IOStreams III

Inserters and Extractors, Manipulators, and friends

Logistics

• Project
  – Part 2 (water) due Sunday, Oct 16th
  • Feedback by Monday
  – Part 3 (block) due Sunday, Oct 30
• Questions?

Exam

• Exam 2
  – Thursday, October 27.
  – Review Session
    • Tuesday, Oct 25th / 9-10am (70-3435)
    • Wednesday, Oct 26th / 8-9pm (70-3445)

Exam 2

• What it will cover
  – C++ classes
    • constructors
    • Inheritance
    • Operator overloading
  – Templates
  – STL
  – Memory Management

Logistics

• Final exam
  – Good news…bad news
  – Good news
    • Last day of finals, November 18th
  – Bad news
    • 8am-10am
  – Room
    • 01-3338

Plan for this week

• I/O Week
  – Today: IOStreams 1
  – Tomorrow: IOStreams 2
  – Thursday: IOStreams 3
  • Friends
    • Review of inserters/extractors
    • Manipulators (FYI)
**IOStream Class Hierarchy**

**Insertion and Extraction**

- **Recall:**
  - Insertion and Extraction is the standard means for writing / reading data and objects to / from text form.
  - Insertion (output) – operator\(<\<\)
  - Extraction (input) – operator\(>>\)

```cpp
int i = 5;
cout << i;
cin >> i;
```

- Inserters and extractors for built-in datatypes (like `int`, `double`, `float`, etc.) are predefined member functions of the IOStream classes.
- It would be nice if we can do I/O on objects of our own classes in the same manner as the basic datatypes.

```cpp
int i = 5;
double d = 7.0;
myClass foo (7);
cout << i << d << foo;
```

**Insertion and Extraction**

- We can do so by overloading the operator\(\ll\) and operator\(<\<\) for our classes.
- Note: these operators must be defined outside of any class.

**Insertion and Extraction**

- Signature
  - Since the operators are being defined outside of a class, the operator will take the 2-argument form:
  ```cpp
  istream & operator\(<\<\) (istream &is, myClass& m);
  ostream & operator\(<\ll\) (ostream &os, myClass& m);
  ```
  Very often, these operators will need access to private members of `myClass` thus are usually declared as `friend`.
  Note that the operators return a reference to the stream acted upon.
Sidebar: friend

- By using friend, a class can grant access to non-member functions or to another class.
- Friend functions can be declared anywhere within a class declaration, but it is common practice to list friends at the beginning of the class.
- The public and protected keywords do not apply to friend functions

Using existing operators

- The simplest approach is to decompose your class into a set of objects/variables that have inserters and extractors defined.

```cpp
friend istream &operator>> (istream& is, date& d) {
    is >> d.day >> d.month >> d.year;
    return is;
}
friend ostream &operator<< (ostream& os, date &d) {
    os << d.day << ' ' << d.month << ' ' << d.year;
    return os;
}
```

This is why operators are declared as friends

Friends and inheritance

- You cannot have virtual friend functions.

```cpp
class date {
    public:
    date (int d, int m, int y);  
date();
    private:
    int day;
    int month;
    int year;

    friend istream &operator>> (istream& is, date& d);
    friend ostream &operator<< (ostream& os, date &d);
};
```

```cpp
class Performer {
    public:
    Performer (char *name, char *talent);
    virtual void calculatePay();
    friend ostream &operator<< (ostream& os, Performer &P);
};

class Musician : public Performer {
    public:
    Musician (char *name);
    void calculatePay();
};
```

Friends and inheritance

- Musician M ("Ringo")
  
  cout << "Info on the drummer for the Beatles: " << M;

  Calls operator<< (ostream &os, Performer &P) displays only Performer data.
Friends and inheritance

class Performer {
public:
    Performer (char *name, char *talent);
    virtual void calculatePay();
    virtual ostream &display (ostream &out);
friend ostream &operator<< (ostream &os, Performer &P);
};
class Musician : public Performer {
public:
    Musician (char *name);
    void calculatePay();
    virtual ostream &display (ostream &out);)
};

Friends and inheritance

ostream &operator<< (ostream &os, Performer &P) {
    return P.display (os);
}

Musician M ("Ringo")
cout << "Info on the drummer for the Beatles: " << M;

Calls operator<<(ostream& os, Performer &P)
calls Musician::display
    displays only Musician data.

Friend classes

- A class can declare a member function of another class as a friend
- A class can declare an entire other class as a friend.

- Useful where one class is tightly coupled to another class.

