -std=c++11 max int.cpp
                                       riceqf@pluto:~/dev/cpp/201701/04$ ./a.out
int main() {
                                       Area of 14 x 10 is 140
                                       Area of 2147483647 x 2 is -2
   int length1 = 14;
                                       ricegf@pluto:~/dev/cpp/201701/04$
   int width1 = 10;
   cout << "Area of " << length1 << " x " << width1 << " is "</pre>
```

int width2 = 2;

The climits library (known in C as limits.h) is a good tool for writing edge case tests.

Post-conditions can detect overflows et al.

```
<< area(length1, width1) << endl;
int length2 = INT MAX; // INT MAX is the largest 2's complement
                     // positive integer supported by C++
cout << "Area of " << length2 << " x " << width2 << " is "
    << area(length2, width2) << endl;
```



Post-conditions

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

```
int area(int length, int width) { // calculate area of a rectangle
    // 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
    // no variables had their values changed
    int result=length*width;
    if (result < 0) throw Bad_area{};
    return result;
}</pre>
```

Actually checking for math overflow should be left to the compiler (e.g., -ftrapv). (Microprocessors detect overflow on every integer operation – the compiler flag adds checks.)

```
ricegf@pluto:~/dev/cpp/04$ g++ -ftrapv int_overflow.cpp
ricegf@pluto:~/dev/cpp/04$ ./a.out
Overflow'd!
Aborted (core dumped)
ricegf@pluto:~/dev/cpp/04$ []
```



- C++ (actually C) provides an assert macro
 - If the parameter is false, the program aborts

```
#include <cassert>
int area(int length, int width) { // calculate area of a rectangle
    // 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
    // no variables had their values changed
    int result=length*width;
    // if (result < 0) throw Bad_area{};
    assert(result > 0);
    return result;
}
```

```
ricegf@pluto:~/dev/cpp/201708/04$ make bad_area_assert
g++ --std=c++11 -o bad_area_assert bad_area_assert.cpp
ricegf@pluto:~/dev/cpp/201708/04$ ./bad_area_assert
15
16
bad_area_assert: bad_area_assert.cpp:7: int area(int, int): Assertion `length>0 && width>0' failed.
Aborted (core dumped)
ricegf@pluto:~/dev/cpp/201708/04$
```

When To Use Assert vs Exception

- In general, assert is used for an *interface* error
 - The caller violated the contract in some way
 - The program calls abort with an error message
 - The assert code can be removed when building for production by defining NDEBUG ("no debug")
 - #define NDEBUG // in your production C++ main OR
 - -DNDEBUG // the g++ command line flag version
- Exceptions are used for recoverable errors
 - Provides an alternate, direct path to error handlers
 - fno-exception (g++ option) disables exceptions in favor of abort (but the std lib's code wasn't built that way!)

Step 1 – Building Your Code with Fewer Bugs Get the Program to Compile

Is every character literal terminated?

Is every block terminated?

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

Is every set of parentheses matched?

- The compiler generally reports this kind of error "late"
 - It often recognizes the error at the end of the file

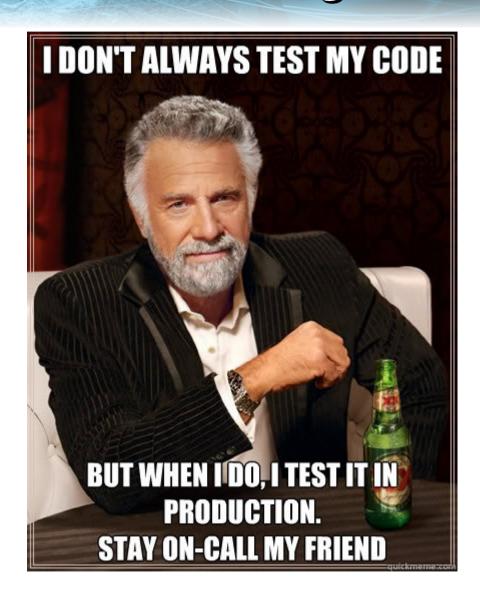
Step 1 – Building Your Code with Fewer Bugs Get the Program to Compile

- Is every name declared?
 - Did you include needed headers? (e.g., iostreams, vector, exception)
 - Are you using the right namespace? (e.g., std)
- Is every name declared before it's used?
 - Did you spell and capitalize all names correctly?

Did you terminate each expression statement with a semicolon?

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

Step 2 Testing



Step 2 – Testing Basic Testing Options

Interactive Testing

- Run the program like a nightmare user try to break it
- Some testing is likely to require a written "test procedure"
- Labor intensive, so avoid this as much as possible

Regression Testing

- Write a test program ("unit test" or "regression test")
- Systematically try expected inputs for correct operation
- Try forbidden inputs for proper exception handling
- When you find a bug, write a new test to ensure it doesn't pop up again

Test-Driven Development

- Write tests for requirements first, then prove the tests fail
- Write your new code, then prove the tests pass
- Repeat for each requirement
- Refactor to create a clean and maintainable solution



How **NOT** to Test

Step 2 – Testing Interactive Testing

- Try the "Normal Case(s)"
 - Run through your Use Cases (the common scenarios you expect from the user – more on Use Cases later)
 - Be a picky user: Spelling errors? Inconsistencies? Unneeded extra steps? Unclear prompts? Document a bug!
- Then be the "User from the Dark Side"
 - Try to break your program or enlist the help of a good "stress tester" (and treat them like the valuable asset they are!)
 - Enter invalid inputs, push buttons at the wrong time, enter weird
 Unicode sequences try to confuse your code!
- Don't Stop Until You Drop!
 - Write down each bug found, but don't fix until the list is complete

Step 2 – Testing Regression Testing

Code what you expect the program to do!

