Object-Oriented Programming and Data Structures

COMP2012: Generic Programming +*-/ Operator Overloading <&% >

Prof. Brian Mak
Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

From Math Notation to Language Operators

- To program the mathematical equation:

```
c = 2(a - 3) + 5b
```

  one may have to write

```
c = add(multiply(2, subtract(a,3)), multiply(5,b));
```

- Most programming languages have operators which allow us to mimic the mathematical notation by writing

```
c = 2*(a - 3) + 5*b;
```

- However, many languages only have operators defined for the built-in types.
- C++ is an exception: it allows you to re-use most, but not all, of its operators and re-define them for new user-defined types.
- E.g., you may re-define "+", "-" etc. for types Complex, Matrix, Array, String, etc. defined by you.

Add 2 Vectors by a Global Add( ) Function

```
class Vector                                          /* File: vector0.h */
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx( ) const { return x; }
    double gety( ) const { return y; }
  private:
    double x, y;
};
```

```
#include <iostream>                                   /* File: vector0-add.cpp */
#include "vector0.h"

Vector add(const Vector& a, const Vector& b)
    { return Vector(a.getx( ) + b.getx( ), a.gety( ) + b.gety( )); }

int main( )
{
    Vector a(1, 3), b(-5, 7), c(22), d;
    d = add(add(a, b), c));                          // d = a + b + c
    std::cout << d.getx( ) << " , " << d.gety( ) << "\n";
}
```

Global Non-member Operator+ Function

- Wouldn't it be nicer if we could write the last addition expression as: $d = a + b + c$ instead of

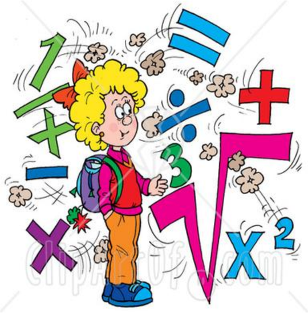
```
d = add(add(a, b), c));
```

- C++ allows you to do that by simply replacing the name of the function **add** by **operator+**.
- Also notice that our global non-member **operator+** works for adding
  - a vector to a vector
  - a vector to a scalar
  - a scalar to a vector

```
#include <iostream>                                  /* File: vector0-op+.cpp */
#include "vector0.h"
using namespace std;

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx( ) + b.getx( ), a.gety( ) + b.gety( )); }

int main( ) {
    Vector a(1, 3), b(-5, 7), c(22), d;
    d = a + b + c; cout << "vector + vector:  a + b + c = ";
    cout << '(' << d.getx( ) << " , " << d.gety( ) << ")\n";

    d = b + 1.0; cout << "vector + scalar:  b + 1.0 = ";
    cout << '(' << d.getx( ) << " , " << d.gety( ) << ")\n";

    d = 8.2 + a; cout << "scalar + vector:  8.2 + a = ";
    cout << '(' << d.getx( ) << " , " << d.gety( ) << ")\n";
}
```

Part I

Overloaded Operator Functions



Operator Function Syntax

- **operator+** is a formal function name that can be used like any other function name.
- We could have called the **operator+** function in the formal way as

  ```
  d = operator+(operator+(a, b), c);
  ```

  But who would want to write code like that?
- Operator functions in C++ are just like ordinary functions, except that they also can be called with a nicer syntax similar to the usual mathematical notations.
- The operator + has a formal name, namely **operator+** (consisting of 2 keywords), and a "nickname," namely +.
- The formal name requires you to call it as

  ```
  operator+(a, b)
  ```

  while the simple nickname let you call it as

  ```
  a + b
  ```

Operator Syntax ...

- The nickname can only be used when calling the function.
- The formal name can be used in any context, when declaring the function, defining it, calling it, or taking its address.
- There is nothing that you can do with operators that cannot be done with ordinary functions. In other words, operators are just syntactic sugar.
- Be careful when defining operators. There is nothing that inhibits you from defining + to denote, e.g., subtraction.
- Similarly, nothing inhibits you from defining **operator+** and **operator+=** so that the following 2 expressions: `a = a + b` and `a += b`, have 2 different meanings.
- However, your code will become unreadable.

Don't shock the user!

C++ Operators

- Almost all operators in C++ can be overloaded except:

  `.` `::` `?:` `.*`

- The C++ parser is fixed. That means that you can only re-define existing operators, but you cannot define new operators (using new symbols).
- Nor can you change the following properties of an operator:

1. Arity: the number of arguments an operator takes.
   e.g. `!x` `x+y` `a%b` `s[j]`
   (So you are not allowed to re-define the + operator to take 3 arguments instead of 2.)
2. Associativity: e.g. `a+b+c` is always identical to `(a+b)+c`.
3. Precedence: which operator is done first?
   e.g. `a+b*c` is treated as `a+(b*c)`.

