CSCI-1200 Data Structures — Fall 2022
Lecture 26 — C++ Exceptions

Announcements

- The Final Exam will be on **Monday December 19th from 6:30-9:30pm America/New York**.

  full schedule: https://rpi.app.box.com/s/g69fsj4lahzqfdv2bryphqtcctec36ik
  final exam conflicts: https://info.rpi.edu/registrar/grades

  Please check the final exam schedule for all of your classes. If you have a conflict, please ask the other instructor of the other course to make arrangements for a makeup exam for you.

- Those of you interested in becoming an undergraduate mentor for Data Structures, or another CSCI course:
  - Shianne Hulbert & David Goldschmidt will email a mentor application website URL to all CSCI majors. You will fill out your qualifications, weekly class schedule availability, and your course mentoring interests.
  - *Note: You don't have to be a CSCI major to apply (ask a friend to forward you the application link!)*
  - Also speak to your graduate lab TA/mentor and ask them to recommend you for the position.
  - Fill out the application **after you have your final Data Structures semester grade.**

Review from Lecture 25

- STL Queue and STL Stack, What's a Priority Queue?
- Definition of a Binary Heap, A Priority Queue as a Heap
- Implementing Pop (Delete Min) and Push (Insert), A Heap as a Vector
- Heapify – Building a Heap (all at once), Heap Sort, Complexity Analysis

Today's Class

- Error handling strategies
- Basic exception mechanisms: `try`/`throw`/`catch`
- Functions & exceptions, constructors & exceptions
- STL `exceptions`
- RAII "Resource Acquisition is Initialization"
- Structured Exception Handling in the Windows Operating System
- Google's C++ Style Guide on Exceptions
- Some examples from today's lecture are drawn from:
  http://www.cplusplus.com/doc/tutorial/exceptions/
  http://www.parashift.com/c++-faq-lite/exceptions.html

26.1 Error Handling Strategy A: Optimism (a.k.a. Naivety or Denial)

- **Assume there are no errors.** Command line arguments will always be proper, any specified files will always be available for read/write, the data in the files will be formatted correctly, numerical calculations will not attempt to divide by zero, etc.

```
double answer = numer / denom;
```

- For small programs, for short term use, by a single programmer, where the input is well known and controlled, this may not be a disaster (and is often fastest to develop and thus a good choice).
- But for large programs, this code will be challenging to maintain. It can be difficult to pinpoint the source of an error. The symptom of a problem (if noticed at all) may be many steps removed from the source. The software system maintainer must be familiar with the assumptions of the code (which is difficult if there is a ton of code, the code was written some time ago, by someone else, or is not sufficiently commented... or all of the above!).

26.2 Error Handling Strategy B: Plan for the Worst Case (a.k.a. Paranoia)

- Anticipate every mistake or source of error (or as many as you can think of). **Write lots of `if` statements everywhere there may be a problem**. Write code for what to do instead, print out error messages, and/or exit when nothing seems reasonable.

```
double answer;
// for some application specific epsilon (often not easy to specify)
double epsilon = 0.00001;
if (fabs(denom) < epsilon) {
  std::cerr << "detected a divide by zero error" << std::endl;
  // what to do now?  (often there is no "right" thing to do)
  answer = 0;
} else {
  answer = numer / denom;
}
```

- Error checking & error handling generally requires a lot of programmer time to write all of this error code.
- The code gets bulkier and harder to understand/maintain.
- If a nested function call might have a problem, and the error needs to be propagated back up to a function much earlier on the call stack, all the functions in between must also test for the error condition and pass the error along. (This is messy to code and all that error checking has performance implications).
- Creating a comprehensive test suite (yes, error checking/handling code must be tested too!) that exercises all the error cases is extremely time consuming, and some error situations are very difficult to produce.

