Chương IV. NẠP CHỒNG TOÁN TỬ (OPERATOR OVERLOADING)

Table of Contents

1. Giới thiệu

- Các toán tử cho phép ta sử dụng cú pháp toán học đối với các kiểu dữ liệu của C++ thay vì gọi hàm (tuy bản chất vẫn là gọi hàm).

Ví dụ thay **a.set(b.add(c));** bằng **a = b + c;**

→ Gần với kiểu trình bày mà con người quen dùng

→ Đơn giản hóa mã chương trình

- C/C++ đã làm sẵn cho các kiểu cài sẵn (int, float…)

```
int a = 5;
int b = 4;
int c = a + b;  // = 9
```

- Đối với các kiểu dữ liệu người dùng tự định nghĩa: C++ cho phép định nghĩa các toán tử cho các thao tác đối với các kiểu dữ liệu người dùng.

→ Đó là operator overload, một toán tử có thể dùng cho nhiều kiểu dữ liệu

```
PhanSo ps1(1, 2);
PhanSo ps2(2, 3);
PhanSo ketQua = ps1 + ps2;
```

- Như vậy, ta có thể tạo các kiểu dữ liệu đóng gói hoàn chỉnh (fully-encapsulated) để kết hợp với ngôn ngữ như các kiểu dữ liệu cài sẵn

2. Nạp chồng toán tử là gì?

- Cũng tương tự như *nạp chồng hàm (overload function)*, bạn có thể định nghĩa nhiều hàm có cùng tên, nhưng khác tham số truyền vào, nạp chồng toán tử cũng tương tự.
- Nạp chồng toán tử (overload operator) là bạn *định nghĩa lại toán tử đã có trên kiểu dữ liệu người dùng tự định nghĩa để dễ dàng thể hiện các câu lệnh trong chương trình.*
- Ví dụ như bạn định nghĩa phép cộng cho kiểu dữ liệu phân số thì sẽ thực hiện cộng hai phân số rồi trả về một phân số mới. So với việc thực hiện gọi hàm, việc overload toán tử sẽ làm cho câu lệnh ngắn gọn, dễ hiểu hơn.

```
PhanSo ps1(1, 2);
PhanSo ps2(2, 3);
PhanSo ketQua;
// Dùng hàm
ketQua = ps1.cong(ps2);
// Dùng Overload operator
ketQua = ps1 + ps2;  // 7/6
```

- Cơ chế hoạt động

+ Về bản chất, việc thực hiện các toán tử cũng tương đương với việc gọi hàm, ví dụ:

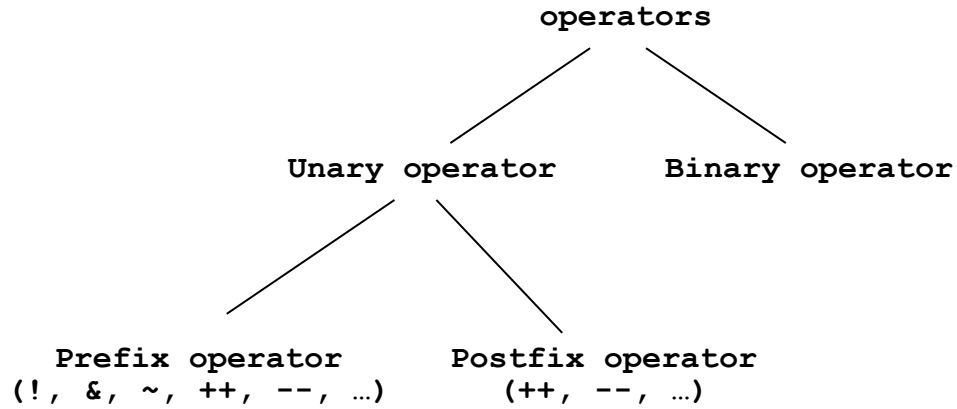
```
PhanSo a(1, 2);
PhanSo b(2, 3);
PhanSo ketQua = a + b;
// Tương đương với
PhanSo ketQua = a.cong(b);
```

+ Nếu bạn thực hiện toán tử trên hai toán hạng có kiểu dữ liệu cơ bản (float, double, int…), trình biên dịch sẽ tìm xem phiên bản nạp chồng toán tử nào phù hợp với kiểu dữ liệu đó và sử dụng, nếu không có sẽ báo lỗi.

+ Ngược lại nếu là kiểu dữ liệu tự định nghĩa như struct, class, trình biên dịch sẽ tìm xem có phiên bản nạp chồng toán tử nào phù hợp không? Nếu có thì sẽ sử dụng toán tử đó, ngược lại thì sẽ cố gắng chuyển đổi kiểu dữ liệu của các toán hạng đó sang kiểu dữ liệu có sẵn để thực hiện phép toán, không được sẽ báo lỗi.

3. Các toán tử của C++

- Các toán tử được chia thành hai loại theo số toán hạng nó chấp nhận
  - + **Toán tử đơn** nhận một toán hạng
  - + **Toán tử đôi** nhận hai toán hạng
- Các toán tử đơn lại được chia thành hai loại
  - + **Toán tử trước** đặt trước toán hạng
  - + **Toán tử sau** đặt sau toán hạng



- Một số toán tử đơn có thể được dùng làm cả toán tử trước và toán tử sau
  - Ví dụ phép tăng ("++") và phép giảm ("--")
- Một số toán tử có thể được dùng làm cả toán tử đơn và toán tử đôi
  - Ví dụ, "*" là toán tử đơn trong phép truy nhập con trỏ, là toán tử đôi trong phép nhân
- Toán tử chỉ mục ("[…]") là toán tử đôi, mặc dù một trong hai toán hạng nằm trong ngoặc
  - Phép lấy chỉ mục có dạng "arg1[arg2]"
- Các từ khoá "new" và "delete" cũng được coi là toán tử và có thể được định nghĩa lại (overload)

- Phần lớn các toán tử của C++ đều có thể overload được, bao gồm:

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

- Các toán tử C++ không cho phép overload

.	.*	::	:?	typeid	sizeof
const_cast		dynamic_cast		reinterpret_cast	static_cast

- Các hạn chế đối với việc overload toán tử

+ *Không thể tạo toán tử mới hoặc kết hợp các toán tử có sẵn theo kiểu mà trước đó chưa được định nghĩa.*
+ *Không thể thay đổi thứ tự ưu tiên của các toán tử*
+ *Không thể tạo cú pháp mới cho toán tử*
+ *Không thể định nghĩa lại một định nghĩa có sẵn của một toán tử*

Ví dụ: không thể thay đổi định nghĩa có sẵn của phép ("+") đối với hai số kiểu int

+ Như vậy, khi tạo định nghĩa mới cho một toán tử, **ít nhất một trong số các tham số (toán hạng) của toán tử đó phải là một kiểu dữ liệu người dùng**

- Lưu ý khi định nghĩa lại toán tử

+ Tôn trọng ý nghĩa của toán tử gốc, cung cấp chức năng mà người dùng mong đợi/chấp nhận

. không ai nghĩ rằng phép "+" sẽ in kết quả ra màn hình hoặc thực hiện phép chia

. Sử dụng "+" để nối hai xâu có thể hơi lạ đối với người chỉ quen dùng "+" cho các số, nhưng nó vẫn tuân theo khái niệm chung của phép cộng

+ Nên cho *kiểu trả về của toán tử khớp với định nghĩa cho các kiểu cài sẵn* (không nên trả về giá trị int từ phép so sánh == của mảng số, nên trả về bool)

+ Cố gắng tái sử dụng mã nguồn một cách tối đa

+ *Ta sẽ thường xuyên định nghĩa các toán tử sử dụng các định nghĩa có sẵn*

- Một số ràng buộc của phép toán

- Hầu hết các phép toán không ràng buộc ý nghĩa, chỉ một số trường hợp cá biệt như operator =, operator [], operator (), operator -> đòi hỏi phải được định nghĩa là hàm thành phần của lớp để toán hạng thứ nhất có thể là một đối tượng trái (lvalue).
- Ta phải chủ động định nghĩa phép toán +=, -=, *=,… dù đã định nghĩa phép gán và các phép toán +,-,*,…

- Lưu ý khi định nghĩa lại toán tử

- Tôn trọng ý nghĩa của toán tử gốc, cung cấp chức năng mà người dùng mong đợi/chấp nhận
- Cố gắng tái sử dụng mã nguồn một cách tối đa
- Trong ví dụ trên, ta định nghĩa hàm thành phần có tên đặc biệt bắt đầu bằng từ khóa operator theo sau bởi tên phép toán cần định nghĩa. Sau khi định nghĩa phép toán, ta có thể dùng theo giao diện tự nhiên

4. Cú pháp operator overloading

- Khai báo và định nghĩa toán tử thực chất không khác với việc khai báo và định nghĩa nghĩa một loại hàm bất kỳ nào khác

- Sử dụng tên hàm là "operator@" cho toán tử "@"

Ví dụ: để overload phép "+", ta dùng tên hàm "operator+"

- Số lượng tham số tại khai báo phụ thuộc hai yếu tố:

+ Toán tử là toán tử đơn hay đôi

- **Toán tử đơn**: Chỉ có một toán hạng (ví dụ: -, ++, --).
- **Toán tử đôi**: Có hai toán hạng (ví dụ: +, -, *, /).

+ Toán tử được khai báo là **hàm toàn cục** hay phương thức của lớp (hàm thành phần)

- *Khi định nghĩa phép toán bằng hàm toàn cục, số tham số bằng số ngôi. Phép toán 2 ngôi cần 2 tham số và phép toán một ngôi cần một tham số*
- *Khi định nghĩa phép toán bằng hàm thành phần, số tham số ít hơn số ngôi một vì đã có một tham số ngầm định là đối tượng gọi phép toán (toán hạng thứ nhất). Phép toán 2 ngôi cần 1 tham số và phép toán 1 ngôi không có tham số*

Giả sử aa, bb là tên biến thuộc lớp.

aa@bb	`Tên_lớp operator@(Tên_lớp& bb)`	`Tên_lớp operator@(Tên_lớp& aa, Tên_lớp& bb)`
@aa	`Tên_lớp operator@( )`	`Tên_lớp operator@(Tên lớp& aa)`
aa@	`Tên_lớp operator@(int)`	`Tên_lớp operator@(Tên_lớp& aa,int)`
	Phương thức của lớp	*Hàm toàn cục*

*** Hàm toàn cục**:

```
const MyNumber operator+(const MyNumber& num1, const MyNumber& num2);
```

"x+y" sẽ được hiểu là "operator+(x,y)"

*Dùng từ khoá **const** để đảm bảo các toán hạng gốc không bị thay đổi, không thay đổi trạng thái của đối tượng mà nó được gọi. Trong trường hợp này, phép cộng không làm thay đổi giá trị của hai số được cộng.*