C++ Operators: Member or Non-member Functions

- All C++ operators already have predefined meaning for the built-in types. It is impossible to change this meaning.
- You can only overload the operator to have a meaning for your own (user-defined) classes (such as **Vector** in the example above).
- Therefore, every operator function you define must have at least one argument of a user-defined class type.
- You may define a (new) operator function as a member function of a new class, or as a global non-member function.
- As a global function, **operator+** has 2 arguments. When it is called in an expression such as **a + b**, this is equivalent to writing **operator+(a, b)**.
- As a member function of class **X**, **operator+** will have only 1 argument. When it is called in an expression such as **a + b**, **a** must be an **X** object and is implicitly passed to **operator+(a,b)** as the first argument.

Global Non-member Operator≪ Function

```
cout << "("  << d.getx( ) << " , " << d.gety( ) << ")\n";
```

- In the previous example, one prints out a Vector *d* as above.
- Let's write a non-member **operator**≪ function to print Vectors more naturally using **ostream** objects.
- The syntax should be similar to the one we use to print values of the basic types (such as **int**). E.g., `cout << x;`
- To allow the usual output syntax with **cout** on the left, the **ostream** object must be the first argument in the function.
- **ostream** is the base class for all possible output streams.
- In particular, **cout** and **cerr** are objects of classes derived from **ostream**.

Question: Why it returns **ostream&**?

Global Non-member Operator<< Function ..

```
#include <iostream>                    /* File: vector0-op+os.cpp */
#include "vector0.h"
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
{
    return (os << '(' << a.getx( ) << " , " << a.gety( ) << ')');
}

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx( ) + b.getx( ), a.gety( ) + b.gety( )); }

int main( )
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    cout << "vector + vector: a + b = " << a + b << endl;
    cout << "vector + scalar: b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector: 8.2 + a = " << 8.2 + a << endl;
}
```

Global Non-member Operator≪ Function ...

- The **operator**≪ returns an ostream object because we like to cascade outputs in one statement such as:

```
Vector a(1, 0);
cout << " a = " << a << "\n";
```

- The second line is equivalent to:

```
operator<<(operator<<(operator<<(cout," a = "),a),"\n");
```

- This can only work if **operator**≪ returns the ostream object itself.

Question: Could we define **operator**≪ as a member function?

Operator+ Member Function

- Member operator functions are called using the same "dot syntax" by specifying an object of, for example, type **Vector**.
- If a is a **Vector** object, then the expression `a+b` is equivalent to `a.operator+(b)`.
- Note that to call the **operator+** member function, the class object must be the left operand. (Here **a**.)
- Thus, when we define **operator+** as a member function of **Vector**, it has only one argument — the first argument is implicitly the object on which the member function is invoked.
- Recall the implicit this pointer in all member functions. Thus,

  ```
  Vector operator+(const Vector& b) const;
  ```

  of the class **Vector** will be compiled into the following global function:

  ```
  Vector operator+(const Vector* this, const Vector& b);
  ```

Operator+ Member Function ..

```
#include <iostream>                                          /* File: vector-op+.h */
class Vector {
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx( ) const { return x; }
    double gety( ) const { return y; }
    Vector operator+(const Vector& b) const;
    const Vector& operator+=(const Vector& b);
  private:
    double x, y;
};

Vector Vector::operator+(const Vector& b) const
{
    // Return by value; any copy constructor?
    return Vector(x + b.x, y + b.y);
}
const Vector& Vector::operator+=(const Vector& b)
{
    x += b.x; y += b.y;
    return *this;                          // Return by const reference. Why?
}
```

Operator+ Member Function ...

```
#include "vector-op+.h"                        /* File: vector-op+test.cpp */
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
{
    return (os << '(' << a.getx( ) << " , " << a.gety( ) << ')');
}

int main( )
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);

    cout << "vector + vector:  a + b = " << a + b << endl;
    cout << "vector + scalar:  b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector:  8.2 + a = " << 8.2 + a << endl; // Error

    a += b;
    cout << "After += :  a = " << a << " b = " << b << endl;
}
```

Operator+ Member Function: Commutative?

