Dynamic casting, error handling, exceptions, namespaces

Review

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

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

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

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

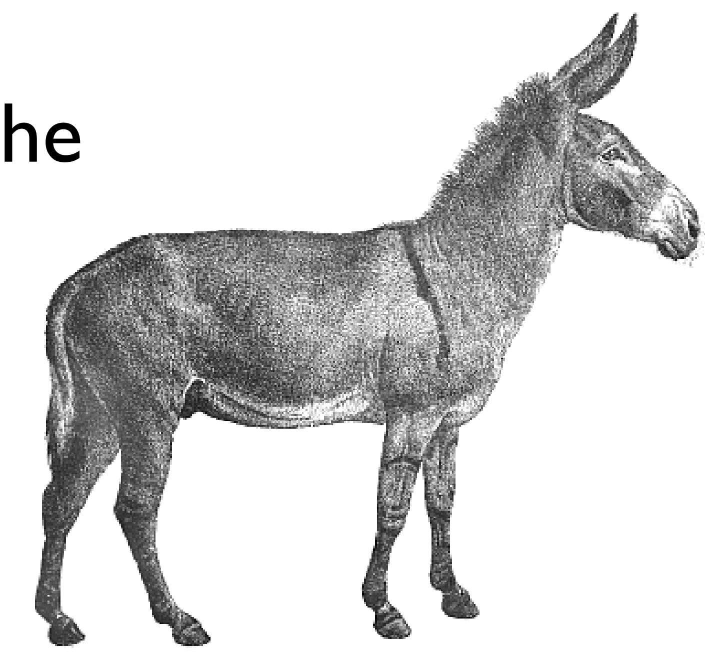
- Yes - but this will **only** work if the pointer is **actually** a Shark!
- This will cause Very Bad Things to happen:

```
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 );
```