*Ký tự **&** được sử dụng để truyền tham chiếu, thay vì truyền giá trị. Điều này có nghĩa là, thay vì tạo ra một bản sao của đối tượng, chúng ta chỉ truy cập đến đối tượng gốc. Điều này giúp tiết kiệm bộ nhớ và cải thiện hiệu suất, đặc biệt khi làm việc với các đối tượng lớn.*

* Khai báo toán tử dưới dạng thành viên của **MyNumber**. *Đối tượng chủ của phương thức được hiểu là toán hạng thứ nhất của toán tử, ta chỉ cần truyền vào một tham số cho hàm, chính là toán hạng thứ hai:*

```
const MyNumber operator+(const MyNumber& num) const;
```

"x+y" sẽ được hiểu là "x.operator+(y)"

** Mỗi từ khoá const đều có mục đích riêng:*

*+ **const trước MyNumber:** Điều này có nghĩa là phương thức operator+ sẽ trả về một đối tượng MyNumber không thể thay đổi. Điều này hữu ích khi bạn không muốn giá trị trả về bị thay đổi sau khi đã được tính toán.*

*+ **const sau danh sách tham số:** Điều này có nghĩa là phương thức operator+ không thể thay đổi trạng thái của đối tượng mà nó được gọi (trong trường hợp này, đối tượng num).*

*+ **const sau MyNumber& num:** Điều này có nghĩa là đối tượng num không thể bị thay đổi bởi phương thức operator+.*

Ví dụ: Sử dụng toán tử "+" để cộng hai đối tượng **MyNumber** và trả về kết quả là một **MyNumber**

Khai báo dưới dạng hàm toàn cục:

```
#include <iostream>
class MyNumber {
    int value;
public:
    MyNumber(int v) : value(v) {}
};

// Định nghĩa hàm toàn cục để nạp chồng toán tử +
const MyNumber operator+(const MyNumber& num1, const MyNumber& num2) {
    return MyNumber(num1.value + num2.value);
}

int main() {
    MyNumber a(10);
    MyNumber b(20);
    MyNumber c = a + b; // Sẽ gọi operator+(a, b)
    std::cout << c.value; // Kết quả sẽ là 30
}
```

Khai báo dưới dạng phương thức của lớp:

```
#include <iostream>
class MyNumber {
    int value;
public:
    MyNumber(int v) : value(v) {}

    // Nạp chồng toán tử + dưới dạng phương thức
    const MyNumber operator+(const MyNumber& num) const {
        return MyNumber(value + num.value);
    }
};

int main() {
    MyNumber a(10);
    MyNumber b(20);
    MyNumber c = a + b; // Sẽ gọi a.operator+(b)
    std::cout << c.value; // Kết quả sẽ là 30
}
```

- Khi có thể định nghĩa bằng hai cách, dùng hàm thành phần sẽ gọn hơn. Tuy nhiên chọn hàm thành phần hay hàm toàn cục hoàn toàn tùy theo sở thích của người sử dụng.
  - ❖ Dùng hàm toàn cục thuận tiện hơn khi ta có nhu cầu chuyển kiểu ở toán hạng thứ nhất.
  - ❖ Các phép toán **=, [ ], ()** bắt buộc phải được định nghĩa là hàm thành phần vì toán hạng thứ nhất phải là lvalue.
  - ❖ Nếu toán hạng thứ nhất không thuộc lớp đang xét thì phải định nghĩa bằng hàm toàn cục.

- Sau khi đã khai báo toán tử bị overload (là phương thức hay hàm toàn cục), cú pháp định nghĩa không có gì khó
  - + Định nghĩa toán tử dạng phương thức không khác với định nghĩa phương thức bất kỳ khác
  - + Định nghĩa toán tử dạng hàm toàn cục không khác với định nghĩa hàm toàn cục bất kỳ khác

- Tuy nhiên, có một số vấn đề liên quan đến hướng đối tượng (đặc biệt là tính đóng gói) mà ta cần xem xét

4.1 Định nghĩa các toán tử là hàm thành viên

- Để bắt đầu, xét định nghĩa toán tử bên trong giới hạn lớp
- Ví dụ: định nghĩa phép cộng:

```
class MyNumber {
    int value;
public:
    MyNumber(int v) : value(v) {}

    // Định nghĩa toán tử + là hàm thành viên
    const MyNumber operator+(const MyNumber& num) const;
};

const MyNumber MyNumber::operator+(const MyNumber& num) const
{
    // Cộng giá trị của đối tượng hiện tại với giá trị của đối tượng truyền vào
    // Sử dụng kết quả phép cộng để tạo ra một đối tượng MyNumber mới bằng cách tái sử dụng
constructor.
    MyNumber result(this->value + num.value);
    // Trả về một đối tượng MyNumber mới chứa kết quả của phép cộng
    return result;
}
```

+ Constructor cho **MyNumber** và định nghĩa có sẵn của phép cộng cho **int** được tái sử dụng

+ Tạo một đối tượng **MyNumber** sử dụng constructor với giá trị là tổng hai giá trị thành viên của các tham số tính bằng phép cộng đã có sẵn cho kiểu **int**.

- Ta sẽ lấy một ví dụ thường gặp khác là phép gán

4.2 Phép gán

- Một trong những toán tử hay được overload nhất

+ Cho phép gán cho đối tượng này một giá trị dựa trên một đối tượng khác

+ Copy constructor cũng thực hiện việc tương tự, cho nên, **định nghĩa toán tử gán gần như giống hệt định nghĩa của copy constructor**

- Ta có thể khai báo phép gán cho lớp **MyNumber** như sau:

```
const MyNumber& operator=(const MyNumber& num);
```

+ Phép gán nên luôn luôn trả về một tham chiếu tới đối tượng đích (đối tượng được gán trị cho)

+ ***Tham chiếu được trả về phải là const*** để tránh trường hợp **a** bị thay đổi bằng lệnh **"(a = b) = c;"** (lệnh đó không tương thích với định nghĩa gốc của phép gán)

```
class MyNumber {
  int value;
public:
    MyNumber(int v) : value(v) {}

    // Định nghĩa toán tử gán
    const MyNumber& operator=(const MyNumber& num) {
        if (this != &num) {  // Kiểm tra tự gán
            this->value = num.value;  // Gán giá trị
        }
        return *this;  // Trả về tham chiếu tới đối tượng hiện tại
    }
};
```

Kiểm tra tự gán (if (this != &num)):

- this là con trỏ tới đối tượng hiện tại.
- &num là địa chỉ của đối tượng num.
- Điều kiện this != &num đảm bảo rằng đối tượng hiện tại không bị gán cho chính nó. Điều này ngăn chặn các vấn đề có thể xảy ra khi sử dụng bộ nhớ động hoặc khi cần xử lý đặc biệt trong quá trình sao chép.

Gán giá trị (this->value = num.value;):

- Nếu đối tượng hiện tại không phải là đối tượng num, ta thực hiện gán giá trị value của num cho đối tượng hiện tại.

Trả về tham chiếu tới đối tượng hiện tại:

- Toán tử gán trả về một tham chiếu tới đối tượng hiện tại (*this). Điều này cho phép các lệnh gán có thể được xâu chuỗi, ví dụ như: a = b = c;.

- Định nghĩa trên có thể dùng cho phép gán

+ Lệnh **if** dùng để ngăn chặn các vấn đề có thể nảy sinh khi một đối tượng được gán cho chính nó (thí dụ khi sử dụng bộ nhớ động để lưu trữ các thành viên)

+ Ngay cả khi gán một đối tượng cho chính nó là an toàn, lệnh **if** trên đảm bảo không thực hiện các công việc thừa khi gán

- Khi nói về copy constructor, ta đã biết rằng C++ luôn cung cấp một copy constructor mặc định, nhưng nó chỉ thực hiện sao chép đơn giản (sao chép nông)
- Đối với phép gán cũng vậy

- Vậy, ta chỉ cần định nghĩa lại phép gán nếu:

+ Ta cần thực hiện phép gán giữa các đối tượng

+ Phép gán nông (memberwise assignment) không đủ dùng vì ta cần sao chép sâu - chẳng hạn sử dụng bộ nhớ động

Sử dụng bộ nhớ động: Nếu lớp của bạn quản lý tài nguyên như bộ nhớ động, bạn cần sao chép sâu (deep copy) thay vì sao chép nông (shallow copy) để tránh việc nhiều đối tượng cùng chia sẻ một vùng bộ nhớ.

+ Khi sao chép đòi hỏi cả tính toán - chẳng hạn gán một số hiệu có giá trị duy nhất hoặc tăng số đếm

4.3 Định nghĩa các toán tử là hàm toàn cục

Quay lại với ví dụ về phép cộng cho **`MyNumber`**, ta có thể khai báo hàm định nghĩa phép cộng tại mức toàn cục:

```
const MyNumber operator+(const MyNumber& num1, const MyNumber& num2);
```

Khi đó, ta có thể định nghĩa toán tử đó như sau:

```
const MyNumber operator+(const MyNumber& num1,
                         const MyNumber& num2) {
   MyNumber result(num1.value + num2.value);
   return result;
}
```

Truy nhập các thành viên private `value`

Ở đây có vấn đề….

*** Làm việc với tính đóng gói**

- Rắc rối: **hàm toàn cục muốn truy nhập thành viên private của lớp**
  - thông thường: không được quyền
  - đôi khi bắt buộc phải overload bằng hàm toàn cục
  - **không thể hy sinh tính đóng gói của hướng đối tượng để chuyển các thành viên private thành public**
- Giải pháp là dạng cuối cùng của quyền truy nhập: **`friend`**

5. Friend & Hàm friend

- Khái niệm **friend** cho phép một lớp cấp quyền truy nhập tới các phần nội bộ của lớp đó cho một số cấu trúc được chọn
- C++ có 3 kiểu friend
  - *Hàm friend (trong đó có các toán tử được overload)*
  - *Lớp friend*
  - *Phương thức friend (hàm thành viên)*
- Các tính chất của quan hệ **friend**
  - Phải được cho, không tự nhận
  - Không đối xứng
  - Không bắc cầu

Hàm friend

- Khi khai báo một hàm bên ngoài là friend của một lớp, hàm đó được cấp quyền truy nhập *tương đương* quyền của các phương thức của lớp đó
  - ⇨ Như vậy, một hàm friend có thể truy nhập cả các **thành viên private và protected** của lớp đó
- Để khai báo một hàm là friend của một lớp, ta phải khai báo hàm đó bên trong khai báo lớp và đặt từ khoá **friend** lên đầu khai báo. Ví dụ:

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      ...
      friend const MyNumber operator+(const MyNumber& num1, const MyNumber& num2);
   ...
};
```

- Lưu ý: tuy khai báo của hàm friend được đặt trong khai báo lớp và hàm đó có quyền truy nhập ngang với các phương thức của lớp, **hàm đó *không phải* phương thức của lớp**
- Không cần thêm sửa đổi gì cho định nghĩa của hàm đã được khai báo là **friend**.
  - Định nghĩa trước của phép cộng vẫn giữ nguyên

