VE281

Data Structures and Algorithms

Linear List Example and Operator Overloading

Announcement

- Written Homework One will be posted on Sakai today.

- Due at 11:40 am next Thursday (Sep. 27, 2012).

Review

- Pointers and Arrays
  - Out-of-bound access error
- Dynamic Memory Management
  - Memory leak error
- Example: A Linear List Class
  - Constructor
  - Initialization syntax
  - Destructor
  - Shallow copy versus deep copy
  - Copy constructor
  - Assignment operator

Outline

- Operational Methods of Linear List
- Operator Overloading
- Overloading **Operator[]** and **Operator<<** for Linear List

Implementation of Linear List Class

- So far, we have

```
class List {
  int *elts;     // pointer to dynamic array
  int sizeElts; // capacity of array
  int numElts;  // current occupancy
public:
  List(int size = MAXELTS);
  // Constructor with default arguments.
  ~List(); // Destructor
  List(const List &l); // copy constructor
  List &operator= (const List &l);
  // assignment operator
};
```

Maintenance methods

Operational Methods

```
void insert(int i, int v); // MODIFIES: this
// EFFECTS: If capacity is full, throws
// FullError; else if 0 <= i <= numElts
// inserts v at position i;
// else throws BoundsError.

void remove(int i); // MODIFIES: this
// EFFECTS: removes the i-th element if 0 <= i
// < numElts; throws BoundsError otherwise.

bool query(int v) const; // EFFECTS: returns true
// if v is in this, false otherwise.

int size() const; // EFFECTS: returns |this|.
```

Const Member Functions

int size() const;

- Each member function of a class has an extra, implicit parameter named **this**.
  - "**this**" is a pointer to the current instance on which the function is invoked.

- **const** keyword modifies the implicit **this** pointer: **this** is now a pointer to a const instance.
  - The member function **size()** cannot change the object on which **size()** is called.
  - By its definition, **size()** shouldn't change the object! Adding **const** keyword prevents any

Const Member Functions

- Implement **size()**

```
int List::size() const {
  return numElts;
}
```

- A const object can only call its const member functions!
- If a const member function calls other member functions, they must be const too!

```
void A::g() const { f(); }
```

```
void A::f() {}
```

```
void A::f() const {}
```

Operational Methods: query

```
bool List::query(int v) const
// EFFECTS: returns true
// if v is in this, false otherwise.
{
    for (int i = 0; i < numElts; i++)
    {
        if (elts[i] == v)
            return true;
    }
    return false;
}
```

Operation Methods: insert

```
class FullError {};
class BoundsError {};

void List::insert(int i, int v) {
    if (numElts == sizeElts) throw FullError();
    if(i >= 0 && i <= numElts) {
        for(int k = numElts-1; k >= i; k--)
            elts[k+1] = elts[k]; //shift right
        elts[i] = v;
        numElts++; // fix numElts invariant
    }
    else throw BoundsError();
}
```

Exercise: Complexity of insert

```
void List::insert(int i, int v) {
    if (numElts == sizeElts) throw FullError();
    if(i >= 0 && i <= numElts) {
        for(int k = numElts-1; k >= i; k--)
            elts[k+1] = elts[k];
        elts[i] = v;
        numElts++;
    }
    else throw BoundsError();
}
```

What is the best case, worst case, and average case?

What are their complexities?

Operation Methods: remove

```
void List::remove(int i) {
    if(i >= 0 && i < numElts) {
        for(int k = i; k <= numElts-2; k++)
            elts[k] = elts[k+1]; //shift left
        numElts--;
    }
    else throw BoundsError();
}
```

Complexity?

Outline

- Operational Methods of Linear List
- Operator Overloading
- Overloading **Operator[]** and **Operator<<** for Linear List

Operator Overloading

Introduction

- C++ lets us redefine the meaning of the operators when applied to objects of class type.
- This is known as operator overloading.
- We have already seen the overloading of the assignment operator.

- Operator overloading makes programs much easier to write and read:

```
List l;
int x = l[5]; // overload [] operator
              // access the list element by index
cout << l << endl; // overload << operator
                   // print all the list elements
```

Operator Overloading

Basics

- Overloaded operators are functions with special names: the keyword **operator** followed by the symbol (e.g., +,-, etc.) of the operator being redefined.

