Error Handling
Introduction to Exceptions

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, February 8, 2013

Chris Hartman

Department of Computer Science
University of Alaska Fairbanks
`cmhartman@alaska.edu`

Based on material by Glenn G. Chappell

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- ✓ Pointers & dynamic allocation
- ✓ Managing resources in a class
- ✓ Templates
- ✓ Containers & iterators
- Error handling
- Introduction to exceptions
- Introduction to Linked Lists

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- ✓ Testing
- ✓ Some principles

Review
Containers & Iterators [1/4]

A **generic container** is a container that can hold items of a client-specified type.

- One kind of generic container is: an array.
- Others are in the C++ **Standard Template Library** (STL).

The STL includes **`std::vector`**, a **smart array** template.

```
std::vector<int> iv(2);
iv[1] = 5;
iv.push_back(4);
cout << iv[2] << endl;       // Prints "4"
cout << iv.size() << endl;  // Prints "3"
```

Review
Containers & Iterators [2/4]

An **iterator** is an object that references an item in a container (or acts like it).

- Iterators do not own what they reference (like a non-owning pointer).

STL containers have **iterator types**.

```
std::vector<int>::iterator iter;          // Like normal pointer
std::vector<int>::const_iterator citer;   // Like pointer-to-const
```

Typical code using iterators:

```
std::vector<int> v(7);
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    *it = 6;
```

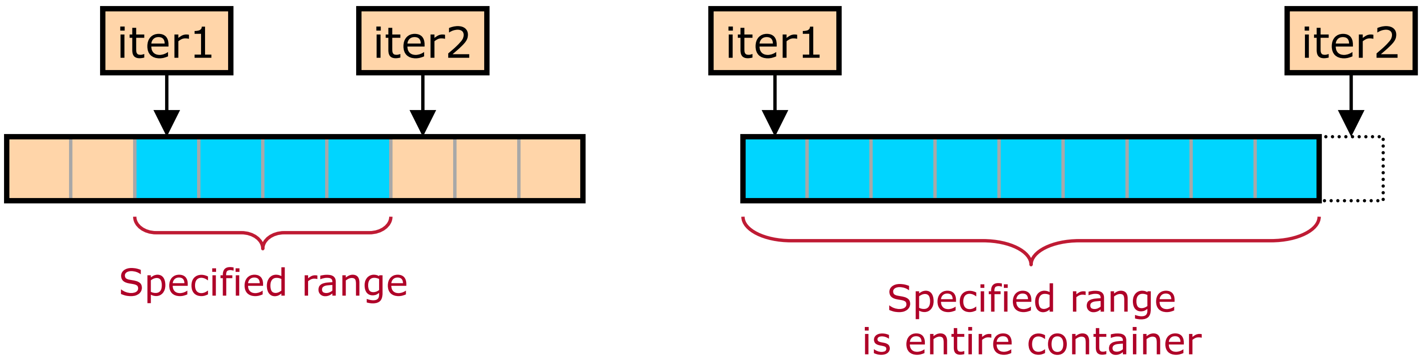
An iterator can be a **wrapper** around data, to make it look like a container.

```
#include <iterator>
std::ostream_iterator<int> myCoolNewIterator(std::cout, "\n"));
*myCoolNewIterator++ = 3;  // Now this does the same as the next line
std::cout << 3 << "\n";
```

Review
Containers & Iterators [3/4]

To specify a **range**, we use two iterators:

- An iterator to the first item in the range.
- An iterator to just past the last item in the range.



STL algorithms use this convention.

```
std::copy(v.begin(), v.end(), v2.begin());
std::for_each(v.begin(), v.end(), myFunc);
std::sort(v.begin(), v.end())
```

Each underlined pair of parameters forms a range specification.

Containers & Iterators
Iterator Basics — Iterators and Kinds of Data

Operations available on an iterator match the underlying data.

