Dynamic casting, error handling, exceptions

Review

```
void swap( int& a, int& b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

- How do we turn these code bits into a template function/class?

```
class ReadOnly
{
    public:
        Data( int v )
        {
            val = v;
        }
        int getVal()
        {
            return val;
        }
    private:
        int v;
};
```

STL review

- Write a simple program that uses the STL vector class:
  - Adds some random integers
  - Sorts them
    - `sort( iterator, iterator );`
  - Prints them all out using iterators

Pointer Problem

- Let's say we have an Animal*.
- We want a Shark*, where Shark is a class derived from Animal*.
- rampage() is a method defined only in Shark.
- Will this work?

```
Animal* a = (some random Animal ptr)
Shark* s = (Shark*)a;
Shark->rampage();
```

Fish/Shark/Boom

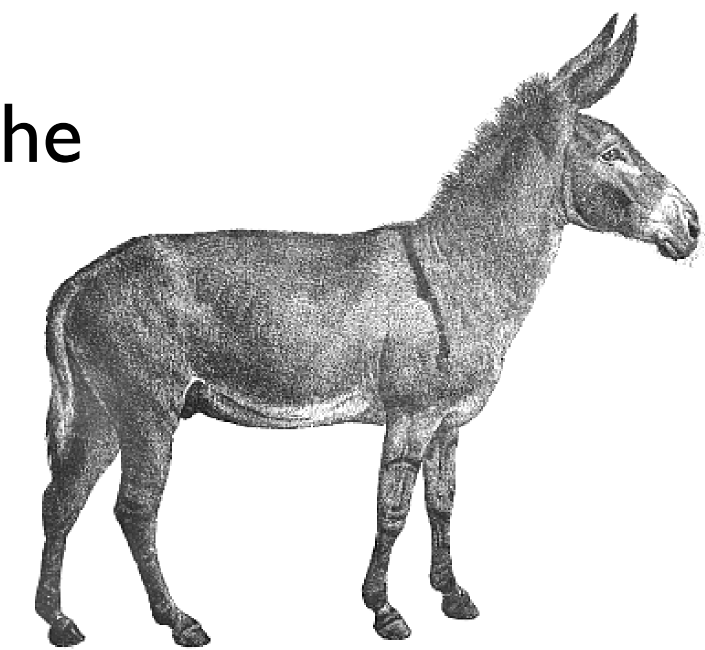
- Yes - but this will **only** work if the pointer is **actually** a Shark!
- This will cause Very Bad Things to happen:

```
Animal* a = new Fish();
Shark* s = (Shark*)a;
s->rampage();
```

- a is **not** a Shark, so there is no rampage method in a! ... *Boom*.

Casting and Type Errors

- This is a *type error* : we're trying to turn a pointer into something it's not
- The C casting operator lets you do this, which is why its use is not encouraged with classes
- Instead, we have something new: the **dynamic_cast** operator thingy



Introducing: **dynamic_cast**

- dynamic_cast attempts to convert the parameter (a) into the requested type (Shark*)
- If successful it returns a valid pointer
- If *not*, it returns NULL!

```
Animal* a = new Fish();
Shark* s = dynamic_cast<Shark*>(a);
if( s )
   s->rampage();
```

Asserts

- C/C++ includes a function called **assert**(), which is widely used in debugging
- **assert** is called with a condition: we want the condition to be true
- If the condition is true, assert() does nothing; if the condition is false, assert() prints a message and ends the program

- Here’s an example:
- We want to make sure a pointer is not NULL
- While debugging, we use assert; if the pointer is NULL when assert is called, the program will terminate with a helpful message
- very helpful, but for “real” programs you often want better debugging can this!

```
// get the first node in the list
Node* ptr = list.getFirstNode();

// this should always return a valid ptr
assert( ptr != NULL );
```

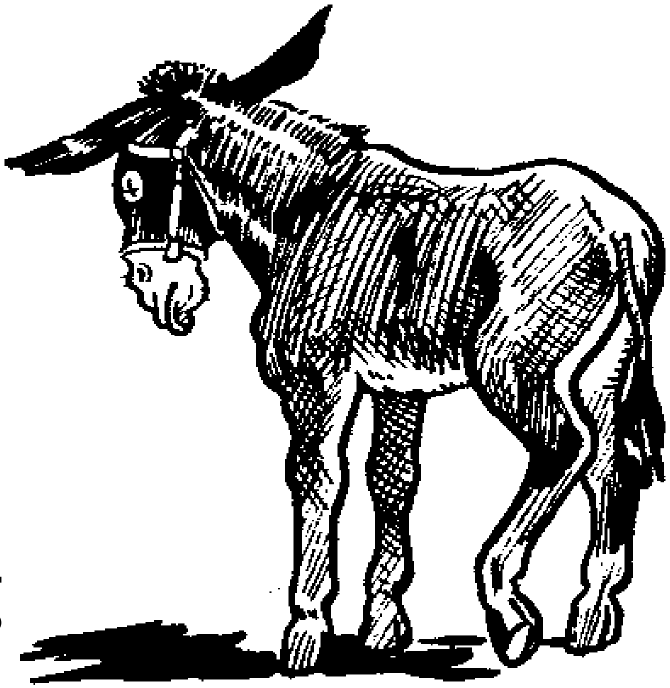
Error Handling

- With simple programs, we assume everything is going to work... but programs sometimes have errors!

Example:

