Programming Design In-class Practices

Operator Overloading

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Problem 1: the comparison operator ==

- Consider the example program of the class **`MyVector`**.
- Overload the comparison operator as follows:
  - Implement an instance function.
  - Compare a given **`MyVector`** object (as the invoking object) with a double value **`d`**.
  - If all vector elements equal **`d`** (more precisely, deviates from **`d`** by no more than 0.00001), return true; otherwise, return false.
- Try it on your own computer.

Problem 2: the indexing operator [ ]

- Recall the two implementations for the overloaded indexing operator **[]**:

```
class MyVector
{
  // ...
  double operator[](int i) const;
  double& operator[](int i);
};
```

```
double MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n) // same
    exit(1);          // implementation!
  return m[i];
}
```

- Explain what will happen if:
  - The non-constant one is removed.
  - The constant one is removed.

Problem 3: the indexing operator `[]`

- One of your friends does not want to have two implementations to overload `[]`.
- He proposed to write only one implementation like this:

```
class MyVector
{
  // ...
  double& operator[](int i) const;
};
```

```
double& MyVector::operator[](int i) const
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
```

  – He claims: "All **`MyVector`** objects, constant or not, may invoke this function. Moreover, the returned value may be put at the LHS of an assignment operator."
- Will this implementation pass compilation? Is there any drawback?

Problem 4: about `this`

- Recall one implementation of the assignment operator:

```
class MyVector
{
  // ...
  void operator=(const MyVector& v);
};
```

```
void MyVector::operator=(const MyVector& v)
{
  if(this != &v)
  {
    if(this->n != v.n)
    {
      delete [] this->m;
      this->n = v.n;
      this->m = new double[this->n];
    }
    for(int i = 0; i < n; i++)
      this->m[i] = v.m[i];
  }
}
```

- This is to avoid copying values for operations like **`a1 = a1`**.
- May we replace **`this != &v`** by **`*this != v`**?
  - Will this pass compilation?
  - Will it avoid value copying?

Problem 5: == again

- Recall that we overloaded **==** in Problem 1.
  - We may write operations like `a1 == 0.2` with that implementation.
- Implement something to allow statements like `0.2 == a1`.
  - Otherwise, show that we do not need to implement any additional thing.

Problem 6: negation

- Let’s implement the negation operator **`-`**.
  - Let **`a1`** by a **`MyVector`** object.
  - **`-a1`** should not modify **`a1`**.
  - **`-a1`** should return a **`MyVector`** object whose element values are the negation of those in **`a1`**.
  - The program at the right should execute successfully and print out **`Equal!`**.
  - **`-a1`** should not be allowed to exist at the LHS of an assignment operator.

```
int main()
{
  double d = 1.23;
  double m[3] = {-d, -d, -d};
  MyVector v(3, m);
  v.print();
  v = -v;
  v.print();

  if(v == d)
    cout << "Equal!";
  else
    cout << "Unequal!";

  return 0;
}
```

Problem 7: streaming out with <<

- May we do operations like **`cout << a1`**?
- We need to overload the stream insertion operator **<<**:

```
class MyVector
{
  // ...
  friend ostream& operator<<(ostream& out, const MyVector& v);
};

ostream& operator<<(ostream& out, const MyVector& v)
{
  out << "(";
  for(int i = 0; i < v.n - 1; i++)
    out << v.m[i] << ", ";
  out << v.m[v.n - 1] << ")";
  return out;
}
```

Problem 7: streaming out with <<

- Now the following program may run successfully:

```
int main()
{
  double d = 1.23;
  double m[3] = {d, d, d};
  MyVector v(3, m);
  cout << v << endl;

  if(d == v)
    cout << "Equal!";
  else
    cout << "Unequal!";

  return 0;
}
```

Problem 8: streaming in with >>

- Let's do operations like **`cin >> a1`** by overloading the stream extraction operator **>>**:

```
class MyVector
{
  // ...
  friend istream& operator>>(istream& in, MyVector& v); // no const!
};
```