Welcome

Object Oriented Programming with C++
CIS 265

Week 9 – Polymorphism

Christopher K. Burns

Schedule

Week		Content
1	1/11	*Review* Chapter 1 Intro to Computers and C++ Programming Chapter 2 Control Structures Chapter 3 Functions Chapter 4 Arrays
2	1/18	Additional Array and Function Topics Chapter 5 Pointers and Strings Lab 1 – Functions, Arrays, and Strings Homework 1 Assigned
3	1/25	Chapter 6 Classes and Data Abstraction
4	2/1	Chapter 7 Classes: Part I Lab 2 – Classes 1 Homework 1 Due
5	2/8	***MID-TERM EXAMINATION*** *(Chapters 1 through 7*)* **only portions of chapter 7 that were covered in class* *Final Project Assigned*
6	2/15	Chapter 7 Classes: Part II Chapter 8 Operator Overloading Homework 2 Assigned
7	2/22	Chapter 8 Operator Overloading Homework 2 Due Lab 3 – Classes 2
8	3/1	Chapter 9 Inheritance: Part I Homework 3 Assigned
9	3/8	Chapter 9 Inheritance: Part II Chapter 10 Polymorphism Homework 3 Due
10	3/15	Chapter 10 Polymorphism Lab 4 – Inheritance and Polymorphism
11	3/22	***FINAL PROJECTS DUE***

Agenda

Tonight's agenda

- Review
- Polymorphism
- Templates
- Streams
- Exception Handling

Object Oriented Programming

Home Work

Final Project!!!

Final Project

Final Project is due in two weeks!!!

Next week's class will be a short lecture, followed by lab time. During the lab time you can work on your project and can receive assistance and help from either myself or other students.

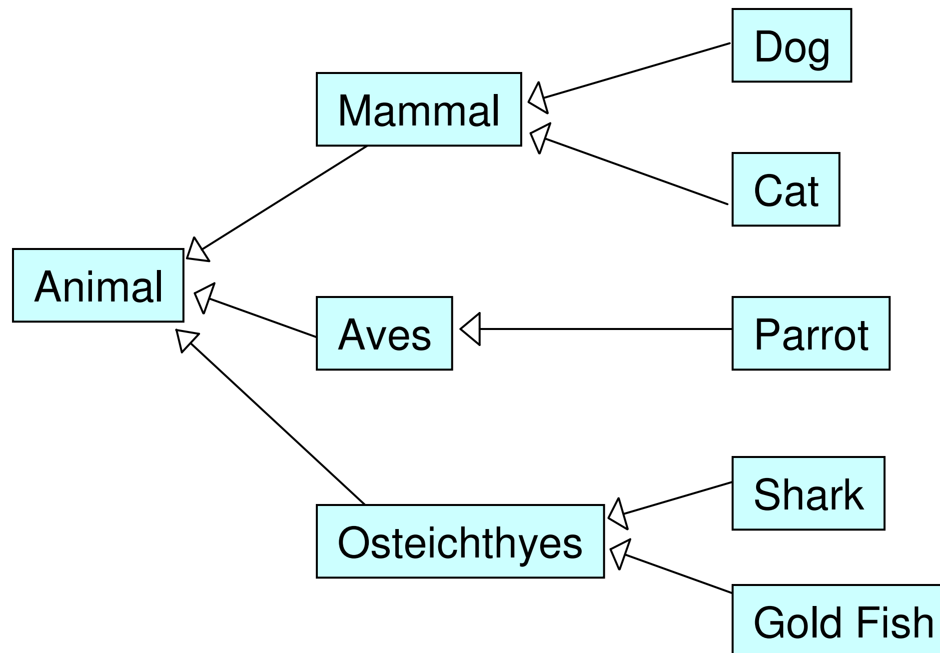
Classes - Review

What we've covered about classes up to this point:

- Basic structure of a class
- Constructors and overloading
- Separating Interface from Implementation
- Data encapsulation (get/set)
- Static members
- this pointer
- Operator Overloading
- Standard C++ Library – String and Vector
- Inheritance

Inheritance

A C++ class can inherit interface and methods from another C++ class.