- Iterators for one-way sequential-access data have the **++** operation.
  - Such an iterator is called a **forward iterator** (example of an **iterator category**).

```
++forwardIterator;
```

- Iterators for two-way sequential-access data also have the **--** operation.
  - These are **bidirectional iterators**.

```
++bidirectionalIterator;
--bidirectionalIterator;
```

- Iterators for random-access data have all the pointer arithmetic operations.
  - These are **random-access iterators**.

```
++randomAccessIter;
--randomAccessIter;
randomAccessIter += 7;
cout << randomAccessIter[5];
std::ptrdiff_t dist = randomAccessIter2 - randomAccessIter1;
```

Containers & Iterators
Wrap-Up: Three STL Algorithms to Know

Be familiar with the following STL algorithms (all in **`<algorithm>`**):

- Copying: **`std::copy`**

```
std::copy(v.begin(), v.end(), v2.begin());
    // Copy items in v to v2 (which must have space!)
```

- For-each loop: **`std::for_each`**

```
std::for_each(v.begin(), v.end(), myFunc);
    // Call myFunc on each item in v
```

- Sorting: **`std::sort`**

```
std::sort(v.begin(), v.end());
    // Arrange items in v in ascending order
```

Error Handling - Review
Dealing with Possible Error Conditions

Sometimes we can **prevent errors**:

- Write a precondition that requires the caller to keep a certain problem from happening.
- Example: Insisting on a non-zero parameter, to prevent a division-by-zero error condition.

Handle the error **before** the function

Sometimes we can **contain errors**, by handling them ourselves:

- If something does not work, fix it.
- Example: To run a fast algorithm, we need a large buffer. Memory is low, and we cannot allocate the buffer. So we run a slower algorithm that needs no buffer.

Handle the error **during** the function

But sometimes we can do neither of these …

Then we must **signal the client code**.

- Rule of thumb: Signal the client code when the function is unable to fulfill its postconditions.
- Example: The earlier file-reading example.

Handle the error **after** the function

Error Handling - Review
Flagging Errors

When we cannot prevent or contain an error, we must signal the client code. **How?**

Method 1: Returning an error code

- Here we indicate an error by our return value (or a reference parameter).
- The old “C” I/O library uses this method:

```
int c = getc(myFile);
if (c == EOF)
    printf("End of file\n");
```

Method 2: Setting a flag to be checked by a separate error-checking function

- Here the caller uses some other function to check whether there was an error.
- C++ file streams use this method by default:

```
char c;
myFileStream >> c;
if (myFileStream.eof())
    cout << "End of file" << endl;
```

Error Handling
Need for Another Method

Return codes and separate error-checking functions are both fine methods for flagging errors, but they do have problems.

- They can be difficult to use in places where a value cannot be returned, or an error condition cannot be checked for.
  - Constructors & destructors. Also bracket operator, etc.
  - In the middle of an expression.
  - When you call someone else's function, and that calls your function, which needs to signal an error condition.
    - Call-back functions, templates, etc.
- They can lead to complicated code.
  - A function calls a function, which calls a function … and an error occurs. To handle the error, we have to back out of all of these. Lots of `if`'s.

To deal with these problems, a third method was developed.

Method 3: **Throwing an exception**

- Exceptions are available in many languages (C++, Java, Python, Ruby, Javascript, etc.), and are generally associated with OOP.
- Shortly, we will look at exception handling in C++.

Summary
Error Handling

An **error condition** (or "error") is a condition occurring during runtime that cannot be handled by the normal flow of execution.

- Not necessarily a bug or a user mistake.
- Example: Could not read file.

Three ways to deal with a *possible* error condition in a function:

before — **Prevention**
Client code must prevent the error (precondition).

during — **Containment**
Fix the problem inside the function.

after — **Signal the Client Code**
Idea: When we cannot fulfill our postconditions.

We **like** these two, but they *might* not be feasible

Methods for signaling an error condition to the client code:

- Return an error code
- Set a flag, checked by a separate function
- Throw an **exception**

Introduction to Exceptions
Exceptions & Catching — The Idea

Exception: an object that is "**thrown**" when a function terminates abnormally.

- Example: "**new**" throws an object of type `std::bad_alloc` if allocation fails.

```
Foo * p = new Foo;  // May throw std::bad_alloc
```

In order to handle exceptions, we **catch** them using a `try` … `catch` construction.

```
#include <new>    // for std::bad_alloc

Foo * p;
bool allocationSuccessful = true;
try {
    p = new Foo;
}
catch (std::bad_alloc & e) {
    allocationSuccessful = false;
    cout << "Allocation failed. Message: " << e.what() << endl;
}
```

The "`catch`" gets an expression of the proper type that is thrown inside the corresponding "`try`".

