CSE 1325: Object-Oriented Programming

Lecture 23 / Chapters 23-24

Text Processing, Numerics, and Anti-Patterns

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

Quick Review

- **Core** is one ALU/register set on a CPU, which runs one thread at a time. **Hyperthreading** is when one ALU has 2 register sets, interleaving 2 threads.
- Match the definition to either process, thread, or concurrency.
  - Performing 2 or more algorithms (as it were) simultaneously **concurrency**
  - A self-contained execution environment including its own memory space **process**
  - An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space **thread**
- List 3 problems with busy loops. What's the better alternative? **Unable to accurately predict the delay, wastes CPU cycles, may be optimized away by a smart compiler. Use sleep instead, e.g.,**
  `this_thread::sleep_for(chrono::milliseconds(100));`
- What does joining a thread accomplish? **The main thread waits for the selected thread to exit, and then continues. This allows the threads to sync before preceding, e.g., ensuring that the thread has completed work before main uses its result.**
- Describe the “garbled output” (sync) problem. How do invariant classes help? How does a mutex solve the problem for the general case? **Two threads executing the same method to update a data structure may corrupt the structure. Invariant classes modify nothing, and so their methods are usually reentrant. A mutex allows access to critical code sections only one thread at a time, avoid this problem.**

Quick Review

- A **Deployment** **Diagram** models the physical deployment of artifacts on nodes.
  - An **artifact** is a result of the software development process
  - A **node** is a physical or logical computational resource
  - A physical node (device) is typically a **CPU**. Give an example. **A server, switch, encrypter, laptop, modem, firewall, NAS, etc.**
  - A logical node (execution environment) is **software** that supports the asset. Give an example. **A virtual machine, operating system, database system such as Oracle, web application server such as Tomcat, etc.**
- Which of the follow are deployment diagram types?
  (A) **Network Architecture** (B) Military (C) .cpp and .h
  (D) **Manifestation of Artifacts by Components** (E) EXE
  (F) **Specification-Level Deployment** (G) **Instance-Level Deployment**
- The **Decorator** **Pattern** dynamically adds new functionality to an object without altering its structure

Strings Matter

- All data can be represented by text
  - Books, articles, web pages
  - Tables of structured information (e.g., XML, JSON)
  - Email, SMS, social media
  - Graphics (e.g., vector formats)
  - Software code(!)
  - Binary data (e.g., uuencode, uudecode)
  - All languages and way too many emojis
- Text is very portable (except that annoying \n, \r, \r\n thingie)
- Text is easily created and edited by your choice of text editor

Options for Representing Strings

- Old C-style strings (zero-terminated char array)
  - #include <cstring> (or <string.h>)
  - strlen(s)
  - strcmp(s1, s2)
- Std::string (a class!)
  - #include <string>
  - s.size()
  - s1 == s2
- Std::basic_string (a template!!!)
  - A template for making string-like classes
  - Std::string is simply **`typedef std::basic_string<char> string;`**
  - Std::basic_string<wchar_t> is the same, for 16-bit chars, etc.
- Proprietary types, e.g., gtkmm’s Glib::ustring (good for Unicode!)

Translating Objects to Strings

- string pi = to_string(3.14159265)
- to_string can be instanced from a simple template
  for any class

