SFU CMPT-212 2008-1 Topic: Operator Overloading

Ján Maňuch

E-mail: jmanuch@sfu.ca

Monday 18th February, 2008

Operator overloading

- operators are already overloaded
  Example: `+`, `-`, `*`, etc.
- what about using those operators for our classes?

```
1    Time t1(10,1),t2=90;
2    Time t=t1+t2;
```

- two ways: overloading *non-member operators* and *member operators*

Non-member operator overloading

- *Example:*

```
1  Time operator+(const Time &t1, const Time &t2) {
2    Time sum;
3    sum.minutes = t1.minutes + t2.minutes;
4    sum.hours = t1.hours + t2.hours + sum.minutes/60;
5    sum.minutes = sum.minutes%60;
6    return sum;
7  }
```

Now we can call `operator+` for objects of type `Time`:

```
1    Time t1(10,1),t2=90;
2    Time t;
3    t=operator+(t1,t2); // function notation;
4    // equivalent to:
5    t=t1+t2;              // operator notation
```

- **a problem:** how can `operator+` access private members of class `Time`?

- we have to declare it `friend` to `Time`:

```
1 class Time {
2 ...
3   friend Time operator+(const Time &,const Time &);
4 };
```

- automatic conversions allows us to do the following:

```
1 Time t1(20,2), t2, t3;
2 t2=t1+10; // convert int to Time and call operator+
3 t3=20+t1; // convert int to Time and call operator+
4 t3=20+10; // add ints and convert the result to Time
```

Example: time2.cpp

Member operator overloading

- *Example:*

```
1   class Time {
2    public:
3     Time operator+(const Time &);
4   };
5   Time Time::operator+(const Time &t) {
6     Time sum;
7     sum.minutes=minutes+t.minutes;
8     sum.hours=hours+t.hours+sum.minutes/60;
9     sum.minutes=sum.minutes % 60;
10    return sum;
11  }
```

- We can call `operator+` for objects of type `Time`:

```
1   Time t1(10,1),t2=90;
2   Time t;
3   t=t1.operator+(t2); // function notation;
4                       // equivalent to:
5   t=t1+t2;            // operator notation
```

- **Notes.**
  - `+` is still a binary operator, but one *operand* is passed implicitly via the `this` pointer

```
1 t=t1+t2; // t=t1.operator+(t2);
```

  - a non-member operator requires as many parameters as the operator has operands
  - a member operator requires one parameter less
  - don't declare both member and non-member versions of the same operator — otherwise, an ambiguous call
- **Overloading restrictions.**
  - at least one operand must be a user-defined type (it's not possible to redefine operators on standard types)
  - only existing operators can be overloaded, they keep their arity and precedence — you cannot define new operators
  - `.`, `::`, `?:`, `sizeof` operators cannot be overloaded
  - `=`, `()`, `[]`, `->` ops can be overloaded only as member functions

Working with ostream

- how does `cout <<i;` works?
- by overloading operator `<<`:

```
1 // option 1 - member operator
2 class ostream {
3 ...
4 ostream & operator<<(int);
5 };
6
7 // option 2 - non-member operator
8 ostream & operator<<(ostream &,int);
```

- we can teach `ostream` (and hence also `cout`) to print our user-defined type in the same way

Example:

```
1  class Date {
2  ...
3    friend ostream & operator<<(ostream &os,const Date &d);
4    // grant access to the private members of Date
5  };
6
7  ostream & operator<<(ostream &os,const Date &d)
8  {
9    os << d.year << "/" << d.month << "/" << d.day;
10   return os;
11 }
12 ...
13 Date d;
14 cout <<d; \\ equivalent to operator<<(cout,d);
```

- why do we return `ostream &`? why not:

```
1 void operator <<(ostream &os, const Date &d);
```

- problem:

```
1   Date d;
2   cout <<d<<endl; // would be a compiler error
```

- how does it work (when `<<` return `ostream&`)?

```
1   Date d;
2   cout <<d<<endl;
```

  is equivalent to:

```
1   (cout <<d)<<endl; \\ eq. to
2   operator<<(operator<<(cout,d),endl);
3   // the inner call displays d and returns cout:
4   operator<<(cout,endl);
5   // displays endl
```

- *Example:* date8.cpp

Sample questions

- Consider the class `Time`:

```
1  class Time {
2    int mins, hours;
3   public:
4    Time(int h, int m) { hours=h+m/60; mins=m%60; }
5  };
```

  Write a code overloading the *binary* operator `-` acting on objects of type `Time` as (a) a member function; (b) a non-member friend function of the class `Time`.

- Consider a class `Point`:

```
1  class Point {
2    int x,y;
3   public:
4    Point() { x=0; y=0; }
5    Point(int nx, int ny) { x=nx; y=ny; }
6    int getX() const { return x; }
7    int getY() const { return y; }
```

```
8 };
```

Enhance the above class so that objects of type `Point` can be printed using `cout` as any built-in type. For example:

```
1 Point p(3,4);
2 cout <<p<<endl;
```

should print `[3,4]`.

Dynamic memory and classes

Task: use classes to design a safe array — which remembers its size and can do some bound checking

First approach:

```
1   class intArray {
2     int *data;
3     int size;
4    public:
5     intArray(int s);
6     ~intArray();
7   };
8   intArray::intArray(int s) {
9     size = s;
10    data = new int[size];
11  }
12  intArray::~intArray() {
13    delete [] data;
14  }
```

Question. How to access elements of the array?