- Whenever the compiler sees an expression of the form **a+b**, it converts the expression to the two possible representations

  operator+(a, b)
  a.operator+(b)

  and verifies whether one of them are defined.
- It is an error to define both.
- **operator+** should be commutative: $a + b$ is equivalent to $b + a$. Thus, we expect we may do (vector + scalar) and (scalar + vector) too.
- However, as a Vector member function, the left operand of **operator+** is always a Vector.
- The current version only works for (vector + vector) and (vector + scalar). Why?

Question: Why **operator+** and **operator+=** have different return types?

Operator+ (Vector, Scalar)

```
#include "vector-op+.h"                    /* File: vector-op+ok.cpp */
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '(' << a.getx( ) << " , " << a.gety( ) << ')'); }

int main( )
{
    Vector a(1.1, 2.2);
    cout << "vector + scalar:  a + 5 = " << a + 5 << endl;
}
```

- It works because the argument to the right of + which is a scalar can be converted to a Vector object.
  Question: Where is the conversion constructor in **vector-op+.h**?
- Thus, the expression `(a + 5)` is converted to

  `a.operator+(Vector(5))`

- Let's do the other way: add a **Vector** object to a scalar.

```
#include "vector-op+.h"                    /* File: vector-op+error.cpp */
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '(' << a.getx( ) << " , " << a.gety( ) << ')'); }

int main( )
{
    Vector a(1.1, 2.2);
    cout << "scalar + vector: 5 + a = " << 5 + a << endl;
}
```

```
vector-op+error.cpp:10:46: error: invalid operands to binary
      expression ('int' and 'Vector')
    cout << "scalar + vector: 5 + a = " << 5 + a << endl;
                                           ~ ^ ~
```

Operator+ (Scalar, Vector): What's the Problem?

- Isn't the **operator+** commutative? Isn't the expression `(5 + a)` equivalent to `(a + 5)`?

  Yes, they are equivalent but `(5 + a)` will be converted to

  ```
  5.operator+(a)
  ```

  but **int** is not a class and there is no **operator+** member function for **int** nor can we re-define it.

- Wouldn't **5** be converted to a Vector object by Vector's conversion constructor and the result calls its **operator+** member function with argument **Vector a**?

  No, compilers will not try to do that for efficiency reason. In theory, there can be many non-Vector objects which may add with a Vector object. How can the compilers check out all of them and make the conversion?

Non-member Operator+ (Scalar, Vector)

- One solution is to write a global non-member **operator+** whose first argument is a scalar, and the function actually calls the **operator+** member function of its 2nd Vector argument.

```
Vector operator+(double a, const Vector& b) { return b + a; }
```

- A better solution is our previous global non-member **operator+** function which takes 2 Vector arguments (if Vector class provides the public getx( ) and gety( ) functions to access x and y).

```
Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx( ) + b.getx( ), a.gety( ) + b.gety( )); }
```

Overload Operator= for Member Assignment

```
#include <iostream>                              /* File: vector-op=.h */

class Vector {
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    const Vector& operator=(const Vector& b);
  private:
    double x, y;
};

const Vector& Vector::operator=(const Vector& b)
{
    if (this != &b)              // Avoid self-assignment to save time
    {
        x = b.x;
        y = b.y;
    }
    return *this;                       // Why return const Vector& ?
};
```

Member Operator= with Owned Data Members

```
class Word                                              /* File: word.h */
{
  private:
    int freq; char* str;

  public:
    Word(const char* s, int k = 1) : freq(k) {
        cout << "conversion\n";
        str = new char [strlen(s)+1]; strcpy(str, s);
    }
    Word(const Word& w) { *this = w; cout << "copy\n"; }

    const Word& operator=(const Word& w) {
        if (this != &w)
        {   cout << "op= with " << w.str << "\n";
            freq = w.freq; delete [ ] str;
            str = new char [strlen(w.str)+1]; strcpy(str, w.str);
        }
        return *this;
    }
};
```

```
#include <iostream>                          /* File: word-test.cpp */
using namespace std;
#include "word.h"

int main( )
{
    Word movie("Titanic"); Word song("My heart will go on");
    song = song; song = movie;
}
```

- If a class contains pointer data members and dynamic memory allocation is required, the default memberwise assignment — shallow copy — is not adequate.
- The copy constructor and **operator=** should be implemented so that each object has its own copy of the "owned" data.
- Since the copy constructor and **operator=** usually do the same thing, they may be defined by making use of the other.
- Here, the copy constructor is defined by calling **operator=**.