```
const MyNumber operator+(const MyNumber& num1, const MyNumber& num2) {
   MyNumber result(num1.value + num2.value);
   return result;
}
```

6. Tại sao sử dụng toán tử toàn cục

- Đối với toán tử được khai báo là phương thức của lớp, đối tượng chủ (xác định bởi con trỏ **this**) luôn được hiểu là toán hạng đầu tiên (trái nhất) của phép toán.
  - Nếu muốn dùng cách này, ta phải được quyền bổ sung phương thức vào định nghĩa của lớp/kiểu của toán hạng trái
- Không phải lúc nào cũng có thể overload toán tử bằng phương thức
  - phép cộng giữa MyNumber và int cần cả hai cách
    MyNumber + int và **int + MyNumber**
  - **cout** << obj;
  - không thể sửa định nghĩa kiểu **int** hay kiểu của **cout**

⇨ lựa chọn duy nhất: overload toán tử bằng hàm toàn cục

Ví dụ:

- Cài đặt với hàm cục bộ

+ Đối với hàm cục bộ hay còn gọi là phương thức của lớp, số tham số sẽ ít hơn hàm toàn cục một tham số vì tham số đầu tiên mặc định chính là đối tượng gọi phương thức (toán hạng đầu tiên). Vậy, đối với toán tử hai ngôi, ta chỉ cần truyền vào một tham số cho hàm, chính là toán hạng thứ hai. Ví dụ:

```
class PhanSo {
private:
	int tu;
	int mau;
public:
	PhanSo() { tu = 0; mau = 1; }
	PhanSo(int a, int b) { tu = a; mau = b; }
	PhanSo operator+(const PhanSo& ps) { // overload toán tử +
		PhanSo kq;
		kq.tu = this->tu * ps.mau + ps.tu * this->mau;
		kq.mau = this->mau * ps.mau;
		return kq;
	}
};
```

+ Sau khi overload toán tử, bạn có thể sử dụng nó trên kiểu dữ liệu bạn đã định nghĩa:

```
PhanSo ps1(1, 2);
PhanSo ps2(2, 3);
PhanSo ps3 = ps1 + ps2; // ps3 = ps1.operator+(ps2)
```

+ Giờ chúng ta hãy xem một ví dụ khác, overload toán tử cộng một phân số với một số nguyên:

```
PhanSo operator+(const int& i) {
	PhanSo kq;
	kq.tu = this->tu + i * this->mau;
	return kq;
}
// Sử dụng
PhanSo ps1(1, 2);
PhanSo ps2 = ps1 + 2; // ps2 = ps1.operator+(2)
```

Do toán tử overload theo cách này là phương thức, được gọi từ một đối tượng, do đó mặc định toán hạng đầu tiên phải là toán hạng có kiểu dữ liệu của lớp đó, điều này cũng có nghĩa là bạn phải đặt toán hạng có kiểu dữ liệu của lớp đó đầu tiên rồi mới đến toán hạng tiếp theo. Và đối với các kiểu dữ liệu có sẵn, ta không thể truy cập vào các lớp định nghĩa nên chúng và overload operator của chúng được. Vậy để giải quyết điều này thì làm như thế nào? Ta sẽ sử dụng hàm toàn cục.

- Cài đặt hàm toàn cục

+ Thay vì để toán hạng đầu tiên luôn phải có kiểu là một lớp đối tượng, chúng ta sẽ sử dụng hàm bạn để có thể tự do lựa chọn thứ tự của các toán hạng. Ví dụ như bạn muốn **1 + ps1**, **ps1 + 1** đều được chứ không nhất thiết phải là **ps1 + 1** nữa. Chúng ta cài đặt với hàm bạn tương tự như sau:

```
class PhanSo {
	//...
	friend PhanSo operator+(const PhanSo& ps, const int& i);
	friend PhanSo operator+(const int& i, const PhanSo& ps);
	// đổi chỗ thứ tự toán hạng bằng cách đổi thứ tự tham số
};
PhanSo operator+(const PhanSo& ps, const int& i) {
	PhanSo kq;
	kq.tu = ps.tu + i * ps.mau;
	return kq;
}
PhanSo operator+(const int& i, const PhanSo& ps) {
	return ps + i;
}
```

+ Lúc này ta có thể thực hiện các phép toán sau:

```
PhanSo a(2, 3);
a + 5; // operator+(a,5)
5 + a; // operator+(5,a)
```

Nhưng vẫn còn 1 nhược điểm đó là ta phải nạp chồng operator+ nhiều lần. Vấn đề này sẽ được giải quyết bằng phương pháp **chuyển kiểu. (Đề cập sau)**

7. Phép nhập và xuất (chèn & tách) (“<<” và “>>”)

- << và >> là hai phép toán thao tác trên từng bit khi các toán hạng là số nguyên.
- C++ định nghĩa lại hai phép toán để dùng với các đối tượng thuộc lớp ostream và istream để thực hiện các thao tác xuất, nhập.
- Lớp ostream (dòng dữ liệu xuất) định nghĩa phép toán << áp dụng cho các kiểu dữ liệu cơ bản (số nguyên, số thực, char*,…)
- Lớp istream (dòng dữ liệu nhập) định nghĩa phép toán >> áp dụng cho các kiểu dữ liệu cơ bản (số nguyên, số thực, char*,…)

Phép dịch bit được ký hiệu: >> (dịch phải) hoặc << (dịch trái)(trong c++) shl(dịch trái); shr(dịch phải) Ví dụ: 5 >> 1 = 2(5 shr 1); 2 >> 1 = 1(2 shr 1); 1 >> 1 = 0;
Giải thích: 5b = 0101 sau khi dịch 1 trở thành 0010 (=2d) và cứ tiếp tục như vậy.

- Khi định nghĩa hai phép toán trên, cần thể hiện ý nghĩa sau:

```
a >> b;             //bỏ a vào b
a << b;             //bỏ b vào a
cout << a << "\n"; // bỏ a và "\n" vào cout
cin >> a >> b;      // bỏ cin vào a và b
```

❖ cout, cerr là các biến thuộc lớp ostream đại diện cho thiết bị xuất chuẩn (mặc nhiên là màn hình) và thiết bị báo lỗi chuẩn (luôn luôn là màn hình).

```
class ostream : virtual public ios {
public:
	// Formatted insertion operations
	ostream& operator<< (signed char);
	ostream& operator<< (unsigned char);
	ostream& operator<< (int);
	ostream& operator<< (unsigned int);
	ostream& operator<< (long);
	ostream& operator<< (unsigned long);
	ostream& operator<< (float);
	ostream& operator<< (double);
	ostream& operator<< (const signed char*);
	ostream& operator<< (const unsigned char*);
	ostream& operator<< (void*);
	// ...
private:
	//data ...
};
```

❖ cin là một đối tượng thuộc lớp istream đại diện cho thiết bị nhập chuẩn, mặc nhiên là bàn phím.

```
class istream : virtual public ios {
public:
	istream& getline(char*, int, char = '\n');
	istream& operator>> (signed char*);
	istream& operator>> (unsigned char*);
	istream& operator>> (unsigned char&);
	istream& operator>> (signed char&);
	istream& operator>> (short&);
	istream& operator>> (int&);
	istream& operator>> (long&);
	istream& operator>> (unsigned short&);
	istream& operator>> (unsigned int&);
	istream& operator>> (unsigned long&);
	istream& operator>> (float&);
	istream& operator>> (double&);
private:
	// data...
};
```

**** Thông thường ta khai báo hai phép toán trên là hàm bạn của lớp để có thể truy xuất dữ liệu trực tiếp.***

7.1 Phép chèn ("<<")

- Xét ví dụ:

```
cout << num;      // num là đối tượng thuộc lớp MyNumber
```

- Toán hạng trái **cout** thuộc lớp **ostream**, không thể sửa định nghĩa lớp này nên ta overload bằng hàm toàn cục
  - *Tham số thứ nhất:* tham chiếu tới **ostream (như cout); Bởi vì bạn muốn in ra đối tượng MyNumber, os sẽ là cout.**
  - *Tham số thứ hai:* kiểu (tham chiếu tới đối tượng) **MyNumber**; **const**: do không có lý do gì để sửa đối tượng được in ra.

- *Giá trị trả về:* tham chiếu tới **ostream** (để thực hiện được `cout << num1 << num2;`)

- Kết luận: `ostream& operator<<(ostream& os, const MyNumber& num)`

(Một đối tượng thuộc lớp istream hoặc ostream thì không sao chép được nên phải sử dụng tham chiếu)

- Khai báo toán tử được overload là **friend** của lớp **MyNumber**

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      ...
      friend ostream& operator<<(ostream& os, const MyNumber& num);
      ...
};
```

- *Định nghĩa toán tử*

```
ostream& operator<<(ostream& os, const MyNumber& num) {
   os << num.value;  // Use version of insertion operator defined for int
   return os;        // Return a reference to the modified stream
};
```

Ví dụ:

```
#include <iostream>
using namespace std;

class MyNumber {
    int value;
public:
    MyNumber(int v) : value(v) {}

    // Định nghĩa toán tử << là hàm toàn cục
    friend ostream& operator<<(ostream& os, const MyNumber& num);
};

// Định nghĩa hàm toàn cục nạp chồng toán tử <<
ostream& operator<<(ostream& os, const MyNumber& num) {
    os << num.value;  // In ra giá trị của đối tượng MyNumber
    return os;  // Trả về ostream để có thể xâu chuỗi lệnh
}

int main() {
    MyNumber num(42);
    cout << num;  // Sẽ in ra "42"
    return 0;
}
```

- Tuỳ theo độ phức tạp của lớp được chuyển sang chuỗi ký tự, định nghĩa của toán tử này có thể dài hơn
- Toán tử tách (">>") được overload tương tự, tuy nhiên, định nghĩa thường phức tạp hơn, do có thể phải xử lý input để kiểm tra tính hợp lệ tuỳ theo cách ta quy định như thế nào khi in một đối tượng ra thành một chuỗi ký tự

7.2 Phép tách (">>")

- Để định nghĩa phép toán >> theo nghĩa nhập từ dòng dữ liệu nhập cho kiểu dữ liệu đang định nghĩa:
  - Ta định nghĩa phép toán >> như **hàm toàn cục** với
    - ***Tham số thứ nhất*** là tham chiếu đến một đối tượng thuộc lớp istream
    - ***Kết quả trả về*** là tham chiếu đến chính istream đó.
    - ***Toán hạng thứ hai*** là tham chiếu đến đối tượng thuộc lớp đang định nghĩa; *nên là một* ***tham chiếu hằng*** *nhằm tránh việc phải thực hiện sao chép quá nhiều (tốn tài nguyên) đồng thời vẫn đảm bảo đối số truyền vào sẽ không bị thay đổi.*

```
istream& operator>>(istream& is, const MyNumber& num)
```

Ví dụ:

```
#include <iostream>
using namespace std;

class MyNumber {
    int value;

public:
    MyNumber(int v = 0) : value(v) {}
    int getValue() {
        return value;
    }

    // Định nghĩa toán tử >> là hàm toàn cục
    friend istream& operator>>(istream& is, MyNumber& num);
};

// Định nghĩa hàm toàn cục nạp chồng toán tử >>
istream& operator>>(istream& is, MyNumber& num) {
    is >> num.value;  // Nhập giá trị vào cho đối tượng MyNumber
    return is;  // Trả về istream để có thể xâu chuỗi lệnh
}

int main() {
    MyNumber num;
    cout << "Enter a number: ";
    cin >> num;  // Sẽ gọi operator>> để nhập giá trị cho num
    cout << "You entered: " << num.getValue() << endl;  // In ra giá trị vừa nhập
    return 0;
}
```

Kết quả:

Enter a number: 21

You entered: 21

** Hàm toàn cục cần thiết khi nạp chồng toán tử << (hoặc >>) để sử dụng với các đối tượng của lớp tự định nghĩa, vì lý do sau:*

Toán hạng trái của toán tử << thuộc về lớp ostream:

- Toán tử << trong lệnh như cout << num; là một phương thức thành viên của lớp ostream.
- cout là một đối tượng của lớp ostream, được định nghĩa trong thư viện chuẩn C++.
- Bạn không thể sửa đổi hoặc thêm phương thức mới vào lớp ostream, vì đây là một lớp đã được định nghĩa sẵn trong thư viện C++.