```
#include <iostream>
#include <exception>
using namespace std;
class Bad area: public exception { };
int area(int length, int width) { // calculate area of a rectangle
   if (length<=0 || width <=0) throw Bad area{ };</pre>
   return length*width;
int main() {
   // TEST #1: Normal Sides (or just use assert!)
   if (area(14, 10) != 140) cerr << "FAIL: <math>10x14 not 140 but " << area(14, 10) << endl;
   // TEST #2: Identical Length Sides
   if (area(10, 10) != 100) cerr << "FAIL: 10x10 not 100 but " << area(10,10) << endl;
   // TEST #3: Zero Length Side
   if (area(0, 10) != 0) cerr << "FAIL: 0x10 not 0 but " << area(0,10) << endl;
   // TEST #4: Negative Length Side
   try {
     int i = area(-1, -2);
     cerr << "FAIL: Negative side not exception but " << area(-1, -2) << endl;
    catch (Bad area e) { // discard the expected exception
```

Step 2 – Testing Testing Edge Cases

- Pay special attention to "edge 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
 - · Null input / end of file
 - Did you open your files correctly?
 - More on this in chapter 11
 - Did you actually read that input?
 - Write that output?





Step 2 – Testing Test-Driven Development (TDD)

- Test-Driven Development swaps steps 0-1 and 2
 - FIRST convert your requirements into test code
 - THEN verify that any existing code fails the tests
 - FINALLY update the code as little as possible to make the tests pass
- The resulting product is likely sub-optimal
 - We schedule "code refactoring" to "clean up" the product code and improve supportability
 - The functionality of the code doesn't change only its structure (for the better, we hope!)
 - Refactoring requires good regression tests

Step 3 – Debugging a Failing Program When Tests Fail, Debug!

What do I expect the program to do?

```
#include <iostream>
                                         ricegf@pluto:~/dev/cpp/201708/04$ g++ -std=c++11 test_area.cpp
#include <exception>
                                         ricegf@pluto:~/dev/cpp/201708/04$ ./a.out
using namespace std;
                                         terminate called after throwing an instance of 'Bad area'
                                          what(): std::exception
class Bad area: public exception { }; Aborted (core dumped)
                                         ricegf@pluto:~/dev/cpp/201708/04$
int area(int length, int width) {
   if (length<=0 || width <=0) throw Bad area{ };</pre>
   return length*width;
                                         Uh oh – Test #4 isn't catching the Bad area exception!
int main() {
   // TEST #1: Normal Sides
   if (area(14, 10) != 140) cerr << "FAIL: <math>10x14 not 140 but " << area(14, 10) << endl;
   // TEST #2: Identical Length Sides
   if (area(10, 10) != 100) cerr << "FAIL: <math>10x10 not 100 but " << area(10, 10) << endl;
   // TEST #3: Zero Length Side
   if (area(0, 10) != 0) cerr << "FAIL: 0x10 not 0 but " << area(0,10) << endl;
   // TEST #4: Negative Length Side
   try {
     int i = area(-1, -2);
     cerr << "FAIL: Negative side not exception but " << area(-1, -2) << endl;
     catch (Bad area e) {
```

Step 3 – Debugging a Failing Program Debugging Options

- Option #1: Mental Execution
 - Pretend that you are the computer running your program
 - Does its output match your expectations? If not, why not?
- Option #2: Add Output
 - Send intermediate data to cout



- Option #3: Invoke the Debugger ← This is usually your best choice!
 - Rebuild your program with additional "hooks"
 - Run it inside a program *specifically designed to help*
 - Examine (and even change) variables while it runs
 - Stop when something "interesting" happens

ricegf@pluto:~/dev/cpp/201708/04\$ g++ -std=c++11 -g test area.cpp ricegf@pluto:~/dev/cpp/201708/04\$ ddd a.out

First, compile and link using the -g option* to include the special debug capabilities

Then, run your program inside of "ddd", the "Data Display Debugger" that provides a GUI for command-line debuggers such as gdb.