Member Operator[ ] To Access Vector Component

```
#include <iostream>                                    /* File: vector-op[ ].h */
class Vector {
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double operator[ ](int) const;   // Read-only; c.f. getx( ) and gety( )
    double& operator[ ](int);                          // Allow read and write
  private:
    double x, y;
};

double Vector::operator[ ](int j) const {
    switch (j) {
        case 0: return x;
        case 1: return y;
        default: std::cerr << "op[] const:  invalid dimension!\n"; } }

double& Vector::operator[ ](int j) {
    switch (j) {
        case 0: return x;
        case 1: return y;
        default: std::cerr << "op[]:  invalid dimension!\n"; } }
```

```
#include "vector-op[].h"                    /* File: vector-op[ ]-test.cpp */
using namespace std;

// Replace getx( ), gety( ) by op[ ]
ostream& operator<<(ostream& os, const Vector& a)
{
    return (os << '(' << a[0] << " , " << a[1] << ')'); // Which op[ ]?
}

int main( )
{
    Vector a(1.2, 3.4);
    cout << "Before assignment:  " << a << endl;

    a[0] = 5.6; a[1] = 7.8;                         // Which op[ ]?
    cout << "After assignment:  " << a << endl;

    a[2] = 9;                                       // Which op[ ]?
}
```

Why 2 Versions of Member Operator[ ]?

- Try to compile "vector-op[ ]-test.cpp" with only having the 2nd version of **operator[ ]**.

```
vector-op[]-test.cpp:7:28: error: no viable overloaded
      operator[] for type 'const Vector'
    return (os << '(' << a[0] << " , " << a[1] << ')');
                         ^~~~
./vector-op[].h:19:17: note: candidate function not viable:
      'this' argument has type 'const Vector', but method
      is not marked const
double& Vector::operator[ ](int j) {
                ^
```

- Try to compile "vector-op[ ]-test.cpp" with only having the 1st version of **operator[ ]**.

```
vector-op[]-test.cpp:15:10: error: expression is not assignable
    a[0] = 5.6; a[1] = 7.8; // Which op[ ]?
    ~~~~ ^
```

Member Operator++

```
class Vector {                                        /* File: vector-op++.h */
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double operator[ ](int) const; // Read-only; c.f. getx( ) and gety( )
    double& operator[ ](int);                        // Allow read and write
    Vector& operator++( );               // Pre-increment returns an l-value
    Vector operator++(int);              // Post-increment returns a r-value
  private:
    double x, y;
};

Vector& Vector::operator++( ) { ++x; ++y; return *this; }

// The int argument is a dummy. Why is it needed?
Vector Vector::operator++(int)
{
    Vector temp(x,y);
    x++; y++; return temp;
}

/* Plus the operator[ ] function definitions */
```

Member Operator++ ..

```
#include <iostream>                    /* File: vector-op++test.cpp */
#include "vector-op++.h"
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '(' << a[0] << " , " << a[1] << ')'); }

int main( )
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    Vector c;

    c = ++a;
    cout << "a = " << a << "\nc = " << c << endl;

    c = b++;
    cout << "b = " << b << "\nc = " << c << endl;
}
```

Summary: Member or Non-member Operator Functions

- The operators: $=$ (assignment), [ ] (indexing), ( ) (call) are required by C++ to be defined as class member functions.
- A member operator function has an implicit first argument of the class. Thus, if the left operand of an operator must be an object of the class, it can be a member function.
- If the left operand of an operator must be an object of other classes, it must be a non-member function. e.g. **operator**$\ll$.
- For commutative operators like +, -, *, it is usually preferred to be defined as non-member functions to allow automatic conversion of types using the conversion constructors.

```
string x("dot"), y("com"), z;
z = x + y;
z = x + "com";
z = "dog" + y;
```

Part II

Friend Functions or Classes

Operator≪ as a Member Function

- Let's try to implement **operator**≪ as a member function.

```
#include <iostream>                    /* File: vector-os-nonfriend.h */

class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    double getx( ) const { return x; }
    double gety( ) const { return y; }
    ostream& operator<<(ostream& os);
  private:
    double x, y;
};

ostream& Vector::operator<<(ostream& os)
{
    return (os << '(' << x << " , " << y << ')');
}
```

Operator<< as a Member Function

```
#include <iostream>                          /* File: vector-os-nonfriend.cpp */
using namespace std;
#include "vector-os-nonfriend.h"

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx( ) + b.getx( ), a.gety( ) + b.gety( )); }

int main( )
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    Vector d = a + b;

    d << (cout << "vector + vector:  a + b = ") << endl;
    (b + 1.0) << (cout << "vector + scalar:  b + 1.0 = ") << endl;
    (8.2 + a) << (cout << "scalar + vector:  8.2 + a = ") << endl;
}
```