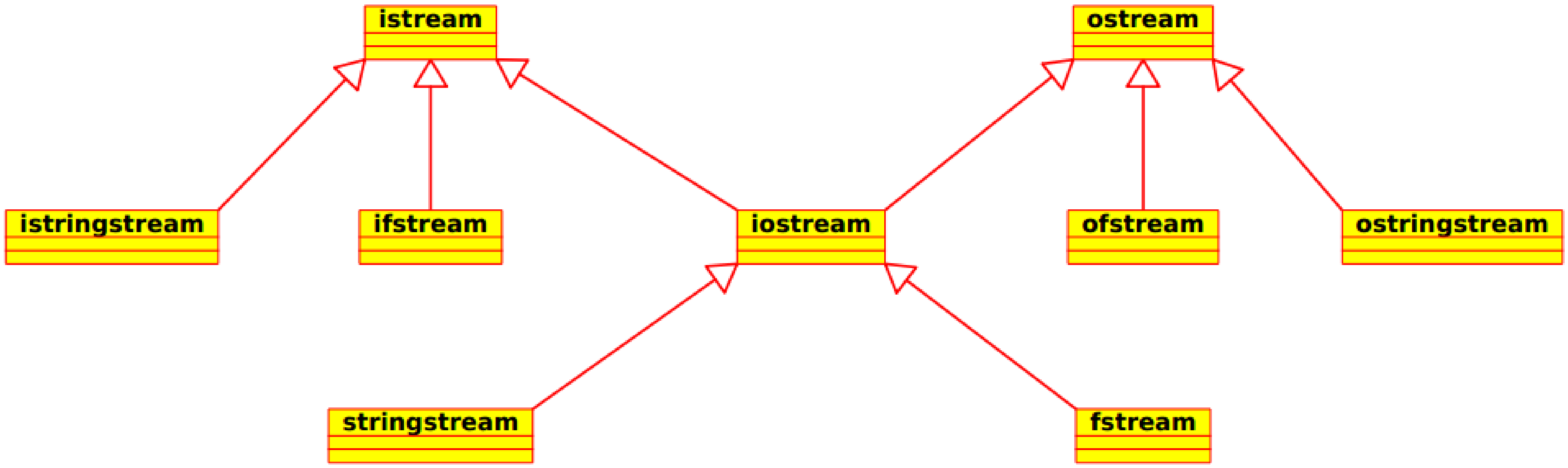
```
template<class T>
string to_string(const T& t) {
    ostringstream os;
    os << t;
    return os.str();
}
class Coordinate {
  public:
    Coordinate(int x, int y) : _x{x}, _y{y} { }
    int x() const {return _x;}
    int y() const {return _y;}
  private:
    int _x, _y;
};
ostream& operator<<(ostream& os, const Coordinate& c) {
    os << "(" << c.x() << "," << c.y() << ")";
    return os;
}
int main() {
    Coordinate c{3,4};
    string s = to_string<Coordinate>(c);
    cout << s << endl;
}
```

Translating Strings to Objects

- stoX (where X is a primitive)
  - int i = stoi(“42”);
  - double e = stod(“2.71828”);
- A template again can be instanced to take care of classes

```
template<class T>
T stox(const string& s) {
    istringstream is(s);
    T t;
    if (!(is >> t)) throw bad_from_string();
    return t;
}
// …
    Coordinate c = stox<Coordinate>(“(3,4)”);
```

String I/O (Partial) Class Hierarchy



Maps

- A std::map is an associative container that stores elements in a key-value pair
  - It is essentially a vector using any type (including *your* class) as the index type
  - planetary_mass[“earth”] = 5.97; // *conceptually*; in yottagrams, of course
- Keys must be unique
  - Only one value is associated with each key*
- Keys are in sorted order
  - Searching for an element in the map through key is thus fast(er)
  - Effectively logarithmic time
- A C++ map is roughly equivalent to a Java NavigableMap or a Python dict

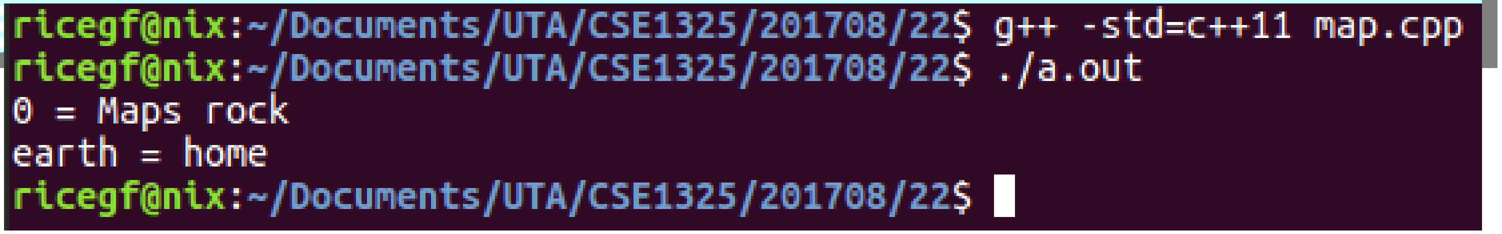
* Use a multimap instead to store multiple values per key

