CIS 190: C/C++ Programming

Lecture 12

Bits and Pieces of C++

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Passing by Value

- the “default” way to pass variables to functions

```
// function prototype
void PrintVal (int x);

int x = 5;
int *xPtr = &x;
PrintVal(x);      // function call
PrintVal(*xPtr); // also valid call
```

Passing by Reference

- uses pointers, and only other alternative in C
  - uses * to dereference, and & to get address

```
void ChangeVal(int *x); //prototype

int x = 5;
int *xPtr = &x;
ChangeVal(&x);    // function call
ChangeVal(xPtr); // also valid call
```

Passing a Reference

- uses references, and is available in C++
  - different from passing **by** reference

```
void ChangeVal(int &x); //prototype


int x = 5;
int *xPtr = &x;
ChangeVal(x);      //function call
ChangeVal(*xPtr); //also valid call
```

Passing a Reference

- uses references, and is available in C++
  - different from passing **by** reference

```
void ChangeVal(int &x); //prototype

int x = 5;
int &xRef = x;      //create reference
ChangeVal(x);       //function call
ChangeVal(xRef);    //also valid call
```

Pointers VS References

- we already know all about pointers… how are references different?

- references **must** be initialized at declaration
- references **cannot** be changed
- references can be treated as another “name” for a variable (no dereferencing)

Reference or Pointer?

- for the following applications, which is more appropriate: a reference, or a pointer?

- arguments in overloaded operators
- as part of a NODE definition
- a function that swaps two arguments
- dynamic memory allocation
- when the value needs to be NULL

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Error Handling

- common errors:
  - file not found/could not be opened
  - could not allocate memory
  - out-of-bounds on vector

- right now, we print out an error message and call **`exit()`**
  - handle the error right where it occurs

Handling Errors at Occurrence

- advantages:
  - easy to find because code is right there
- disadvantages:
  - error handling scattered throughout code
    - code duplication
    - code inconsistency (even worse!)
  - errors are handled however the original coder decided would be best

Two “Coders” with Classes

- class ***implementer***
  - creates the class definition
  - knows what constitutes an error
    - decides how to handle errors
- class ***user***
  - uses the class implementation
  - knows how they want to handle errors
    - (if handled internally, the class user may not even know an error occurred)

Example: Classy Trains

- how did we handle inappropriate/incorrect information for our trains?

Example: Classy Trains

- how did we handle inappropriate/incorrect information for our trains?
- why?

Example: Classy Trains

- how did we handle inappropriate/incorrect information for our trains?
- why?
- what if we were getting this information directly from a user instead of a file?

Example: Classy Trains

- what if we wanted this to be usable for both methods of inputting data?

- we need to separate ***error detection*** from ***error handling***

Example: Classy Trains

- what if we wanted this to be usable for both methods of inputting data?
- we need to separate ***error detection*** from ***error handling***
- implementer knows how to detect, and the user can decide how to handle

Exceptions

- ***exceptions*** are used to handle exceptional cases, or cases that shouldn't normally occur

- allow us to indicate an error has occurred without explicitly handling it
  - C++ uses these too, like when we try to use `.at()` to examine an out-of-bounds element

Try / Catch / Throw

- exceptions are implemented using the keywords try, catch, and throw

Try / Catch / Throw

- exceptions are implemented using the keywords try, catch, and throw

- the ***try*** keyword means we are going to try something, even though we are not sure it is going to perform correctly

Try / Catch / Throw

- exceptions are implemented using the keywords try, catch, and throw

- the ***throw*** keyword is used when we encounter an error, and means we are going to “throw” two things :
  - a value (explicit)
  - control flow (implicit)

Try / Catch / Throw

- exceptions are implemented using the keywords try, catch, and throw

- the ***catch*** keyword means we are going to try to catch at most **one** value
  - to catch different types of values, we need multiple catch statements

Exception Example

```
// inside SetCarID() function

  if (newID < MIN_ID_VAL ||
      newID > MAX_ID_VAL) {
    cerr << "ID invalid, no change";
  }
```