Catch exceptions by reference.

`e` is the **exception**.

Member function of standard exception types. Returns `string`.

Introduction to Exceptions
Exceptions & Catching — What is Caught? [1/4]

A **`catch`** only gets an exception that is:

- Thrown inside the corresponding **`try`** block.
- Of an appropriate type.

Once an exception is thrown, the **`try`** block is exited.

If no exception is thrown, the **`catch`** block is not executed.

```
Foo * p1, p2;
p1 = new Foo;
try {
    p2 = new Foo;
    myFunc(p2);
}
catch (std::bad_alloc & e) {
    [exception-handling code goes here]
}
```

The **`catch`** block below will **not** catch any exception thrown by this statement.

If the **`new`** fails, then this function call is not made.

If this function throws an exception that is not **`std::bad_alloc`** or a derived type, then the **`catch`** block below is **not** executed.

Introduction to Exceptions
Exceptions & Catching — What is Caught? [2/4]

A **`catch`** gets exceptions of the proper type that are thrown inside the corresponding **`try`** block. This includes exceptions thrown in function calls, if they are not caught inside the functions.

```
void myFunc()
{
    globalP1 = new Foo;
    globalFlag = true;
    try {
        globalP2 = new Foo;
    }
    catch (std::bad_alloc & e) {
        globalFlag = false;
    }
}

int main()
{
    try {
        myFunc();
    }
    catch (std::bad_alloc & e) {
```

The **`catch`** in function **`main`** will catch an exception thrown by this statement …

… but not by this statement.

Introduction to Exceptions
Exceptions & Catching — What is Caught? [3/4]

Exceptions can propagate out of deeply nested function calls.

```
void f1();  // May throw std::bad_alloc

void f2()
{ f1(); }

void f3()
{ f2(); }

void f4()
{ f3(); }

void f5()
{
    try {
        f4();
    }
    catch (std::bad_alloc & e) {
```

An exception thrown **here** is caught **here**.

Exceptions & Catching — What is Caught? [4/4]

When we catch by reference (recommended), we also catch derived types.

```
#include <stdexcept>  // for std::exception

void myFunc2();       // May throw std::range_error

int main()
{
    try {
        myFunc2();
    }
    catch (std::exception & e) {
```

This will catch a **std::range_error**. All standard exception classes are derived from **std::exception**.

An uncaught exception terminates the program.

```
void myFunc3();  // May throw std::range_error

int main()
{
    Foo * p1 = new Foo;
    try {
        myFunc3();
    }
    catch (std::bad_alloc & e) {
```

An exception **here** or **here** will terminate the program.

Introduction to Exceptions
Throwing

We can throw our own exceptions, using “**`throw`**”.

```
class Foo {
public:
    int & operator[](int index)  // May throw std::range_error
    {
        if (index < 0 || index >= arraySize)
            throw std::range_error("Foo: index out of range");
        return theArray[index];
    }
private:
    int * theArray;
    std::size_t arraySize;
};
```

We do this only do it when we must signal the client code that an error condition has occurred.

Introduction to Exceptions
Catch All & Re-Throw

We can catch **all** exceptions, using "`...`".

- In this case, we do not get to look at the exception, since we do not know what type it is.

```
try {
    myFunc4(17);
}
catch (...) {
    fixThingsUp();
    throw;
}
```

- Inside any **`catch`** block, we can **re-throw the same exception** using **`throw`** with no parameters.

Introduction to Exceptions
Put It All Together — Example Code

```
void f(const Foo & x)  // throw(std::runtime_error)
{ if (!xtest(x)) throw std::runtime_error("xtest failed"); }

void g()  // throw(std::runtime_error)
{
    Foo x;
    f(x);
    do_something(x);
}

void h()  // throw()  // Does not throw any exceptions
{
    try
    { g(); }
    catch (std::runtime_error & e)
    { cout << "Runtime error: " << e.what(); }
```

[*More code here …*]

Introduction to Exceptions
Put It All Together — Thoughts

When throwing your own exception (which we won't do very often), it is a good idea to use or derive from one of the standard exception types.