Map Example

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <iterator>
using namespace std;

int main() {
    // With vectors (using int as the index type)
    vector<string> s;
    s.push_back("Maps rock");
    for (int i=0; i < s.size(); ++i)
       cout << i << " = " << s[i] << endl;

    // With maps (using string as the index type)
    map<string, string> m;
    m.insert(make_pair("earth", "home"));
    for (map<string, string>::iterator it = m.begin(); it != m.end(); ++it) {
        cout << it->first << " = " << it->second << endl;
    }
}
```



```
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ g++ -std=c++11 map.cpp
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ ./a.out
0 = Maps rock
earth = home
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$
```

Pairs

- A std::pair is a template that couples two values of different types
  - The first value is accessed via ->first
  - The second value is accessed via ->second
  - Constructors are available
    - pair<string,string> planet("earth", “home”);
  - More common is the factory make_pair
    - auto planet = make_pair("earth", "home");
- **A map thus manages pairs** – key-value pairs
- A pair is a type of *tuple*, a finite ordered list of elements – specifically, a 2-tuple

Common Map Operations

- empty() returns true if the map contains no pairs
- size() returns the number of pairs in the map
- operator[ ] (rvalue) provides random access by key, adding a default value to the map if not already defined for that key
  - at(key) is like operator[ ], but instead throws an out_of_range exception if key isn’t defined
- operator[ ] (lvalue) silently overwrites the existing value for a key
  - insert(pair p) adds a pair to the map only if the key doesn’t already exist
- begin() and end() return the usual iterators
- find(key) returns an iterator to the associated value, or map::end if the key is not in the map
- erase(key) or erase(iterator1[, iterator2]) deletes the values pointed to by the iterators from the map

Other Map Types

- Multimap allows multiple values per key
- Set uses the key as the value, essentially storing a list of values alone
- Multiset counts the number of times that a key is added, with the count as the value
- Unordered_X (where X is map, set, multimap, or multiset) don’t automatically sort the keys, and are thus somewhat faster for non-search operations when searching is rare or isn’t needed

Regular Expressions

- A regular expression (regex) is a string that defines a search pattern for other strings
  - The regex syntax is terse, cryptic, and borderline write-only, yet exceeding useful – learn it!
- A non-special character matches itself
  - A special character will match 0 or more characters
  - A special character preceded by a \ matches itself
    - And remember that you need another \ to escape the \ in the regular expression – so an actual regex is more likely defined *in C++ code* as “\\s*” rather than “\s*”

Special Characters

- . matches any single character except a newline or carriage return
