CMSC433, Fall 2001 Programming Language Technology and Paradigms

Alan Sussman
September 4, 2001

Administrivia

- Quiz answers posted on Exams web page
- C++ readings from textbook posted on Readings web page
  - supplements material in lectures
- C++ lecture notes will be posted on Lectures web page regularly

Administrivia (cont.)

- Class account info emailed Friday
  - if you didn't get it, see me after class or in office hours
- My office hours will be posted on class home page
  - TA's coming soon too
- Project 1 - C++ - due Thursday, Sept. 20

Project 1

- C++ Linked List and Expanding Int Array
  - don't use other C++ libraries
  - Expanding Int array should use shared representation, copy on write
    - so that copying is fast
- Specification posted today on web page under Projects link
  - more discussion Thursday

Quiz answers

Copy constructor

```
SimpleArray(const SimpleArray & that) {
    length = that.length;
    data = new int[length];
    for(int i = 0; i < length; i++)
        data[i] = that.data[i];
    }
```

BAD Copy constructor

```
// DO NOT DO THIS
  SimpleArray(const SimpleArray & that) {
      *this = that;
      }
```


Operator=

```
SimpleArray& operator=(const SimpleArray & that)
 {
    if (this == &that) return *this;
    delete [] data;
    length = that.length;
    data = new int[length];
    for(int i = 0; i < length; i++)
          data[i] = that.data[i];
    return *this;
  }
```


operator= notes

- operator= is not operator==
- generally better to define as member function than standalone
- check for this == &that
- free storage for this

Destructor

```
~SimpleArray() {
     delete [] data;
   }
```


destructor notes

- no need to set length = 0 or data = null
- Everything is much simpler if you never allow data==null

implementing OnePlaceBuffer

```
public class Buf implements cmsc433.OnePlaceBuffer {
  private Object data;
  public synchronized Object take()
                  throws InterruptedException {
      while (data == null)
            wait();
      Object tmp = data;
      data = null;
      notifyAll();
      return tmp;
  }
```

OnePlaceBuffer (cont.)

```
public synchronized void put(Object o)
                throws InterruptedException {
    while (data != null)
        wait();
    data = o;
    notifyAll();
    }
}
```


C++

C++ without Classes

- Don't need to say "struct"
- New libraries
- function overloading
  - confusing link messages
- default parameters
- **void ***
  - C++ requires explicit conversion from **void ***

More C++ without Classes

- Dynamic initialization of globals
  - no way to control order between files
- References
  - Compiler treats as pointers
  - Programmer doesn't
  - Once initialized to reference an address, cannot be made to reference a different address
  - Used for pass-by-reference parameters

References

```
int x,y;
int *p = &x;
int &q = y;
q++; // increments y
(*p)++; // increments x
q = x; // assigns value of x to y
p=&y; // makes p point to y;
```


new/delete

- use **new A** to allocate one A
- use **new A[size]** to allocate an array of **size** A's
- use **delete** to free one A
- use **delete[]** to free an array of A
- invokes constructors and destructors
  - for all elements of an array
  - **delete**[] must be given pointer to first element

Data Abstraction

- Avoid name clashes
- Hide implementation
- Override "standard" functionality

IntArray example

- Without OO:

```
void IA_init(IntArray *ia);
void IA_cleanup(IntArray * ia);
void IA_setSize(IntArray * ia, int value);
int IA_getSize(IntArray * ia);
void IA_setElem(IntArray * ia,
      int index, int value);
int IA_getElem(IntArray * ia, int index);
```

- With OO:

```
class IntArray{
public:
    IntArray();
    ~IntArray();
    void setSize(int value);
    int getSize();
    void setElem(int index,
          int value);
    int getElem(int index);
private:
…}
```


Access control

- A member can be public/protected/private
- An access specification controls access to all following members (until the next access specification)
- For classes, initial/default access is private
  - For structs, initial/default access is public
  - no other difference between classes and structs

Access levels

- public – any function/method allowed access
- private - only methods in the class allowed access
- protected - like private, except that subclasses (derived classes) can access inherited members

Protected access

- Class A {
  private: int x;
  protected: int y; }
- Class B : public A {

```
static int foo(A a, B b) {
        int i = a.x; // illegal
        int j = a.y; // illegal
        int k = b.x; // illegal
        int n = b.y; // LEGAL
        …}
```


Friends

- A class X can declare a class Y or a function f as a friend
  - Gives Y or f access to all of X's private and protected data
- Friendship is not transitive
  - If X declares Y as a friend, and Y declares Z as a friend, Z can't see X's private data
- It's not inherited either
  - a class derived from Y cannot access X's private or protected data

Hidden functions

- class A {
  A(); // void (or default) constructor
  A(const A &); // copy constructor
  A& operator=(const A &) // operator=
  ~A(); // destructor
  }

void/default constructor

- Invoked whenever an instance is created without being initialized.
- If no constructor is provided, default public one is created by compiler
  - invokes void constructor on each base class and instance variable (member)
  - members of built-in types not initialized

copy constructor

- Invoked to create a new value from an old value
  - for initialization
  - for parameters passed by value
  - for objects returned as values
- If no copy constructor supplied, default public one created by compiler
  - invokes copy constructor for each base class and instance variable (and copies bits for built-in types)
  - almost never correct if you have pointers
- If copy constructor supplied, must also supply void constructor

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

Alan Sussman

September 6, 2001

Administrivia

- Office hours – Tu 3:30-4:30, Wed 11-noon
  - and by appointment
  - subject to change
  - still no TA
- Project 1

Project 1

- Issues
  - reusing IntList cells for operator=
  - IntListf::pop() and push() are to front of list (like a stack)
  - IntList::appendCopyAtEnd() appends to end of list
  - 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

- C++ without classes
  - function overloading
  - default parameters
  - dynamic global initialization
  - references
  - new/delete
- Classes
  - data abstraction
  - access control – public, private, protected, friend
  - hidden functions – void constructor, copy constructor

operator=

- If no operator=() defined, compiler defines default public one
  - does operator= on each base class and instance variable
- Watch for a = a
- A::operator=(const A&) usually returns A&
  - allows a = b = c;

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;
                }
```


operator =

```
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;
        }
  }
```


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 Subytping 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* pbb = &b; B* pbd = &d;
  - b = d; // B::operator=(const B&)
  - d = b; // Illegal (why?)
  - *pbb = *pbd; // B::operator=(const B&)
  - *pbd = *pbb; // B::operator=(const B&)

Virtual operator=

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

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?