CMSC 433, Spring 2001 Programming Language Technology and Paradigms More C++

Alan Sussman
February 6, 2001

Administrivia

- Project 1
  - `const` error in IntList::empty() fixed
  - issues with reusing IntList cells for operator=
  - IntList pop() and push() are to front of list (like a stack)
  - appendCopyAtEnd() appends to end of list
- Quiz
  - average: 37    median: 30
  - 25%: 20       75%: 50

Last time

- new/delete – calls constructors/destructors
- access control
  - public, protected, private
  - friends
- Hidden functions
  - void constructor, copy constructor, operator=
  - created for you, if not defined
    - and do the obvious thing (call the versions for each base class and each member that is an object)

Pointer obligations

- For the hidden functions, if you have a member variable that is a pointer, the default functions are almost certainly wrong
- For each pointer, decide if it is the unique pointer to an object
  - if not, need to control updates of that object and figure out when to free referenced object
  - if so, need to duplicate object rather than pointer

Pointer obligations, with concise syntax

class IntArray {
  int *rep;
  int size;
public:
  IntArray(int sz) : rep(new int[sz]), size(sz) {};
  IntArray(const IntArray &a) :
    rep(new int[a.size]), size(a.size)
    {memcpy(rep,a.rep,sizeof(int)*a.size);}
  ~IntArray()
  {delete [] rep;}
  IntArray & operator=(const IntArray &a) {
    if (this == &a) return *this;
    if (size != a.size) {
      delete [] rep;
      rep = new int[a.size];
      size = a.size;
      memcpy(rep, a.rep, sizeof(int)*size);
      return *this;
    }
  }
};

operator =
One arg constructors

- A one argument constructor is also used for type conversion
  - A(int sz) allows an int to be used anywhere an A is expected
  - only one level of conversion is supported
  - if a B is required and an int is supplied, the system won’t convert an int to an A, which is then converted to a B
  - Can mark constructor as explicit to avoid this feature

Subtyping – Derived classes

- if class D has class B as a public base type
  - a pointer to a D can be provided anywhere a pointer to a B is expected
  - a reference to a D can be provided anywhere a reference to a B is expected
- class D should fulfill B’s public contract
  - something that expects a pointer to a B
  - and is given a pointer to a D
  - shouldn’t be surprised

Virtual functions

- Without virtual functions, B’s functionality will be invoked on objects pointed to by a B pointer
  - determined from explicit type of pointer
- With virtual functions
  - dispatching based on run-time type of object pointed to
    - and compile-time type of arguments
    - must match argument types exactly to override in derived class

Non-virtual functions

- non-virtual functions have confusing semantics
  - function you get depends on type of pointer used to reference the object
- but virtual functions may be more expensive to invoke
- should probably make functions non-virtual only if never overridden
  - final in Java

Common Subtyping Errors

- An array of D’s can be given to someone
  - who expects an array of B’s
- B’s public contract may involve assigning another B to it
- What is the right thing to do
  - when a D is assigned a B?
  - when a B is assigned a D?

Operator= and subtyping

- struct B {
  B& virtual operator=(const B & b) […]
};
- struct D : public B {
  D & virtual operator=(const D & d) […]
};
- B b; D d; B* pb = &b; B* ph = &d;
  - b = d; // B::operator=(const B&)
  - d = b; // Illegal (why?)
  - *pb = *pb; // B::operator=(const B&)
  - *ph = *ph; // B::operator=(const B&)

Virtual operator=

- virtual declaration didn’t matter; arguments not identical
  - struct B {
    B& virtual operator=(const B & b) {...};
    public B {
      D & virtual operator=(const B & b) {...});
      virtual operator=(const D & d) {...};
  }
- struct D : public B {
  D & virtual operator=(const B & b) {...});
  virtual operator=(const D & d) {...});

Similar problems elsewhere

- Copy constructor
  - overriding not an issue
  - watch out for omitting & for a reference parameter
- operator==
  - almost identical set of problems to operator=

Covariance of return types

- When you override a function, your return type can be a subtype of the method you are overriding
  - struct B {
    virtual B* getChild();
    public B {
      virtual B* getChild();
    }
  }
- struct D : public B {
  virtual D* getChild();
  public D {
    virtual D* getChild();
  }

Covariance

- Covariance of return types is OK
  - Somebody who expects a B*, is given a D*
  - invokes getChild() on it
  - Expects a B*, is given a D*
  - No surprise…
- Not OK for argument types
- Co-variance = (in the same way)(vary)
  - as opposed to contravariance
    * allowed in some OO languages for arguments, but not that useful
- Covariance = (in the same way)(vary)
  - as opposed to contravariance
    * allowed in some OO languages for arguments, but not that useful

Argument types can’t be covariant

- Consider if D::f overrode B::f in
  - struct B { virtual void f(B* p); }
  - struct D : public B { virtual void f(D* p); }
- Somebody who expects a B*, is given a D*
  - invokes f(new B()) on it
  - D::f would be surprised!
    * expected a D*, got a B*

Function hiding

- In class B, if you define f(int)
- In derived class D, you define f(char *)
- f(int) can’t be invoked on objects of type D
  - except though a B* pointer
  - violates contractual obligation
- Why?
  - Hides operator== ?
  - prevents surprises
Object Hierarchy Design

• composition/aggregation
  – have components as instance variables
• derivation
  – have components as base classes

Engine vs. Car

• Should a Car have an Engine as a component or as a base type?

Animal class

• Say we have classes for Fish, Dog, Eagle, Whale and Hippo.
• Should they have a common base class?
• What methods should it support?
  – age()?
  – speed()?
  – fly()?
  – eat()?

Heterogeneous collections

• Will you ever need a collection that contains objects that all is-a/has-a/are-a A
  – If so, use is-a (derivation)
• Will you need a collection of engines and cars?
• Will you need a collection of animals?

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

Alan Sussman
February 8, 2001

Administrivia

• Office hours today changed to 3-4PM
• Project 1
  – be careful appending an IntList to itself
  – ElementRef for IntEArray
    • don’t create objects of type ElementRef explicitly
    • ElementRef really should be an inner class of IntEArray
Last time

- Pointer obligations
- One argument constructors for implicit conversions
- Subtyping with virtual functions
- Covariance of return types
  - but not argument types
- Function hiding by name overloading
- Object hierarchy design
  - derivation vs. composition

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)
  - 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
- Make 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 defined
- 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 file or namespace
- 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

```cpp
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(); } 
}