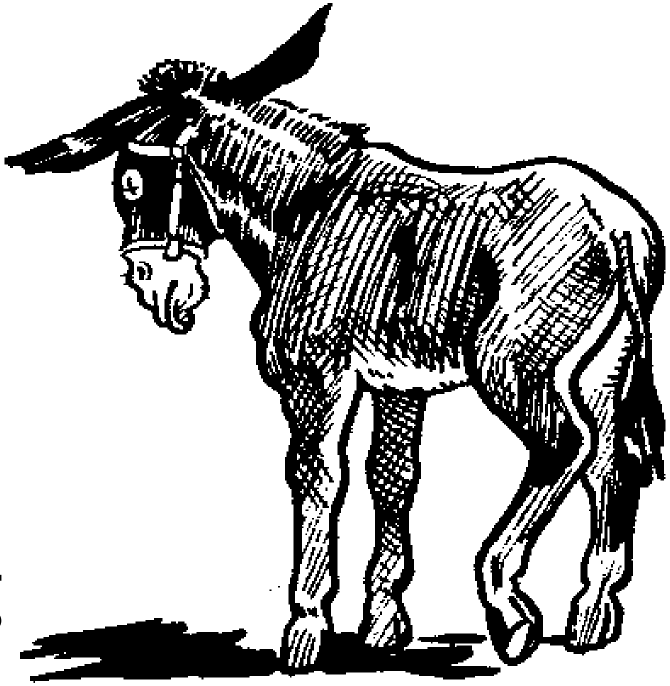
Error Handling

- 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



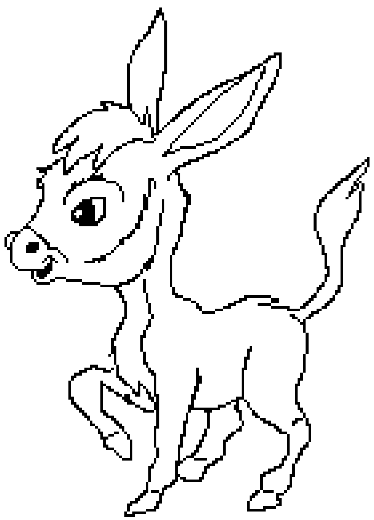
Return Codes

- 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!!");
```

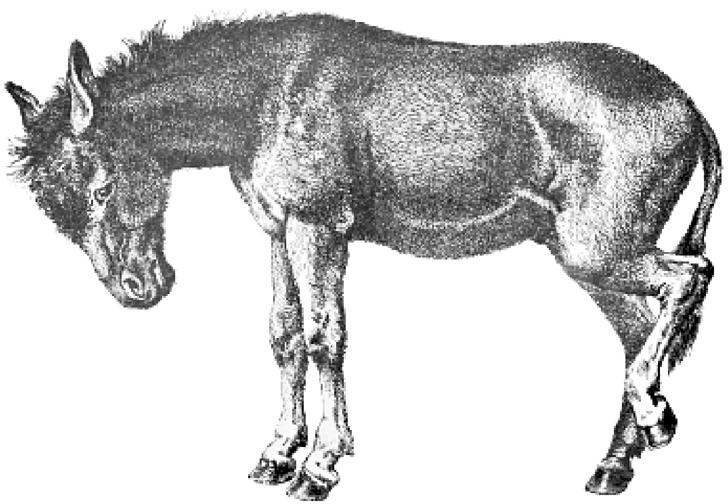
Try/Catch Blocks

- 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?

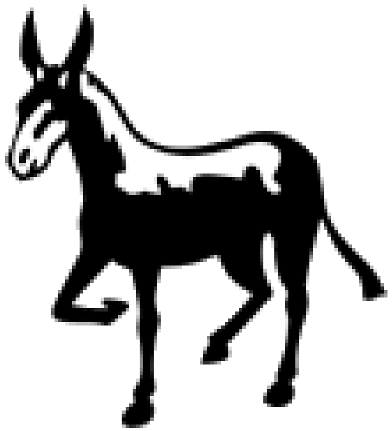
Nesting Exceptions

- 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!

```
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";
```

Example

- 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

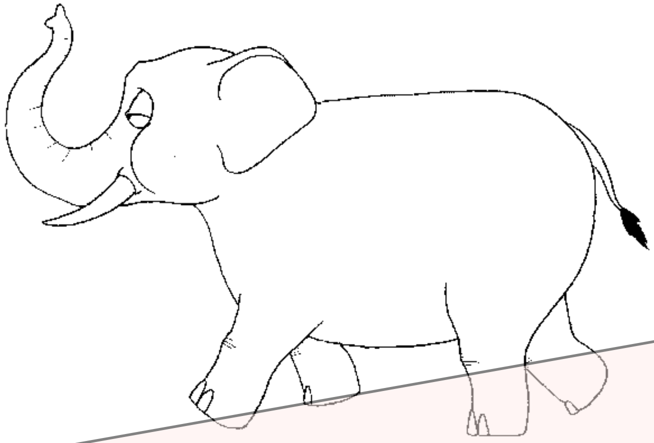
```
class myexception: public exception
{
  virtual const char* what() const
  {
    return "My exception happened";
  }
}
```

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

Problem...

vendorone.h

```
class Data
{
   // contents ignored
};
```

vendortoo.h

```
class Data
{
   // contents ignored
};
```

code.cpp

```
#include "vendorone.h"
#include "vendortoo.h"

int main()
{
    return 0;
}
```

- This code is problematic.
- What kind(s) of error(s) is it going to cause?
- What is the scope of these classes?

Namespaces

- **Namespaces** are a new(ish) C++ feature designed to solve this problem by "grouping" symbols so they don't clash with each other
- We've been using this all semester: we can use `cout` two different ways:
  - ```
    using namespace std;
    cout << "hello";
    ```
  - `std::cout << "hello" << endl;`
    ```

Applying Namespaces

vendorone.h

```
namespace VendorOne
{
  class Data
  {
    // contents ignored
  };
}
```

vendortoo.h

```
namespace VendorToo
{
  class Data
  {
    // contents ignored
  };
}
```

