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

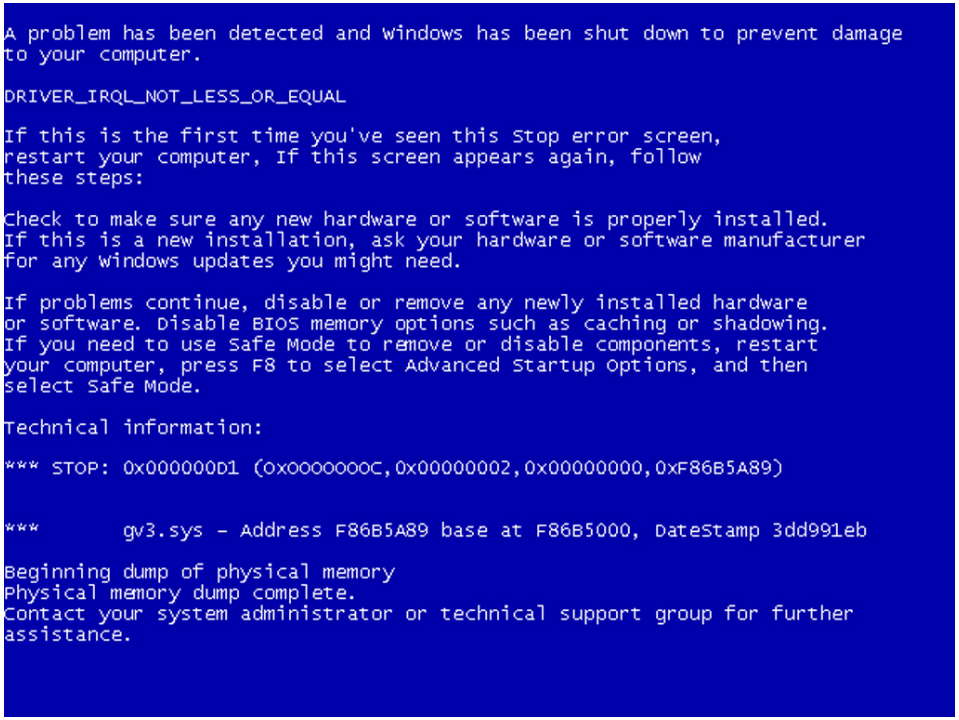
Sources of errors

- **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!"

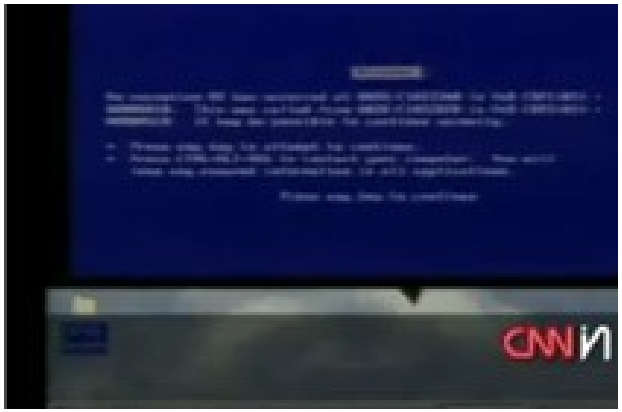
Your JOB is to reasonably respond to all possible events without introducing undue burden on the primary use case(s)