26.3 Error Strategy C: If/When It Happens We'll Fix It (a.k.a. Procrastination)

- Again, anticipate everything that might go wrong and just **call `assert` in lots of places**. This can be somewhat less work than the previous option (we don't need to decide what to do if the error happens, the program just exits immediately).

```
double epsilon = 0.00001;
assert (fabs(denom) > epsilon);
answer = numer / denom;
```

- This can be a great tool during the software development process. Write code to test all (or most) of the assumptions in each function/code unit. Quickly get a prototype system up and running that works for the general, most common, non-error cases first.
- If/when an unexpected input or condition occurs, then additional code can be written to more appropriately handle special cases and errors.
- However, the use of assertions is generally frowned upon in real-world production code (users don't like to receive seemingly arbitrary & total system failures, especially when they paid for the software!).
- Once you have completed testing & debugging, and are fairly confident that the likely error cases are appropriately handled, then the `gcc` compile flag `-DNDEBUG` flag can be used to remove all remaining `assert` statements before compiling the code (conveniently removing any performance overhead for assert checking).

26.4 Error Handling Strategy D: The Elegant Industrial-Strength Solution

- **Use exceptions.** Somewhat similar to Strategy B, but in practice, code written using exceptions results in more efficient code (and less overall code!) and that code is less prone to programming mistakes.

```
double epsilon = 0.00001;
try {
  if (fabs(denom) < epsilon) {
    throw std::string("divide by zero");
  }
  double answer = numer / denom;
  /* do lots of other interesting work here with the answer! */
}
catch (std::string &error) {
  std::cerr << "detected a " << error << " error" << std::endl;
  /* what to do in the event of an error */
}
```

26.5 Basic Exception Mechanisms: Throw

- When you detect an error, `throw` an exception. Some examples:

```
throw 20;
throw std::string("hello");
throw Foo(2,5);
```

- You can throw a value of any type (e.g., `int`, `std::string`, an instance of a custom class, etc.)
- When the throw statement is triggered, the rest of that block of code is abandoned.

26.6 Basic Exception Mechanisms: Try/Catch

- If you suspect that a fragment of code you are about to execute may throw an exception and you want to prevent the program from crashing, you should wrap that fragment within a try/catch block:

```
try {
  /* the code that might throw */
}
catch (int x) {
  /* what to do if the throw happened
     (may use the variable x)
  */
}
/* the rest of the program */
```

- The logic of the try block may throw more than one type of exception.
- A catch statement specifies what type of exception it catches (e.g., `int`, `std::string`, etc.)
- You may use multiple catch blocks to catch different types of exceptions from the same try block.
- You may use `catch (...) { /* code */ }` to catch *all* types of exceptions. (But you don't get to use the value that was thrown!)
- If an exception is thrown, the program searches for the closest *enclosing* try/catch block with the appropriate type. That try/catch may be several functions away on the call stack (it might be all the way back in the main function!).
- If no appropriate catch statement is found, the program exits, e.g.:

```
terminate called after throwing an instance of 'bool'
Abort trap
```

26.7 Basic Exception Mechanisms: Functions

- If a function you are writing might throw an exception, you can specify the type of exception(s) in the prototype.

```
int my_func(int a, int b) throw(double,bool) {
  if (a > b)
    throw 20.3;
  else
    throw false;
}

int main() {
  try {
    my_func(1,2);
  }
  catch (double x) {
    std::cout << " caught a double " << x << std::endl;
  }
  catch (...) {
    std::cout << " caught some other type " << std::endl;
  }
}
```

- If you use the throw syntax in the prototype, and the function throws an exception of a type that you have not listed, the program will terminate immediately (it can't be caught by any enclosing try statements).
- If you don't use the throw syntax in the prototype, the function may throw exceptions of any type, and they may be caught by an appropriate try/catch block.

