

Error Handling, cont'd

Introduction to Exceptions

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, February 13, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

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.
- 1/2 ✓ • Error handling.
 - Introduction to exceptions.

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.
- Other generic container types are in the C++ **Standard Template Library** (STL).

Kinds of Data

- **Random-access**
- **Sequential-access**
 - One-way
 - Two-way

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]

You are familiar with **counter-controlled loops**:

- Also called “**for**” loops.

```
for (int i = 0; i < 100; ++i)
    cout << i * i << endl;
```

... and **condition-controlled loops**:

- Also called “**while**” loops. But in C++ we *can* write these using the “for” construct.

```
while (!infile.eof())
{
    readFrom(infile);
    if (!infile) break;
}
```

Now we look at a third kind: **iterator-controlled loops**:

- Also called “**for-each**” loops.

```
int array[7];
for (int * p = array; p != array+7; ++p)
    *p = 6;
```

Review

Containers & Iterators [3/4]

Iterator-Controlled Loops

- With an array:

```
int array[7];  
for (int * p = array; p != array+7; ++p)  
    *p = 6;
```

- With a `std::vector`:

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

- As above, but using `typedef`:

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

Points to
first item

Points to
one-past last item

Review

Containers & Iterators [4/4]

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
```

An iterator can be a **wrapper** around data, to make it look like a container. We specify a **range** using two iterators: to first item, to just past last item. The C++ STL includes a number of **generic algorithms** that use iterators.

```
#include <algorithm>
std::copy(v.begin(), v.end(), arr1);
std::sort(arr1+5, arr1+20); // sorts items 5 .. 19
std::for_each(v.begin(), v.end(), printInt);
```

Review

Error Handling [1/2]

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

An error condition is not the same as a bug in the code.

- So we are not referring to compilation errors.
- But *some* error conditions are caused by bugs.
- However, in our discussion of error handling, we will usually assume that our code is properly written.

An error condition does not mean that the user did something wrong.

- Although *some* error conditions are caused by user mistakes.

Example

- Suppose we have a function `copyFile`, which opens a file, reads its contents into a buffer, and writes them to another file.
- Function `copyFile` is called in order to read a file on a device that is accessed via a network.
- Halfway through reading the file, the network goes down.
- The function is supposed to read the file. This is now impossible. The normal flow of execution cannot handle this. We have an error condition.

Review

Error Handling [2/2]

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.

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.

But sometimes we can do neither of these ...

Then we must **signal the client**.

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

Error Handling, cont'd

Goals and Guarantees (Preview)

In situations in which the client might need to be informed of errors, there are three things we would *like* to happen:

- First, we want to be sure that an error will not “mess up” our program. It should be able to continue to run, and, later, to terminate properly. Objects must still be usable. Also, resources should not be leaked.
- In addition, we would like it if, when we want to perform an operation, either the operation completes successfully with no errors, or, if there is an error, no change is made to the program’s data.
- Best of all, we would like it if the client never needs to be informed of an error at all.

Later in the class, we will formalize these as “safety guarantees”.

- The first idea above is the fundamental standard that all quality code *must* meet. We will call it the “Basic Guarantee”.
- The second is preferred, although sometimes we do not achieve it. We will call it the “Strong Guarantee”.
- The third is mostly wishful thinking. Errors happen, and sometimes the client needs to be informed. But in special cases (often involving “finishing things up”), we may need to insure that the client never needs to be informed of an error. We will call this the “No-Throw Guarantee”, or “No-Fail Guarantee”.

Error Handling, cont'd

Flagging Errors

When we cannot prevent or contain an error, we must signal the client. **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, cont'd

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 OOD.
- Shortly, we will look at exception handling in C++.

Error Handling, cont'd

Summary

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.

Sometimes
these two are
not feasible.

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.

Introduction to Exceptions

Exceptions & Catching

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;
```

```
}
```

e is the exception.



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



Introduction to Exceptions

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

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

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.

Introduction to Exceptions

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()
```

```
{
```

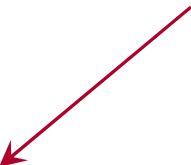
```
    try {
```

```
        myFunc2();
```

```
    }
```

```
    catch (std::exception & e) {
```

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



Introduction to Exceptions


Uncaught Exceptions

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 not do this very much. And we only do it when we need to signal the client 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

Example Code

```
void f(const Foo & x) // throw(std::runtime_error)
{ if (!my_test(x)) throw std::runtime_error("my_test 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

Thoughts

When throwing your own exception (which you won't do very much!), 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 are 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, except when debugging.