Phương thức thành viên chỉ có thể được định nghĩa trong lớp mà nó thuộc về:

- Nếu bạn muốn định nghĩa toán tử << như một phương thức thành viên, bạn phải định nghĩa nó trong lớp ostream. Nhưng như đã nói, bạn không thể thay đổi lớp ostream.
- Trong khi đó, lớp MyNumber của bạn không biết gì về ostream, và toán tử << trong lớp này sẽ không có tham số ostream.

Hàm toàn cục không thuộc về bất kỳ lớp nào:

- Bằng cách nạp chồng toán tử << như một **hàm toàn cục** (ngoài lớp), bạn có thể định nghĩa hàm này với tham số đầu tiên là một tham chiếu tới ostream và tham số thứ hai là một tham chiếu tới đối tượng MyNumber.
- Điều này cho phép bạn "kết hợp" cả ostream và MyNumber trong cùng một hàm, mà không cần thay đổi định nghĩa của ostream hay MyNumber.

Cho phép xâu chuỗi các lệnh:

- Hàm toàn cục trả về một tham chiếu tới ostream (thường là cout), cho phép bạn xâu chuỗi nhiều lệnh << với nhau như cout << num1 << num2;.
- Điều này sẽ không dễ dàng thực hiện nếu toán tử << được định nghĩa như một phương thức thành viên của MyNumber, vì khi đó hàm sẽ trả về một đối tượng MyNumber thay vì ostream.

- *Phép toán << và >> cũng có thể được định nghĩa với toán hạng thứ nhất thuộc lớp đang xét, không thuộc lớp ostream hoặc istream.*
- *Trong trường hợp đó, ta dùng hàm thành phần. Kiểu trả về là chính đối tượng ở vế trái để có thể thực hiện phép toán liên tiếp.*
- *Các ví dụ về sử dụng phép toán trên theo cách này là các lớp Stack, Tập hợp, Danh sách, Mảng, Tập tin…*

8. Phép tăng & phép giảm ("++" & "--")

- ++ là phép toán một ngôi có vai trò tăng giá trị một đối tượng lên giá trị kế tiếp. Tương tự -- giảm giá trị một đối tượng xuống giá trị trước đó.
- ++ và -- chỉ áp dụng cho các ***kiểu dữ liệu đếm được***, nghĩa là mỗi giá trị của đối tượng ***đều có giá trị kế tiếp hoặc giá trị trước đó.***
- ++ và -- có thể được dùng theo hai cách, tiếp đầu ngữ (tăng/giảm trước) hoặc tiếp vị ngữ (tăng/giảm sau).

8.1 Phép tăng ("++")

`@aa`	➔ `aa.operator@( )`	hoặc	`operator@(aa)`
`aa@`	➔ `aa.operator@(int)`	hoặc	`operator@(aa,int)`

- Khi gặp phép tăng trong một lệnh, trình biên dịch sẽ sinh một trong 4 lời gọi hàm trên, tuỳ theo toán tử là toán tử trước (prefix) hay toán tử sau (postfix), là phương thức hay hàm toàn cục.
- Giả sử ta overload phép tăng dưới dạng phương thức của **MyNumber** và overload cả hai dạng đặt trước và đặt sau
  - Nếu gặp biểu thức dạng "++**x**", trình biên dịch sẽ sinh lời gọi `MyNumber::operator++()`
  - Nếu gặp biểu thức dạng "**x**++", trình biên dịch sẽ sinh lời gọi `MyNumber::operator++(int)`
  - Tham số **int** chỉ dành để phân biệt danh sách tham số của hai dạng prefix và postfix ***(Tham số giả - dummy parameter. Nó không được sử dụng bên trong thân hàm, mà chỉ đóng vai trò làm dấu hiệu phân biệt giữa hai phiên bản của toán tử tăng.)***

- **Toán tử tăng trước (prefix): ++num**
  - Prototype: `MyNumber& MyNumber::operator++();`
  - Giá trị trả về:
    - trả về tham chiếu đến đối tượng hiện tại (**MyNumber &**)
    - giá trị trái - lvalue (có thể được gán trị)
  - Khi bạn sử dụng ++x, giá trị của x sẽ được tăng lên trước và sau đó trả về tham chiếu đến x. Điều này có nghĩa là bạn có thể tiếp tục gán giá trị cho x sau khi tăng.

- **Toán tử tăng sau (postfix): num++**
  - Prototype: `const MyNumber MyNumber::operator++(int);`
  - Giá trị trả về:
    - **trả về giá trị (giá trị cũ trước khi tăng)**
    - trả về đối tượng tạm thời chứa giá trị cũ **(const MyNumber).**
    - giá trị phải - rvalue (không thể làm đích của phép gán)
  - Khi bạn sử dụng x++, giá trị hiện tại của x sẽ được trả về trước, sau đó x mới được tăng lên. Vì vậy, toán tử tăng sau trả về một đối tượng tạm thời chứa giá trị cũ của x.

- Nhớ lại rằng phép tăng trước tăng giá trị trước khi trả kết quả, trong khi phép tăng sau trả lại giá trị trước khi tăng
- Ta định nghĩa từng phiên bản của phép tăng như sau:

```
MyNumber& MyNumber::operator++() { // Prefix
   this->value++;        // Increment value
   return *this;         // Return current MyNumber
}

const MyNumber MyNumber::operator++(int) { // Postfix
   MyNumber before(this->value); // Create temporary MyNumber
                                 // with current value
   this->value++;        // Increment value
   return before;        // Return MyNumber before increment
}
```

before là một đối tượng địa phương của phương thức và sẽ chấm dứt tồn tại khi lời gọi hàm kết thúc. Khi đó, tham chiếu tới nó trở thành bất hợp lệ

Không thể trả về tham chiếu

** Cách khác để viết toán tử tăng:*

```
MyNumber& MyNumber::operator++() {
    ++value;  // Tăng giá trị trước khi sử dụng
    return *this;
}
```

```
MyNumber MyNumber::operator++(int) {
    MyNumber temp = *this;  // Tạo một bản sao tạm thời của đối tượng hiện tại
    ++value;  // Tăng giá trị sau khi đã lưu lại giá trị cũ
    return temp;  // Trả về bản sao tạm thời chứa giá trị cũ
}
```

Ở phiên bản tiền tố (tăng trước) (++a):

+ Con trỏ **this** ở đây giữ **địa chỉ của đối tượng đang gọi phương thức**, **return *this** tức là trả về đối tượng đang gọi thực hiện phương thức. Kiểu dữ liệu trả về là tham chiếu chỉ để tránh phải thực hiện sao chép nhiều lần.

+ Trả về tham chiếu giúp cho toán tử tăng trước có thể tiếp tục sử dụng đối tượng này trong các biểu thức khác, ví dụ như trong ++x + y.

Ví dụ 1:

```cpp
#include <iostream>
using namespace std;

class MyNumber {

    int value;
public:
    MyNumber(int v = 0) : value(v) {}
    int getValue() {
        return value;
    }
    // Tăng trước (prefix)
    MyNumber& operator++() {
        ++value;  // Tăng giá trị thành viên
        return *this;  // Trả về tham chiếu đến đối tượng hiện tại
    }

    // Tăng sau (postfix)
    const MyNumber operator++(int) {
        MyNumber temp = *this;  // Lưu lại giá trị cũ
        ++value;  // Tăng giá trị thành viên
        return temp;  // Trả về đối tượng tạm thời chứa giá trị cũ
    }
};

int main() {
    MyNumber num(5);

    // Tăng trước
    ++num;
    cout << "After prefix increment (++num): " << num.getValue() << endl;  // Kết quả là 6

    // Tăng sau
    num++;
    cout << "After postfix increment (num++): " << num.getValue() << endl;  // Kết quả là 7

    return 0;
}
```

Ví dụ 2:

```cpp
#include <iostream>
using namespace std;

class MyNumber {
    int value;
public:
    MyNumber(int v = 0) : value(v) {}
    int getValue() {
        return value;
    }

    const MyNumber operator++(int) { // Postfix
        MyNumber before(this->value); // Tạo đối tượng tạm thời với giá trị hiện tại
        this->value++;    // Tăng giá trị của đối tượng hiện tại
        return before;    // Trả về đối tượng tạm thời với giá trị trước khi tăng
    }

};
```

```
int main() {
    MyNumber x(5);
    MyNumber y = x++; // y sẽ nhận giá trị cũ của x, trước khi x bị tăng
    cout << y.getValue();// Kết quả là 5
    return 0;
}
```

* Khi sử dụng từ khóa const trước kiểu trả về MyNumber, điều này có nghĩa là **đối tượng tạm thời được trả về không thể bị thay đổi sau khi trả về**. Điều này giúp ngăn chặn việc vô tình sửa đổi đối tượng tạm thời, ví dụ như trong các biểu thức phức tạp.
* Ở ví dụ trên, **y sẽ nhận giá trị 5 (giá trị trước khi tăng)**, và **x sẽ trở thành 6.**
    - **Trước khi tăng**: `x` có giá trị là 5.
    - **Tạo đối tượng tạm thời `before`**: `before` được tạo ra với giá trị 5.
    - **Tăng giá trị của `x`**: Sau đó `x` được tăng lên 6.
    - **Trả về `before`**: `y` nhận giá trị của `before`, tức là 5.
* Việc sử dụng const đảm bảo rằng đối tượng tạm thời before không thể bị thay đổi sau khi trả về.

