CS 368 Announcements
Wednesday, November 6, 2013

Program p2 – graded

Program p3
- due 10 pm today, November 6th
- add destructor, copy constructor, operator= to classes from p2

Program p4 – assigned

Last Time
- Unix utilities: gdb and valgrind
- `const` member functions
- operator overloading
- `operatorX` syntax
- overloading assignment ops
- overloading arithmetic ops

Today
- continue Ch. 5
- overloading arithmetic ops
- `explicit`
- overloading output operator (**<<**)
- overloading relational ops
- `friend`
- overloading increment (**++**) and decrement (**−−**)

Next Time
- start Ch. 9 (Input and Output)
- console I/O
- error states
- file I/O

Recall: 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);
```

Recall: Member vs Non-member Function?

Consider using a member function with assignment ops:

`Polynomial p1, p2;`
`p1 = 1.1;` **actually** `p1.operator=(1.1)`
`p2 += p1;` **actually** `p1.operator+=(p.1)`

- use member functions with assignment operators since left-hand operand will be a Polynomial object

Consider using a member function with arithmetic ops:

`Polynomial p1 = 11.22, p2;`
`p2 = p1 * 1.1;` **actually** `p1.operator*(1.1)` **okay**
`p2 = 1.1 * p1;` **actually** ~~`1.1`~~`.operator*(p1)` **error**

- arithmetic operators should be symmetrical (as shown above)
- can't use member function and allow for symmetry
- use non-member function instead:

`p2 = p1 * 1.1;` **now is** `operator*(p1, 1.1)` **okay**
`p2 = 1.1 * p1;` **now is** `operator*(1.1, p1)` **okay**

- but this requires two versions to be symmetrical
  ```
  operator*(double, Polynomial)
  operator*(Polynomial, double)
  ```
- better – code a single arithmetic operator
  ```
  operator+(Polynomial, Polynomial)
  ```
  and code constructors to convert to Polynomial
  ```
  Polynomial(double const_term);
  ```

Type Converting Constructors

`explicit` keyword

<< Op: 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);
```

<< Op: Member vs Non-member Function?

Polynomial.cpp

```
// Prints the polynomial to the given ostream.  If no
// ostream is given, the polynomial is printed to cout
void Polynomial::print(ostream & out) const {
    if (size == 0) {
        return;
    }

    for (int i = size - 1; i > 0; i--)
        out << coefs[i] << "x^" << i << " + ";
    out << coefs[0];
}


// Overload << for output
ostream & operator<<(ostream & out, const Polynomial & p) {
    p.print(out);
    return out;
}
```

Relational 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);
```

Relational Ops and `friend` keyword

Relational Ops: Polynomial.cpp

```
// Overload ==
bool operator==(const Polynomial & lhs, const Polynomial & rhs)
{

    if (lhs.size != rhs.size) {
        return false;
    }

    for (int i = 0; i < lhs.size; i++) {
        if (lhs.coefs[i] != rhs.coefs[i]) {
            return false;
        }
    }

    return true;
}
```

Overloading ++ and --