Bài 10: **Chồng hàm và toán tử** (***Function and operator overload***)

Chồng hàm

- C++ cho phép nhiều hàm trong cùng một phạm vi (toàn cục, trong cùng namespace, hàm static trong một file nguồn,...) có thể có trùng tên, nhưng phải khác nhau về các tham số gọi (số tham số, kiểu từng tham số)

```
1. int compare(int n1, int n2);
2. int compare(float x1, float x2);
3. bool compare(float x1, float x2);    // lỗi
4. int compare(string& s1, string& s2);
5. int compare(const string& s1, const string& s2);
```

- Để xác định đúng hàm gọi, trình biên dịch sẽ ưu tiên hàm có các kiểu tham số chính xác như các tham số khi gọi, nếu không có thì sẽ dùng hàm nào mà các tham số có thể chuyển kiểu được sang

```
string ss1("xyz"), ss2("mpnq");
const string cs("aaa");
compare(1.3, 2.5);          // lỗi
compare("abcd", "12345"); // hàm 5
compare(ss1, ss2);          // hàm 4
compare(ss1, cs);           // hàm 5
```

Chồng phương thức trong lớp

- Tương tự, các phương thức trong cùng một lớp cũng có thể được định nghĩa chồng

```
class C {
public:
    int compare(int x, int y);
    int compare(int x, int y) const;
    int compare(float x, float y);
};
```

- Định nghĩa chồng ở lớp con sẽ che mất các phương thức cùng tên của lớp mẹ

```
class D: public C {
public:
    int compare(string s1, string s2);
};

D d;
d.compare("1234", "abcd");   // OK
d.compare(10, 20);           // lỗi
d.C::compare(10, 20);        // OK
```

Tham số mặc định của hàm/phương thức

- Các tham số của hàm có thể có giá trị mặc định (là giá trị được dùng nếu bỏ qua khi gọi)
- Tham số mặc định phải là các tham số cuối cùng của hàm
  - ```
    void out(double x, int width = 7, int prec = 3) {...}
    out(1.2345, 10, 5);
    out(1.2345, 10);  // → out(1.2345, 10, 3);
    out(1.2345);      // → out(1.2345, 7, 3);
    ```
  - ```
    void f(char c = 'y', int n, bool b = true) {...} // lỗi
    ```
- Tham số mặc định có thể chỉ cần khai báo ở prototype
  - ```
    double df(double x, int order = 1);
    // ...
    double df(double x, int order) {...}
    ```
- Có thể dùng biểu thức làm giá trị mặc định, nhưng không được chứa các tham số khác của hàm đó
  - ```
    UserProfile usr;
    double out(double x, int prec = getPrecOption(usr));
    ```
  - ```
    double next(double x, double dx = diff(x));  // lỗi
    ```
    ```

Tham số mặc định của hàm/phương thức *(tiếp)*

- Tránh gây nhầm lẫn với các hàm chồng
  - ```
    void input(double& x);
    void input(double& x, const char* prompt = "Nhap so: ");
    input(y);   // lỗi
    ```
- Tham số mặc định của phương thức: tương tự như hàm
  - ```
    class Vehicle {
       void out(int prec = 3);
    };
    ```
- Tham số mặc định của constructor
  - ```
    class Vehicle {
    public:
       Vehicle();  // cons mặc định
       Vehicle(Color c = Color::black, int wheels = 4);
    };
    Vehicle v1(Color::red);
    Vehicle v2(Color::white, 8);
    Vehicle v3;  // lỗi
    ```
- Hàm/phương thức có số tham số tuỳ ý: tự tìm hiểu thêm
    ```

Định nghĩa chồng toán tử (*operator overload*)

Khái niệm

- Các toán tử trong C++ có thể được định nghĩa lại cho các kiểu mới:
  - VD: sau khi đã định nghĩa lớp Vector, ta có thể định nghĩa các toán tử +, -, * để có thể thực hiện các phép toán như sau:
    - ```
      Vector v1, v2, v3;
      v3 = -v1 + v2*2;  // câu lệnh sử dụng 4 toán tử
      ```
  - Tuy nhiên, phép toán giữa các kiểu cơ bản là có sẵn, không thể định nghĩa lại:
    - ```
      int x = 3 + 2*5;
      double y = 2.54/1.23 + 3.11;
      ```
- Để định nghĩa lại toán tử, ta viết một hàm gọi là hàm toán tử (operator function) với các tham số và kiểu trả về tương ứng
  - Hàm toán tử có thể là hàm toàn cục hoặc là phương thức của một lớp
  - Không được định nghĩa tham số mặc định cho các hàm toán tử
  - Nếu được định nghĩa trong lớp, tham số thứ nhất của toán tử luôn là chính đối tượng được gọi, không cần phải khai báo

Khái niệm *(tiếp)*

- Hầu hết các toán tử có thể được định nghĩa lại trong C++:

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
||	++	--	,	->*	->	()	[]	new	delete
new[]	delete[]	sizeof							