Nếu trong hàm main ta thay đổi như sau:

```
int main() {
    MyNumber x(5);
    x++;  // Tăng giá trị của x, nhưng không gán kết quả cho biến khác
    cout << x.getValue();  // Giả sử có hàm getValue() để lấy giá trị
    return 0;
}
```

Lệnh x++:
- Giá trị ban đầu của x là 5.
- x++ đầu tiên ghi nhớ giá trị 5 trong một bản sao tạm thời.
- Sau đó, x được tăng lên thành 6.
- Bản sao tạm thời với giá trị 5 bị bỏ qua nếu không được gán cho biến khác.

Khi bạn gọi x.getValue():
- Hàm getValue() sẽ trả về giá trị hiện tại của x, tức là 6, vì x đã được tăng trong lệnh trước đó.

Cách hoạt động của x++:

Tăng sau (x++):
- Toán tử tăng sau hoạt động bằng cách đầu tiên ghi nhớ giá trị hiện tại của x.
- Sau đó, x được tăng lên.
- Cuối cùng, giá trị trước khi tăng (bản sao tạm thời) được trả về nếu nó được gán hoặc sử dụng ở một nơi khác.

Khi bạn sử dụng trực tiếp x++:
- Khi bạn gọi x++, giá trị của x được tăng ngay lập tức sau khi giá trị ban đầu được ghi nhớ trong một đối tượng tạm thời.
- Sau khi thực hiện phép tăng, x sẽ có giá trị mới (tức là x + 1).
- Nếu bạn sử dụng x ngay sau khi x++ mà không gán kết quả của x++ vào biến khác, bạn đang thực sự truy cập giá trị mới của x sau khi nó đã được tăng.

Sự khác biệt giữa việc trả về tham chiếu trong toán tử tăng trước (++a) và không trả về tham chiếu trong toán tử tăng sau (a++) xuất phát từ bản chất và mục đích của từng toán tử:

Toán tử tăng trước (++a):

- **Cách hoạt động**: Toán tử tăng trước *tăng giá trị của đối tượng rồi trả về đối tượng đã được tăng.*
- **Trả về tham chiếu**:
  - *Toán tử tăng trước trả về tham chiếu đến đối tượng hiện tại (bản thân đối tượng đó sau khi đã được tăng giá trị).*
  - Việc trả về tham chiếu cho phép bạn sử dụng đối tượng đã tăng giá trị này trong các biểu thức khác mà không cần tạo bản sao mới.
  - Trả về tham chiếu cũng giúp tránh việc tạo ra một đối tượng tạm thời không cần thiết, tiết kiệm tài nguyên.
  - Cho phép các phép gán liên tiếp như ++a = b;, trong đó a được tăng giá trị trước rồi sau đó gán giá trị b cho a.

Toán tử tăng sau (a++):

- **Cách hoạt động**: Toán tử tăng sau *trả về giá trị hiện tại của đối tượng (trước khi tăng) và sau đó mới tăng giá trị của đối tượng.*
- **Trả về giá trị (không phải tham chiếu)**:
  - Toán tử tăng sau phải trả về giá trị của đối tượng trước khi nó được tăng, vì vậy nó trả về một bản sao của đối tượng trước khi thực hiện việc tăng giá trị.
  - Không thể trả về tham chiếu trong trường hợp này vì bạn cần giữ lại giá trị ban đầu để trả về, trong khi đối tượng gốc sẽ bị tăng giá trị sau đó.
  - Giữ nguyên giá trị cũ trong một bản sao tạm thời để sử dụng trong biểu thức, trước khi tăng giá trị thực của đối tượng gốc.

8.2 Phép giảm ("--")

- Toán tử giảm (--) trong C++ cũng có thể được nạp chồng tương tự như toán tử tăng (++). Nó cũng có hai dạng: **giảm trước (prefix)** và **giảm sau (postfix)**. Cách hoạt động của toán tử giảm rất giống với toán tử tăng, chỉ khác ở chỗ giá trị được giảm thay vì tăng.

- Cú pháp nạp chồng toán tử giảm

Toán tử giảm trước (prefix): --x (giảm giá trị trước khi trả kết quả).

- *Prototype:* `MyNumber& MyNumber::operator--();`
- *Giá trị trả về:* Tham chiếu đến đối tượng hiện tại (MyNumber&).
- Khi bạn sử dụng --x, giá trị của x sẽ được giảm trước và sau đó trả về tham chiếu đến x. Điều này có nghĩa là bạn có thể tiếp tục gán giá trị cho x sau khi giảm.

```
MyNumber& MyNumber::operator--() {
    --value;  // Giảm giá trị thành viên
    return *this;  // Trả về tham chiếu đến đối tượng hiện tại
}
```

Toán tử giảm sau (postfix): x-- (trả lại giá trị cũ trước khi giảm).

- *Prototype:* `const MyNumber MyNumber::operator--(int);`
- *Giá trị trả về:* Đối tượng tạm thời chứa giá trị cũ (const MyNumber).
- Khi bạn sử dụng x--, giá trị hiện tại của x sẽ được trả về trước, sau đó x mới được giảm. Toán tử giảm sau trả về một đối tượng tạm thời chứa giá trị cũ của x.

```
const MyNumber MyNumber::operator--(int) {
    MyNumber temp = *this;  // Lưu lại giá trị cũ
    --value;  // Giảm giá trị thành viên
    return temp;  // Trả về đối tượng tạm thời chứa giá trị cũ
}
```

Ví dụ:

```
#include <iostream>
using namespace std;

class MyNumber {
    int value;
public:
    MyNumber(int v = 0) : value(v) {}
    int getValue() {
        return value;
    }
    // Giảm trước (prefix)
    MyNumber& operator--() {
        --value;  // Giảm giá trị thành viên
        return *this;  // Trả về tham chiếu đến đối tượng hiện tại
    }

    // Giảm sau (postfix)
    const MyNumber operator--(int) {
        MyNumber temp = *this;  // Lưu lại giá trị cũ
        --value;  // Giảm giá trị thành viên
        return temp;  // Trả về đối tượng tạm thời chứa giá trị cũ
    }
};

int main() {
    MyNumber num(5);

    // Giảm trước
    --num;
    cout << "After prefix decrement (--num): " << num.getValue() << endl;  // Kết quả là 4

    // Giảm sau
    num--;
    cout << "After postfix decrement (num--): " << num.getValue() << endl;  // Kết quả là 3

    return 0;
}
```

**** Ví dụ tổng hợp quá tải toán tử***

```
class Fraction
{
private:
      int numerator;
      int denominator;
public:
      Fraction(int num, int denom);
      //Fraction add (const Fraction &f) ;
      Fraction operator+(const Fraction& other) const;
      Fraction operator+(int val) const;
      Fraction& operator+=(const Fraction& other);
      Fraction& operator+=(int val);

      int getNum() const { return numerator; }
      int getDenom() const { return denominator; }
      ~Fraction(void);
};
```

```
Fraction Fraction::operator+(const Fraction& other) const
{
	// Tìm bội số chung nhỏ nhất của mẫu số
	int lcm = other.denominator;
	while (lcm % denominator != 0) {
		lcm += other.denominator;
	}
		// Return a new fraction
		return Fraction(
			(lcm / denominator) * numerator +
			(lcm / other.denominator) * other.numerator, lcm);
}

Fraction Fraction::operator+(int val) const
{

	// Return a new fraction
	return Fraction(numerator + denominator * val, denominator);
}

Fraction& Fraction:: operator +=(const Fraction& other)
{
	Fraction temp(*this);
	// su dung toan tu "+" da dinh nghia
	temp = temp + other;
	numerator = temp.numerator;
	denominator = temp.denominator;

	// Return a reference to ourself
	return *this;
}
Fraction& Fraction:: operator+=(int val) {

	// su dung toan tu "+" da dinh nghia
	Fraction temp(*this);
	temp = temp + val;

	numerator = temp.numerator;
	denominator = temp.denominator;

	// Return a: reference to ourself
	return *this;
}
```

9. Các tham số và kiểu trả về

- Cũng như khi overload các hàm khác, khi overload một toán tử, ta cũng có nhiều lựa chọn về việc truyền tham số và kiểu trả về
  - chỉ có hạn chế rằng ***ít nhất một trong các tham số phải thuộc kiểu người dùng tự định nghĩa***
- Ở đây, ta có một số lời khuyên về các lựa chọn:

✍ **Các toán hạng:**

- **Nên sử dụng tham chiếu** mỗi khi có thể (đặc biệt là khi làm việc với các đối tượng lớn)
- Luôn luôn **sử dụng tham số là hằng tham chiếu** khi đối số sẽ không bị sửa đổi

```
bool String::operator==(const String &right) const
```

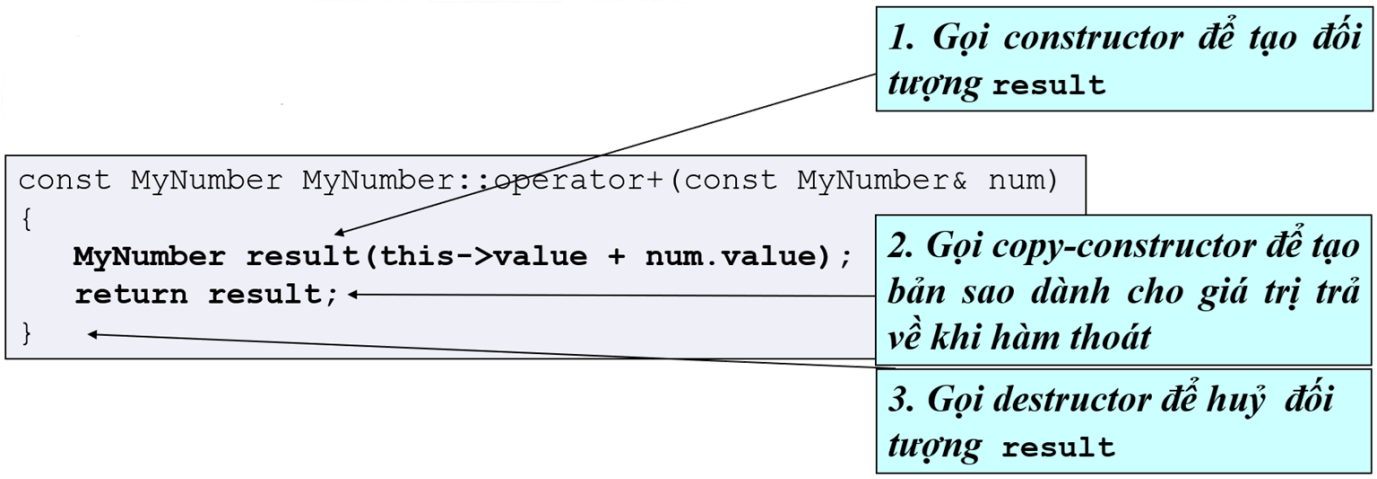
  - Đối với các **toán tử là phương thức**, điều đó có nghĩa ta nên *khai báo toán tử là hằng thành viên nếu toán hạng đầu tiên sẽ không bị sửa đổi*
  - Phần lớn các **toán tử (tính toán và so sánh)** không sửa đổi các toán hạng của nó, do đó ta sẽ rất hay *dùng đến hằng tham chiếu*

