CS 368 Announcements
Wednesday, March 20, 2013

Homework hw – graded, regrades contact Chetan by 4/3

Program p3
- due Monday, April 8
- add copy constructor, operator= and destructor to classes from p2

Last Time
- finish Ch. 4 – the Big Three
- copy constructor
- copy assignment (operator=)
- destructor
- Polynomial example

Today
- Polynomial example Big Three
- Unix utilities: gdb and valgrind
- `operatorX` syntax
- overloading assignment ops
- overloading arithmetic ops

Next Time
- cont. Ch. 5, and Polynomial example
- `explicit`
- conventions for arithmetic operators
- `friend`
- conventions for relational and I/O operators

Polynomial.h (Big 3)

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include
using namespace std;

class Polynomial {

    friend bool operator==(const Polynomial & lhs,
                           const Polynomial & rhs);

    public:

        // constructors
        Polynomial();
        Polynomial(double coefficients[], int number);
        Polynomial(const Polynomial & rhs);
        explicit Polynomial(double const_term);

        // destructor
        ~Polynomial();

        // named member functions
        int degree() const { return size - 1; }
        void print(ostream & out = cout) const;

        // assignment operators
        Polynomial & operator= (const Polynomial & rhs);
        Polynomial & operator+= (const Polynomial & rhs);
        Polynomial & operator*= (double rhs);

    private:
        int size;   // size of the coefs array ( = degree + 1)
        double * coefs; // coefs will be an array
};

Polynomial operator+(const Polynomial & lhs,
                     const Polynomial & rhs);

Polynomial operator*(const Polynomial & lhs, double rhs);
Polynomial operator*(double lhs, const Polynomial & rhs);

ostream & operator<<(ostream & out, const Polynomial & p);

#endif
```

Polynomial.cpp (Big 3)

```
// Copy constructor
// Creates a polynomial from the given polynomial
Polynomial::Polynomial(const Polynomial & rhs) :
  size(rhs.size), coefs(new double[rhs.size]) {
    for (int i = 0; i < size; i++) {
        coefs[i] = rhs.coefs[i];
    }
}



// Destructor
Polynomial::~Polynomial() {
    delete [] coefs;
}



// Overload assignment =
const Polynomial & Polynomial::operator=(
                                        const Polynomial & rhs) {
    if (this == &rhs) {
        return *this;
    }

    else {
        delete [] coefs;
        coefs = new double[rhs.size];
        size = rhs.size;
        for (int i = 0; i < size; i++) {
            coefs[i] = rhs.coefs[i];
        }
    }
    return *this;
}
```

Unix utilities: gdb and valgrind

gdb

valgrind

valgrind and Makefiles

```
main: testIntList.o IntList.o
     g++ testIntList.o IntList.o

testIntList.o: testIntList.cpp IntList.h
     g++ -c testIntList.cpp

IntList.o: IntList.cpp IntList.h
     g++ -c IntList.cpp

valgrind: main
     valgrind --leak-check=full main

clean:
     rm *.o
```

Operator Overloading

Why overload operators?

Operators that cannot be overloaded:

Operators that shouldn't be overloaded:

(Non-obvious) operators that can be overloaded:

Can't

- create new operators
- change precedence or associativitiy
- change arity (# of parameters)

Keep in mind

- don't overload just because you can
- be aware of commutative (symmetric) operations
- be aware of what happens if operands are of different types

Polynomial.h (operator overloading)

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H

#include
using namespace std;

class Polynomial {

    friend bool operator==(const Polynomial & lhs,
                           const Polynomial & rhs);

    public:

        // constructors
        Polynomial();
        Polynomial(double coefficients[], int number);
        Polynomial(const Polynomial & rhs);
        explicit Polynomial(double const_term);

        // destructor
        ~Polynomial();

        // named member functions
        int degree() const { return size - 1; }
        void print(ostream & out = cout) const;

        // assignment operators
        Polynomial & operator= (const Polynomial & rhs);
        Polynomial & operator+= (const Polynomial & rhs);
        Polynomial & operator*= (double rhs);

    private:
        int size;   // size of the coefs array ( = degree + 1)
        double * coefs; // coefs will be an array
};

Polynomial operator+(const Polynomial & lhs,
                     const Polynomial & rhs);

Polynomial operator*(const Polynomial & lhs, double rhs);
Polynomial operator*(double lhs, const Polynomial & rhs);

ostream & operator<<(ostream & out, const Polynomial & p);

#endif
```