```
deleteFile( “c:\\temp.txt” );
```

- the file might not exist
- It might not be delete-able
- something else might go wrong



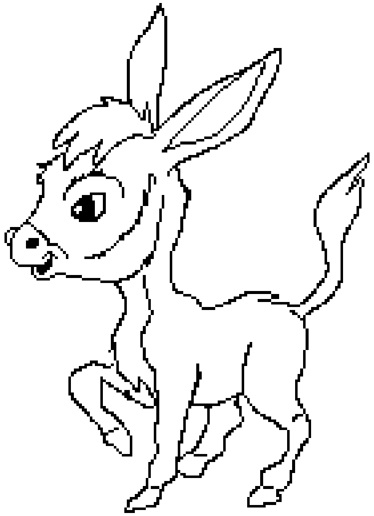
Return Codes

- By convention, C functions use return values to indicate success/failure (sometimes known as return codes)
- This can be a pain, because you may have to sometimes check for multiple different errors every time you call a function

```
int returnVal = deleteFile( “c:\\temp.txt” );
if( returnVal == ERR_FILE_DOES_NOT_EXIST )
    cout << “File Does Not Exist”;
else if( returnVal == ERR_FILE_NOT_WRITEABLE )
    cout << “File not writable!”;
// ... etc.
```

Introducing C++ Exceptions

- So... we can't ignore error checking and just assume everything is going to work
- But error-checking every single function is a pain
- C++ introduced an alternative mechanism, called **exceptions**



Exceptions

- Basic idea: you *try* to do something in C++, specifically the sorts of things that might fail
  - opening a file, requesting memory, etc.
- If that fails, your code *throws* an exception: a small object, an integer, etc.
- Your code *catches* that exception, and deals with it in an *exception handler*
- If nothing goes wrong, none of the error handing code gets called - the program proceeds normally and all handlers are ignored

Exception Structure

- We arrange code that uses exceptions in **try/catch** blocks:

```
try
{
   // Do something that could cause an error
   // throw an exception on error
}
catch( exception )
{
   // handle the error: print a
   // message, quit the program...
   // whatever.
}
```

Throwing Exceptions

- To throw an exception, you simply use the **throw** keyword:

```
throw 42;
```

or...

```
throw MadCow("moo!!");
```

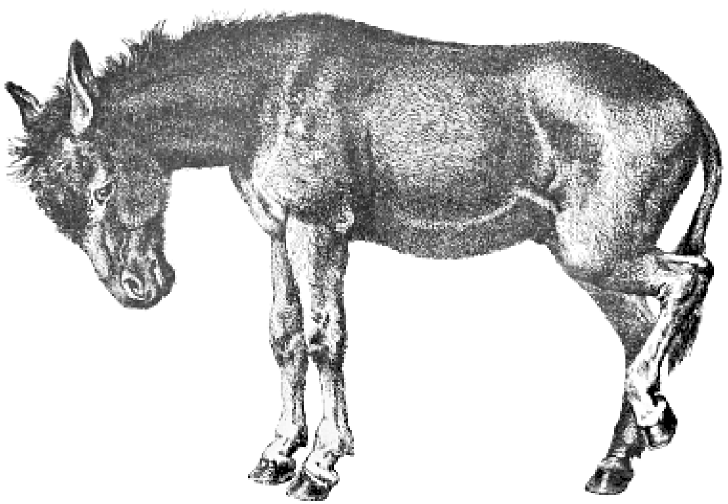
Try/Catch Blocks

- Every **try** block requires at least one **catch** (there can be more than one).
- Each **catch** block needs to accept a single parameter of a specific type:
- The appropriate *exception handler* will get called, depending on what kind of exception gets thrown

```
catch( int e )
{
    cout << "INT:" << e;
}
catch( MadCow e )
{
    cout << "MOO!" << e.moo();
}
```

Catch-all Block

- We can also define a *catch-all* exception handler: this will get called if none of the other exception handlers “match”
- There’s no parameter to the catch-all! (why not?)