- Some people throw strings. Do not do this.
  - It would mean you cannot catch by type.
- Standard exception classes have a string member, to use as a message.
  - This is a parameter to the ctor and is accessed through the `what()` member.
- To make your own exception type, derive from a standard exception class.
  - All standard exception classes are set up to allow this.

Catch exceptions **by reference**.

- Thrown objects may be copied, regardless. Catching by value copies the copy.
- Catching by reference allows for derived types, which are commonly used.

`throw-catch` is just another flow-of-control structure, like "`if`", "`for`", etc.

- Recommendation: **Use C++ exceptions only to handle error conditions**.

Exception specifications allow you to tell the compiler what types of exceptions a function might throw.

- These are present, but commented out, on the "Example Code" slide.
- Recommendation: Avoid exception specifications. (Deprecated in C++11, except for noexcept, which is encouraged.)

Introduction to Exceptions
What Throws?

The following can throw in C++:

- "**`throw`**" throws.
- "**`new`**" may throw **`std::bad_alloc`** if it cannot allocate.
  - There is a non-throwing version of **`new`**. See the applicable doc's.
- A function that (1) calls a function that throws, and (2) does not catch the exception, will throw.
- Functions written by others may throw. See their doc's.

The following do *not* throw:

- Built-in operations on built-in types.
  - Including the built-in **`operator[]`**.
- Deallocation done by the built-in version of "**`delete`**".
  - Note: "**`delete`**" also calls destructors. These can throw.
- C++ Standard I/O Libraries (default behavior)
  - You *can* tell standard file streams to throw when an error occurs. However, they are non-throwing by default.

Introduction to Exceptions
Double Exceptions & Dctors

Fact 1. An automatic object's dctor is called when it goes out of scope, even if this is due to an exception.

- This is why the dctor is the place to do clean-up operations (deallocate memory, release resources, etc.; think "RAII"). An exception may bypass your carefully written clean-up code, but it will not bypass the dctor.
- Note: Dctors are only called for **fully constructed** objects. If a ctor throws, then the dctor will not be called.

Fact 2. If an exception is thrown, and one of the destructors called before it is caught also throws, then the program terminates.

Put these two facts together, and we conclude:
Destructors generally should not throw.

Other Thoughts:

- If a destructor throws, this says that the object cannot be properly destroyed. So the program cannot end (?).
- It is okay for *constructors* to throw.

Introduction to Exceptions
Example 1

TO DO

- Run some code that throws & catches.

Introduction to Exceptions
Example 2

TO DO

- Write a function **`allocate1`** that:
  - Takes a **`size_t`**, indicating the size of an array to be allocated.
  - Attempts to allocate an array of **`int`**s, of the given size.
  - Returns a pointer to this array, using a reference parameter.
  - If the allocation fails, throws **`std::bad_alloc`**.
  - … and has no memory leaks.
- Write a function **`allocate2`** that:
  - Takes a **`size_t`**, the size of **two arrays** to be allocated.
  - Attempts to allocate **two arrays** of **`int`**s, both of the given size.
  - Returns pointer to these arrays, using reference parameters.
  - If the allocation fails, throws **`std::bad_alloc`**.
  - … and has no memory leaks.

Introduction to Exceptions
Final Thoughts

When to Do Things:

- **Throw** when a function you are writing is unable to fulfill its postconditions.
- **Try/Catch** when you can handle an error condition that may be signaled by some function you call.
  - Or simply to prevent a program from crashing.
- **Catch all and re-throw** when you call a function that may throw, you cannot handle the error, but you do need to do some clean-up before your function exits.

Typically we do not do more than one of the above.

- For example, someone else throws, and we catch.

Some people do not like exceptions.

- A *bad reason* not to like exceptions is that they require lots of work.
  - Dealing with **error conditions** is a lot of work. Exceptions are one method of dealing with them. Handling exceptions properly is hard work simply because **writing correct, robust code is hard work**.
- A *good reason* might be that they add hidden execution paths.