- [ ] matches any single character inside the brackets
  - [^] matches any character NOT inside the brackets
  - [a-z] matches any characters in the range a to z
  - [cat|dog] matches cat OR dog
- {m,n} matches the previous element at least m but no more than n times
  - * matches the previous element 0 or more times (same as {0,})
  - ? matches the previous element 0 or 1 times (same as {0,1})
  - + matches the previous element 1 or more times (same as {1,})
- ^ matches the start and $ the end of a line
- ( ) forms a sub-match for later reference
  - \N matches the Nth sub-match's actual text (start at 1, NOT 0!)

Character Matching Examples

- a.c matches abc, acc, a/c
- r[au]n matches ran or run
- [^2-9][0-9] matches 17 or 03 or K9, but not 21 or 99
- 3\.3* matches 3., 3.3, 3.33, or 3.3333333333
- ([0-9]+) = \1 matches 3 = 3 and 42 = 42, but not 21 = 39
- ^[hH]ello matches hello or Hello, but only at the start of a line
- Sincerely(yours)?,$ matches Sincerely, or Sincerely yours, but only at the end of a line
- Subject: (F[Ww]:|R[Ee]:)?(.*) matches an email subject line for forwards and replies only
- [a-zA-Z] [a-zA-Z_0-9]* matches a C++ identifier

Character Classes
(C++ supports both syntax families)

- Unix / Linux traditional syntax
  - \s matches a whitespace char
    - \S anything else
  - \w matches a word char ([A-Za-z0-9_])
    - \W anything else
  - \d matches a digit ([0-9])
    - \D anything else
  - \x represents a hexadecimal digit
    - \X anything else
  - \n, \r, \t represent newline, carriage return, and tab as usual

- ECMA class syntax

[:alnum:]	alpha-numerical character
[:alpha:]	alphabetic character
[:blank:]	blank character
[:cntrl:]	control character
[:digit:]	decimal digit character
[:graph:]	character with graphical rep
[:lower:]	lowercase letter
[:print:]	printable character
[:punct:]	punctuation mark character
[:space:]	whitespace character
[:upper:]	uppercase letter
[:xdigit:]	hexadecimal digit character
[:d:]	decimal digit character
[:w:]	word character
[:s:]	whitespace character

http://www.cplusplus.com/reference/regex/ECMAScript/

More Character Matching Examples

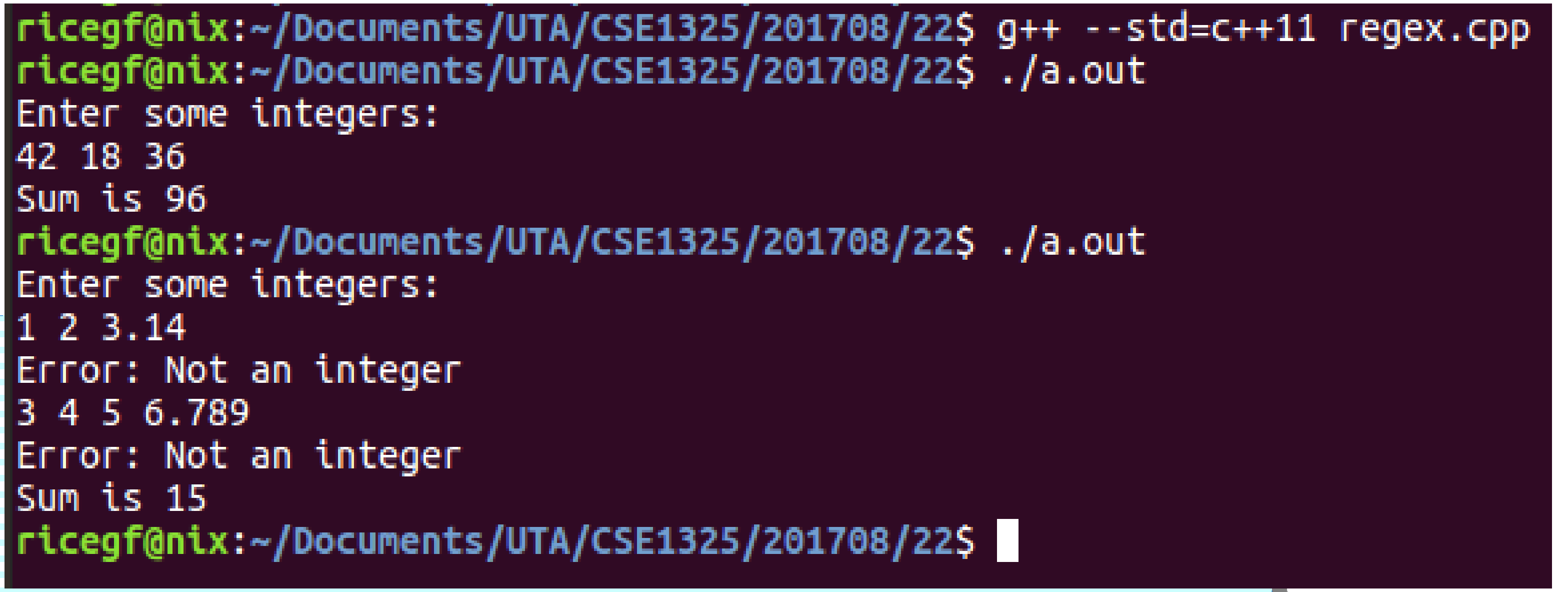
- Xa{2,3}   matches Xaa   Xaaa
- Xb{2}   matches Xbb
- Xc{2,}   matches Xcc   Xccc   Xcccc   Xccccc
- int\s+x\s*=\s*\d+; matches an integer definition
- CSE\d{4} matches a CSE class
- \w{2}-\d{4,5}   matches 2 letters and 4 or 5 digits
- \d{5}(-\d{4})? matches a 5 or 9 digit zip code
- [^aeiouy]   matches not an English vowel

Some C++ Regex-Related Classes and Functions

- The regex class is constructed with the regular expression string as a parameter
  - If the regular expression has syntax errors, the constructor throws a regex_error exception
  - Once instanced, the regex may be used in multiple matches and searches
- regex_match(string_to_search, regex) returns true if the entire string_to_search matches the regex
- regex_search(string_to_search, regex) returns true if any substring of string_to_search matches regex

http://www.cplusplus.com/reference/regex/

A Simple C++ Regex Example