Issues of Operator$\ll$ as a Member Function

- **operator**$\ll$ is a binary operator. As a member function, the Vector object must be on the left of $\ll$ and **cout** on the right.
- To print a Vector **x**, now you have to write: x $\ll$ cout; instead of the usual output syntax: cout $\ll$ x;
- Furthermore, to cascade outputs, say, to print Vectors **x**, **y** and then **z**, now you will have to write:

  z $\ll$ (y $\ll$ (x $\ll$ cout));

  instead of the usual output syntax: cout $\ll$ x $\ll$ y $\ll$ z;
- For such kinds of operators, it is better to implement them as global non-member functions.
- Two issues:
  1. Since global non-member functions can't access private data members, don't forget to provide the latter with public assessor member functions.
  2. Compared to member operators, non-member operators are less efficient as additional calls to assessor functions are made.

Friend Member Operator≪

```
#include <iostream>                                 /* File: vector-friend.h */

class Vector
{
    friend ostream& operator<<(ostream& os, const Vector& a);
    friend Vector operator+(const Vector& a, const Vector& b);
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
  private:
    double x, y;
};

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '(' << a.x << " , " << a.y << ')'); }

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.x + b.x, a.y + b.y); }
```

Friend Member Operator<<

```
#include <iostream>                    /* File: vector-friend.cpp */
using namespace std;
#include "vector-friend.h"

int main( )
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    cout << "vector + vector:  a + b = " << a + b << endl;
    cout << "vector + scalar:  b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector:  8.2 + a = " << 8.2 + a << endl;
}
```

friend Functions and friend Classes

- A class **X** may grant a global function or another class as its friends.
- Friend functions are not considered member functions.
- Member access qualifiers are irrelevant to friend functions.
- Friend functions or classes of class **X** can be declared by **X** anywhere inside its class definition, but usually before all the members.
- Friends of **X** may access all its data members — both public and non-public members. So be careful!
- All member functions of an **X**'s friend class can access all data members of **X**.

Properties of C++ Friendship

- Friendship is granted not taken. The designer of a class determines who are its friends during the design. Afterwards, he cannot add more friends without rewriting the class definition.
- Friendship is not symmetric: if A is B's friend, B is not necessarily A's friend.
- Friendship is not transitive: if A is B's friend and B is C's friend, A is not necessarily C's friend.
- Friendship is not inherited: friends of a base class do not become friends of its derived classes automatically.

Student with a Hacker Friend: v-student.h

```
#include "course.h"                                   /* File: v-student.h */
#include "v-uperson.h"

class Student : public UPerson {
    friend class Hacker;                      // Got a Hacker friend! Good luck!
  private:
    float GPA; Course* enrolled[50]; int num_courses;
  public:
    Student(string n, Department d, float x) :
        UPerson(n, d), GPA(x), num_courses(0) { }
    ~Student( ) { for (int j = 0; j < num_courses; ++j) delete enrolled[j]; }
    float get_GPA( ) const { return GPA; }
    bool add_course(const string& s)
        { enrolled[num_courses++] = new Course(s); return true; };
    virtual void print( ) const
    {
        cout << "--- Student Details --- \n"
             << "Name:   " << get_name( ) << "\nDept:   " << get_department( )
             << "\nGPA:  " << GPA
             << "\n" << num_courses << " Enrolled courses:   ";
        for (int j = 0; j < num_courses; ++j)
           { enrolled[j]->print( ); cout << ' '; } cout << "\n";
    }
};
```

Student with a Bad Hacker Friend: hacker.h

```
#ifndef HACKER_H                                    /* File: hacker.h */
#define HACKER_H

class Hacker
{
  private:
    string name;

  public:
    Hacker(const string& s) : name(s) { }
    void add_course(Student& s) { s.GPA = 0.0; }
};
#endif
```

Student with a Bad Hacker Friend: Ooops

```
#include <iostream>                    /* File: bad-friend.cpp */
using namespace std;
#include "v-student.h"
#include "hacker.h"

int main( )
{
    Student freshman("Naive", CIVL, 4.0);
    Hacker cool_guy("$#%&");

    freshman.print( );
    freshman.add_course("COMP2012");
    freshman.print( );

    cool_guy.add_course(freshman);
    freshman.print( );
}
```