Friend classes

void PointCollection::set(const double x, const double y) {
    // Get the number of elements in the collection.
    const int nElements = vecPoints.size();
    // Set each element.
    for (int i=0; i<nElements; i++) {
        vecPoints[i].x = x;
        vecPoints[i].y = y;
    }
}

Friend classes

// Forward declaration of friend class.
class PointCollection;

// Point class.
class Point {
friend PointCollection;
public:
    Point(const double x, const double y);
    ~CPoint(void)
    // ...
private:
    double x;
    double y;
};

Friend classes

- Friendship is not mutual

void Point::illegallyAccessCollection(PointCollection PC, int i) {
    cout << "Point " << i << " is " << PC.vecPoints[i];
}

Friend classes

void Point::illegallyAccessCollection(PointCollection PC, int i) {
    cout << "Point " << i << " is " << PC.vecPoints[i];
}
Friend classes

• Friendship is not transitive
  – Will not travel down class hierarchy

// Forward declaration of friend class.
class PointCollection;

// Point class.
class Point
{
friend PointCollection;  // subclasses of PointCollection cannot access

public:
  Point(const double x, const double y);
  ~Point(void)
  
private:
  double x;
  double y;
};

Using existing operators

• Good safety tip:
  – It is usually a good idea to make extraction and
    insertion complementary operations.

  – Thus,
    date d (10, 10, 2003);
    date dd;
    cin >> dd;
    // dd == d

Writing an inserter (operator<<)

friend ostream &operator<<( ostream &out, const Point &p )
{
  out << '(' << p.x << ',' << p.y << ')';
  return out;
}

Output:
(0,0)

Friends

• Questions?

Writing an inserter (operator<<)

1. The first argument should be a reference to an
   ostream. The second argument should be a constant
   reference to your class.
2. The function should return a reference to an ostream
   so that insertions can be chained.
3. The body of the function should perform whatever
   output is appropriate for your class, but nothing more!
4. If you need to access the private data members of your
   class directly, then your class must declare this function
   to be a friend

Writing an extractor (operator>>)

• Like inserter except:
  – Must handle possible errors
  – Argument cannot be const reference.
  – Almost surely will have to declare as a friend.
Writing an extractor (operator>>)

friend istream & operator>> (istream &in, Point &p)
{
    char c;
    int ok = FALSE;
    in >> c;
    if (c == '(') {  
in        in >> p.x >> c;
        if (c == ',') {
            in >> p.y >> c;
            if (c == ')') ok = TRUE;
        }
    }
    if (!ok) in.clear (in.rdstate() | ios::failbit
                            return in;

• Things to note:
  – Checks to see if in same format as output.
  – Stops reading as soon as an error is found.
  – Sets failbit if format is not correct.

• Questions?

Inserters / Extractors

• Questions

Manipulators

• The iostreams have a version of operator<< and operator>> that take a function pointer as an argument:

  ostream & operator<< {ostream& (*f)(ostream&)}
  
  return (*f)(*this); }

Manipulators

• What’s the deal with manipulators?

  cout << setw(5) << setfill ('0') << 345;

  How does C++ know what to do with setw and set_fill

• Example:
  – cout << flush

  – Note that there is a non-member function

  ostream & flush (ostream& os)
  
  return os.flush();

}
Manipulators

When the compiler sees:
\[\text{cout} \ll \text{flush;}\]

It will convert to
\[\text{cout.operator\ll} (\text{flush});\]

Which will call
\[\text{flush} (\text{cout});\]

Which will result in
\[\text{cout.flush();}\]

And all this is done at compile time!!

Manipulators

• Let’s kick it up a notch

• What if your manipulator takes arguments?
\[\text{cout} \ll \text{setw}(5) \ll 123;\]

There is no operator\ll with a function pointer and an argument

Manipulators

• functor!!!

• Recall: what differentiates functors from pointers to function is that functors are objects
  – They can retain state.

Manipulator

• Assume there is a non-member function
\begin{verbatim}
ostream & setw (ostream& os, int w)
{    return os.width(w);
}

smanip setw(int i)
{    return smanip(setw, i);
}
\end{verbatim}

Manipulator

\begin{verbatim}
class smanip
{
    public:
    ostream & (*f) (ostream &os, int i); // function to call
    int intarg; // integer argument

    smanip (ostream & (*ff)(ostream&, int, int ii)) :
    f (ff), intarg(ii) {}

    operator(ostream &os) { return (*f)(os, intarg); }
    
    ostream operator\ll (ostream &os, smanip sm)
    {    return os(sm);
    }

    
}
\end{verbatim}

Manipulator

When the compiler sees:
\[\text{cout} \ll \text{setw}(5);\]

It will convert to
\[\text{cout} \ll \text{smanip} (\text{setw}, 5); // lets call smanip (setw, 5), setsm then\]

\[\text{cout.operator\ll} (\text{setsm});\]

Which will call
\[\text{setsm(cout);}\]

Which will result in
\[\text{setw(os, 5)}\]

Which will call
\[\text{os.setw(5);}\]

And all this is done at compile time!!
Manipulator

• Rolling your own

```cpp
ostream& hello (ostream& os) {
    os << "Hello" << endl;
    return os;
}

main (int argc, char argv[]) {
    cout << hello;
}
```

Summary

• Questions?
  – Have a nice weekend.