- A namespace introduces a scope...
- Neither **Data** class is now in the global scope, so they can both co-exist happily
- Their names are now **VendorOne::Data** and **VendorToo:Data**

Declaring Namespaces

- Namespaces can be declared more than once
- The contents of a namespace are the *accumulation* of all the declarations the compiler has seen so far

```
namespace bob
{
   int x;
   int y;
}

// at this point bob
// contains x and y

namespace bob
{
   int z;
}

// at this point bob
// contains x, y, and z
```

Using Namespace’d Items

- We have a few options to use the Data class:

```
#include “VendorOne.h”

int main()
{
   VendorOne::Data bob;
}
```

```
#include “VendorOne.h”
using namespace VendorOne;

int main()
{
   Data bob;
}
```

- The **using** keyword makes everything from that namespace available in the global scope
- What happens if we do this...

```
using namespace VendorOne
using namespace VendorToo;
```

Conflicting Namespaces

- This isn't a problem unless we try and *use* the Data class
- If we do, *then* there's a conflict!

```
#include "VendorOne.h"
#include "VendorToo.h"
using namespace VendorOne;
using namespace VendorToo;

int main()
{
   Data bob;
}
```

error

- How do you fix this?

The **using** keyword

- As we've seen, the **using** keyword can make an entire namespace available for us
- It can also make *individual pieces* of a namespace available

```
#include "VendorOne.h"
using VendorOne::Data;

int main()
{
   Data bob;
}
```

- This makes *only* the Data class from VendorOne available - not anything else in that namespace
- You can use this to import functions, too, and even class member functions

Unnamed Namespaces

```
#include <iostream>
using namespace std;

namespace
{
    int x;
}

namespace
{
    int q;
}

int main()
{
    q = 0;
    cout << ::x;
    return 0;
}
```

- A namespace without a name is (duh) called an **unnamed namespace**
- Elements in an unnamed namespace can be accessed with or without the scope resolution operator
- Internally, an unnamed namespace has a privately generated name
- Can't clash with unnamed namespaces from other source files!

A new thing...

- We often find ourselves doing stuff like this:

```
int bob;

if( someConditionIsTrue )
    bob = 17;
else
    bob = 96;
```

- ... where we just want to execute a single statement based on the outcome of some condition (here, setting a value).

A Shortcut:

- C++ provides us a nifty shortcut to do this sort of thing:
- The ternary operator!
  - (what does ternary mean?)

An Example

This unwieldy piece of code:

```
int bob;

if( someCondition )
    bob = 17;
else
    bob = 96;
```

can be reduced to this:

```
int bob = someCondition ? 17 : 96;
```

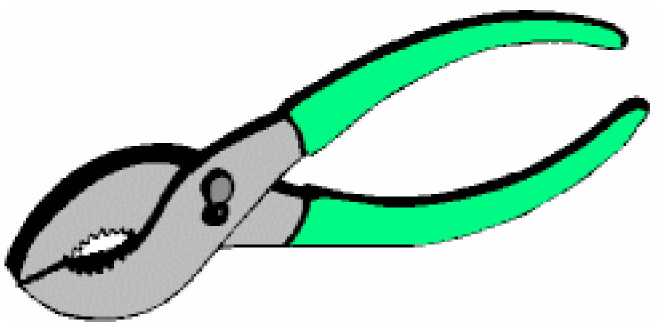
Anatomy of the Ternary Operator

```
condition ? truePart : falsePart
```