Virtual Methods

Virtual Methods differ from static (standard) methods in that they are dynamically bound to the object.

If a function may be overridden by a derived class it should be defined as virtual. We will see why when we begin discussing polymorphism.

```
class Animal
{
public:
  string name;
  virtual void produceSound() { };
};
```

Pure Virtual Methods

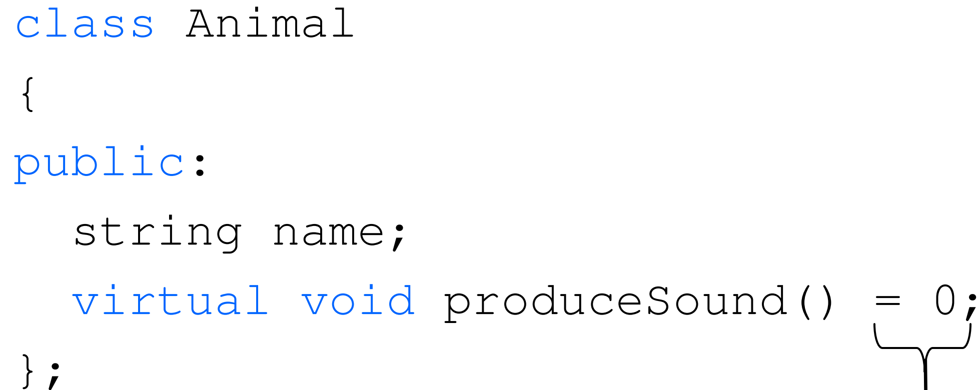
Often, it is known what the interface of the derived classes should look like, but the implementation cannot always be defined.

We can use pure virtual methods to define common functions that the derived classes must implement.

This would avoid problems as with the Cat class in the previous example.

Pure Virtual Methods

A Pure Virtual Method is defined by:



```
class Animal
{
public:
   string name;
   virtual void produceSound() = 0;
};
```

"= 0" means that there is no function defined. No class with a pure virtual function can be instantiated. Thus, before a derived class can be used, this function must be implemented.

Overriding

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

class Animal
{
public:
  string name;
  virtual void produceSound() = 0;
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound()
  {
    cout << "woof\n";
  }
};
```

```
class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
  void produceSound()
  {
    cout << “meow\n";
  }
};

void main()
{
  Dog dog;
  Cat cat;
  dog.produceSound();
  cat.produceSound();
}
```

Polymorphism

…but wait, in some of the earlier examples the dog class is derived from animal, and the cat class is also derived from animal…

…since they are both “animals”, can’t they be treated as such…?

Yes, this is called polymorphism

Polymorphism

Main Entry: **poly·mor·phism**

Pronunciation: "pä-lE-'mor-"fi-z&m

Function: *noun*

: the quality or state of being able to assume different forms: as **a :** existence of a species in several forms independent of the variations of sex **b :** the property of crystallizing in two or more forms with distinct structure

- **poly·mor·phic** /-fik/ *adjective*
- **poly·mor·phi·cal·ly** /-fi-k(&-)lE/ *adverb*

from Merriam-Webster.com

Polymorphism

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

class Animal
{
public:
  string name;
  virtual void produceSound() = 0;
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound()
  {
    cout << "woof\n";
  }
};
```

```
class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
  void produceSound()
  {
    cout << "meow\n";
  }
};

void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    animals[i]->produceSound();
  }
}
```

Polymorphism

The “is-a” relationship allows derived classes to be treated like their base class.

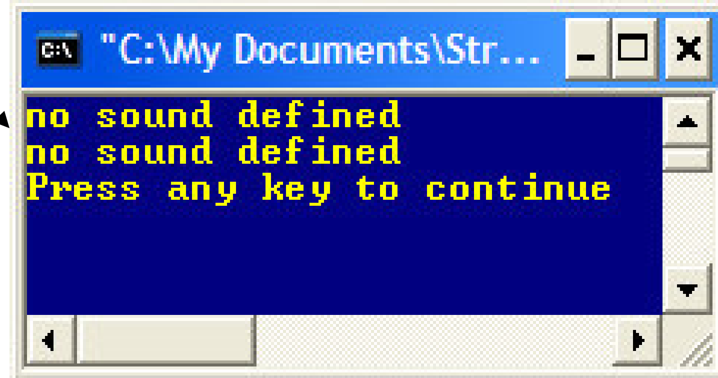
Dog is-a Animal

Cat is-a Animal

Polymorphism

```
class Animal
{
public:
  string name;
  void produceSound() { cout << "no sound defined\n"; }
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound() { cout << "woof\n"; }
};

class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
};

void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    animals[i]->produceSound();
  }
}
```

What happened???



