CMSC433, Fall 2001 Programming Language Technology and Paradigms More C++

Alan Sussman

September 13, 2001

Administrivia

- Extra office hour today – 4-5PM
- Project 1
  - public drivers posted
    - a gzipped tar file with driver codes and corresponding outputs
    - 4 list driver files, 2 array driver files
  - questions?

Last time

- One arg constructor – for type conversion
- Derived classes and subtypes
  - in C++ generally means virtual classes
  - derived class should obey public contract of base class
  - overriding only based on *declared* types of parameters
  - covariance of return types allowed
    - but not of parameters/arguments
- Object hierarchy design
  - has-a vs. is-a

Casts in C++

- static_cast
  - C style cast, with different syntax
- dynamic_cast
  - cast from base class ptr to derived class ptr
  - returns null if not instance of derived class
- const_cast
  - to remove/add **const** attribute from a pointer
- reinterpret_cast
  - interpret bits

Casting between classes

- In C++ with multiple inheritance, casting between a ptr to a base type and a ptr to a derived type
  - may require adding/subtracting offset
  - may require run-time lookups
    - if you have virtual base classes
- Reusing the bit pattern likely to be badly wrong
- So C++ provides various cast operators to provide some safety guarantees

static_cast

- For pointers, assumes correct, doesn't look at object
  - B *b = static_cast<B *>(d); // OK
  - D *d = static_cast<D *>(b); // ?
- Also used for casting between built-in types
  - integers, doubles, ...
  - replaces C style cast
  - also can be used to cast from a void*

dynamic_cast

- Looks at object to determine:
  - if legal (returns null otherwise)
  - if pointer value adjustment needed
- Generally used for safely casting from base type to derived type
- Requires that code be compiled with RTTI
  - Run time type information

const_cast

- To remove or add **const** or **volatile** declarations
  - sometimes necessary because **const** is part of a (member) function's type signature
  - can be useful to invoke a non-**const** member function on a **const** object
    - assuming that the function behaves properly in this situation

reinterpret_cast

- interpret bits
- VERY dangerous, no checks
- No adjustments applied
- one valid use is to cast out of void*
- and potentially useful for explicitly placing object in memory at a fixed location with **new**

Abstract classes

- An abstract class cannot be directly instantiated
- Although subclasses may
- In C++, any class with an abstract function is abstract
  - define a method to be abstract by setting it = 0 (no implementation)

Constructor chaining

- For a constructor, you can give the arguments to the constructors for:
  - the base classes
  - the instance variables
- If nothing provided, void constructor used
- Ex. class B : public A {
  C c1, c2; D d1;
  public: B() : A(42), c1("hello"), c2("hi"), d1() {
  …. }

What does B() print?

- struct A {
  A() { g(); };
  virtual void f() { cout << "A"; };
  virtual void g() { f() };
  }
- class B : public A {
  B() { g(); };
  virtual void f() { cout << "B"; };
  }

What happened?

- Constructor B() chains to constructor for A()
  - while doing constructor for A(), object is an A
- Then, perform void constructor for B
  - now, object is a B

Namespaces

- What if I've defined a List class
- And you've written a different List class
- Can my code and your code be used in the same program?
  - No - Can't have two different classes named the same
  - Namespaces to the rescue
- Similar to packages in Java

Using namespaces

- namespace Foo {
  class List { … };
  };
- namespace Bar {
  class List { … };
  };
- Foo::List queue = new Foo::List();

namespace Foo { … };

- All new definitions are in namespace Foo
- Makes all definitions previously declared in Foo available
- OK to open and close a namespace:
  namespace A { … };
  namespace B { … };
  namespace A { … };

namespace needed to add definitions

- Can use :: to give definition of a name previously declared
- But not to define a new name
- void Foo::f(int x) { … };
  - Foo::f(int) must have been previously declared

using *namespace*

- using Foo;
  - makes all names from Foo available
  - until end of namespace (or file)
- using Foo::List;
  - makes List from namespace Foo available as List

Inheriting namespaces

- namespace Foo {
  using A;
  using B;
  class List { … };
  };
- using Foo {
  // Foo::List and all names from A
  // and B available
  }

namespaces aren't databases

- You see X in namespace Foo only if the compiler sees X declared in Foo at some point earlier in the file.
  - doesn't affect need to carefully link header files

namespace aliases

- What if two developers both write code using the namespace Parser?
  - back to our old problem of name clashes
- Partial solution using namespace aliases;
- namespace Parser = als_Parser;
- Makes long names bearable
  - still have to come up with unique long names
  - Java style: edu_umd_cs_als_Parser ?

Exceptions

- throw *exp*; -- throws an exception *exp*
- try { … } catch (ExceptionType e) { … }
  - if, while executing a try block, an exception is thrown that could be passed to a function taking an ExceptionType as an argument, the catch clause is invoked with e bound to value thrown

Exception example

```
struct IndexOutOfBounds{};
char String::getChar(int i) {
        if (i < 0 || i >= length)
                throw IndexOutOfBounds();
        };
try {
        c = s.getChar(42); }
catch (IndexOutOfBounds err) {
        cout << "Index out of bounds"; }
```


Exception hierarchy

- You can create a hierarchy of exceptions
  - Exception
    - IOException
    - BoundsException
- Using derived classes
  - catch a reference to the base class (parameter Base&)
    - if a derived class is thrown, you catch it
  - catch value of the base class (parameter Base)
    - if derived class is thrown, copy constructor for the base class is invoked, using the obj thrown as arg

More exceptions

- If you aren't going to do anything with the exception you caught
  - omit variable name in catch
- You can throw other values
  - ints, doubles
  - don't do this …

Exception declarations

- You can declare the set of exceptions a function might throw
  - not checked at compile time
- Checked at run-time
  - if you declare the exceptions you throw
  - and you throw something not on that list
  - function unexpected() is called
    - which by default calls terminate()
      - which by default calls abort()

Declaring exceptions

- Declare you might throw X or Y
  - int f(int i) throw (X,Y) { … };
- Declare you don't throw any exceptions
  - int g(int i) throw () { … };
- Declare that you don't know what exceptions you throw
  - int h(int i) { … };

Using destructors for final actions

- Sometimes, you want to ensure you take some action when leaving a scope
  - no matter how you leave the scope
    - exception, return, fall off the bottom
- Destructors get called no matter how you leave the scope
  - use them

Example - File closer

```
class FileCloser {
    const FILE *f;
public:
    FileCloser(FILE *f) { this.f = f; };
    virtual ~FileCloser() { f.close(); }
    }
```


Example - Monitor lock

```
class MonitorLock {
    Monitor & m;
public:
    MonitorLock(Monitor & mon) : m(mon)
        { m.lock(); }
    virtual ~MonitorLock()
        { m.unlock(); }
    }
```