```
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ g++ --std=c++11 regex.cpp
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ ./a.out
Enter some integers:
42 18 36
Sum is 96
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ ./a.out
Enter some integers:
1 2 3.14
Error: Not an integer
3 4 5 6.789
Error: Not an integer
Sum is 15
ricegf@nix:~/Documents/UTA/CSE1325/201708/22$ 
```

```
#include <iostream>
#include <regex>
#include <string>

using namespace std;

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

    int sum = 0;
    while(cin>>input) {
        if(regex_match(input,integer))
            sum += stoi(input);
        else
            cerr << "Error: Not an integer" << endl;
    }
    cout << "Sum is " << sum << endl;
}
```

Regex are often VERY useful for data validation!

Regex and Beyond

- Regex are extremely powerful text manipulators
  - We’ve only scratched the thin outer surface
- Regex are widely used outside C++
  - The gedit text editor can search (and replace) using regex
  - Command line tools like grep (literally “global regular expression print”), find, and awk use regex
  - Perl famously extends regex power to ludicrous (and incredibly useful!) extremes
- If you deal with much text, you need to learn regex!

Random Numbers

- A random number is a number from a defined but non-obvious sequence that matches a specified distribution
- We've used cstdlib's rand thus far, which is suitable for simplistic games but nothing more sophisticated
- The <random> library is more professional, producing random numbers using a combination of a generator and a distribution

http://www.cplusplus.com/reference/random/

Generators and Distributions

- Generator: An object that generates uniformly distributed numbers
  - <random> includes 3 generators: Linear congruential, Mersenne twister, and subtract-with-carry
  - Also included are 3 adapters that modify the sequence of the above generators
  - Several pre-instantiated generators are also available
- Distribution: An object that transforms a sequence of numbers from a generator into a number sequence that follows a specific random variable distribution, e.g., Uniform, Normal or Binomial.

Using random

```
#include <random>
#include <chrono>
#include <iostream>

using namespace std;

int main() {
    // Seed the generator
    typedef std::chrono::high_resolution_clock myclock;
    myclock::time_point beginning = myclock::now();
    myclock::duration d = myclock::now() - beginning;
    unsigned seed = d.count();

    std::default_random_engine generator{seed};
    std::uniform_int_distribution<int> distribution(1,6);

    for(int i=0; i<100; ++i) {
        int dice_roll = distribution(generator);  // generate [1, 6]
        cout << dice_roll << ' ';
    }
    cout << endl;
}
```

Bind

- Bind is a functional programming template (with defaults) that among other uses can define a single, intuitive name to distribution(generator)
  - Shorter code is *usually* more readable code

http://www.cplusplus.com/reference/functional/bind/

Using random with bind