Kinds of Errors

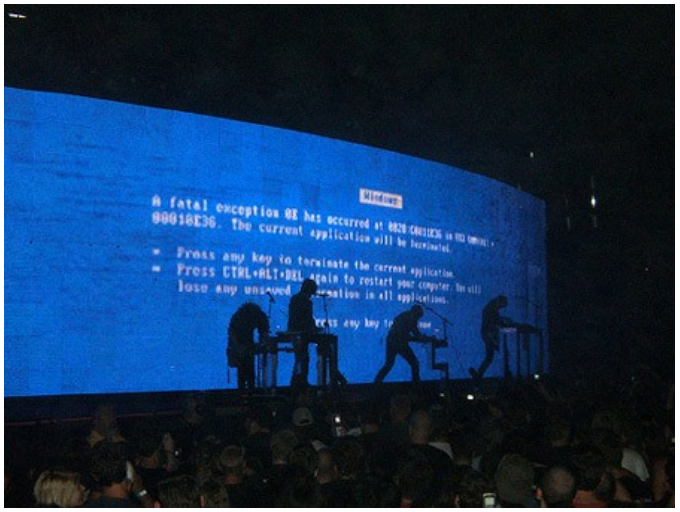
- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
  - Missing libraries
  - Missing .o files
- Run-time errors
  - Unreported (crash)
  - Reported (exceptions)
  - Handled (exception handlers)
- Logic errors
  - Detected by automated tests (regression test, specification tests)
  - 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

Even highly regarded business software fails sometimes. Don’t let this happen to you, though...

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**)

“Data Validation”

- EXAMPLE: SQL Injection Attack (common against poorly coded websites)

```
// userName has been authenticated by matching password
cin >> itemName;
string query = "SELECT * FROM items WHERE owner = '" + userName + "'
               AND itemname = '" + itemName + "'";
sda = new SqlDataAdapter(query, conn);
```

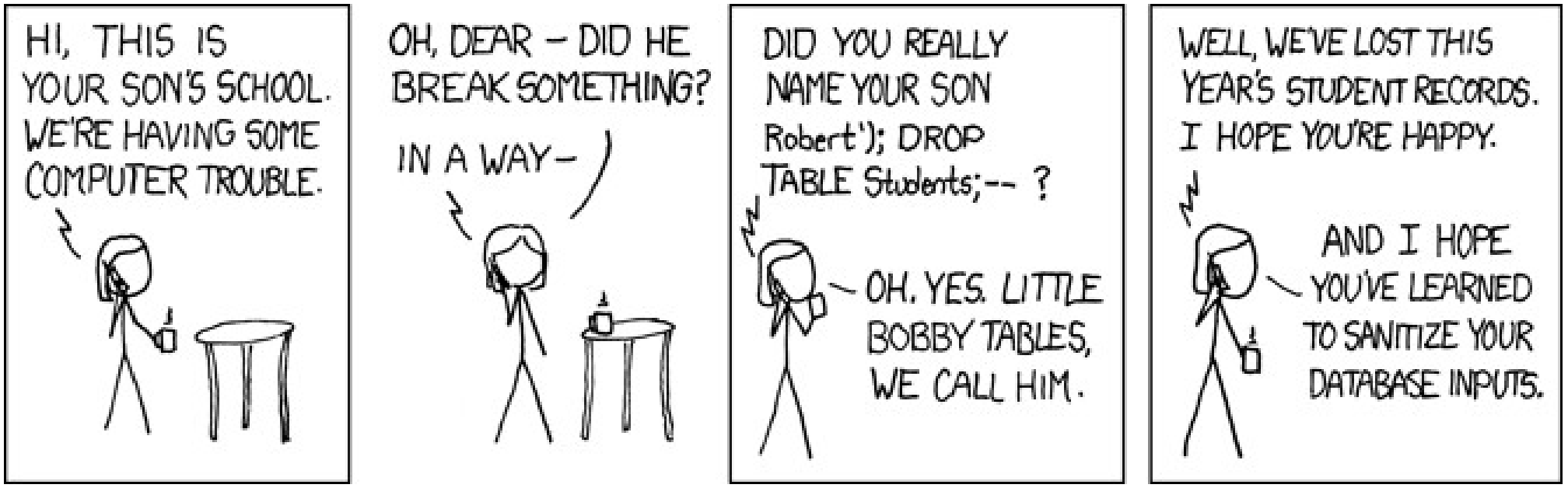
- sda where the user enters “book”:

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

- sda where the user enters “book' OR 'a'='a”:

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

Messing with Modern Technology



Avoiding Errors
Regular Expressions

- A very powerful, compact technology - “regex” - is available via the Standard Class Library
  - Lets you define the “look” of valid data
  - The match method will let you know if the data “looks” valid

```
ricegf@pluto:~/dev/cpp/201801/04$ ./a.out
Enter some integers:
42
That’s an int!
hello
***INVALID INPUT***
-42
That’s an int!
3.14
***INVALID INPUT***
379h
***INVALID INPUT***
0x42
***INVALID INPUT***
-192342153243
That’s an int!
^C
```

```
#include <iostream>
#include <regex>
using namespace std;

int main() {
    string input;
    regex integer{"-?\\d+"};
    cout << "Enter some integers:" << endl;

    while(cin>>input) {
        if(regex_match(input,integer)) cout << "That’s an int!" << endl;
        else cout << "***INVALID INPUT***" << endl;
    }
}
```

(Very) Basic Regex Rules

- Most characters match themselves
- A special character matches one of a group
  - \s matches any whitespace except newline
  - \w matches a “word” character – A-Z, a-z, 0-9, or _
  - \d matches a “digit” – 0-9
  - \S, \W, and \D match the opposite of their lowercase version
- Repeaters compactly express multiple characters
  - \d? represents either 0 or 1 digits
  - \d* represents 0 or more digits
  - \d+ represents 1 or more digits
- Parentheses work as expected
  - (ha)+ represents ha, haha, hahahahaha, etc.

```
regex integer{"-?\\d+"};
```

Matches itself
Matches “-” 0 or 1 times
Matches a digit
Matches the digit 1 or more times

What Regex Would Validate These?

- “Professor Rice” or “Professor George Rice”
  Professor(George)? Rice
- A C++ name, e.g., “_counter3” \w+
- A negative integer, e.g., “-108” -\d+
- Adding two positive integers, e.g., “42+38”
  \d+\+\d+
- Exactly 3 words, e.g., “Go Mavs Go”
  \w+\s\w+\s\w+
- One or more words, e.g., “Hello” or “Bye Bye Baby Bye Bye”
  \w+(\s\w+)*

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

Catching Bad Function Arguments

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

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

int x3 = area(7, 10);       // correct
int x1 = area(7);           // error: wrong number of arguments
int x2 = area("seven", 2); // error: 1st argument has a wrong type
int x5 = area(7.5, 10);     // ok, but dangerous: 7.5 truncated to 7;
                            //       compiler may warn you
int x = area(10, -7);       // undetected error: bad assumptions
                            //   the types are correct,
                            //   but the values make no sense
```

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 ← **Usually the "right answer" in OO**
- 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;
}
```

- So, "let the caller beware"

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

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

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;
}
```

- So, “let the caller check”

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

- 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 <exception>
using namespace std;

class Bad_area: public 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;
    cout << area(8, 2) << endl;
    cout << area(3,-5) << endl;
    return 0; // success
  } catch(Bad_area e) {
    cerr << "Bad area call - fix program!" << endl;
    return 1; // failure
  }
}
```

Include the exception library

Define a new exception
(don't sweat the syntax yet)

Create (instance) the exception object

Begin watching for an exception

Catch the exception with a matching type
(e.g., Bad_area) and then execute the code
in the new scope (i.e., inside the { })

```
ricegf@pluto:~/dev/cpp/04$ ./a.out
15
16
Bad area call - fix program!
```

Use cerr to send text to STDERR, similar
to using cout to send text to STDOUT.

Return a non-zero result from main
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;
     cout << area(8, 2) << endl;
     cout << area(3,-5) << endl;
   } catch(Bad_area e) {
     cout << "Bad area call - fix program!" << endl;
     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”

Why return != 0?

```
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{};   // note the {} - a value
   return length*width;
 }
 int main() {
   try {
     cout << area(3, 5) << endl;
     cout << area(8, 2) << endl;
 //  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
16
Bad area call - fix program!
ricegf@pluto:~/dev/cpp/04$ if [ $? -eq 0 ]
> then
> echo Success!
> else
> echo Failure!
> fi
Failure!
ricegf@pluto:~/dev/cpp/04$
```

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

Bash responds to success of the program.

