Operator Overloading

Lập trình hướng đối tượng

Tài liệu đọc

- Eckel, Bruce. *Thinking in C++, 2nd Ed.* Vol. 1.
  - Chapter 8: Operator Overloading
- Dietel. *C++ How to Program, 4th Ed.*
  - Chapter 8: Operator Overloading

Operator Overloading

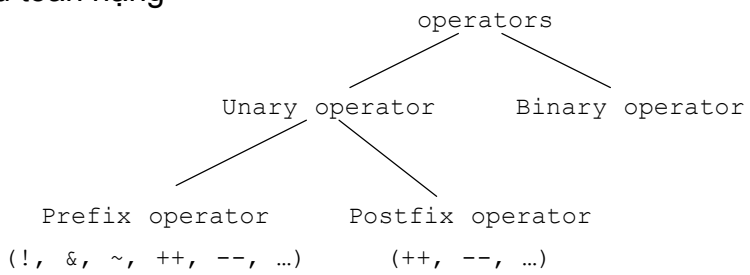
- Giới thiệu
- Các toán tử của C++
- Lý thuyết về operator overloading
- Cú pháp operator overloading
- Định nghĩa các toán tử thành viên
- Phép gán
- Định nghĩa các toán tử toàn cục
- Làm việc với tính đóng gói
- friend
- Tại sao sử dụng toán tử toàn cục
- Phép chèn ("<<")
- Phép tăng ("++")
- Các tham số và kiểu trả về
- Thành viên hay hàm toàn cục?
- Chuyển đổi kiểu tự động

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…)
- Đối với các kiểu dữ liệu người dùng: 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
- 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

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



Các toán tử của C++.

- 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)

Các toán tử overload được

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

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


Các toán tử không overload được

- 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

Cú pháp của Operator Overloading

```
class MyNumber {
   public:
      MyNumber(int value = 0);
      ~MyNumber();
      …
   private:
      int value;
};
```

- Ta sẽ sử dụng ví dụ trên:
  - Đây là lớp bọc ngoài (wrapper class) cho kiểu int
- Ta sẽ overload các toán tử để cho phép cộng, trừ, so sánh, … các đối tượng của lớp

Cú pháp của 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 "@"
  - 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 được khai báo là hàm toàn cục hay phương thức của lớp

```
aa@bb    è aa.operator@(bb)  hoặc operator@(aa,bb)
@aa      è aa.operator@( )   hoặc operator@(aa)
aa@      è aa.operator@(int) hoặc operator@(aa,int)
```

là phương thức của lớp

là hàm toàn cục

Cú pháp của Operator Overloading

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

```
MyNumber x(5);
MyNumber y(10);
...
z = x + y;
```

- Ta có thể khai báo hàm toàn cục sau

```
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

- Hoặc khai báo toán tử dưới dạng thành viên của **`MyNumber`**:

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

  - đối tượng chủ của phương thức được hiểu là toán hạng thứ nhất của toán tử.
  - "`x+y`" sẽ được hiểu là "`x.operator+(y)`"

Cú pháp của Operator Overloading

- 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

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:

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

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)

Phép gán "="

```
const MyNumber& MyNumber::operator=(const MyNumber& num) {
   if (this != &num) {
      this->value = num.value;
   }
   return *this;
}
```

- 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

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

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;
}
```

  - Ở đây có vấn đề....

truy nhập các thành viên private **`value`**

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`**

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);
   ...
};
```

Hàm **friend**

- 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;
}
```


Tại sao dùng 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

Toán tử chèn (‘<<‘)

- prototype như thế nào? 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`**
- Tham số thứ hai : kiểu **`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)`

Toán tử chèn ("<<")

- 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);
      ...
};
```

Toán tử chèn ("<<")

- 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
};
```

- 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

Phép tăng ("++")

@aa è aa.operator@( ) hoặc **operator@(aa)**
aa@ è 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

Phép tăng ("++")

- giá trị trả về
  - tăng trước **++num**
    - trả về tham chiếu (**MyNumber &**)
    - giá trị trái - lvalue (có thể được gán trị)
  - tăng sau **num++**
    - trả về giá trị (giá trị cũ trước khi tăng)
    - trả về đối tượng tạm thời chứa giá trị cũ.
    - giá trị phải - rvalue (không thể làm đích của phép gán)
- prototype
  - tăng trước: **MyNumber& MyNumber::operator++()**
  - tăng sau: **const MyNumber MyNumber::operator++(int)**