- Like any other function, an overloaded operator has a return type and a parameter list.

```
A operator+(const A &l, const A &r);
```

Operator Overloading

Basics

- Most overloaded operators may be defined as ordinary **nonmember** functions or as class **member** functions.

- Overloaded functions that are members of a class may appear to have one less parameter than the number of operands.
  - Operators that are member functions have an implicit **this** parameter that is bound to the first operand.

Operator Overloading

Basics

- An overloaded unary operator has no (explicit) parameter if it is a member function and one parameter if it is a nonmember function.

- An overloaded binary operator would have one parameter when defined as a member and two parameters when defined as a nonmember function.

```
A operator+(const A &l, const A &r);
// returns l "+" r
A A::operator+(const A &r);
// returns *this "+" r
```

Example

- Overload **operator+=** for a class of complex number.

```
class Complex {
  // OVERVIEW: a complex number class
  double real;
  double imag;
public:
  Complex(double r=0, double i=0); // Constructor
  Complex &operator += (const Complex &o);
  // MODIFIES: this
  // EFFECTS: adds this complex number with the
  // complex number o and return a reference
  // to the current object.
};
```

Example

```
Complex &Complex::operator += (const Complex &o)
{
    real += o.real;
    imag += o.imag;
    return *this;
}
```

Example

- `operator+=` is a member function.
- We can also define a nonmember function that adds two numbers.

```
Complex operator + (const Complex &o1,
     const Complex &o2)
{
  Complex rst;
  rst.real = o1.real + o2.real;
  rst.imag = o1.imag + o2.imag;
  return rst;
}
```

- However, there is a problem with this. What is it?
- Since `operator+` is a nonmember function, it

Friend

- So, we'll need some other mechanism to make the function as a "friend".
- The "friend" declaration allows you to expose the private state of one class to another function or class (and only that function or class) explicitly.

```
class foo {
  friend class bar;
  friend baz();
  int f;
};
class bar { ... };
void baz() { ... }
```

The function **baz** and the methods of class **bar** all have access to **f**, which would otherwise be private to class **foo**.

Friend

```
class foo {
  friend class bar;
  friend baz();
  int f;
};
class bar { ... };
void baz() { ... }
```

- Understanding that friendship is something given, not taken (i.e., **foo** gives friendship to **bar**, but not **foo** takes friendship from **bar**), will help you remember that "**friend class bar;**" goes inside **foo**, not the other way around.

Friend

```
class foo {
  friend class bar;
  friend baz();
  int f;
};
class bar { ... };
void baz() { ... }
```

- Although "friendship" is declared inside **foo**, **bar** and **baz()** are not the members of **foo**!
- "friend" declaration may appear anywhere in the class.
  - It is a good idea to group friend declarations together either at the beginning or end of the

Example

- In our example of complex number class, we will declare **operator+** as a friend:

```
class Complex {
  // OVERVIEW: a complex number class
  double real;
  double imag;
public:
  Complex(double r=0, double i=0);
  Complex &operator += (const Complex &o);
  friend Complex operator+(const Complex &o1,
        const Complex &o2);
};
```

Its implementation is the same as before.

Outline

- Operational Methods of Linear List
- Operator Overloading
- Overloading **Operator[]** and **Operator<<** for Linear List

Linear List

Overloading Operator []

- We want to access each individual element in the list through subscript operator [], just like how we access an ordinary array.
  - For example, **`l[5]`** accesses the sixth element in the list **`l`**.

- We need to overload the **`operator[]`**.
  - It is a binary operator: The first operand is the list and the second one is the index.

Linear List

Overloading Operator []

- We write two versions with bound checking

```
int List::operator[](int i) const {
    if(i >= 0 && i < numElts) return elts[i];
    else throw BoundsError();
}
```

const version returning a plain int

```
int &List::operator[](int i) {
    if(i >= 0 && i < numElts) return elts[i];
    else throw BoundsError();
}
```

nonconst version returning a reference to int

Linear List

Overloading Operator []

- Why we need a nonconst version that returns a reference to int?
  - We need to assign to an element through subscript operation
    `l[5] = 2;`

- Why we need a const version that returns a plain int?
  - We may call the subscript operator with some const list objects. Const objects can only call their const member functions.

- A variation of the const version