Bash responds to failure of the program.

Make halts the build on a non-zero return value

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 (the "try scope") 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 are Objects of Class exception

- Exception has one method and one operator

Member functions

(constructor)	Construct exception (public member function)
operator=	Copy exception (public member function)
what (virtual)	Get string identifying exception (public member function)
(destructor) (virtual)	Destroy exception (public virtual member function)

http://www.cplusplus.com/reference/exception/exception/what/

Call e.what() to obtain the text passed as a parameter to the thrown exception!

- Many exception classes “derive” from exception

Derived types (scattered throughout different library headers)

bad_alloc	Exception thrown on failure allocating memory (class)
bad_cast	Exception thrown on failure to dynamic cast (class)
bad_exception	Exception thrown by unexpected handler (class)
bad_function_call C++11	Exception thrown on bad call (class)
bad_typeid	Exception thrown on typeid of null pointer (class)
bad_weak_ptr C++11	Bad weak pointer (class)
ios_base::failure	Base class for stream exceptions (public member class)
logic_error	Logic error exception (class)
runtime_error	Runtime error exception (class)

It is very common to throw runtime_error rather than creating a custom exception.

Custom Exceptions vs runtime_exception

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

class Bad_dates: public exception {
  public:
    const char* what() const noexcept {
      return "Bad dates";
    }
};

int main() {
  try {
    throw Bad_dates{};
  } catch (exception& e) {
    cerr << e.what() << endl;
  }

  try {
    throw runtime_error{"Bad dates"};
  } catch (exception& e) {
    cerr << e.what() << endl;
  }
}
```

Bad_dates “isa” exception

Sorry – required by C++ standard!

Throw a custom exception

Throw a predefined runtime_error

```
ricegf@pluto:~/dev/cpp/201801/04$ g++ --std=c++14 test_exception.cpp
ricegf@pluto:~/dev/cpp/201801/04$ ./a.out
Bad dates
Bad dates
ricegf@pluto:~/dev/cpp/201801/04$
```

Exceptions to Exceptions

- Try this

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  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 (ahem)
    cout << "v[" << i << "] == " << v[i] << endl;
}
```

```
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
v[10] == 132033
ricegf@pluto:~/dev/cpp/201701/04$
```

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!