Exception Example

```
// inside SetCarID() function
try {
   if (newID < MIN_ID_VAL ||
       newID > MAX_ID_VAL) {
     cerr << "ID invalid, no change";
   }
}
catch () {


}
```

Exception Example

```
// inside SetCarID() function
try {
   if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
     throw(newID);
   }
}
catch () {


}
```

Exception Example

```
// inside SetCarID() function
try {
   if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
      throw(newID);
   }
}
catch (int ID) {


}
```

Exception Example

```
// inside SetCarID() function
try {
   if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
      throw(newID);
   }
}
catch (int ID) {
   cerr << "ID invalid, no change";
}
```

Using Catch

- the catch keyword requires:
  - one parameter
    - typename (int, exception, out_of_range, etc)
    - name (newID, e, oor, etc.) [optional]
- to catch multiple types of exceptions, you need to use multiple ***catch blocks***

Using Catch

- you **`can`** throw from inside a catch block, but this should be done sparingly and only after careful consideration
  - most of the time, a nested try-catch means you should re-evaluate your program design

- uncaught exceptions will cause the **`terminate()`** function to be called

Using Catch

- catch blocks are run in order, so exceptions should be caught in order from most specific to least specific

- to catch all possible exceptions, use:

  `catch(...)`

- literally use three periods as a parameter

Throwing Out of a Function

- we can throw exceptions without try/catch
  - most commonly done within functions

- requires that we list possible exception types in the function prototype and definition
  - called a ***throw list***

Throw List Example: Inside

```
void SetCarID(int newID) throw (int) {
   if (newID < MIN_ID_VAL ||
       newID > MAX_ID_VAL) {
     throw(newID);
   }
   else {
     m_carID = newID;
   }
}
```

Throw List Example: Inside

```
void SetCarID(int newID) throw (int) {
   if (newID < MIN_ID_VAL ||
       newID > MAX_ID_VAL) {
     throw(newID);
   }
   else {
     m_carID = newID;
   }
}
```

Throw List Example: Inside

```
void SetCarID(int newID) throw (int) {
   if (newID < MIN_ID_VAL ||
       newID > MAX_ID_VAL) {
      throw(newID);
   }
   else {
      m_carID = newID;
   }
}
```

this function might throw an integer

Throw List Example: Outside v0

```
// inside main()

    train[0].SetCarID(-1);
```

- what will happen if we run this code?

Throw List Example: Outside v0

```
// inside main()


    train[0].SetCarID(-1);
```

- what will happen if we run this code?
  - the exception won't be caught
  - the **`terminate()`** function will be called

Throw List Example: Outside v1

```
// inside main()

  try {
    train[0].SetCarID(-1);

  } catch (int ID) {
    cerr << "ID invalid, no change";
  }
```

Throw List Example: Outside v1

```
// inside main()

   try {
      train[0].SetCarID(-1);

   } catch (int ID) {
      cerr << "ID invalid, no change";
   }
```

this user has based their code on getting input from a file

Throw List Example: Outside v2

```
// inside main()
while(set == false) {
   try {
      train[0].SetCarID(userID);
      set = true;
   } catch (int ID) {
      cerr << "ID invalid, try again:";
      cin >> userID;
   }
}
```

Throw List Example: Outside v2

```
// inside main()
while(set == false) {
   try {
      train[0].SetCarID(userID);
      set = true;
   } catch (int ID) {
      cerr << "ID invalid, try again:";
      cin >> userID;
   }
}
```

this user has based their code on getting input from a user, and being able to repeat requests

Throw Lists

- warn programmers that functions throw exceptions without catching them
- throw lists should match up with what is thrown and not caught inside the function
  - otherwise, it can lead to a variety of errors, including the function **`unexpected()`**
- can also have empty throw lists for clarity:

```
int GetCarID() throw ();
```

Exception Planning

- how does the exception in **`SetCarID()`** affect the performance of our constructor?

Exception Planning

- how does the exception in **`SetCarID()`** affect the performance of our constructor?