this would go in the if statement

the *single statement* that gets executed if **condition** is true

the *single statement* that gets executed if **condition** is false



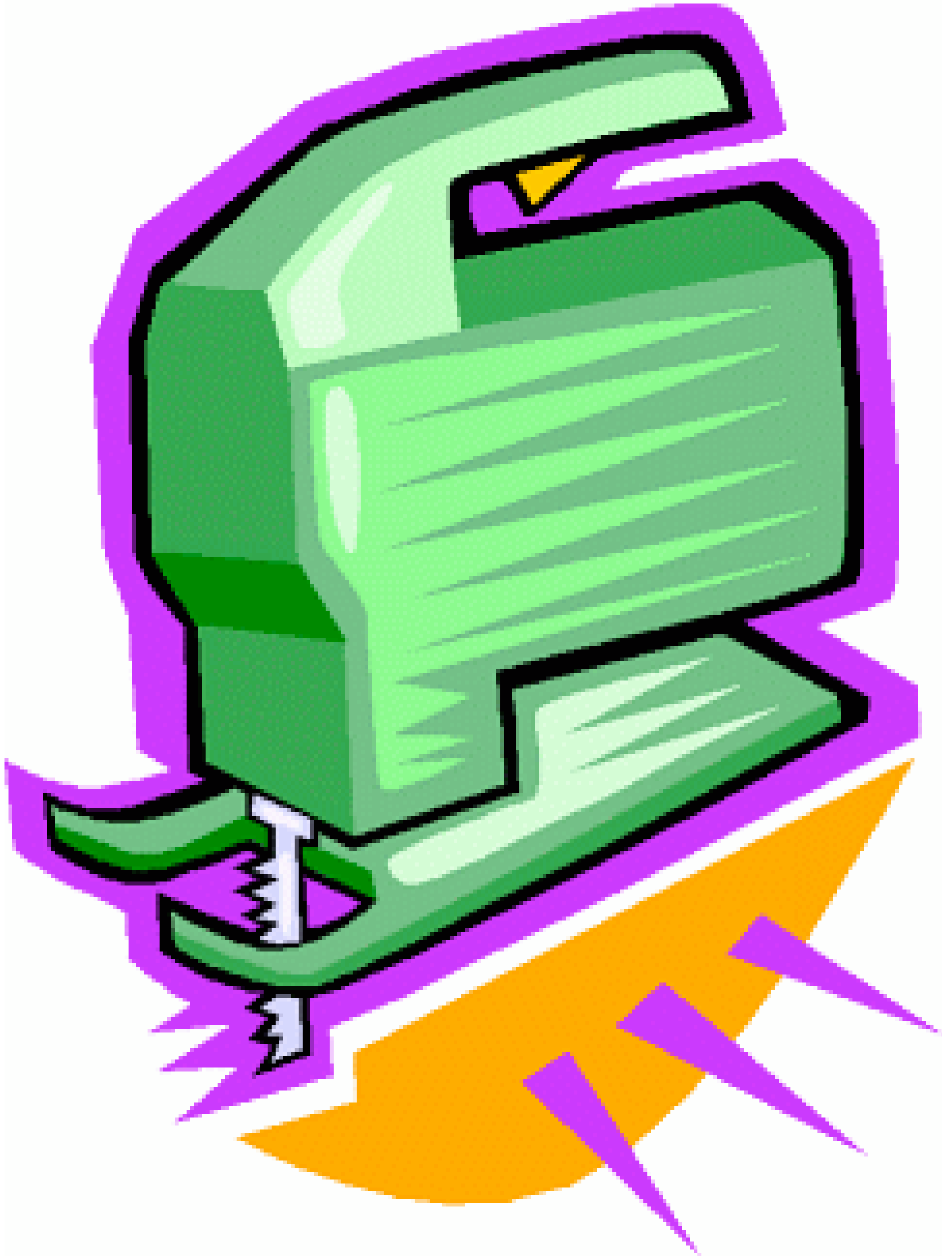
Usages

- What is this good for?
- Shortening code

```
int max( int a, int b )
{
    return a > b ? a : b;
}
```

- Assigning const values conditionally

```
bool correct = getValue();
const int PI = correct ? 3.14 : 92.8;
```



Question

- Hopefully you should know the answer to this by now...
- Why might the ternary operator not always be a good idea?

Bad Code!