Phép tăng ("++")

- 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

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

Tham số và kiểu trả về

- 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

Tham số và kiểu trả về

- 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

Tham số và kiểu trả về

- Giá trị trả về …
  - 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)
```

Tham số và kiểu trả về

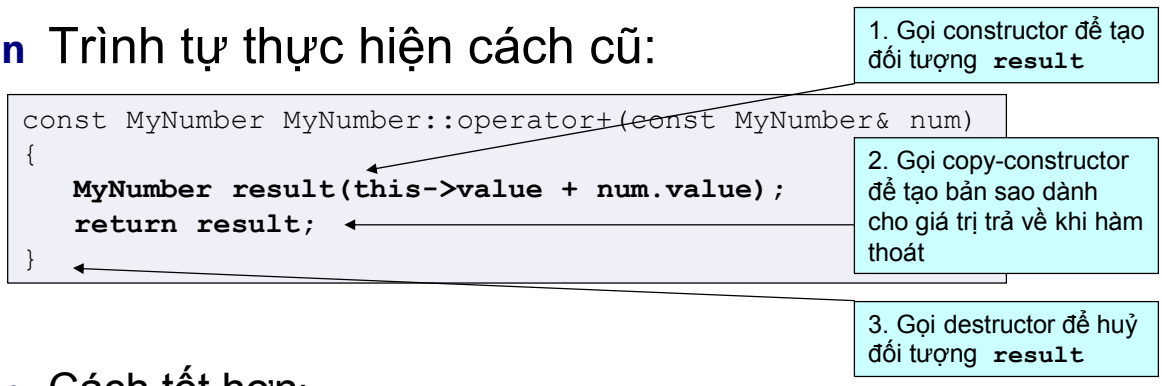
- Lời khuyên cuối cùng:
  - Xem lại cách ta đã dùng để trả về kết quả của toán tử:

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

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

Tham số và kiểu trả về

- Trình tự thực hiện cách cũ:



- Cách tốt hơn:

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

Tham số và kiểu trả về

```
return MyNumber(this->value + num.value);
```

- Cú pháp của ví dụ trước 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

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 "`->`", "`->*`" phải là thành viên
- 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

Phương thức hay hàm toàn cục?

- 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 du: 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

```
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++();           // prei
17    Date operator++( int );       // postincrement operator
18
19    const Date &operator+=( int ); // add days, modify object
20
21    bool leapYear( int ) const;   // is this a leap year?
22    bool endOfMonth( int ) const; // is this end of month?
```

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

```
23
24    private:
25    int month;
26    int day;
27    int year;
28
29    static const int days[];      // array of days per month
30    void helpIncrement();         // utility function
31
32    }; // end class Date
33
34    #endif
```

```cpp
// Fig. 8.11: date1.cpp
// Date class member function definitions.
#include <iostream>
#include "date1.h"

// initialize static member at file scope;
// one class-wide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y )
{
   setDate( m, d, y );

} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;

   } // end function setDate

// overloaded preincrement operator
Date &Date::operator++()
{
   helpIncrement();

   return *this;  // reference return to create an lvalue

} // end function operator++

// overloaded postincrement operator
// integer parameter does not have a
Date Date::operator++( int )
{
   Date temp = *this;  // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp;   // value return; not a reference return

} // end function operator++
```

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.

```
53 // add specified number of days to date
54 const Date &Date::operator+=( int additionalDays )
55 {
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
86    void Date::helpIncrement()
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
107   } // end function helpIncrement
108
```

```
109   // overloaded output operator
110   ostream &operator<<( ostream &output, const Date &d )
111   {
112      static char *monthName[ 13 ] = { "", "January",
113         "February", "March", "April", "May", "June",
114         "July", "August", "September", "October",
115         "November", "December" };
116
117      output << monthName[ d.month ] << ' '
118             << d.day << ", " << d.year;
119
120      return output;   // enables cascading
121
122   } // 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
```

```
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
```

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
  - 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

Đổi kiểu 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**

Đổi kiểu tự động - constructor

- Cách 1: 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.

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

```
void foo(MyNumber num) {
…
}
```

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

Đổi kiểu tự động - constructor

- 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
```


Đổi kiểu tự động – toán tử

- Cách 2: 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 thành viên 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

Đổi kiểu tự động – toán tử

- 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

Đổi kiểu tự động - 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`**

Đổi kiểu tự động

- 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