- need to think carefully about when, how, and why we throw exceptions

Exception Classes

- we can create, throw, and catch exception classes that we have created

- we can even create hierarchies of exception classes using inheritance
  - catching the parent class will also catch all child class exceptions

Exception Class Example

```
class MathError { /*...*/ };

class DivideByZeroError:
  public MathError { /*...*/ };
class InvalidNegativeError:
  public MathError { /*...*/ };
```

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Friend Functions

- non-member functions that have member-style access
- function is declared inside the class
  - will be public regardless of specifier
- designate using the ***friend*** keyword

```
friend void AFriendFunction();
```

Friend Classes

- classes can also be declared to be friends of another class

```
class Milo {
public:
   ...
};

class Otis { ... };
```

Friend Classes

- classes can also be declared to be friends of another class

```
class Milo {
public:
   friend class Otis;
};

class Otis { ... };
```

Friend Classes

- classes can also be declared to be friends of another class

```
class Milo {
public:
   friend class Otis;
};

class Otis { ... };
```

the Otis class now has access to all of the private members of the Milo class

Friend Classes

- when one class references another in its definition, we need a ***forward declaration***
  - we've used these before: remember this?

```
typedef struct node* NODEPTR;
```

- in order to reference the **`Otis`** class before it's defined, we need something similar:

```
class Otis;
```

  - before the **`Milo`** class declaration

Using Friends

- why do we want to give access to private members?

Using Friends

- why do we want to give access to private members?
  - use for testing
  - increased speed
  - operator overloading
    - non-member functions get automatic type conversion
  - enhances encapsulation
    - a function being a friend is specified **in** the class

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Inline Functions

- an ***inline function*** gives the complete definition in the class declaration

```
// inside declaration
int GetCarID() {
   return m_carID;
}
```

- no definition of the function in the .cpp file

Inline Functions

- used **only** for short functions

Inline Functions

- used **only** for short functions
  - accessors, empty constructors, one-line functions
- compiler treats inline functions a special way

Inline Functions

- used **only** for short functions
  - accessors, empty constructors, one-line functions
- compiler treats inline functions a special way
  - the function code is inserted in place of each function call at compile time
  - why?

Inline Functions

- used **only** for short functions
  - accessors, empty constructors, one-line functions
- compiler treats inline functions a special way
  - the function code is inserted in place of each function call at compile time
  - saves overhead of a function invocation

Non-Class Inline Functions

- we can make any function an inline function

- use the **inline** keyword

```
inline void PrintHello() {
   cout << "Hello";
}
```

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Namespaces

- we already know and use one namespace:

```
using namespace std;
```

- we can also define and use our own namespaces

Namespace Declarations

```
namespace Alice {
   void Hello();
}

namespace Bob {
 void Hello();
}
```

Namespace Definitions

```
namespace Alice {
   void Hello() {
      cout << "Hello from Alice!"; }
}
namespace Bob {
 void Hello() {
    cout << "Hello from Bob!"; }
}
```

Using Namespaces v1

```
using namespace Alice;
int main() {
   Hello();
   Hello();

   return 0;
}
```

Using Namespaces v1

```
using namespace Alice;
int main() {
   Hello();
   Hello();

   return 0;
}
```

what do each of these calls to Hello() print out?

Using Namespaces v2

```
int main() {
   {
     using namespace Alice;
     Hello();
   } {
     using namespace Bob;
     Hello();
   }
   return 0;
}
```

Using Namespaces v2

```
int main() {
   {
     using namespace Alice;
     Hello();
   } {
     using namespace Bob;
     Hello();
   }
   return 0;
}
```

what do each of these calls to Hello() print out?

Using Namespaces

- What if we use Alice as a universal namespace? Can we call Bob's Hello()?
- How else can we explicitly call one function or the other?
- What if we nest namespaces?

LIVECODING

Outline

- Pass by value VS by reference VS a reference
- Exceptions
- Friends
- Inline Functions
- Namespaces
- Project

Project

- signup for presentation slots next class
- alpha due next Sunday night (the 23rd)

- mini-course project demo day (optional)
  - December 10th or 11th (reading days)
  - poster-session style presentation

Survey

- 1% extra credit **overall**
- please fill out honestly (it's anonymous, and won't be looked at until after grades are in)
- online course evaluation: fill out for **this** class, not for the lecture portion
- pick up your feedback after turning in survey