```
#include <random>
#include <chrono>
#include <functional>
#include <iostream>

using namespace std;

int main() {
    typedef std::chrono::high_resolution_clock myclock;
    myclock::time_point beginning = myclock::now();
    myclock::duration d = myclock::now() - beginning;
    unsigned seed = d.count();

    std::default_random_engine generator{seed};
    std::uniform_int_distribution<int> distribution(1,6);
    auto dice = std::bind(distribution, generator);

    for(int i=0; i<100; ++i) {
        cout << dice() << ' ';
    }
    cout << endl;
}
```

Review

Patterns

A **software design pattern** is a general reusable algorithm solving a common problem

- Patterns are **discovered**, not invented
- Patterns are documented and validated by the software development community at large - **meritocracy**

Patterns: What To Do

Types of Patterns

- Creational Patterns
  - These patterns address creating objects from classes via mechanisms more flexible than “new”
- Structural Patterns
  - These patterns address class and object composition, which is in general favored over inheritance
- Architectural Patterns
  - These patterns address design of loosely coupled subsystems of objects
- Behavioral Patterns
  - These patterns address communication between objects
- We covered 9 of the 23 original GoF patterns
  - It's good to know others, but only these 9 will be on the exam!

Patterns We Covered

- Creational
  - Singleton – restricts a class to a single instance
  - Factory – creates new objects based on specified criteria
- Structural
  - Decorator – dynamically adds new functionality to a class
  - Adapter – implements a bridge between 2 classes
  - Façade – provides a simple interface to a complex class or package
- Architectural
  - MVC – separates the business logic from the user interface
- Behavioral
  - Observer – notifies dependent objects of events of interest
  - Strategy – enable algorithm behavior to be modified at runtime
  - State Design – supports full scalable encapsulation of unlimited states

Anti-Patterns

- Anti-Patterns are common errors reflected in software systems
  - Common enough to be given a name
  - One or more strategies are known to exist
- This is a small subset of the most common / irksome (IMHO) anti-patterns
  - Wikipedia actually has an extensive list
  - However, only these will be on the exam!

Anti-Pattern: Comment Inversion and Documentation Inversion

- **Comment Inversion**: Adding obvious code comments ("adds two variables") and omitting useful comments such as <u>why</u> those variables should be added in the first place!
- **Documentation Inversion**: Explaining the system in terms of itself ("select 'New Foo' to create a new Foo") rather than documenting what a Foo is and why the user may want one
- In both cases, empathy is power. Consider the reader and what they are likely to <u>want</u> to know, rather than just writing down what you happen to know.
  - User-testing documentation is helpful to learn empathy

Anti-Pattern: Error Hiding

- Catching an error and either doing nothing with it or displaying a meaningless message
  - May also refer to destroying the evidence, e.g., overwriting the stack trace or disposing of problematic input
  - May also refer to addressing the error inappropriately, e.g., printing an error message in a library routine
- Catch exceptions only for specific reasons for which reasonable remediation is well-understood

Anti-Pattern: The "God Class"

- Functionality migrates upward in the class hierarchy until most methods are at the root
- This is a symptom that inheritance *may be* less appropriate for your design than composition
  - Use UML to understand your current class hierarchy
  - Test multiple redesigns with UML to more properly partition and encapsulate your data and allocate responsibilities to managable units of code.

AntiPattern: The Big Ball of Mud

- A software system lacking any discernible architecture, and is thus a “haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle” (Wikipedia).
  - Often result of a project with high programmer turnover and a long life of unstable requirements, bad managers, and tight budgets
  - Sometimes called “legacy code”
- *Usually* avoid temptation to rewrite – instead, try to *refactor*
  - Start by writing good test code that captures correct behavior
  - Next, define a clear and appropriate architecture for the system
  - Replace portions of the system one-by-one with well-designed code, using your test code to verify equivalent functionality
  - Continue until the Big Ball of Mud becomes maintainable enough

AntiPattern: Not Invented Here (NIH) Syndrome

- NIH is reimplementing an existing solution, e.g., a standard class library, due to the mistaken belief that code “not invented here” is inferior
  - If the longevity of the existing solution is in question, delegate to the Buyer to escrow or otherwise ensure access