- **Giá trị trả về:**
  - **không có hạn chế về kiểu trả về** đối với toán tử được overload, nhưng nên cố gắng tuân theo tinh thần của các cài đặt có sẵn của toán tử
    - Ví dụ, các phép so sánh (==, !=…) thường trả về giá trị kiểu **bool**, nên các phiên bản overload cũng nên trả về **bool**
  - **là tham chiếu** (tới đối tượng kết quả hoặc một trong các toán hạng) **hay** *một vùng lưu trữ mới*
  - Hằng hay không phải hằng
  - Các **toán tử sinh một giá trị mới** cần có **kết quả trả về là một giá trị (thay vì tham chiếu), và là const** (để đảm bảo kết quả đó không thể bị sửa đổi như một l-value)
    - *Hầu hết các phép toán số học* đều sinh giá trị mới
    - ta đã thấy, các phép tăng sau, giảm sau tuân theo hướng dẫn trên
  - Các toán tử **trả về một tham chiếu tới đối tượng ban đầu** (đã bị sửa đổi), chẳng hạn *phép gán và phép tăng trước*, nên **trả về tham chiếu** không phải là hằng
    - để kết quả có thể được tiếp tục sửa đổi tại các thao tác tiếp theo

```
const MyNumber MyNumber::operator+(const MyNumber& right) const
MyNumber& MyNumber::operator+=(const MyNumber& right)
```

- **Xem lại cách ta đã dùng để trả về kết quả của toán tử:**

Trình tự thực hiện cách cũ:



⇨ **Cách trên không sai,** nhưng C++ cung cấp một cách hiệu quả hơn

Cách tốt hơn:

```
const MyNumber MyNumber::operator+(const MyNumber& num)
{
    return MyNumber(this->value + num.value);
}
```

- Cú pháp này tạo một **đối tượng tạm thời (temporary object)**
- Khi trình biên dịch gặp đoạn mã này, nó hiểu *đối tượng được tạo chỉ nhằm mục đích làm giá trị trả về,* nên nó tạo thẳng một đối tượng bên ngoài (để trả về) - ***bỏ qua việc tạo và huỷ đối tượng*** bên trong lời gọi hàm
- Vậy, chỉ có **một lời gọi duy nhất đến constructor** của MyNumber (***không phải copy-constructor***) thay vì dãy lời gọi trước
- Quá trình này được gọi là **tối ưu hoá giá trị trả về**
- Ghi nhớ rằng quá trình này không chỉ áp dụng được đối với các toán tử. **Ta nên sử dụng mỗi khi tạo một đối tượng chỉ để trả về**

10. Phương thức hay hàm toàn cục?

Khi lựa chọn overload toán tử tại lớp hoặc tại mức toàn cục, trường hợp nào nên chọn kiểu nào?

- Một số toán tử ***phải*** là thành viên: **"=", "[]", "()", và "->", "->*"**
- Các toán tử **đơn *nên*** là thành viên (để đảm bảo tính đóng gói)
- Khi toán hạng trái có thể được gán trị, toán tử ***nên*** là thành viên **("+=", "-=", "/=",…)**
- **Mọi toán tử đôi khác không nên là thành viên**, trừ khi ta muốn các toán tử này là *hàm ảo* trong cây thừa kế
- Các **toán tử là thành viên** nên là ***hằng hàm*** mỗi khi có thể, điều này cho phép tính mềm dẻo khi làm việc với hằng
- Nếu ta cảm thấy *không nên cho phép sử dụng một toán tử nào đó với lớp của ta* (và không muốn các nhà thiết kế khác định nghĩa nó), **ta khai báo toán tử đó dạng private** (và không cài đặt toán tử đó)

Ví dụ: Kiểu `Date`

- **`Date`** class
  - Overload phép tăng: thay đổi ngày, tháng, năm
  - Overloaded +=
  - hàm kiểm tra năm nhuận
  - hàm kiểm tra xem một ngày có phải cuối tháng

Ví dụ:

```
1  // Fig. 8.10: date1.h
2  // Date class definition.
3  #ifndef DATE1_H
4  #define DATE1_H
5  #include <iostream>
6
7  using std::ostream;
8
9  class Date {
10    friend ostream &operator<<( ostream &, const Date & );
11
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
14    void setDate( int, int, int ); // set the date
15
16    Date &operator++();              // preincrement operator
17    Date operator++( int );          // postincrement operator
18
19    const Date &operator+=( int ); // tăng ngày lên một số ngày chỉ định
20
21    bool leapYear( int ) const;    // is this a leap year?
22    bool endOfMonth( int ) const;  // is this end of month?
23
24 private:
25    int month;
26    int day;
27    int year;
28
29    static const int days[];       // Mảng tĩnh lưu số ngày trong mỗi tháng
30    void helpIncrement();          // Hàm hỗ trợ để tăng ngày lên.
31
32    }; // end class Date
33
34    #endif
```

cho phép khởi tạo ngày tháng với các giá trị mặc định

Lưu ý sự khác nhau giữa tăng trước và tăng sau.

```
1  // Fig. 8.11: date1.cpp
2  // Date class member function definitions.
3  #include <iostream>
4  #include "date1.h"
5
6  // initialize static member at file scope;
7  // one class-wide copy
8  const int Date::days[] = // với số ngày tương ứng trong mỗi tháng;
9     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
10
11 // Date constructor
12 Date::Date( int m, int d, int y )
13 {
14    setDate( m, d, y );
15
16 } // end Date constructor
17
18 // set month, day and year
19 void Date::setDate( int mm, int dd, int yy )
20 {
21    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
22    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
23
24    // test for a leap year
25    if ( month == 2 && leapYear( year ) )
26       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27    else
28       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29
30    } // end function setDate
31
32 // overloaded preincrement operator
33 Date &Date::operator++()
34 {
35    helpIncrement();
36
37    return *this;  // reference return to create an lvalue
38
39 } // end function operator++
40
41 // overloaded postincrement operator; note that the dummy
42 // integer parameter does not have a parameter name
43 Date Date::operator++( int )
44 {
45    Date temp = *this;  // hold current state of object
46    helpIncrement();
47
48    // return unincremented, saved, temporary object
49    return temp;   // value return; not a reference return
50
51 } // end function operator++
52
53 // add specified number of days to date
54 const Date &Date::operator+=( int additionalDays )
55 {
```

- Biến month được gán giá trị của mm nếu mm nằm trong khoảng từ 1 đến 12 (tháng hợp lệ).
- Nếu mm nằm ngoài khoảng này, giá trị mặc định là 1 (tháng 1) sẽ được gán cho month.

- Biến year được gán giá trị của yy nếu yy nằm trong khoảng từ 1900 đến 2100 (năm hợp lệ).
- Nếu yy nằm ngoài khoảng này, giá trị mặc định là 1900 sẽ được gán cho year.

Mảng days[] được khai báo 13 phần tử, mặc dù chỉ có 12 tháng trong năm vì mảng này sử dụng chỉ số mảng tương ứng với số tháng (từ 1 đến 12) để truy cập số ngày trong tháng đó, và phần tử đầu tiên của mảng (chỉ số 0) không được sử dụng.

*- Nếu **month là tháng 2 và year là năm nhuận** (xác định bằng hàm leapYear()), **thì giá trị day được gán bằng dd nếu dd nằm trong khoảng từ 1 đến 29**. Nếu dd không hợp lệ, day sẽ được gán giá trị mặc định là 1.*
*- **Nếu month không phải là tháng 2 hoặc year không phải là năm nhuận**, day được gán bằng dd nếu dd nằm trong khoảng **từ 1 đến số ngày tối đa của tháng đó (dựa trên mảng days[])**. Nếu dd không hợp lệ, day sẽ được gán giá trị mặc định là 1.*

Lưu ý: biến int không có tên.

Phép tăng sau sửa giá trị của đối tượng và trả về một bản sao của đối tượng ban đầu. Không trả về tham số tới biến tạm vì đó là một biến địa phương và sẽ bị hủy.

```
56    for ( int i = 0; i < additionalDays; i++ )
57       helpIncrement();
58
59    return *this;    // enables cascading
60
61 } // end function operator+=
62
63 // if the year is a leap year, return true;
64 // otherwise, return false
65 bool Date::leapYear( int testYear ) const
66 {
67    if ( testYear % 400 == 0 ||
68       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
69       return true;   // a leap year
70    else
71       return false;  // not a leap year
72
73 } // end function leapYear
74
75 // determine whether the day is the last day of the month
76 bool Date::endOfMonth( int testDay ) const
77 {
78    if ( month == 2 && leapYear( year ) )
79       return testDay == 29; // last day of Feb. in leap year
80    else
81       return testDay == days[ month ];
82
83 } // end function endOfMonth
84
85 // function to help increment the date, tăng ngày của Date lên một đơn vị
86 void Date::helpIncrement()//& tự động điều chỉnh tháng và năm khi cần thiết.
87 {
88    // day is not end of month
89    if ( !endOfMonth( day ) )
90       ++day;
91
92    else
93
94       // day is end of month and month < 12
95       if ( month < 12 ) {
96          ++month;
97          day = 1;
98       }
99
100      // last day of year
101      else {
102         ++year;
103         month = 1;
104         day = 1;
105      }
106 } // end function helpIncrement
107
108  // overloaded output operator
109  ostream &operator<<( ostream &output, const Date &d )
110  {
```

```
111     static char *monthName[ 13 ] = { "", "January",
112        "February", "March", "April", "May", "June",
113        "July", "August", "September", "October",
114        "November", "December" };
115
116     output << monthName[ d.month ] << ' '
117            << d.day << ", " << d.year;
118
119     return output;   // enables cascading
120
121  } // end function operator<<
```

```
1     // Fig. 8.12: fig08_12.cpp
2     // Date class test program.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include "date1.h"  // Date class definition
9
10    int main()
11    {
12       Date d1;  // defaults to January 1, 1900
13       Date d2( 12, 27, 1992 );
14       Date d3( 0, 99, 8045 );  // invalid date
15
16       cout << "d1 is " << d1 << "\nd2 is " << d2
17            << "\nd3 is " << d3;
18       //--------------------------------------------------
19       cout << "\n\nd2 += 7 is " << ( d2 += 7 );
20       //--------------------------------------------------
21       d3.setDate( 2, 28, 1992 );
22       cout << "\n\n  d3 is " << d3;
23       cout << "\n++d3 is " << ++d3;
24       //--------------------------------------------------
25       Date d4( 7, 13, 2002 );
26
27       cout << "\n\nTesting the preincrement operator:\n"
28            << "  d4 is " << d4 << '\n';
29       cout << "++d4 is " << ++d4 << '\n';
30       cout << "  d4 is " << d4;
31
32       cout << "\n\nTesting the postincrement operator:\n"
33            << "  d4 is " << d4 << '\n';
34       cout << "d4++ is " << d4++ << '\n';
35       cout << "  d4 is " << d4 << endl;
36
37       return 0;
38
39    } // end main
```

