CS 213 -- Lecture #19

"Late Night Guide to C++"
Chapter 11

MORE ABOUT EXCEPTIONS

Administrative...

- Assignment #8 due Today:
- Remember, second prelim on 11/23

Exceptions and Inheritance

- Last lecture, we found out that when an exception is thrown there is no attempt to implicitly coerce the type of data thrown to a catch statement which "could" receive it.
- For example, if I throw the value `1.4` it will be caught by a catch statement looking for a `double`, not a `float`.
- This is different from the more general practice of C++ always doing whatever type conversions it can (such as when an integer value is assigned to a `float` variable).
- When it comes to inheritance, however, the behavior is more consistent.
- The normal "rules" of inheritance apply.
- Consider the following base class used for your own exceptions:

Exceptions and Inheritance (cont)

```
class BaseException
{
public:
  BaseException(string msg,int err=0):message(msg),
       errorCode(err){}
  virtual string getMessage()
       { return "BASE EXCEPTION: " + message; }
  int getErrorCode() { return errorCode; }
protected:
  string message;
  int errorCode;
};
```

- Here we have a simple exception class which can be used to store simple information about exceptional events. It can be thrown with:
  - `throw BaseException("Divide By Zero",-3);`

Exceptions and Inheritance (cont)

- Now, it can be caught with any of the following:

```
catch (BaseException e)    // Puts a copy of BaseException in e
catch (BaseException &e)   // Puts a reference in e
catch (BaseException)      // BaseException not accessible
catch (…)                  // BaseException not accessible
```

```
int main()
{
  try {
    cout << "4/0 is " << divide(4,0) << endl;
  }
  catch(BaseException e) {
    cout << e.getMessage() << ", error code: "
         << e.getErrorCode() << endl;
  }
  return 0;
}
```

Exceptions and Inheritance (cont)

- Now, having BaseException by itself is useful.
- You can store messages and error codes in a convenient class and throw it!
- However you can also derive from it as well…
- Consider a class specifically designed for array index problems...

```
class ArrayIndexException : public BaseException
{
public:
  ArrayIndexException(string msg,int err,int index):
    BaseException(msg,err),badIndex(index){}
  string getMessage()
  { return "ARRAY INDEX EXCEPTION: " + msg; }
  int getBadIndex() { return badIndex; }
private:
  int badIndex;
};
```

Exceptions and Inheritance (cont)

- Now, `ArrayIndexException` can be caught with :

```
catch (ArrayIndexException e)  // copy of ArrayIndexException
catch (ArrayIndexException &e) // reference to ArrayIndex...
catch (ArrayIndexException)    // ArrayIndex... not accessible
catch (BaseException e)        // copy of BaseException in e
catch (BaseException &e)       // BaseException reference in e
catch (BaseException)          // BaseException not accessible
catch (…)                      // BaseException not accessible
```

```
int main()
{
  MyIntArray<5> anIntArray;
  for (int i=0; i<5; i++)   anIntArray[i] = i;
  try { cout << "anIntArray[6] = " << anIntArray[6] << endl; }
  catch(ArrayIndexException e) {
    cout << e.getMessage() << endl;
  }
}
```

Exceptions and Inheritance (cont)

- If, in general, you would like to catch ArrayIndexExceptions and BaseExceptions separately, you can catch instances of the derived exception first, followed by instances of the base exception.
- This might look like this:

```
try {
  someArbitraryFunction(someIntArray[someIndex]);
}
catch (ArrayIndexException e)
{
  // take appropriate action for an ArrayIndexException
  …
}
catch (BaseException e)
{
  // take appopriate action for a more general exception
…
}
```

Exceptions and Inheritance (cont)

- However, if your base exception class makes use of virtual member functions, you might only need to catch a *reference* to the base class.
- If you don't catch a reference, you get a *copy* of the originally thrown object which short circuits any virtual functions...

```
try {
  someArbitraryFunction(someIntArray[someIndex]);
}
catch (BaseException &e)
{
  // Print out an exception-specific message
  cout << e.getMessage() << endl;
}
```

Demonstration #1

Exceptions and Inheritance

A Little OOP Concept

- While using inheritance to neatly organize different exceptions has its advantages, there is at least one disadvantage.
- If we catch references to the base class all the time, how do we know exactly what *type* of exception was thrown?
- We could have many classes derived from `BaseException`:
  - `ArrayIndexException`
  - `MathException`
  - `FileException`
  - etc…
- These, in turn, might have more specific exception classes derived from them.
- When I catch BaseException, what do I have?
- You could implement another member function which returns an exception type code, something like this:

A Little OOP Concept (cont)

```
class BaseException
{
public:
  BaseException(string msg,int err=0):message(msg),
       errorCode(err){}
  virtual string getMessage()
        { return "BASE EXCEPTION: " + message; }
  virtual int getExceptionType() { return BASE_TYPE; }
  int getErrorCode() { return errorCode; }
protected:
  enum { BASE_TYPE, ARRAYINDEX_TYPE, MATH_TYPE, FILE_TYPE };
  string message;
  int errorCode;
};
```

- Naturally, we'd change `ArrayIndexException` as well...