  - Các toán tử `+ - * &` có ý nghĩa khác nhau khi dùng một hoặc hai ngôi, nhưng đều có thể được định nghĩa lại
  - Tất cả các toán tử trên khi định nghĩa trong một lớp, thì sẽ được thừa kế, chỉ trừ toán tử =
- Chỉ một số ít toán tử không thể định nghĩa lại:

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

- Không thể thay đổi thứ tự ưu tiên của các toán tử và thứ tự thực hiện chúng trong biểu thức

Chồng toán tử một ngôi

- Dùng hàm toán tử toàn cục với một tham số, hoặc phương thức không có tham số trong một lớp
- Cú pháp:
  - `<kiểu trả về> operator <toán tử>(<kiểu> <tham số>) {...}`

  hoặc:
  - ```
    class <tên lớp> {
        <kiểu trả về> operator <toán tử>() [const] {...}
    };
    ```
- Ví dụ:
  - ```
    Vector operator -(const Vector& v)
        { return Vector(-v.x, -v.y, -v.z); }
    ```

  hoặc:
  - ```
    class Vector {
    public:
        Vector operator -() const
            // tham số chính là *this
            { return Vector(-x, -y, -z); }
    };
    ```
    ```

Chồng toán tử một ngôi *(tiếp)*

- Các hàm toán tử nếu khai báo ngoài lớp thường được khai báo là friend để sử dụng các biến ẩn

```
class Vector {
public:
    friend Vector operator -(const Vector& v);
};

Vector operator -(const Vector& v)
    { return Vector(-v.x, -v.y, -v.z); }
```

- Ví dụ sử dụng:

```
Vector v1(1.2, 2.3), v2;
v2 = -v1;
```

- Có thể gọi tường minh các hàm toán tử:

```
v2 = operator -(v1);   // hàm toán tử ngoài lớp
```

hoặc:

```
v2 = v1.operator -();  // hàm toán tử trong lớp
```

Toán tử ++ và --

- Hai toán tử này có thể dùng ở trước (tiền tố) hoặc sau (hậu tố). Để phân biệt, toán tử tiền tố được định nghĩa như bình thường, còn toán tử hậu tố có thêm tham số thứ hai với kiểu int (dù không dùng).
- Ví dụ định nghĩa trong lớp:

```
class LimitedNum {
private:
   int n, lim;
public:
   LimitedNum& operator ++() {     // tiền tố
       if (++n > lim) n = lim;
       return *this; }
   LimitedNum& operator ++(int) { // hậu tố
       return ++(*this); }
};
```

- Gọi hàm toán tử trực tiếp:

```
n.operator ++();   // gọi toán tử tiền tố
n.operator ++(0);  // gọi toán tử hậu tố
```

- Ghi chú: tương tự nếu hàm toán tử định nghĩa ngoài lớp

Toán tử chuyển kiểu

- Tương tự như các toán tử một ngôi khác, nhưng không cần khai báo kiểu trả về khi viết hàm toán tử (chỉ định nghĩa ở trong lớp)

```
class Fraction {
private:
   int a, b;
public:
   operator double() { return (double)a/(double)b; }
   operator string() { ... }
   operator const char*() { ... }
   ...
};
```

- Sử dụng:

```
Fraction f(4, 5);
double d = (double)f + 1.2;
string s(f);
strcpy(cstr, f);
```

- Chú ý phân biệt toán tử chuyển kiểu (chuyển tử lớp → kiểu khác) và constructor chuyển kiểu (chuyển từ kiểu khác → lớp)

Chồng toán tử hai ngôi

- Dùng hàm toán tử toàn cục với hai tham số, hoặc phương thức có một tham số trong một lớp
- Ví dụ:

```
Vector operator -(const Vector& v1, const Vector& v2)
    { return Vector(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z); }
```

hoặc:

```
class Vector {
public:
    Vector operator -(const Vector& v) const
        // tham số thứ nhất là *this
        { return Vector(x-v.x, y-v.y, z-v.z); }
};
```

- Ví dụ sử dụng:

```
v3 = v2-v1;
```

- Tương tự với toán tử một ngôi:
  - Thường khai báo các hàm toán tử hai ngôi ngoài lớp là friend để sử dụng biến ẩn
  - Có thể gọi tường minh các hàm toán tử hai ngôi

Toán tử so sánh

- Ví dụ:
  - ```
    class Vector {
    public:
       bool operator ==(const Vector& v) const // trong lớp
          { return x == v.x && y == v.y; }
       friend bool operator !=(const Vector&, const Vector&);
    };
    ```
  - ```
    // ngoài lớp:
    bool operator !=(const Vector& v1, const Vector& v2)
       { return !(v1==v2); }  // dùng lại toán tử ==
    ```
- Các toán tử so sánh khác có thể định nghĩa tương tự:

  `>  <  >=  <=`

Các toán tử gán

- Các toán tử gán chỉ có thể được định nghĩa trong lớp
  - ```
    class Complex {
    public:
       Complex& operator =(const Complex& c);
       Complex& operator =(double x);
       Complex& operator +=(const Complex& c);
       Complex& operator -=(const Complex& c);
       Complex& operator *=(double x);
    };
    ```
- Các toán tử gán khác có thể định nghĩa tương tự:

  `=  +=  -=  *=  /=  ^=  &=  |=  <<=  >>=`
    ```

Toán tử =

- Có một số điểm khác các toán tử gán khác:
  - Còn được coi là toán tử copy
  - Nếu không khai báo, có một toán tử copy mặc định được định nghĩa cho lớp với tham số cùng kiểu để copy các biến thành phần
  - Không được thừa kế bởi các lớp dẫn xuất (bị toán tử mặc định của lớp con che mất)
- Chú ý phân biệt với constructor copy
  - `Vector v2(v1), v3 = v2; // đều dùng constructor copy`
  - `v3 = v2;  // toán tử copy`
- Phân biệt với constructor chuyển kiểu
  - `string s1("12"), s2 = "ab"; // các cons chuyển kiểu`
  - `s2 = "xyz"; // toán tử copy`

Toán tử new, new[] và delete, delete[]

- Dùng để cấp phát bộ nhớ động
- Chú ý: việc gọi constructor, destructor là tự động, không thể can thiệp

```
class Obj {
public:
    void* operator new(size_t sz) {
        return malloc(sz);
    }
    void* operator new[](size_t sz) {
        return malloc(sz);
    }
    void operator delete(void* p) {
        free(p);
    }
    void operator delete[](void* p) {
        free(p);
    }
};
```

Các toán tử đặc biệt khác tự tìm hiểu thêm

- Toán tử gọi hàm: `p(x, y)`
- Toán tử chỉ số: `arr[i]`
- Toán tử phảy: `a, b`
- Toán tử tham chiếu: `*ptr`
- Toán tử lấy phần tử: `pnt->mem`
- Toán tử con trỏ tới thành phần (pointer to a member): `obj->*mem`
- Toán tử new có địa chỉ (placement new): `new (p)[n]`

cout, cin và toán tử xuất/nhập

- cout, cin là hai đối tượng thuộc các lớp ostream và istream. Các toán tử << và >> đã được định nghĩa chồng dùng để xuất/nhập.
- Ví dụ: (*)
  - ```
    ostream& operator <<(int x) {...}
    ostream& operator <<(float x) {...}
    ostream& operator <<(double x) {...}
    ostream& operator <<(char x) {...}
    ostream& operator <<(const char* s) {...}
    ...
    istream& operator >>(int& x) {...}
    istream& operator >>(float& x) {...}
    istream& operator >>(double& x) {...}
    istream& operator >>(char& x) {...}
    istream& operator >>(char* s) {...}
    ...
    ```

* Ví dụ ở đây chỉ mang tính chất minh hoạ. Trên thực tế các lớp ostream và istream được định nghĩa không hoàn toàn giống như ở đây. Xem thêm ở phần về STL.
    ```

Chồng toán tử << và >> để xuất/nhập

- Muốn các lớp mới tạo ra có thể dùng được với cout, cin thì định nghĩa chồng các toán tử này cho lớp đó

```
class Vector {
    // khai báo friend cho các toán tử
};

ostream& operator <<(ostream& s, const Vector& v) {
  s << '(' << v.x << ", " << v.y << ", " << v.z << ')';
  return s;
}
istream& operator >>(istream& s, Vector& v) {
  s >> v.x >> v.y >> v.z;
  return s;
}
```

- Sử dụng:

```
Vector v1, v2;
cout << "v1 = " << v1;
cin >> v2;
```

Bài tập

1. Định nghĩa đầy đủ các toán tử cho lớp Vector: cộng, trừ, nhân với số, tích vô hướng và có hướng
2. Định nghĩa các toán tử cho lớp Complex
3. Định nghĩa các toán tử cho lớp String: + (cộng chuỗi hoặc ký tự), chuyển kiểu, xuất/nhập, [ ] (lấy phần tử)
4. Viết một lớp BigInt để làm việc với các số lớn tuỳ ý và định nghĩa các toán tử cần thiết: +, -, *, /, ++, --, chuyển kiểu sang string/long long
5. Viết một lớp Array cho mảng động với các toán tử: += (thêm phần tử, nối hai mảng), [ ], chuyển kiểu
6. Viết một lớp Iterator để duyệt DSLK với toán tử ++ (tới phần tử tiếp theo), ! (kiểm tra đã ở cuối danh sách chưa), * (lấy giá trị tại vị trí hiện tại). Sau đó định nghĩa toán tử ~ (tạo đối tượng Iterator) với lớp LList. Mục tiêu là sau đó ta có thể duyệt DSLK như sau:

```
LList lst;
for (Iterator itr = ~lst; !itr; itr++) {
    int& data = *itr;
    // ...
}
```