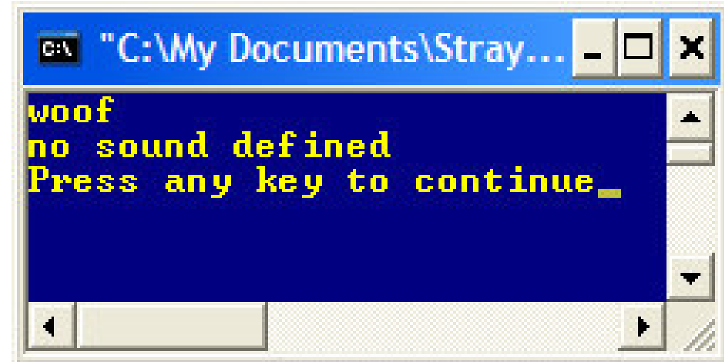
Polymorphism

If you are going to override a function of a base class, and use polymorphism, you must make the function virtual.

When a call is made to a virtual function, the function is looked up dynamically in a virtual table (or vtable). This adds a few clock cycles to the call, but solves our overriding problem.

Polymorphism

```
class Animal
{
public:
  string name;
  virtual void produceSound() { cout << "no sound defined\n"; }
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound() { cout << "woof\n"; }
};

class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
};

void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    animals[i]->produceSound();
  }
}
```



Polymorphism

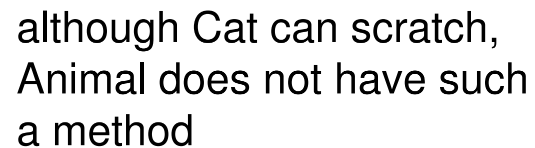
When objects of type Cat and Dog are both contained in an Animal collection, they have only the characteristics of Animal.

If Cat or Dog have members in addition to those of Animal, they are not directly visible.

Polymorphism

If a function exists in a derived class, but not base:

```
class Animal
{
public:
  string name;
  virtual void produceSound() = 0;
};

class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
  void produceSound() { cout << "meow\n"; }
  void scratch() { cout << "scratch\n"; }
};

void main()
{
  Animal* animal;
  animal = new Cat();
  animal->scratch();
}
```



Polymorphism

How can I tell what type a derived object is when it is part of a collection of an abstract base type?

(How can you tell if it is a Cat or a Dog in the Animal array?)

Run Time Type Information

C++ includes a special operator called the “typeid”.

