Exceptions II

Logistics

• Project
  – Part 3 (block) due Sunday, Oct 30
    • Functionality (10)
    • Use of framework (5)
    • Code Style (5)
    • Memoization (5)

• Questions?

Exam

• Exam 2
  – Thursday, October 27.

  – Review Session
    • Tuesday, Oct 25th / 9-10am (70-3435)
    • Wednesday, Oct 26th / 8-9pm (70-3445)

Exam 2

• What it will cover
  – C++ classes
    • constructors
    • Inheritance
    • Operator overloading
  – Templates
  – STL
  – Memory Management

Logistics

• Final exam
  – Good news…bad news
  – Good news
    • Last day of finals, November 18th
  – Bad news
    • 8am-10am
  – Room
    • 01-3338

Plan for this week

• I/O Week
  – Today: Exceptions
  – Tomorrow: Exceptions II
  – Thursday: Exam 2
Before we begin

- Any questions?

Enter...the exception

- Exceptions allow a method to tell the caller when an error has occurred
  - Many times it is the calling function that knows what to do when an error occurs.
  - Exceptions allow the caller to respond to the error rather than the method itself.
  - Different callers may wish to respond to particular errors differently.

Throwing exceptions

- In C++, exceptions are thrown by using the `throw` keyword.
  - Unlike Java, there is not a Throwable class.
  - In C++, any item can be thrown
    - Basic datatypes (int, float, etc.)
    - Class objects
    - Pointers to class objects
    - References to class objects

Catching Exceptions

- Like in Java, C++ uses a `try/catch` block for catching exceptions.

```cpp
void f()
{
    try {
        // call to a method that may throw something
        catch (Overflow) {
            // code that handles an overflow error
            ...
        }
        ...
    }
}
```

Question: Catching exceptions

- Erroneous ordering:

```cpp
try {// something}
    catch (...)
    { // handle anything}
    catch (MathError)
    { // it'll never get here}
    catch (Overflow)
    { // or here}
```

Answer: Catching exceptions

- CC and g++ flags as an error

```cpp
try {// something}
    catch (MathError)
    { // it'll never get here}
    catch (Overflow)
    { // or here}
```

Both flag this as a warning
**Question: Throwing things you shouldn’t**

```c
void foo1 () throw (int){ throw 7;}
void foo2 () throw (float) { foo1();}
main (int argc, char *argv[])
{
try {
  foo2();
} catch (int) { // do something }
}
```

**Answer: Throwing things you shouldn’t**

- foo2 violated it’s specification
- Unexpected handler will be called.
- Not catching == throwing up the stack
- Note: did not convert to float!

---

**Question: Throwing things you shouldn’t**

```c
void foo1 () throw (int){ throw 7;}
void foo2 () throw (float) { foo1();}
main (int argc, char *argv[])
{
try {
  foo1();
} catch (int) { // do something }
}
```

**Answer: Automatic type conversions**

```c
void foo1 () throw (int){ throw 7;}
void foo2 () throw (float) { foo1();}
main (int argc, char *argv[])
{
  try {
    foo1();
  } catch (long) { // do something }
}
```

**Question: Automatic type conversions**

- Sorry, Charlie
  - C++ will NOT do automatic type conversions for exceptions.
  - This code will call terminate.

---

**Question: Is this a good idea?**

```c
void f()
{
  throw Overflow();
}
void g()
{
  try { f (); } catch (MathError &E) { E.printMessage(); }
}
```
Answer: Maybe yes, maybe no

- Implementation may store/transmit exception specially.
  - I.e. not on stack

- Your mileage may vary

- Stroustrup claims it’s acceptable practice.

Any others?

Catching Exceptions

- In C++, there is no `finally` section of the try/catch block.
- In Java, the `finally` code is executed regardless of whether an exception was caught.
  - Allows for cleanup of system resources.

Stack unwinding

- When an exception is thrown in C++
  - Call stack is searched for first function to catch the data thrown.
    - If none found, program will terminate.
    - If one is found:
      - All local variables from all methods on stack from method that threw the exception to that which caught it, will have its destructor called.
      - Note that this is not true for objects allocated on the heap.

void use_file (const char *name)
{
  FILE *f = fopen (...);
  try {
    // some file operation
  } catch (...) {
    fclose (f);
    throw;
  }
  fclose(f);
}

Catching Exceptions

void use_file (const char *name)
{
  FILE *f = fopen (...);
  try {
    // some file operation
  } catch (...) {
    fclose (f);
    throw;
  }
  fclose(f);
}

Stack unwinding

- Advantageous to wrap system resource calls into class objects.
  - The resource can be cleaned up during object destruction.

void use_file (const char *name)
{
  FileObject f (...)
  // do something with f
}
Exceptions and Constructors

• If an exception is thrown during the call to an object’s constructor, only the data members that have been completely constructed are destroyed.

```cpp
class X {
    Y yy;
    Z zz;
public:
    X (char *foo, int bar) :
        yy (foo), zz (bar) {}
    ...
}
```

Exceptions and Constructors

• An object is not completely constructed until its constructor completes.
  – Beware of data members allocated on the heap

```cpp
class Y {
private:
    int *p;
    void init();
public:
    Y (int s) { p = new int [s]; init();}
    ~Y () { delete [] p;}
    ...
}
```

Exceptions and Constructors

```cpp
class Y {
private:
    vector<int> p;
    void init();
public:
    Y (int s) [ p (s); init();]
}
```

Exceptions and Constructors

• Questions?
Using exceptions

• Important safety tips (from Stroustrup, the inventor of C++)
  – Use exceptions for error handling
  – Throw an exception to indicate failure during construction
  – Use exception specifications (good style...in style guide)
  – Beware of dynamically allocated data members when throwing an exception from a constructor

Using exceptions

• Important safety tips
  – Assume that every exception that can be thrown by a function will be thrown.
  – Don’t assume that exceptions will be derived from the “standard” exception class.
  – Libraries should not terminate a program. Throw an exception instead.
  – Think about error handling early in a design

Assertions

• Debugging mechanism to test condition at any point in the code
  – If condition is false, the program aborts and dumps core.
  – Useful for testing preconditions, postconditions and invariant checks.

Assertions

# include <cassert>

void foo (int *p)
{
    // At this point p should not be null
    assert (p != 0);
    ...
}

// constructor

// Preconditions:
// last & first are not empty (emptyString)
// age is not negative

Person::Person( string last, string first, int age, string firstJobName ): lastName(last), firstName(first), currentAge(age), currentJob(0)
{
    assert( last != emptyString );
    assert( first != emptyString );
    assert( age >= 0 );
    if ( firstJobName != noJob ) {
        currentJob = new Job( firstJobName );
    }
}

Exceptions vs. Assertions

• Exceptions
  – Let the caller decide how to handle the error.

• Assertions
  – Aborts the program
  – Debugging tool
  • Should not be included in the release version of software.
Exceptions vs. Assertions

• What about for testing preconditions?

// push
//
// Description: adds a new element to "the top of" the stack
// Arguments:   the element to be added
// Pre:         stack is not full
// Post:        size has increased by one
// Post:        top is equal to the argument newElement
//
virtual void push( char newElement ) = 0;

Exceptions vs. Assertions

• My own humble philosophy
  – Use assertions for errors that are under your control, as a programmer.
  – Use exceptions for error that are under the control of a user of the system or user of your code.

Exceptions

• Questions?