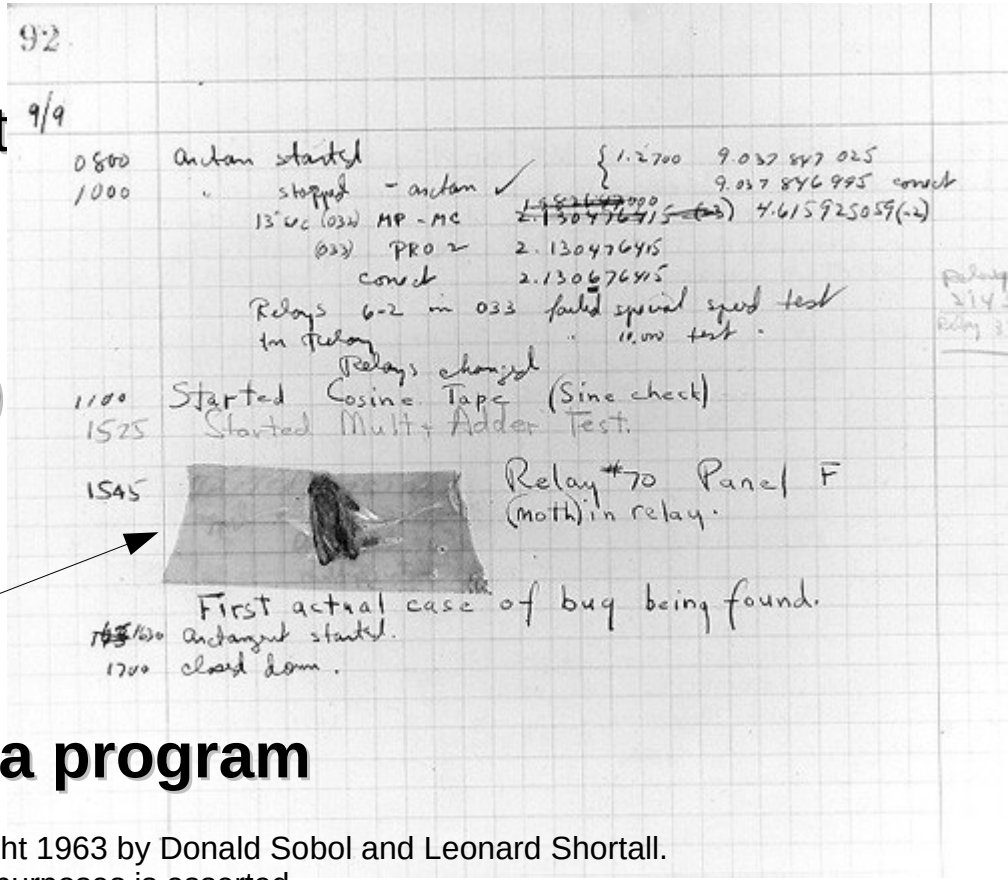
After the Exception is Caught

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 many 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

Bugs

- Every program that you write **will** 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

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

Defensive Coding

- 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

* The latter is called “comment inversion”

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;
```

- 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

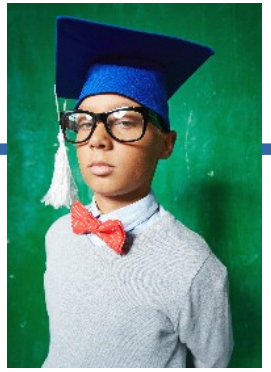
More on Assertions
Pre-conditions

- 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;
}
```

Challenge

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



More on Assertions

Edge Cases and Post-conditions

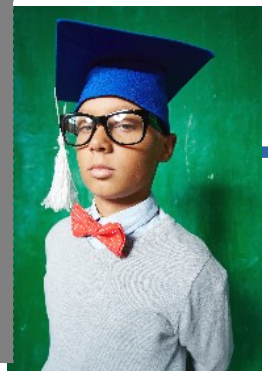
```
#include <iostream>
#include <exception>
#include <climits>
using namespace std;

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;
}

int main() {
    int length1 = 14;
    int width1 = 10;

    cout << "Area of " << length1 << " x " << width1 << " is "
         << area(length1, width1) << endl;

    int length2 = INT_MAX; // INT_MAX is the largest 2's complement
    int width2 = 2;        // positive integer supported by C++

    cout << "Area of " << length2 << " x " << width2 << " is "
         << area(length2, width2) << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201701/04$ g++ -std=c++11 max_int.cpp
ricegf@pluto:~/dev/cpp/201701/04$ ./a.out
Area of 14 x 10 is 140
Area of 2147483647 x 2 is -2
ricegf@pluto:~/dev/cpp/201701/04$
```

???

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.