26.8 Comparing Method B (explicit if tests) to Method D (exceptions)

- Here's code using exceptions to sort a collection of lines by slope:

```
class Point {
public:
  Point(double x_, double y_) : x(x_),y(y_) {}
  double x,y;
};

class Line {
public:
  Line(const Point &a_, const Point &b_) : a(a_),b(b_) {}
  Point a,b;
};

double compute_slope(const Point &a, const Point &b) throws(int) {
  double rise = b.y - a.y;
  double run = b.x - a.x;
  double epsilon = 0.00001;
  if (fabs(run) < epsilon) throw -1;
  return rise / run;
}

double slope(const Line &ln) {
  return compute_slope(ln.a,ln.b);
}

bool steeper_slope(const Line &m, const Line &n) {
  double slope_m = slope(m);
  double slope_n = slope(n);
  return slope_m > slope_n;
}

void organize(std::vector<Line> &lines) {
  std::sort(lines.begin(),lines.end(), steeper_slope);
}

int main () {
  std::vector<Line> lines;
  /* omitting code to initialize some data */
  try {
    organize(lines);
    /* omitting code to print out the results */
  } catch (int) {
    std::cout << "error: infinite slope" << std::endl;
  }
}
```

- Specifically note the behavior if one of the lines has infinite slope (a vertical line).
- Note also how the exception propagates out through several nested function calls.
- **Exercise:** Rewrite this code to have the same behavior but *without exceptions*. Try to preserve the overall structure of the code as much as possible. (Hmm... it's messy!)

26.9 STL `exception` Class

- STL provides a base class `std::exception` in the `<exception>` header file. You can derive your own exception type from the exception class, and overwrite the `what()` member function

```
class myexception: public std::exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
};

int main () {
  myexception myex;
  try {
    throw myex;
  }
  catch (std::exception& e) {
    std::cout << e.what() << std::endl;
  }
  return 0;
}
```

- The STL library throws several different types of exceptions (all derived from the STL `exception` class):

`bad_alloc`	thrown by new on allocation failure
`bad_cast`	thrown by `dynamic_cast` (when casting to a reference variable rather than a pointer)
`bad_exception`	thrown when an exception type doesn't match any catch
`bad_typeid`	thrown by typeid
`ios_base::failure`	thrown by functions in the iostream library

26.10 Exceptions & Constructors

- The only way for a constructor to fail is to throw an exception.
- A common reason that a constructor must fail is due to a failure to allocate memory. If the system cannot allocate sufficient memory resources for the object, the `bad_alloc` exception is thrown.

```
try {
  int* myarray= new int[1000];
}
catch (std::exception& e) {
  std::cout << "Standard exception: " << e.what() << std::endl;
}
```

- It can also be useful to have the constructor for a custom class throw a descriptive exception if the arguments are invalid in some way.

26.11 Resource Acquisition Is Initialization (RAII)

- Because exceptions might happen at any time, and thus cause the program to abandon a partially executed function or block of code, it may not be appropriate to rely on a `delete` call that happens later on in a block of code.
- RAII describes a programming strategy to ensure proper deallocation of memory despite the occurrence of exceptions. The goal is to ensure that resources are released before exceptions are allowed to propagate.
- Variables allocated on the stack (not dynamically-allocated using `new`) are guaranteed to be properly destructed when the variable goes out of scope (e.g., when an exception is thrown and we abandon a partially executed block of code or function).
- Special care must be taken for dynamically-allocated variables (and other resources like open files, mutexes, etc.) to ensure that the code is *exception safe*.

26.12 Structured Exception Handing (SEH) in the Windows Operating System

- The Windows Operating System has special language support, called Structured Exception Handing (SEH), to handle hardware exceptions. Some examples of hardware exceptions include divide by zero and segmentation faults (there are others!).
- In Unix/Linux/Mac OSX these hardware exceptions are instead dealt with using signal handlers. Unfortunately, writing error handling code using signal handlers incurs a larger performance hit (due to `setjmp`) and the design of the error handling code is less elegant than the usual C++ exception system because signal handlers are global entities.

26.13 Google's C++ Style Guide on Exceptions

https://google.github.io/styleguide/cppguide.html#Exceptions

Pros:

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an Init() method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons:

- When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if f() calls g() calls h(), and h throws an exception that f catches, g has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

Decision:

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

There is an exception to this rule (no pun intended) for Windows code.