```
int main() {
    // TEST #1: Normal Sides
    if (area(14, 10) != 140) cerr << "FAIL: 10x14 not 140 but " << area(14,10) << endl;
    // TEST #2: Identical Length Sides
    if (area(10, 10) != 100) cerr << "FAIL: 10x10 not 100 but " << area(10,10) << endl;
    // TEST #3: Zero Length Side
    if (area(0, 10) != 0) cerr << "FAIL: 0x10 not 0 but " << area(0,10) << endl;
    // TEST #4: Negative Length Side
GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea LReading symbols from a.out...done.
(gdb)
                                                                                                DDD
                                                                                               Run
                                                                                             Interrupt
                                                                                            Step
                                                                                                  Stepi
                                                                                             Next
                                                                                                  Nexti
                                                                                             Until
                                                                                                  Finish
                                                                                             Cont
                                                                                                  Kill
                                                                                                  Down
                                                                                            Undo:
                                                                                                  Redo
                                                                                             Edit
                                                                                                  Make
```

Debugging

Now, we want the program to stop just before throwing the exception.

So right-click on the line that throws Bad_area and select "Set Breakpoint". A small "stop sign" then marks the breakpoint.

A "breakpoint" stops the program as soon as it starts to execute the selected line, and allows you to interact with it (e.g., check variable values).

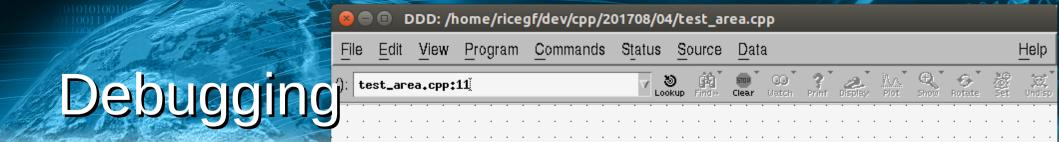
```
#include <exception>
#include <stdexcept>
using namespace std;
class Bad_area: public exception { };
int area(int length, int width) { // calculate area of a rectangle
      // length and width must be positive
  if (length <=0 | | width <=0)
      throw Rad area83+
Set Breakpoint
Set Temporary Breakpoint linux-gnu), by Dorothea LReading symbols from a.out...done.
Continue Until Here
                                                                                       DDD
                             Next. click "Run" in the toolbar.
                                                                                      Run
Set Execution Position
                             The program will run normally
                                                                                    Interrupt
                             until it reaches the breakpoint,
                                                                                   Step
                                                                                        Stepi
                             then stop and wait for your
                                                                                   Next:
                                                                                        Nexti
                             direction. The Program Counter
                                                                                   Until
                                                                                        Finish
                             is indicated by a green arrow.
                                                                                   Cont
                                                                                         Kill
                              class Bad_area: public exception { };
                                                                                        Down
                              int area(int length, int width) { // calcula
                                                                                   Undo:
                                                                                        Redo
                                     // length and width must be positive
                                 if (length<=0 || width <=0)
                                                                                   Edit
                                                                                        Make
                                     throw Bad_area{};
                                 return length*width;
 Set a breakpoint at the selected position
```

Help

DDD: /home/ricegf/dev/cpp/201708/04/test_area.cpp

Edit View Program Commands Status Source

test_area.cpp:11



Finally, we select Up on the tool palette and...

The problem is NOT Test #4, but rather Test #3.

```
// TEST #1: Normal Sides
   if (area(14, 10) != 140) cerr << "FAIL: 10x14 not 140 but " << area(14,10) << endl;
   // TEST #2: Identical Length Sides
    if (area(10, 10) != 100) cerr << "FAIL: 10x10 not 100 but " << area(10,10) << endl;
    // TEST #3: Zero Length Side
   if (area(0, 10) != 0) cerr << "FAIL: 0x10 not 0 but " << area(0,10) << endl;
   // TEST #4: Negative Length Side
    try {
GNU DDD 3.3.12 (x86_64-pc-linux-gnu), by Dorothea LReading symbols from a.out...done.
(gdb) break test_area.cpp:11
                                                                                             DDD
Breakpoint 1 at 0x400d68; file test_area.cpp, line 11.
(gdb) run
```

Do you see the problem?

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

(gdb)

Up. Down Select and print stack frame that called this one Undo Redo

Run

Interrupt

Stepi

Nexti

Finish

Kill

Down

Redo

Make

Step

Next

Until

Cont

Up

Undo

Edit

Edit

Make

When in doubt, don't cout – use the debugger!

#1 0x0000000000400e5e in main () at test_area.cpp:23

Starting program: /home/ricegf/dev/cpp/201708/04/a.out

#1 0x000000000000400e5e in main () at test_area.cpp:23

Breakpoint 1, area (length=0, width=10) at test_area.cpp:11

Other Debuggers

- Gcc provides excellent (and representative) debugger support
 - Compile with the -g option to enable debugging
 - Use 'ulimit -c unlimited' in bash to enable core dumps (if desired)
 - gcc includes the gdb debugger Command Line Interface (CLI)
 - http://www.gnu.org/software/gdb/documentation/
 - https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html
 - ddd is an alternate front end to gdb Graphical User Interface (GUI)
 - You may need to install ddd separately, e.g., in Linux:

```
- sudo apt-get install ddd
```

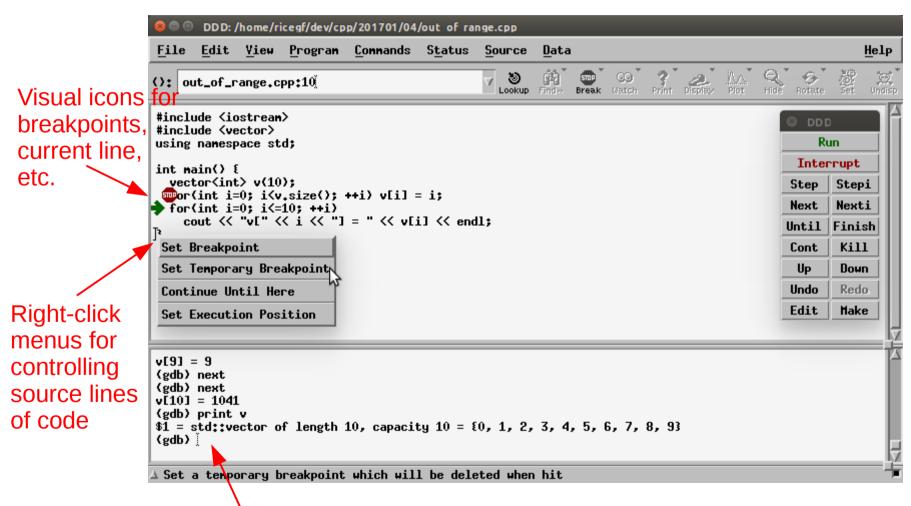
- Other environments may have different debuggers
 NOTE: We offer NO support for these environments!
 - Code::Blocks see http://wiki.codeblocks.org/index.php/Debugging_with_Code::Blocks
 - Visual Studio see https://msdn.microsoft.com/en-us/library/k0k771bt.aspx
 - Geany see https://plugins.geany.org/debugger.html
 - Clang see http://lldb.llvm.org/

More on 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 very old and very sad programmer's joke

99 bugs in the code that I wrote, 99 bugs in my code. Fix one fast and patch it around, 127 bugs in the code that I wrote.

Debugging Out_Of_Range with ddd



Quick access buttons for stepping through the program with various granularities

Command line interface (gdb) always available, showing equivalent of GUI actions

Typical Debugger Capabilities

- Breakpoint (stop) at a specified source line or event
 - Events such as exceptions, I/O, or threads (covered later)
 - Optionally only when a given expression is true
- Single step through the program
 - Either dive into or skip over function calls
- View the value of variables and memory
 - Automatically view variables at each breakpoint
 - View all of a function's local variables in a table
 - View stack frames up through the call stack
 - CHANGE a variable's value interactively
- Debug at the assembly level
 - View all registers and the memory to which they point
 - Change register values (or as pointers) interactively

Quick Review

- Common types of errors include which of the following:
 - (a) Edit-time errors
- (b) Compile-time errors
- (c) Link-time errors (d) Run-time errors
- (e) Logic errors
- (f) Math errors
- To handle bad assumptions, a method should usually
 - (a) Document the assumptions and expect callers to comply
 - (b) Force compliant types on parameters so the compiler can detect
 - (c) Detect the invalid input and throw an exception
- What are the pros and cons of each strategy for handling errors?
 - (a) Return an error code
 - (b) Set a global error variable
 - (c) Throw an exception

Quick Review

- Why should error messages be printed to cerr instead of cout?
- When should assert be used instead of throwing an exception?
- Match the name to the associated type of testing
 - Interactive testing (a) Write numerous automated unit tests
 - Regression testing
 (b) Write the test before writing the code
 - Test-driven development (c) Run program like "user from the dark side"
- What are some common useful debugger capabilities?

For Next Class

- (Optional) Read Chapter 5 in Stroustrop
 - More advanced students should read Chapter 26 –
 Not on the exam, but important information for software developers
 - Do the Drills!
- Skim Chapter 8 for next lecture
 - This is the "Eclectics" lecture a lot of review and an eclectic mix of knowledge to (hopefully) "fill in the gaps" in your OO and C++ competence
- Homework #2 is due by 8 am Thursday