Operator Overloading

Ho Dac Hung

Operator Overloading

- Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones.

```
d3.addobjects(d1, d2);
d3 = d1.addobjects(d2);
d3 = d1 + d2;
```

Overloading Unary Operators

- ++
- --
- -

Overloading Unary Operators

```
class Counter
{
    private:
        unsigned int count; //count
    public:
        Counter() : count(0) //constructor
        { }
        unsigned int get_count() //return count
        { return count; }
        void operator ++ () //increment (prefix)
        {++count;}
};
```

Overloading Unary Operators

```
class Counter
{
    private:
        unsigned int count; //count
    public:
        Counter() : count(0) //constructor
        { }
        unsigned int get_count() //return count
        { return count; }
        void operator ++ (int) //increment (postfix)
        {count++;}
};
```

```
class Counter
{
    private:
        unsigned int count; //count
    public:
        Counter() : count(0) //constructor no args
        { }
        Counter(int c) : count(c) //constructor, one arg
        { }
        unsigned int get_count() const //return count
        { return count; }
        Counter operator ++ () //increment count (prefix)
        { //increment count, then return
            return Counter(++count); //an unnamed temporary object
        } //initialized to this count
        Counter operator ++ (int) //increment count (postfix)
        { //return an unnamed temporary
            return Counter(count++); //object initialized to this
        } //count, then increment count
```

Overloading Binary Operators

- Binary operators can be overloaded just as easily as unary operators.
  - Arithmetic Operators
  - Comparison Operators
  - Arithmetic Assignment Operators

```
class Distance //English Distance class
{
    private:
        int feet;
        float inches;
    public: //constructor (no args)
        Distance() : feet(0), inches(0.0)
        { } //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void getdist() //get length from user
        {
            cout << "\nEnter feet: "; cin >> feet;
            cout << "Enter inches: "; cin >> inches;
        }
        void showdist() const //display distance
        { cout << feet << "\'-" << inches << '\"'; }
        Distance operator + ( Distance ) const; //add 2 distances
};
```

```
Distance Distance::operator + (Distance d2) const
{
    int f = feet + d2.feet; //add the feet
    float i = inches + d2.inches; //add the inches
    if(i >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        i -= 12.0; //by 12.0 and
        f++; //increase feet by 1
    } //return a temporary Distance
    return Distance(f,i); //initialized to sum
}
```

```
class Distance //English Distance class
{
    private:
        int feet;
        float inches;
    public: //constructor (no args)
        Distance() : feet(0), inches(0.0)
        { } //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void getdist() //get length from user
        {
            cout << "\nEnter feet: "; cin >> feet;
            cout << "Enter inches: "; cin >> inches;
        }
        void showdist() const //display distance
        { cout << feet << "\'-" << inches << '\"'; }
        bool operator < (Distance) const; //compare distances
};
```

```
bool Distance::operator < (Distance d2) const {
    float bf1 = feet + inches/12;
    float bf2 = d2.feet + d2.inches/12;
    return (bf1 < bf2) ? true : false;
}
```

```
class Distance //English Distance class
{
    private:
        int feet;
        float inches;
    public: //constructor (no args)
        Distance() : feet(0), inches(0.0)
        { } //constructor (two args)
        Distance(int ft, float in) : feet(ft), inches(in)
        { }
        void getdist() //get length from user
        {
            cout << "\nEnter feet: "; cin >> feet;
            cout << "Enter inches: "; cin >> inches;
        }
        void showdist() const //display distance
        { cout << feet << "\'-" << inches << '\"'; }
        void operator += ( Distance );
};
```

```
void Distance::operator += (Distance d2)
{
    feet += d2.feet; //add the feet
    inches += d2.inches; //add the inches
    if(inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
    inches -= 12.0; //by 12.0 and
    feet++; //increase feet
    } //by 1
}
```

The Subscript Operator ([])