Kết quả:

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postincrement operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

11. Chuyển đổi kiểu tự động (Automatic type conversion)

- Ta đã nói về nhiều công việc mà đôi khi C++ ngầm thực hiện
- Một trong những việc đó là thực hiện **đổi kiểu tự động**
  - *Trong C++, có khả năng chuyển đổi kiểu tự động, một tính năng cho phép chuyển đổi một kiểu dữ liệu này sang một kiểu dữ liệu khác mà không cần chỉ định tường minh. Điều này hữu ích khi làm việc với các kiểu dữ liệu người dùng định nghĩa, như các lớp (classes).*
    - Ví dụ, nếu ta gọi một hàm đòi hỏi một kiểu dữ liệu có sẵn nhưng ta cho hàm một kiểu hơi khác, C++ đôi khi sẽ tự đổi kiểu tham số truyền vào

```
void foo(double x);
...
foo(2);
```

  - Đối với các lớp dữ liệu người dùng, C++ cũng cung cấp khả năng định nghĩa các phép chuyển đổi tự động

- **Có hai cách định nghĩa phép đổi kiểu tự động**

**** Khai báo và định nghĩa một constructor lấy một tham số duy nhất***

```
class MyNumber {
   public:
      MyNumber(int value = 0);
   ...
```

→ Đổi từ **int** sang **MyNumber**

**** Khai báo và định nghĩa một toán tử đặc biệt (special overloaded operator)***

```
class MyOtherNumber {
   public:
      MyOtherNumber(…);
      …
      operator MyNumber() const;
```

→ Đổi từ **MyOtherNumber** sang **MyNumber**

*** Cách 1: Đối với chuyển kiểu bằng Constructor:**

- Định nghĩa một constructor lấy một tham số duy nhất (giá trị hoặc tham chiếu) **thuộc một kiểu nào đó,** constructor đó sẽ được ngầm gọi khi cần đổi kiểu.
- Được sử dụng để **chuyển đổi từ kiểu của tham số đó sang kiểu của lớp.**

```
class MyNumber {
   public:
      MyNumber(int value = 0);
   ...
```

```
void foo(MyNumber num) {
…
}
```

```
...
int x = 5;
foo(x); // Automatically converts the argument
...
```

• Trong ví dụ này, `foo(x)` gọi hàm `foo` với một biến `int x`. Tuy nhiên, hàm `foo` yêu cầu tham số kiểu `MyNumber`.

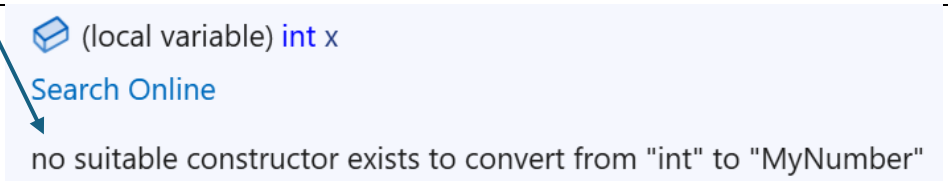
• C++ tự động sử dụng constructor `MyNumber(int v)` để chuyển đổi `x` thành một đối tượng của lớp `MyNumber`.

- Nếu ta muốn ngăn chặn đổi kiểu tự động kiểu này, ta có thể dùng từ khoá **explicit** khi khai báo constructor

```
class MyNumber {
   public:
      explicit MyNumber(int value = 0);
   ...
```

→ Kết quả là việc đổi kiểu chỉ xảy ra khi được gọi một cách tường minh

```
int x = 5;
foo(x); // This will generate an error
foo(MyNumber(x)); // Explicit conversion - ok
```



*** Cách 2: Toán tử:**

- Định nghĩa phép đổi kiểu sử dụng một toán tử đặc biệt
  - Tạo một **toán tử là thành viên của lớp cần đổi**, toán tử này sẽ *chuyển đổi tượng của lớp này sang kiểu mong muốn*
  - Toán tử này hơi đặc biệt: **không có kiểu trả về** → Thực ra, **tên của toán tử chính là kiểu trả về**
- Giả sử cần một phép chuyển đổi tự động từ kiểu MyOtherNumber sang kiểu **MyNumber**
  - Khai báo một toán tử có tên **MyNumber** :

```
class MyOtherNumber {
   public:
      MyOtherNumber(…);
      …
      operator MyNumber() const;
```

- Thân toán tử chỉ cần tạo và trả về một thể hiện của lớp đích → ***Nên là hàm inline***

```
operator MyNumber() const { return MyNumber(…); }
```

(…) : Các tham số cần thiết để tạo thể hiện mới

Ví dụ:

```cpp
#include <iostream>

class MyNumber {
private:
    int value;

public:
    MyNumber(int v = 0) : value(v) {}

    void display() const {
        std::cout << "Value: " << value << std::endl;
    }
};

class MyOtherNumber {
private:
    double otherValue;

public:
    MyOtherNumber(double v = 0.0) : otherValue(v) {}

    // Toán tử chuyển đổi từ MyOtherNumber sang MyNumber
    operator MyNumber() const {
        return MyNumber(static_cast<int>(otherValue));
    }
};

void foo(MyNumber num) {
    num.display();
}

int main() {
    MyOtherNumber y(7.5);
    foo(y); // Chuyển đổi tự động từ MyOtherNumber sang MyNumber
    return 0;
}
```

Kết quả: Value: 7

Giải thích:

- **Toán tử operator MyNumber() const:** Toán tử này chuyển đổi đối tượng MyOtherNumber thành MyNumber *bằng cách chuyển giá trị otherValue (kiểu double) thành int, rồi tạo và trả về một đối tượng MyNumber mới.*
- **foo(MyNumber num):** Hàm foo yêu cầu một tham số kiểu MyNumber.
- **foo(y);** Khi gọi foo(y), đối tượng y (kiểu MyOtherNumber) cần được chuyển đổi sang kiểu MyNumber để phù hợp với tham số của hàm foo. C++ tự động gọi toán tử chuyển đổi operator MyNumber() để thực hiện chuyển đổi này.

Kết quả là đối tượng y được chuyển đổi thành một đối tượng MyNumber, và giá trị của y (7.5) sẽ được chuyển thành int (7), sau đó được truyền vào hàm foo.

**** Đổi kiểu ẩn***

```
class Fee {
   public:
      Fee(int) {}
};

class Fo {
      int i;
   public:
      Fo(int x = 0) : i(x) {}
      operator Fee() const { return Fee(i); }
};

int main() {
   Fo fo;
   Fee fee = fo;
} ///:~
```

3. Không có copy constructor để tạo **Fee** *từ* **Fee**, *nhưng lại có copy constructor mặc định do trình biên dịch tự sinh (phép gán mặc định)*

2. Tuy nhiên, có phép chuyển đổi tự động từ **Fo** *sang* **Fee**

1. Không có constructor tạo **Fee** *từ* **Fo**

1. Không có constructor tạo Fee từ Fo

- *Lớp Fee chỉ có một constructor nhận một tham số kiểu int*, nhưng không có constructor nào nhận tham số kiểu Fo. Điều này có nghĩa là bạn *không thể tạo một đối tượng Fee trực tiếp từ một đối tượng Fo* bằng cách sử dụng constructor của Fee.

2. Tuy nhiên, có phép chuyển đổi tự động từ Fo sang Fee

- Lớp Fo định nghĩa một toán tử chuyển đổi tự động (operator Fee() const) để chuyển đổi một đối tượng Fo thành một đối tượng Fee.
- Toán tử này có chức năng *tạo một đối tượng Fee từ giá trị của thành viên dữ liệu i của Fo, bằng cách gọi constructor Fee(int) với i làm tham số.*

3. Không có copy constructor để tạo Fee từ Fee, nhưng lại có copy constructor mặc định do trình biên dịch tự sinh (phép gán mặc định)

- Điều này *cho phép sao chép các đối tượng Fee với nhau mà không cần phải định nghĩa lại constructor.*

*** Sự nhập nhằng**

Sự nhập nhằng xảy ra khi lớp của bạn có *chuyển kiểu bằng constructor lẫn chuyển kiểu bằng toán tử chuyển kiểu.* Nó khiến cho trình biên dịch không xác định được nên chuyển kiểu bằng cái nào, dẫn đến việc mất đi cơ chế chuyển kiểu tự động (ngầm định).

```
class PhanSo {
    //...
public:
    PhanSo(int a);
    operator float();
};

int main() {
    PhanSo a(2, 3);
    a + 5; // lỗi do sự nhập nhằng, không biết nên chuyển
    5 + a; // 5 thành Phan_so hay a thành float
    return 0;
}
```

*** Cách xử lý sự nhập nhằng**

Để tránh sự nhập nhằng, bạn cần quyết định rõ ràng phương pháp chuyển đổi nào sẽ được ưu tiên trong lớp của bạn:

Loại bỏ một trong hai phương pháp chuyển đổi:

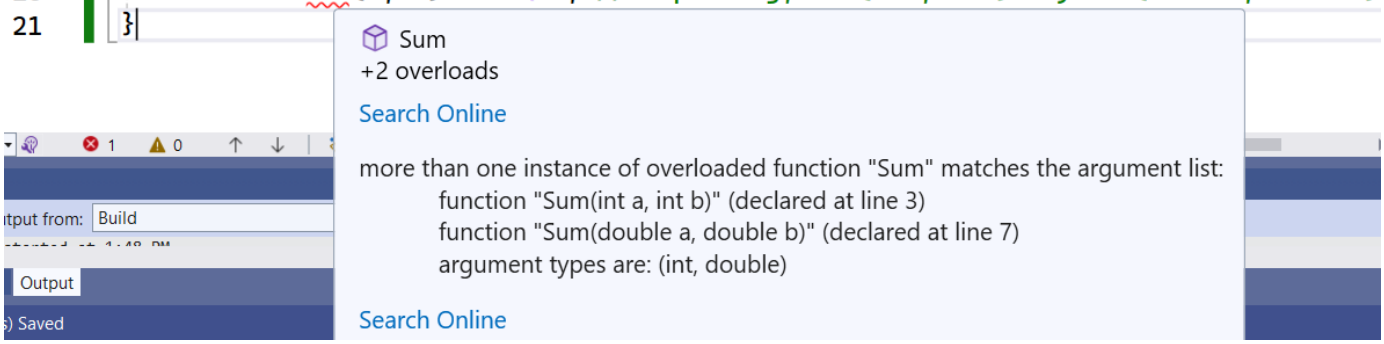
- Nếu bạn chỉ cần một phương pháp chuyển đổi (ví dụ, chỉ cần chuyển đổi từ int sang PhanSo bằng constructor), bạn có thể loại bỏ toán tử chuyển kiểu (operator float()).