Assignment Ops: Polynomial.h

```
class Polynomial {
    friend bool operator==(const Polynomial & lhs,
                           const Polynomial & rhs);

    public:
        Polynomial();
        Polynomial(double coefficients[], int number);
        Polynomial(const Polynomial & rhs);
        explicit Polynomial(double const_term);
        ~Polynomial();
        int degree() const { return size - 1; }
        void print(ostream & out = cout) const;


        const Polynomial & operator= (const Polynomial & rhs);
        const Polynomial & operator+= (const Polynomial & rhs);
        const Polynomial & operator*= (double rhs);

    private:
        int size;
        double * coefs;
};

Polynomial operator+(const Polynomial & lhs,
                     const Polynomial & rhs);
Polynomial operator*(const Polynomial & lhs, double rhs);
Polynomial operator*(double lhs, const Polynomial & rhs);

ostream & operator<<(ostream & out, const Polynomial & p);
```

Assignment Ops: Member vs Non-member Function?

Consider:

```
Polynomial p1, p2;

p1 = 1.1;

p2 += p1;
```

Assignment Ops: Polynomial.cpp

```
// Overload assignment +=
Polynomial & Polynomial::operator+= (
                                    const Polynomial & rhs) {
    int newSize = (rhs.size > size) ? rhs.size : size;
    double *newCoef = new double [newSize];

    for (int i = 0; i < newSize; i++) {
        newCoef[i] = 0;
    }
    for (int i = 0; i < rhs.size; i++) {
        newCoef[i] += rhs.coefs[i];
    }
    for (int i = 0; i < size; i++) {
        newCoef[i] += coefs[i];
    }
    delete [] coefs;
    coefs = newCoef;
    return *this;
}


// Overload assignment *= so it supports scalar multiplication
Polynomial & Polynomial::operator*=(double rhs) {
    for (int i = 0; i < size; i++) {
        coefs[i] *= rhs;
    }
    return *this;
}
```

Arithmetic Ops: Polynomial.h

```
class Polynomial {
    friend bool operator==(const Polynomial & lhs,
                           const Polynomial & rhs);

    public:
        Polynomial();
        Polynomial(double coefficients[], int number);
        Polynomial(const Polynomial & rhs);
        explicit Polynomial(double const_term);
        ~Polynomial();
        int degree() const { return size - 1; }
        void print(ostream & out = cout) const;


        const Polynomial & operator= (const Polynomial & rhs);
        const Polynomial & operator+= (const Polynomial & rhs);
        const Polynomial & operator*= (double rhs);

    private:
        int size;
        double * coefs;
};

Polynomial operator+(const Polynomial & lhs,
                     const Polynomial & rhs);
Polynomial operator*(const Polynomial & lhs, double rhs);
Polynomial operator*(double lhs, const Polynomial & rhs);

ostream & operator<<(ostream & out, const Polynomial & p);
```

Arithmetic Ops: Member vs Non-member Function?

Consider:

```
Polynomial p1 = 11.22, p2;

p2 =  p1 + 1.1;

p2 = 1.1 + p1;
```

Arithmetic Ops: Polynomial.cpp

```
// Overload +
Polynomial operator+(const Polynomial & lhs,
                     const Polynomial & rhs) {
    Polynomial answer(lhs);
    answer += rhs;
    return answer;
}


// Overload * so it supports scalar multiplication.  Note that
// we overload it twice so we can do either:
//    polynomial * scalar
// or
//    scalar * polynomial

Polynomial operator*(const Polynomial & lhs, double rhs) {
    Polynomial answer(lhs);
    answer *= rhs;
    return answer;
}

Polynomial operator*(double lhs, const Polynomial & rhs) {
    Polynomial answer(rhs);
    answer *= lhs;
    return answer;
}
```