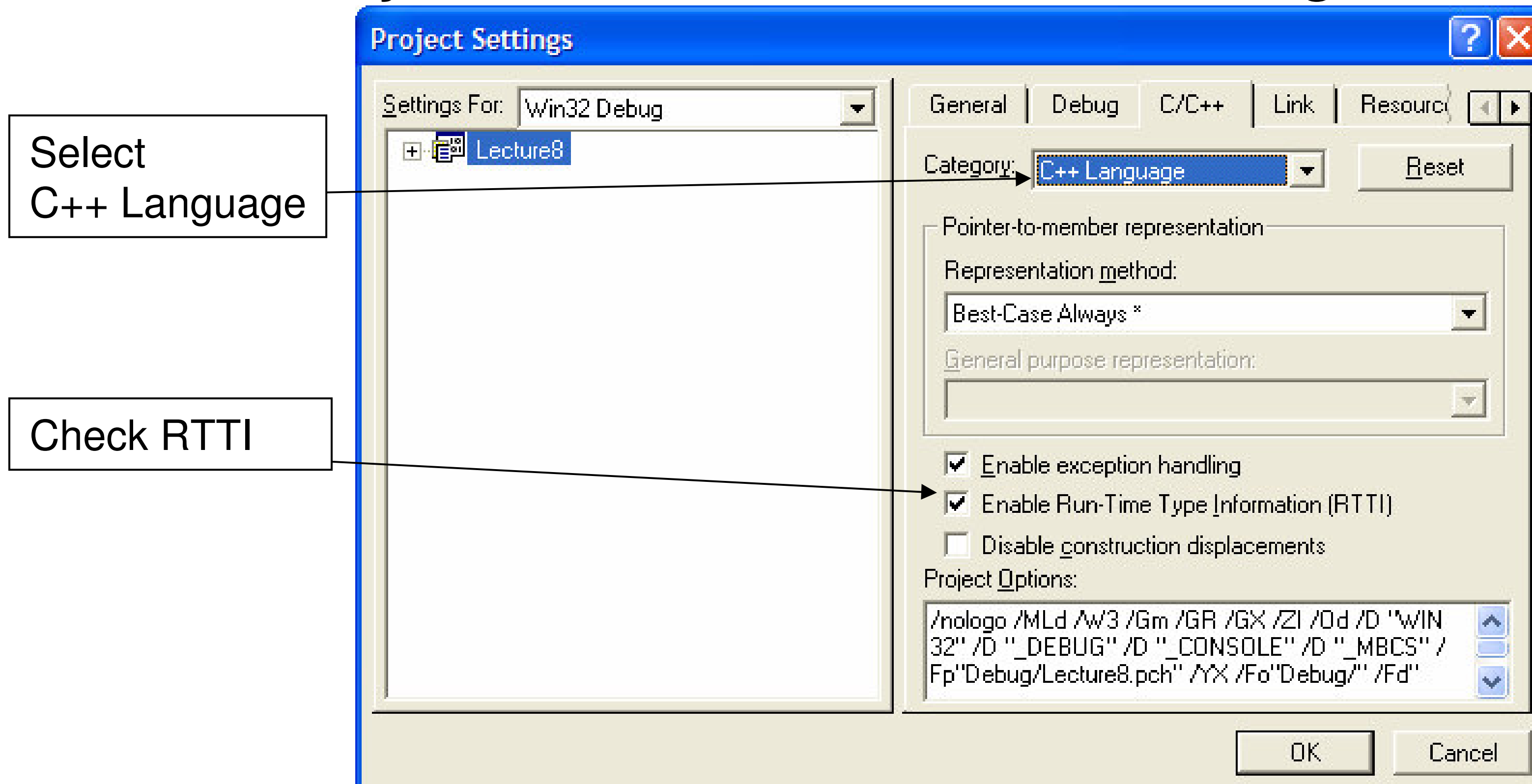
You can only use typeid if your code is compiled with run time type checking

Run Time Type Information

To enable run time type information (RTTI) go to the Project menu and choose settings:

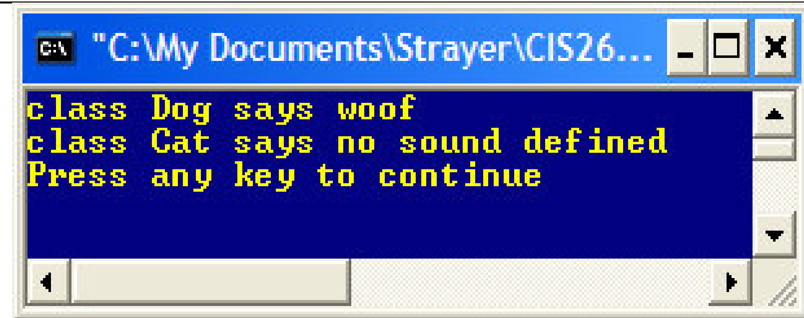
Select C++ Language

Check RTTI



Run Time Type Information

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

class Animal
{
public:
  string name;
  virtual void produceSound() { cout << "no sound defined\n"; }
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound() { cout << "woof\n"; }
};

class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
};

void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    cout << typeid( *animals[i] ).name() << " says ";
    animals[i]->produceSound();
  }
}
```



Polymorphism

```
void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    cout << typeid( *animals[i] ).name() << " says ";
    animals[i]->produceSound();
    if ( typeid( *animals[i] ) == typeid( Cat ) )
    {
      dynamic_cast< Cat* >(animals[i])->scratch();
    }
  }
}
```

Polymorphism

typeid – a special operator that returns a type_info class:

```
class type_info {
public:
   virtual ~type_info();
   int operator==(const type_info& rhs) const;
   int operator!=(const type_info& rhs) const;
   int before(const type_info& rhs) const;
   const char* name() const;
   const char* raw_name() const;
private:
   ...
};
```

Casting operators in C++

dynamic_cast – Used for conversion of polymorphic types.

static_cast – Used for conversion of nonpolymorphic types.

const_cast – Used to remove the const, volatile, and __unaligned attributes.

reinterpret_cast – Used for simple reinterpretation of bits.

Going Beyond

Tonight we have finished the required topics for CIS 265.

Before we finish the lecture portion of this class, there are a few C++ topics that must be mentioned for completeness.

We will cover some of these tonight, and a few other topics next week.

Templates

A quick treatment of templates:

The purpose of a template, is to create a bit of code that is generic for a given data type (a.k.a. generics in Java or C#)

Templates are considered to be one of the most powerful features of the C++ language.

Templates

Writing Generic Classes

A Template Class is similar to a regular C++ class, but it is able to take a “data type” parameter.

```
SomeClass< data type? >
```

Templates

Typically storage classes benefit the most from Template programming.

A Stack template class:

```
Stack < int >
Stack < string >
Stack < MyClass >
```

Templates

When you create a template class, you need to put the keyword “template” before the class definition:

```
template< typename SomeType >
class Stack
{
  SomeType array[ 100 ];
}
```

Templates

Here is our template class:

```
template< typename SomeType >
class Stack
{
  SomeType array[ 100 ];
}
```

Here is how our template is compiled if it is used this way

```
Stack< int > aStack
…
class Stack
{
  int array[ 100 ];
}
```

Templates

Our template can be used multiple times for the same or different types in a program:

```
template< typename SomeType >
class Stack
{
   SomeType array[ 100 ];
}
```

Stack< string > aStack

```
…
class Stack
{
      string array[ 100 ];
}
```

Stack< int > aStack

```
…
class Stack
{
      int array[ 100 ];
}
```

Stack< char > aStack

```
…
class Stack
{
      char array[ 100 ];
}
```

Templates

Templates can also take parameters.

If we are really going create a stack template using an array as the internal data structure, it might be nice if the template has an idea how large the stack might become.

Templates Example

```
template < typename SomeType, int size >
class Stack
{
private:
  SomeType data[ size ];
  int index;
public:
  Stack() { index = 0; }
  void putItemInList( SomeType item ) {
    if ( index < size )
      data[ index++ ] = item;
  }
  SomeType popItemFromList() {
    if ( index > 0 ) {
      return data[ --index ];
    }
    return data[0];
  }
  bool empty() {
    return( index == 0 );
  }
};
```

```
int main()
{
  Stack< int, 10 > myIntStack;
  Stack< string, 10 > myStringStack;

  myIntStack.putItemInList( 1 );
  myIntStack.putItemInList( 2 );
  myIntStack.putItemInList( 3 );
  while ( myIntStack.empty() == false )
    cout << myIntStack.popItemFromList()
         << endl;

  myStringStack.putItemInList( "Once" );
  myStringStack.putItemInList( "there" );
  myStringStack.putItemInList( "was" );
  while ( myStringStack.empty() == false )
    cout << myStringStack.popItemFromList()
         << endl;

