

# Introduction to Exceptions, cont'd

---

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, February 15, 2008

Glenn G. Chappell

Department of Computer Science  
University of Alaska Fairbanks

[CHAPPELLG@member.ams.org](mailto:CHAPPELLG@member.ams.org)

© 2005–2008 Glenn G. Chappell

# Unit Overview

## Advanced C++

---

### Major Topics

- ✓ • The structure of a package.
- ✓ • Parameter passing & “const”.
- ✓ • Operator overloading.
- ✓ • Silently written and silently called functions.
- ✓ • Pointers and dynamic allocation.
- ✓ • Managing resources in a class.
- ✓ • Templates.
- ✓ • Containers & iterators.
- ✓ • Error handling.
- $\frac{1}{2}$ ✓ • Introduction to exceptions.

# Review

## Error Handling

---

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

- Not the same as a bug.
- Example: Could not read file.

Three ways of dealing with an error condition:

- Prevention
  - Require the client (via a precondition?) to prevent an error condition.
- Containment
  - Fix the problem ourselves.
- **Signal the Client**
  - Rule of thumb: When we cannot fulfill our postconditions.

Three ways to signal an error condition to the client:

- Returning an error code.
- Setting a flag to be checked by a separate error-checking function.
- Throwing an exception.

# Review

## Introduction to Exceptions — Catching

---

**Exceptions** are objects that are “**thrown**”, generally to signal error conditions.

- We **catch** exceptions using a `try ... catch` construction.
- A **throw** causes repeated backing out of blocks & functions, until a matching **catch** is found.
- An uncaught exception terminates the program.

```
Foo * makeAFoo() // throw(std::bad_alloc)
{
    return new Foo(2, 3);
}

void myFunc() // throw()
{
    Foo * p;
    try {
        p = makeAFoo();
    }
    catch (std::bad_alloc & e) {
        allocationSuccessful = false;
        cout << "Oops! Message: " << e.what() << endl;
    }
    [more stuff here]
}
```

Commented-out **exception specifications**.  
If uncommented, these are legal C++; I do not recommend using them in release code.

# Review

## 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 not do this very much. And we only do it when we need to signal the client that an error condition has occurred.

## Review

### 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, cont'd

## 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, cont'd

## 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.

- So: Don't throw exceptions in such circumstances.

Put these two facts together, and we conclude:

- **Destructors generally should not throw.**
  - This means that exceptions are not good for one their original purposes: error handling in dctors.
  - They are fine for errors in ctors, however, as long as the ctors clean up when necessary.

# Introduction to Exceptions, cont'd

## Example 1

---

### TO DO

- Run some code that throws & catches.

*Done. See `throwing.cpp`,  
on the web page.*

# Introduction to Exceptions, cont'd

## 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 `ints`, 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 `ints`, 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.

*Done. See `allocate2.cpp`,  
on the web page.*

# Introduction to Exceptions, cont'd

## Final Thoughts: C++

---

### When to Do Things:

- **Throw** when a function you are writing is unable to fulfill its postconditions.
- **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.

# Introduction to Exceptions, cont'd

## Exceptions in Other Languages [1/2]

---

A number of other languages have a slightly different way of dealing with exceptions. I explain Python's method here. Java and Javascript are both similar.

### Execution

- Block A is executed.
- If an exception is raised, then the first matching exception handler (one of B, C, D) is executed (D handles any exception not matched by B, C).
- If an exception is **not raised** then the else-block (E) is executed.
- When all this is done, the finally-block (F) is executed, no matter what, even if a second exception was raised in B, C, D, or E.

Parts of this can be left out. If there is no handle-any-exception block (here: D), then an uncaught exception will search elsewhere for a handler, just as in C++. However, Block F will be executed first.

```
try:
    [Block A]
except ExceptionClass1:
    [Block B] # catch EC1
except ExceptionClass2:
    [Block C] # catch EC2
except:
    [Block D] # catch other
else:
    [Block E] # if no exception
finally:
    [Block F] # ALWAYS executes
```

# Introduction to Exceptions, cont'd

## Exceptions in Other Languages [2/2]

---

This alternate way of dealing with exceptions leads to a different way of doing “catch-all-and-re-throw”.

For example, the code to the right reads from a file and closes the file when it is done.

If there is an error during opening, an exception is raised.

If there is an error during reading, an exception is raised, but the file is closed before control is passed to the handler (which would be elsewhere in the code).

- We might enclose all of this in a “`try: ... except IOError:`”.

```
f = open(filename, "r")
    # open file for reading
try:
    [read from file here]
finally:
    f.close()
    # close file if it is open
```

# Unit Overview

## Recursion & Searching

---

Next we begin a unit on Recursion & Searching.

- Major Topics
  - Introduction to Recursion.
  - Search Algorithms.
  - Recursion vs. Iteration.
  - Eliminating Recursion.
  - Recursive Search with Backtracking.
- We will follow the text more closely than we have been.
  - Recursion & Searching material is in chapters 2 & 5.

After this, we will look at Algorithmic Efficiency & Sorting.

- Chapter 9.