  - If licensing concerns exist, discuss with Legal
  - If the code doesn't fit the problem precisely, engineer an adapter or façade

AntiPattern: Cargo Cult Programming

- Inclusion of code, language features, data structures, or (ahem) patterns without really understanding why
  - Often occurs when an inexperienced programmer copies code without understanding it
  - Lack of understanding results in redundant code, subtle errors, and unnecessary or misleading comments
- Complete due diligence and thoroughly understand EVERYTHING you include in your designs, code, and documentation
  - Ask questions! That what Sr. Programmers are for!

Quick Review

- List 4 options for representing strings in C++ and the most significant advantage of each.
- How is a map similar to a vector? What's the most significant difference?
- How are key / value pairs accessed in a map?
  (a) value = map[key]
  (b) map.key and map.value
  (c) iterator->key and iterator->value
  (d) iterator->first and iterator->second
- Which are common map operations?
  (a) navigate (b) begin and end (c) operator[ ] (d) find
- List at least 3 advantages of <random> over rand.
- Which type(s) compose a <random> number?
  (a) rand (b) generator (c) to_string (d) at (e) distribution

Quick Review

- Match Creational Pattern, Structural Pattern, and Behavioral Pattern to their definitions
  - These patterns address class and object composition, which is in general favored over inheritance
  - These patterns address communication between objects
  - These patterns address creating objects from classes via mechanisms more flexible than “new”
- Match the pattern name to the application and identify its type

Observer	1	Creates new objects without exposing the creation logic to the client
Decorator	2	Enables an algorithm behavior to be modified at runtime
Strategy	3	Implements a bridge between two classes with incompatible interfaces
Singleton	4	Restricts a class to instancing a single object
Adapter	5	Dynamically adds new functionality to an object without altering its structure
Factory	6	Implements a simplified interface to a complex class or package
MVC	7	Notifies dependent objects when the observed object is modified
State Design	8	Separates business logic from data visualization and human or machine user
Façade	9	Supports full scalable encapsulation of unlimited states

Quick Review

- Match the anti-pattern name to its definition

Documentation Inversion	1	Adding obvious code comments while omitting useful comments
The “God Class”	2	Explaining the system in terms of itself
Comment Inversion	3	Catching an error and ignoring it or displaying a meaningless message
Not Invented Here (NIH) Syndrome	4	Migrating functionality upward in the class hierarchy until most methods are at root
Error Hiding	5	A software system lacking any discernable architecture
The Big Ball of Mud	6	Reimplementing an existing solution because “our code is always better”
Cargo Cult Programming	7	Inclusion of code, language features, data structures or patterns with understanding why

Quick Review

- Suggest a pattern to address each of the following concerns
  - "I want to create an object based on subjective criteria"
  - "I want to switch to a more precise algorithm as my self-driving car approaches another vehicle"
  - "I want my software to be notified every time the user clicks the left mouse button"
  - "I want to extend a set of Boolean methods so that each one, when called, keeps re-running until it returns true"
- Define and suggest strategies to overcome each of the following anti-patterns
  - The "God Class"
  - Not Invented Here (NIH) Syndrome
  - The Big Ball of Mud
  - Comment and Documentation Inversion
  - Error Hiding
  - Cargo Cult Programming

Unsolicited Career Advice

(1) Integrity matters most. Never lie, mislead, or claim undue credit for any reason.

(2) Make your team and your boss look good. Credit others for success, but accept responsibility for your own mistakes quickly. **Focus on the solution**, not the blame. **Be a teacher** and help others grow and succeed, then cheer their success.

(3) Network broadly. Remember peoples' names. Cherish and practice loyalty.

(4) Being the person who knows who knows the answer is often more valuable than actually knowing the answer.

(5) Prefer "Yes". Consider no task beneath you, and do what is necessary for the team to win. But say "no" when necessary, e.g., to optimize your work load.

(6) Code early and often. A software manager or architect who doesn't code is clueless.

(7) Keep an engineering log. Write down everything, especially solutions and reasons.

(8) Learn new languages and tools often. **Automate everything.**

(9) Start stuff. Show initiative. Motivate your team, even if you're the junior member. Make good things happen. It's contagious.

(10) Keep a personal task backlog. Keep it updated. Work it by priority.