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;
}
```

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.)

https://gist.github.com/mastbaum/1004768

```
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$
```

cassert (assert.h)

- 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$
```

http://www.cplusplus.com/reference/cassert/assert/

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

- Any errors in this program?

```
int main() {
{
  String name;
  cin >> name;
  std:cout << “Hello, << name << ‘!’
  if (Name = 'George' cout << " (prof)" << end1;
};
```

Step 1 – Building Y
Get the Pro

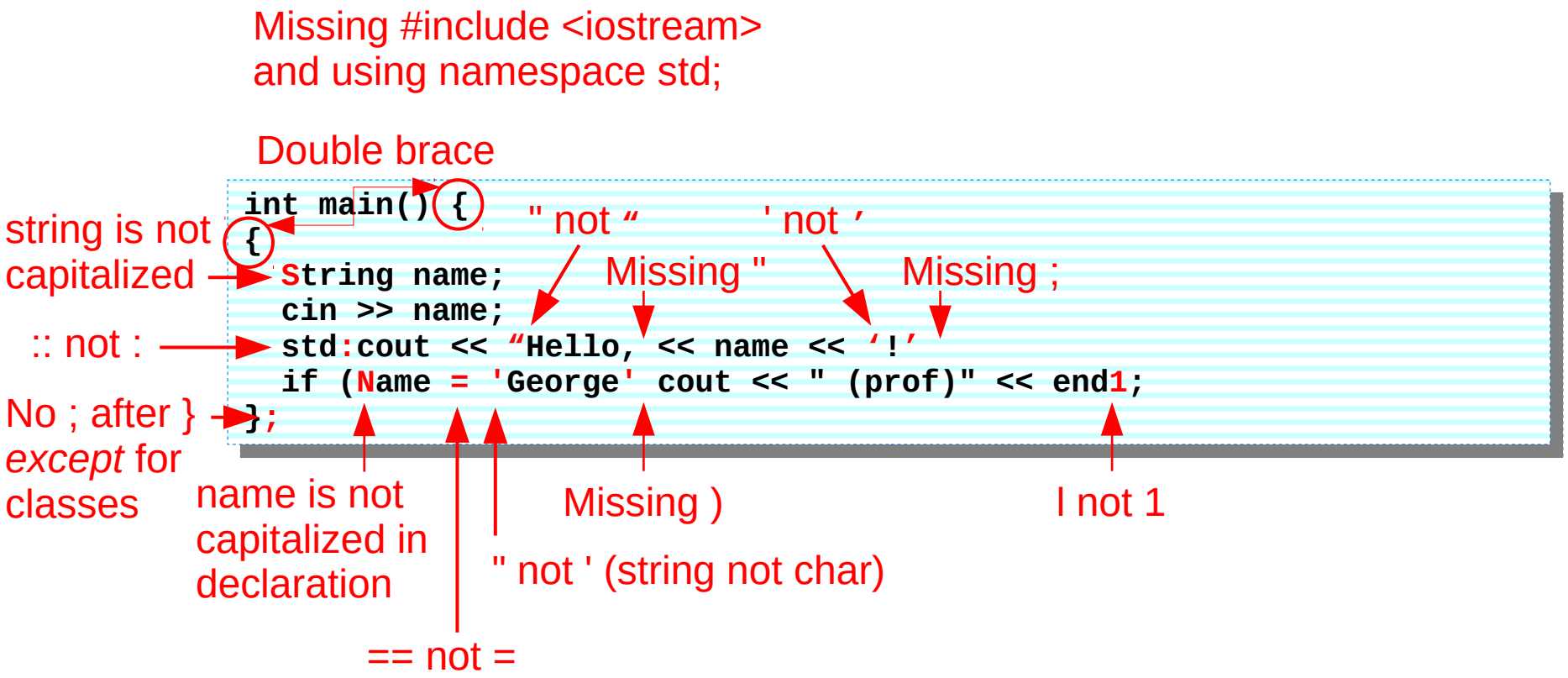
- Any errors in this pro
  - A few...

```
int main() {
{
  String name;
  cin >> name;
  std:cout << “Hello, << name
  if (Name = 'George' cout <<
};
```

```
ricegf@pluto:~/dev/cpp/201801/04$ g++ bad_code.cpp
bad_code.cpp:5:3: error: stray ‘\342’ in program
   std:cout << “Hello, << name << ‘!’
   ^
bad_code.cpp:5:3: error: stray ‘\200’ in program
bad_code.cpp:5:3: error: stray ‘\234’ in program
bad_code.cpp:5:3: error: stray ‘\342’ in program
bad_code.cpp:5:3: error: stray ‘\200’ in program
bad_code.cpp:5:3: error: stray ‘\230’ in program
bad_code.cpp:5:3: error: stray ‘\342’ in program
bad_code.cpp:5:3: error: stray ‘\200’ in program
bad_code.cpp:5:3: error: stray ‘\231’ in program
bad_code.cpp:6:3: error: stray ‘\342’ in program
   if (Name = ‘George’ cout << “ (prof)” << end1;
   ^
bad_code.cpp:6:3: error: stray ‘\200’ in program
bad_code.cpp:6:3: error: stray ‘\230’ in program
bad_code.cpp:6:3: error: stray ‘\342’ in program
bad_code.cpp:6:3: error: stray ‘\200’ in program
bad_code.cpp:6:3: error: stray ‘\231’ in program
bad_code.cpp:6:3: error: stray ‘\342’ in program
bad_code.cpp:6:3: error: stray ‘\200’ in program
bad_code.cpp:6:3: error: stray ‘\234’ in program
bad_code.cpp:6:3: error: stray ‘\342’ in program
bad_code.cpp:6:3: error: stray ‘\200’ in program
bad_code.cpp:6:3: error: stray ‘\235’ in program
bad_code.cpp: In function ‘int main()’:
bad_code.cpp:3:3: error: ‘String’ was not declared in this scope
   String name;
   ^
bad_code.cpp:4:3: error: ‘cin’ was not declared in this scope
   cin >> name;
   ^
bad_code.cpp:4:10: error: ‘name’ was not declared in this scope
   cin >> name;
          ^
bad_code.cpp:5:7: error: ‘cout’ was not declared in this scope
   std:cout << “Hello, << name << ‘!’
       ^
bad_code.cpp:5:18: error: ‘Hello’ was not declared in this scope
   std:cout << “Hello, << name << ‘!’
                  ^
bad_code.cpp:5:25: error: expected primary-expression before ‘<<’ token
   std:cout << “Hello, << name << ‘!’
                         ^
bad_code.cpp:6:3: error: expected primary-expression before ‘if’
   if (Name = ‘George’ cout << “ (prof)” << end1;
   ^
bad_code.cpp:7:2: error: expected ‘}’ at end of input
 };
  ^
ricegf@pluto:~/dev/cpp/201801/04$
```