- The subscript operator, [], which is normally used to access array elements, can be overloaded. This is useful if you want to modify the way arrays work in C++.

```
class safearay
{
    private:
        int arr[LIMIT];
    public:
        void putel(int n, int elvalue) //set value of element
        {
            if( n< 0 || n>=LIMIT )
            { cout << “\nIndex out of bounds”; exit(1); }
            arr[n] = elvalue;
        }
        int getel(int n) const //get value of element
        {
            if( n< 0 || n>=LIMIT )
            { cout << “\nIndex out of bounds”; exit(1); }
            return arr[n];
        }
};
```

```
class safearay
{
    private:
        int arr[LIMIT];
    public:
        int& access(int n) //note: return by reference
        {
            if( n< 0 || n>=LIMIT )
            { cout << “\nIndex out of bounds”; exit(1); }
            return arr[n];
        }
};
```

```
class safearay
{
    private:
        int arr[LIMIT];
    public:
        int& operator [](int n) //note: return by reference
        {
            if( n< 0 || n>=LIMIT )
            { cout << "\nIndex out of bounds"; exit(1); }
            return arr[n];
        }
};
```

Data Conversion

- Assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign. But what happens when the variables on different sides of the = are of different types?

Conversions Between Basic Types

- There are of course many conversions which were built into the compiler and called up when the data types on different sides of the equal sign so dictate. We say such conversions are implicit because they aren't apparent in the listing.
- Casting provides explicit conversion: It's obvious in the listing that static_cast<int>() is intended to convert.

Conversions Between Objects and Basic Types

- When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler doesn't know anything about user-defined types besides what we tell it. Instead, we must write these routines ourselves.

```
class Distance
{
    …
    public:
    …
        Distance(float meters) : MTF(3.280833F)
        { //convert meters to Distance
            float fltfeet = MTF * meters; //convert to float feet
            feet = int(fltfeet); //feet is integer part
            inches = 12*(fltfeet-feet); //inches is what's left
        } //constructor (two args)
```

```
    operator float() const //conversion operator
    { //converts Distance to meters
        float fracfeet = inches/12; //convert the inches
        fracfeet += static_cast<float>(feet); //add the
                feet
        return fracfeet/MTF; //convert to meters
    }
};
```

Conversions Between Objects of Different Classes

- You can use a one-argument constructor or you can use a conversion operator. The choice depends on whether you want to put the conversion routine in the class declaration of the source object or of the destination object.

```
class time12
{
    private:
        bool pm; //true = pm, false = am
        int hrs; //1 to 12
        int mins; //0 to 59
    public: //no-arg constructor
        time12() : pm(true), hrs(0), mins(0)
        { }
        //3-arg constructor
        time12(bool ap, int h, int m) : pm(ap), hrs(h),
          mins(m)
        { }
```

```
    void display() const //format: 11:59 p.m.
    {
        cout << hrs << ':';
        if(mins < 10)
        cout << '0'; //extra zero for "01"
        cout << mins << ' ';
        string am_pm = pm ? "p.m." : "a.m.";
        cout << am_pm;
        }
};
```

```
class time24
{
    private:
        int hours; //0 to 23
        int minutes; //0 to 59
        int seconds; //0 to 59
    public:
        time24() : hours(0), minutes(0), seconds(0)
        { }
        time24(int h, int m, int s) : hours(h), minutes(m),
          seconds(s)
        { }
        void display() const //format: 23:15:01
        {…}
        operator time12() const; //conversion operator
};
```

```
time24::operator time12() const
{
    int hrs24 = hours;
    bool pm = hours < 12 ? false : true; //find am/pm
    int roundMins = seconds < 30 ? minutes : minutes+1;
    if(roundMins == 60) //carry mins?
    {
        roundMins=0;
        ++hrs24;
        if(hrs24 == 12 || hrs24 == 24) //carry hrs?
        pm = (pm==true) ? false : true;
    }
    int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
```

```
    if(hrs12==0) //00 is 12 a.m.
    { hrs12=12; pm=false; }
    return time12(pm, hrs12, roundMins);
}
int main()
{
    …
    time24 t24(h, m, s);
    time12 t12 = t24;
    …
    return 0;
}
```