- by overloading index operator `[]`:

```
1  class intArray {
2      ...
3   public:
4      ...
5    int & operator[](long i);
6    const int & operator[](long i) const;
7  };
8  int & intArray::operator[](long i) {
9    // check i
10   return data[i];
11 }
12 const int & intArray::operator[](long i) const {
13   // check i
14   return data[i];
15 }
```

- usage:

```
1  intArray A(10);
2  A[0]=5;                 // equivalent to
3  A.operator[](0)=5; // calls non-const version
4  ...
5  const intArrayB(20);
6  cout <<B[2];            // calls const version
7  ...
8  cout <<A[3];            // could call either
9                          // if both defined,
10                         // calls non-constant version
```

Example: intarray1.cpp

Question: What will the following code do?

```
1 intArray A(5);
2 intArray B=A; // initializing B from A
```

- as for structures: *memberwise copying of data members* — not exactly what we want
- more precisely, a *copy constructor* is called:

```
1 intArray(const intArray &);
```

  - if not defined by the creator of the class, provided by compiler
  - the *default copy constructor* performs *memberwise copying*

    Example:

```
1 intArray::intArray(const intArray &a) {
2   size=a.size;
3   data=a.data;
4 }
```

```
1 Date today;
```

Question: When is the *copy constructor* used?

- explicitly:

  ```
  Date d = Date(today);
  ```

- implicitly:

  ```
  Date d(today);
  ```

- dynamic memory allocation:

  ```
  Date *pdate = new Date(today);
  ```

- automatic conversion from the unary (copy) constructor:

  ```
  Date d = today;
  ```

- passing parameter to and/or returning from a function *by value*

```
1 Date f(Date d) {
2   ...
3   return Date;
4 }
```

- when temporary object is created

Fixing intArray

- the *copy constructor* should allocate new memory and copy elements of the array:

```
1  intArray::intArray(const intArray & a)
2  {
3    size = a.size;
4    data = new int[size];
5    for (int i=0; i<size; i++)
6      data[i]=a.data[i];
7      // also could be: data[i]=a[i]
8      //   <- using index operator
9  }
```

Example: intarray2.cpp

Question: What will the following code do?

```
1 intArray A(5),B(10);
2 B=A; // memberwise assignment
```

- again, if not defined, compiler provides a default assignment operator which performs *memberwise assignment*:

```
1 intArray & intArray::operator=(const intArray & a) {
2   size=a.size;
3   data=a.data;
4 }
```

- *Question:* what problems this will cause for `intArray`?
  - loosing pointer `B.data` (memory leak)
  - `A` and `B` point to same dynamic array
  - this dynamic array is then deleted twice!
- the *assignment operator* is used when assigning to the *existing* object

Fixing intArray

- the *assignment operator* should free used memory, then allocate new memory and copy elements of the array:

```
1  intArray &intArray::operator=(const intArray & a)
2  {
3    if (this == &a) // if assigning object to itself,
4      return *this; //   do nothing
5    delete [] data;
6    size = a.size;
7    data = new int[size];
8    for (int i=0; i<size; i++)
9      data[i]=a.data[i];
10
11   return *this;
12 }
```

Example: intarray3.cpp

- *Q.* Why do we have to avoid self-assignment?

```
1   intArray A(10), &B=A;
2   A=B; // self-assignment
```

`data` would be deleted before they are copied!

- *Q.* What happens if we use the following code?

```
1   Time t;
2   t=90;
```

Example: time3.cpp

  - the unary constructor `Time(int)` is used to create temporary object
  - calls the assignment operator
  - destroys the temporary object

- if an assignment from some particular type `T` is used often in the code, we can optimize the code:
  define the assignment operator taking `T` as its argument
  (hence we can avoid creating and destroying a temporary object)
- *Example:*

```
1   Time &Time::operator=(int m) {
2     minute = m % 60;
3     hours  = m / 60;
4     return *this;
5   }
```

Summary for classes with dynamic data

- if a class dynamically allocates memory, the destructor of the class should free this memory
- if the destructor is freeing the memory pointed to by a pointer `p`, then make sure that every constructor initializes `p` to newly allocated memory or to the `NULL` pointer
  (calling `delete` on `NULL` is safe)
- define a copy constructor (that is: redefine the default copy constructor)
- define the assignment operator — remember to check for the self-assignment
- once again, only pair `new` with `delete`, and `new []` with `delete []`

Sample questions

- List 3 similarities and 3 differences between the copy constructor and the assignment operator for a given class.
- What is the output of the following code?

```
1  class A {
2  private:
3    int v;
4  public:
5    A(int v) { v=v; }
6    A(const A& a)
7    { cout <<"- <- "<<a.v<<endl; v=a.v; }
8    A& operator=(const A& a)
9    { cout <<v<<" <- "<<a.v; v=a.v; }
10   friend A f(A);
11 };
```

```
12  A f(A o) {
13   o.v++;
14   return o;
15  }
16
17  int main()
18  {
19    A x(1),y(2);
20    x=f(y);
21  }
```

Explain why!

- Assuming that a class is using dynamic memory allocations, which member functions is necessary to define?

Illustrate this on the following example. Consider a class `Array` with constructors and data members as follows:

```
1  class Array {
2   private:
3    long size;
4    int *data;
5   public:
6    explicit Array(long s, int *inidata=NULL) : size(s)
7    {
8      data = new int[size];
9      if (inidata)
10       for (long i=0; i<size; i++)
11         data[i]=inidata[i];
12   }
13   // ...
14 };
```

Add the necessary functions to finalize the class.