Step 1 – Building Your Code with Fewer Bugs

Get the Program to Compile



A good syntax-checking editor is a great tool! Experience helps, too...

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 every possible 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{ };
   return length*width;
}
int main() {
   // TEST #1: Normal Sides (or just use assert!)
   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 {
     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?

https://www.youtube.com/watch?v=kYUrqdUyEp
http://www.cs.jhu.edu/~jorgev/cs106/bug.pdf

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>
#include <exception>
using namespace std;

class Bad_area: public exception { };
int area(int length, int width) {  //
   if (length<=0 || width <=0) throw Bad_area{ };
   return length*width;
}
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
   try {
     int i = area(-1, -2);
     cerr << "FAIL: Negative side not exception but " << area(-1, -2) << endl;
   } catch (Bad_area e) {
   }
}
```

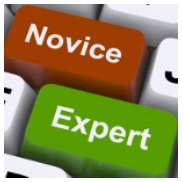
```
ricegf@pluto:~/dev/cpp/201708/04$ g++ -std=c++11 test_area.cpp
ricegf@pluto:~/dev/cpp/201708/04$ ./a.out
terminate called after throwing an instance of 'Bad_area'
  what():  std::exception
Aborted (core dumped)
ricegf@pluto:~/dev/cpp/201708/04$
```

Uh oh – Test #4 isn't catching the Bad_area exception!

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



Newbie Alert!

- 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