A Little OOP Concept (cont)

```
class ArrayIndexException : public BaseException
{
public:
  ArrayIndexException(string msg,int err,int index):
    BaseException(msg,err),badIndex(index){}
  string getMessage()
  { return "ARRAY INDEX EXCEPTION: " + message; }
  int getExceptionType() { return ARRAYINDEX_TYPE; }
  string getBadIndex() { return badIndex; }
private:
  int badIndex;
}
```

- Now, we could have code take explicit action based on the exception type...

A Little OOP Concept (cont)

```
try {
  divide(someIntArray[34343],someIntArray[2222]);
}
catch(BaseException &e)
{
  cout << e.getMessage() << endl;
  switch( e.getExceptionType() )
  {
    case ARRAYINDEX_TYPE:     // some specific code here
                              break;
    case MATH_TYPE:                 // some specific code here
                              break;
    default:                  // default code here
                              break;
  }
}
```

- Hmmmmm….

A Little OOP Concept (cont)

- Doesn't having a constant which represents the type of exception defeat the purpose of inheritance?
- Yes and No.
- The practice of placing a constant indentifier within a class is a technique used when dealing with a concept called *composition*.
- It's usually pitted against inheritance as an either/or proposition:
  - Either you use inheritance and virtual functions to illicit different behavior from a "base class"
  - Or you use an identifier and have conditional code which performs different behavior depending on the value of the identifier (without using inheritance)
- There are many people who argue that inheritance is overused and that, most of the time, composition is all we need...
- I have found that mixing the two can be appropriate as well, depending on the task.
- Bottom line is "think carefully" before you go one way or the other.

Generic Exceptions

- C++ defines a standard exception class called `exception`.
- It's most interesting member function is a simple call to return an error message associated with the exception.
- That member function's name is `what()`.
- In Lecture 4 we talked about pointers for the first time, and how to allocate them.
- I mentioned then that there is a much better way to check for allocation failure than comparing the pointer to NULL.
- When `new` fails to allocate memory requested, it throws an exception class named `bad_alloc` which is derived from `exception`.
- Now that we know this, we actually *must* wrap calls to new in a try block to keep from having an unhandled exception halt our program!
- Consider the following mods to `MyString::growStorage()`

Generic Exceptions (cont)

```
bool MyString::growStorage(int spaceNeeded)
{
  if (spaceNeeded < allocatedSpace)
    return true;

  int padding = spaceNeeded % 32;
    spaceNeeded += (32 - padding);
  char *newStoragePtr = NULL;
  try {
    newStoragePtr = new char[spaceNeeded];
  }
  catch(exception e)
  {
    return false;
  }
  // rest of code here...
}
```

Generic Exceptions (cont)

- We can be reasonably sure that any exception thrown in that try block will be a bad allocation exception.
- But suppose our size parameter were actually an array element?
- We might have an invalid array index exception.
- Just to be sure, we should catch `bad_alloc`...

```
char *aPtr;
try {
  aPtr = new char[someIntArray[1234]];
}
catch (bad_alloc &e)
{
  // Bad allocation exception caught
  cout << e.what() << endl;
}
catch (…) {}
```

Demonstration #2

Catching Bad Allocations

Constructors and Destructors

- There's one small detail I've left out which is actually quite important.
- When an exception is thrown, the stack is "unwound" to the place at which the exception is caught.
- That means for every function call you "abort" due to the exception being thrown, the local variables of those functions are properly disposed of (that is, their destuctors are called).
- The same cannot be said for dynamically allocated memory or other resources not associated with any object.
- LNG gives an example of this in the form of a MacOS programming technique.
- Since I'm a Mac person myself, I thought "What a great example".
- Let's have a look…

Constructors and Destructors (cont)

- Consider the following code which performs a graphical operation in a MacOS environment.
- It needs to save the current GrafPtr, set the current GrafPtr to the window we want to draw in and then restore the original GrafPtr...

```
void drawInWindow(WindowPtr wPtr)
{
  GrafPtr savePort;
  GetPort(&savePort);   // save the old GrafPtr
  SetPort(wPtr)         // WindowPtr & GraPtr are the same

  Rect aRect = { 0,0,100,100 };
  PaintRect(wPtr,&aRect);   // Draw a rectangle

  SetPort(savePotr);
}
```

Constructors and Destructors (cont)

- The problem here is what happens if an exception occurs when performing the graphics operation?
- The final call to `SetPort` (which restores the original GrafPtr) is never made.
- This is bad!
- A workaround is to use a local object to do the same task:

```
class PortSaver {
public:
  PortSaver()  { GetPort(&savePort); }
  ~PortSaver() { SetPort(&savePort); }
private:
  GrafPtr savePort;
};
```

Constructors and Destructors (cont)

- Now, we can implement the same operation safely!
- You see, since we're saving the current port in the constructor and restoring it in the destructor of `SavePort`, any exception that occurs will call `SavePort::~SavePort()` and restore the original GrafPtr!

```
void drawInWindow(WindowPtr wPtr)
{
  PortSaver saveMe;
  SetPort(wPtr)         // WindowPtr & GraPtr are the same

  Rect aRect = { 0,0,100,100 };
  PaintRect(wPtr,&aRect);   // Draw a rectangle
}
```

Final Thoughts

- Assignment #9 is up!
- Remember, Prelim on 11/23
- We'll get to STL (Standard Template Library) next time.