```
catch( int e ) {}
catch( MadCow e ) {}

catch( ... ) // catchall handler
{
   cout << “default!” << endl;
}
```

Code Flow

```
int main()
{
    cout << "1";
    try
    {
        cout << "2";
        throw 42;
        cout << "3";
    }
    catch(...)
    {
        cout << "BOOM!";
    }
    cout << "4";

    return 0;
}
```

- After an exception is thrown and caught, execution picks up again *after the exception handler!*
- It does **not** start again after the throw statement
- What is the output of this program?

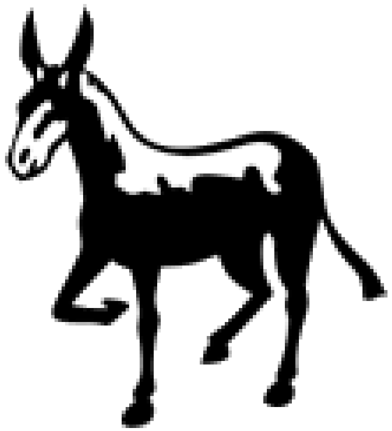
Nesting Exceptions

- You can have multiple levels of **try/catch** blocks (much like if/else statements)
- If an exception is thrown:
  - The first matching exception handler in the current level is called
  - If there isn't one, higher levels are tried
  - If no matching handler is found at any level, the program terminates
  - This is also what happens if you **throw** outside a try/catch block!

Example

```
cout << "1";
try
{
   cout << "2";
   try
   {
      cout << "3";
      throw 42.3f;
      cout << "4";
   }
   catch( int a )
   {
      cout << "boom one;
   }
   cout << "5";
}
catch( float f )
{
   cout << "boom two;
}
cout << "6";
```

- What is the output of this impressively dense chunk of code?
- Remember: after an exception has been handled, the next code to be executed is the code after the handler



What Do We Throw?

- Most any type (object, built-in, etc) can be thrown
- Often there will be a special exception class:
  - C++ has a standard base class for exceptions called **exception** that can be used as a base class

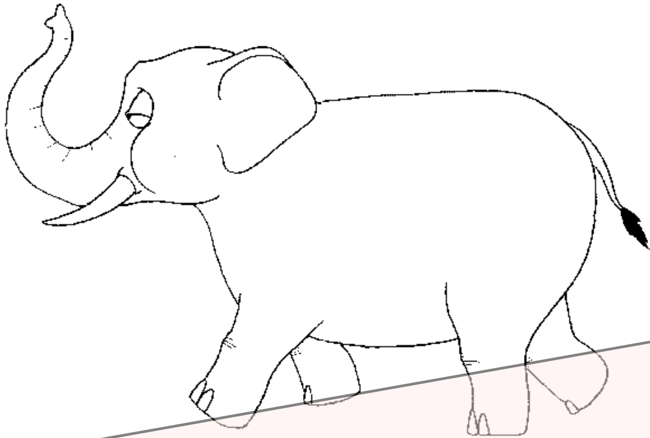
```
class myexception: public exception
{
  virtual const char* what() const
  {
    return "My exception happened";
  }
}
```

Putting This Into Context

- Earlier we used the (fictional) **deleteFile** function as an example:

```
int returnVal = deleteFile( “c:\\temp.txt” );
if( returnVal == ERR_FILE_DOES_NOT_EXIST )
    cout << “File Does Not Exist”;
else if( returnVal == ERR_FILE_NOT_WRITEABLE )
    cout << “File not writable!”;
// ... etc.
```

- If we rewrite this to use exceptions, we can make the code cleaner to read



```
try
{
    deleteFile( “c:\\temp.txt” );
}
catch( exception& e )
{
    cout << “Delete Error: “
         << e.what() << endl;
}
```

```
// in deleteFile
...
if( somethingWrong )
{
   FileException fs;
   throw fs;
}
```

- Since we’re catching a *reference* to an **exception**, we can catch derived classes too (such as FileException)
- Also note that exceptions can be thrown by functions (aka code *outside* of this function)

Exceptions Philosophy

```
try
{
    One();
    Two();
    Three();
    Four();
    Five();
}
catch( ... )
{
    cout << "err";
}
```

- There's disagreement on how widely exceptions should be used...
- ... they sometimes make it hard to tell whether code will be executed
- Can you tell whether Two() will be executed just by looking?
- Three()? Four()?

Goodly Exceptions

- When using exceptions:
  - Use them for *exceptional* circumstances -
  - don't have your code *depend* on them!
    - one reason: exceptions are expensive
  - try to structure your code so that exceptions are only used when needed
    - helps keep things readable