Thực hiện chuyển đổi kiểu tường minh:

- Nếu bạn muốn giữ cả hai phương pháp, bạn cần chuyển đổi kiểu một cách tường minh khi sử dụng các đối tượng để tránh sự nhập nhằng. Tuy nhiên việc này làm mất đi sự tiện lợi của cơ chế chuyển kiểu tự động.

```
int main() {
    PhanSo a(2, 3);
    float result = a + static_cast<PhanSo>(5);
    result = static_cast<float>(a) + 5;
    return 0;
}
```

```
1  #include <iostream>
2  using namespace std;
3  int Sum(int a, int b)
4  {
5      return a + b;
6  }
7  double Sum(double a, double b)
8  {
9      return a + b;
10 }
11
12 void main() {
13     int a = 3, b = 7;
14     double r = 3.2, s = 6.3;
15     cout << a + b << "\n";     // Ok
16     cout << r + s << "\n";     // Ok
17     cout << a + r << "\n";     // Ok: double(a)+r
18     cout << Sum(a, b) << "\n"; // Ok Sum(int, int)
19     cout << Sum(r, s) << "\n";  // Ok Sum(double, double)
20     cout << Sum(a, r) << "\n"; //Nhap nhang, Sum(int, int) hay Sum(double, double)
21 }
```



- Nói chung, **nên tránh đổi kiểu tự động**, do chúng có thể *dẫn đến các kết quả người dùng không mong đợi*
  - Nếu có constructor lấy đúng một tham số khác kiểu lớp chủ, hiện tượng đổi kiểu tự động sẽ xảy ra
  - Chính sách an toàn nhất là khai báo các constructor là **explicit**, trừ khi kết quả của chúng đúng là cái mà người dùng mong đợi

12. Phép toán lấy phần tử mảng: [ ]

- Ta có thể định nghĩa phép toán [ ] để truy xuất phần tử của một đối tượng có ý nghĩa mảng, đối tượng `std::vector`, hoặc đối tượng tương tự có thể dùng chỉ số để truy cập phần tử.
- Trong C++, toán tử [] có thể được nạp chồng (overload) trong các lớp để cho phép truy cập các phần tử thông qua chỉ số, giống như cách bạn truy cập phần tử trong mảng thông thường.
- Toán tử này thường được định nghĩa như một hàm thành viên không tĩnh (non-static) trong lớp.

```cpp
#include <iostream>

class Array {
private:
    int data[10]; // Mảng giả định có 10 phần tử

public:
    Array() {
        // Khởi tạo mảng với các giá trị từ 0 đến 9
        for (int i = 0; i < 10; ++i) {
            data[i] = i;
        }
    }

    // Nạp chồng toán tử []
    int& operator[](int index) {
        if (index < 0 || index >= 10) {
            std::cerr << "Index out of bounds!" << std::endl;
            // Có thể xử lý lỗi theo nhiều cách, đây chỉ là ví dụ
            exit(EXIT_FAILURE);
        }
        return data[index];
    }

    // Nạp chồng phiên bản const cho toán tử []
    const int& operator[](int index) const {
        if (index < 0 || index >= 10) {
            std::cerr << "Index out of bounds!" << std::endl;
            exit(EXIT_FAILURE);
        }
        return data[index];
    }
};

int main() {
    Array arr;
    arr[3] = 100;    // Đặt giá trị 100 vào vị trí thứ 3 (chỉ số 2)
    std::cout << arr[3] << std::endl; // In ra giá trị của phần tử thứ 3: 100

    return 0;
}
```

- **int& operator[](int index):** Toán tử [] được nạp chồng để trả về tham chiếu đến phần tử tại vị trí index. Điều này cho phép bạn vừa truy xuất (get) vừa gán (set) giá trị cho phần tử.
- **const int& operator[](int index) const:** Phiên bản const của toán tử [] đảm bảo rằng toán tử này có thể được sử dụng với các đối tượng const, chỉ cho phép truy xuất mà không cho phép gán giá trị.

13. Phép toán gọi hàm: ()

- Phép toán [ ] *chỉ có thể có một tham số*, vì vậy dùng phép toán trên không thuận tiện khi ta muốn lấy phần tử của một ma trận hai chiều.
- Phép toán gọi hàm cho phép có thể ***có số tham số bất kỳ***, vì vậy thuận tiện khi ta muốn truy xuất phần tử của các đối tượng thuộc loại mảng hai hay nhiều chiều hơn.

Ví dụ 1:

```
#include <iostream>

class Multiply {
private:
    int factor;

public:
    Multiply(int f) : factor(f) {}

    // Nạp chồng toán tử () để nhân giá trị với factor
    int operator()(int x) const {
        return x * factor;
    }
};

int main() {
    Multiply timesTwo(2); // Tạo một đối tượng Multiply nhân với 2

    int result = timesTwo(5); // Sử dụng đối tượng như một hàm
    std::cout << "5 * 2 = " << result << std::endl; // In ra: 5 * 2 = 10

    return 0;
}
```

Multiply timesTwo(2); Tạo một đối tượng timesTwo của lớp Multiply, với hệ số nhân là 2. Constructor sẽ gán giá trị 2 cho factor.

Toán tử () được nạp chồng để thực hiện phép nhân với một số đã cho. Khi bạn gọi timesTwo(5), nó tương đương với việc gọi timesTwo.operator()(5), trả về giá trị 10 vì 5 * 2 = 10.

Ví dụ 2 (Functor với nhiều đối số):

```
#include <iostream>
class Calculator {
public:
    // Nạp chồng toán tử () cho phép nhận hai đối số
    int operator()(int a, int b, char op) const {
        switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return b != 0 ? a / b : 0; // Tránh chia cho 0
        default: return 0;
        }
    }
};

int main() {
    Calculator calc;

    std::cout << "3 + 4 = " << calc(3, 4, '+') << std::endl; // In ra: 3 + 4 = 7
    std::cout << "10 * 5 = " << calc(10, 5, '*') << std::endl; // In ra: 10 * 5 = 50

    return 0;
}
```

Toán tử () với nhiều đối số: Trong ví dụ này, toán tử () được nạp chồng để thực hiện các phép toán số học cơ bản dựa trên đối số op. Nó nhận ba đối số: hai số nguyên và một ký tự biểu thị phép toán (+, -, *, /).

14. Toán tử so sánh (Relational operator)

- Một số toán tử so sánh thường gặp: ==, !=, >, <, >=, <=.
- Một phương thức đa năng hóa toán tử so sánh thường trả về giá trị true (1) hay false (0). Điều này phù hợp với ứng dụng thông thường của những toán tử này (sử dụng trong các biểu thức điều kiện).

Ví dụ: Định nghĩa toán tử so sánh > cho lớp PhanSo

```
#include <iostream>
using namespace std;
class PhanSo {
private:
	int tu;
	int mau;
public:
	PhanSo() { tu = 0; mau = 1; }
	PhanSo(int a, int b) { tu = a; mau = b; }
	bool operator>(const PhanSo& x);
};
bool PhanSo::operator>(const PhanSo& x) {
	float gt1 = (float)tu / mau;
	float gt2 = (float)x.tu / x.mau;
	if (gt1 > gt2)
		return true;
	return false;
}

int main() {
	PhanSo a(2, 3), b(4, 1);
	if (a > b) // a.operator>(b)
		cout << "Phan so a lon hon phan so b" << endl;
	else
		cout << "Phan so a khong lon hon phan so b" << endl;
}
```

Kết quả: Phan so a khong lon hon phan so b

** Tương tự với các phép so sánh còn lại.*

Operator	Name or Category	Method or Global Friend Function	When to Overload	Sample Prototype

`operator+`
`operator-`
`operator*`
`operator/`
`operator%`
	Binary arithmetic	Global friend function recommended	Whenever you want to provide these operations for your class	`friend const T operator+(const T&, const T&);`

`operator-`
`operator+`
`operator~`
	Unary arithmetic and bitwise operators	Method recommended	Whenever you want to provide these operations for your class	`const T operator-() const;`

`operator++` `operator--`	Increment and decrement	Method recommended	Whenever you overload binary + and -	`T& operator++();` `const T operator++(int);`
`operator=`	Assignment operator	Method required	Whenever you have dynamically allocated memory in the object or want to prevent assignment, as described in Chapter 9	`T& operator=(const T&);`

`operator+=`
`operator-=`
`operator*/`
`operator/=`
`operator%=`
	Shorthand arithmetic operator assignments	Method recommended	Whenever you overload the binary arithmetic operators	`T& operator-=(const T&);`

`operator<<`
`operator>>`
`operator&`
`operator|`
`operator^`
	Binary bitwise operators	Global friend function recommended	Whenever you want to provide these operations	`friend const T operator<<(const T&, const T&);`

`operator<<=`
`operator>>=`
`operator&=`
`operator|=`
`operator^=`
	Shorthand bitwise operator assignments	Method recommended	Whenever you overload the binary bitwise operators	`T& operator<<=(const T&);`

`operator<`
`operator>`
`operator<=`
`operator>=`
`operator==`
	Binary comparison operators	Global friend function recommended	Whenever you want to provide these operations	`friend bool operator<(const T&, const T&);`

`operator<<` `operator>>`	I/O stream operators (insertion and extraction)	Global friend function recommended	Whenever you want to provide these operations	`friend ostream &operator<<(ostream&, const T&);` `friend istream &operator>>(istream&, T&);`
`operator!`	Boolean negation operator	Member function recommended	Rarely; use `bool` or `void*` conversion instead	`bool operator!() const;`
`operator&&` `operator||`	Binary Boolean operators	Global friend function recommended	Rarely	`friend bool operator&&(const T& lhs, const T& rhs);` `friend bool operator||(const T& lhs, const T& rhs);`

`operator[]`	Subscripting (array index) operator	Method required	When you want to support subscripting: in array-like classes	`E& operator[](int);` `const E& operator[](int) const;`
`operator()`	Function call operator	Method required	When you want objects to behave like function pointers	Return type and arguments can vary; see examples in this chapter
`operator new` `operator new[]`	Memory allocation routines	Method recommended	When you want to control memory allocation for your classes (rarely)	`void* operator new(size_t size) throw(bad_alloc);` `void* operator new[](size_t size) throw(bad_alloc);`
`operator delete` `operator delete[]`	Memory deallocation routines	Method recommended	Whenever you overload the memory allocation routines	`void operator delete(void* ptr) throw();` `void operator delete[](void* ptr) throw();`
`operator*` `operator->`	Dereferencing operators	Method required for operator-> Method recommended for operator*	Useful for smart pointers	`E& operator*() const;` `E* operator->() const;`
`operator&`	Address-of operator	N/A	Never	N/A
`operator->*`	Dereference pointer-to-member	N/A	Never	N/A
`operator,`	Comma operator	N/A	Never	N/A
`operator type()`	Conversion, or cast, operators (separate per type)	Method required	When you want to provide conversions from your class to other types	`operator type() const;`