- On the other end of the conditional execution scale:
- When you are testing a single value against a lot of conditions, you get a lot of hard-to-read code
- Like this!

```
int input = getInput();

if( input == 0 )
    doStuff();
else if( input == 1 )
    doSomethingElse();
else if( input == 2 )
    doAThirdThing();
else if( input == 3 )
    playSpades();
else if( input == 4 )
    watchScrubs();
else if( input == 5 )
    goBirdWatching();
else if( input == 6 )
    eatHamburger();
```

the switch statement

- The switch statement is often a more elegant, sometimes faster way to do this
- **switch** tests a single *integer* variable against a large number of conditions
- Here we're checking input against 0 - 6

```
int input = getInput();

switch( input )
{
    case 0: doStuff();
            break;
    case 1: doSomethingElse();
            break;
    case 2: doAThirdThing();
            break;
    case 3: playSpades();
            break;
    case 4: watchScrubs();
            break;
    case 5: goBirdWatching();
            break;
    case 6: eatHamburger();
            break;
}
```

ARROWED!!!

this can be any *integer* expression - in parenthesis, just like an if statement

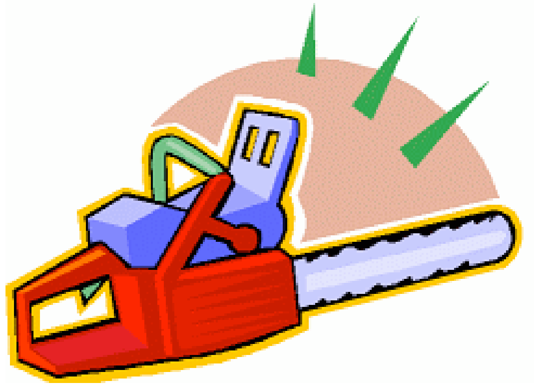
```
int input = getInput();

switch( input )
{
    case 0: doStuff();
            break;
    case 1: doSomethingElse();
            break;
    case 2: doAThirdThing();
            break;
    case 3: playSpades();
            break;
    case 4: watchScrubs();
            break;
    case 5: goBirdWatching();
            break;
    case 6: eatHamburger();
            break;
}
```

switch keyword

case statement: must be unique!

entire switch statement enclosed in curly braces



Case Statements

- When the input value is equal to a *case value*, everything until the next **break** is executed
- Even code in other case statements!
  - this is called falling through
- Any code that can go in a function can go in a case statement

```
char grade = getGrade();

switch( grade )
{
    case 'A': callMom();
              cout << "yay!";
              postOnFridge();
              break;

    case 'D': sigh();

    case 'F': grumble();
              cout << "boo.";
              studyHarder();
              break;
}
```

Default Statements

- Code in the **default** statement is executed if none of the case statements are true
- There can be only one of these per switch statement

```
char grade = getGrade();

switch( grade )
{
    case 'A': callMom();
              cout << "yay!";
              postOnFridge();
              break;

    case 'D': sigh();

    case 'F': grumble();
              cout << "boo.";
              studyHarder();
              break;

    default:  cout << "meh.";
              eatHamburger();
              break;
}
```

A Random Note About C++ Conditionals

```
bool one()
{
   cout << "one()" << endl;
   return false;
}

bool two()
{
   cout << "two()" << endl;
   return false;
}

int main()
{
    if( one() && two() )
         cout << "true" << endl;
    return 0;
}
```

What is the output of this program?

Minimal Evaluation

- C++ uses a strategy called *minimal evaluation* or *short circuit evaluation* to avoid doing unnecessary work
- This comes into play with the && operator, which is evaluated left-to-right:

(returns false)

```
if( one() && two() )
    cout << "true" << endl;
```

Minimal Evaluation

- Keep minimal evaluation in mind when writing conditional expressions
- This can actually be really handy!

```
if( ptr && ptr->value == 42 )
{
    // do stuff
}
```

- Here, we won’t access ptr->value unless ptr is non-null