  return 0;
}
```

Streams

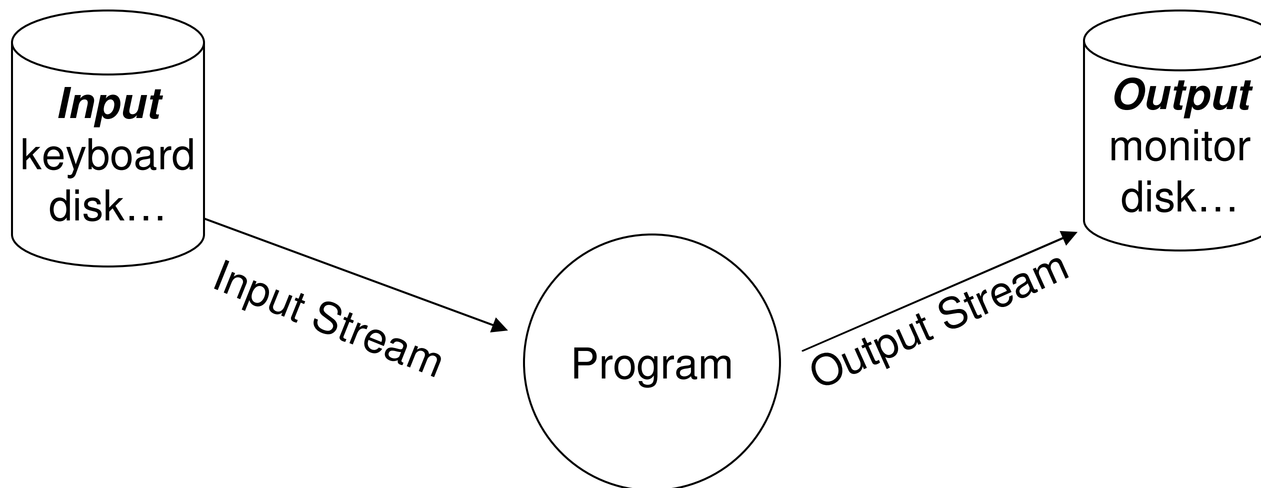
A quick treatment of streams:

You've already been using streams (cin, cout, overloading the << and >> operators…)

Now let's give them a more detailed look

Streams

A C++ stream is an input or output flow of data:



Streams – Console I/O Example

```
#include <iostream>
#include <string>
using namespace std;
const int MAX_STRING = 10;

int main() {
  cout << "Please enter some characters:" << endl;

  // let's take a look at the cin buffer without affecting it
  char input = cin.peek();

  if ( input > '0' && input < '9' ) {
    string input;
    cin >> input;
    int i = atoi(input.c_str());
    cout << "Number: " << i << endl;
  }
  else {
    char input[ MAX_STRING ];
    cin.get( input, MAX_STRING );
    cout << "Text: " << input << endl;
  }
  return 0;
}
```

Streams – Console I/O

C++ *cin* and *cout* are portable

That is, they support very basic console input and output functionality.

Some functions such as "move cursor", "clear screen", "get a single character *without* pressing enter" cannot be done with *cin/cout*.

- The reason is these are OS/Hardware dependent, thus not portable (console I/O on Linux differs from DOS)
- You need to use libraries that work with the OS to perform these function

Streams – Simple File I/O Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
  // file output
  ofstream myOutputFile( "file.test" );
  myOutputFile << "Hello, this a file" << endl;
  myOutputFile.close();

  // file input
  ifstream myInputFile( "file.test" );
  cout << "my input file contains:\n" << myInputFile.rdbuf();
  myInputFile.close();
  return 0;
}
```

Exceptions

An exception is an that occurs while your program is running.

There are two ways to deal with runtime errors in your program:

- Avoid errors
- Handle errors

Exceptions

Let's look at a divide by zero scenario

Avoiding the error

```
float divide( float a, float b )
{
  if ( b != 0 )
    return ( a / b );
  else
    return 0;
}
```

Exceptions

Let's look at a divide by zero scenario

Handle an error

```
float divide( float a, float b )
 {
   try
   {
     return ( a / b );
   }
   catch( ... )
   {
     return 0;
   }
 }
```

Exceptions

Handling Exceptions:

```
try  // to do something
{
  // something
}
catch( ... )  // any errors here
{
  // an error happened
}
```

Exceptions

Handling Specific Exceptions:

```
try
{
   return ( static_cast<float>(a) /
            static_cast<float>(b) );
}
catch( bad_cast &e )
{
   cout << "Cast Error!" << endl;
}
catch ( int i )
{
   cout << "Error #" << i << endl;
}
catch ( ... )
{
   cout << "Error!" << endl;
}
```

More C++

Topics for next week:

Making a “real” program – user input

Writing a program that could actually be used by a user.

Object Oriented Programming Methodologies

(aka putting it all together)

Suggestions?