Chương 10

Lập trình hướng đối tượng

--Cơ bản--

Lê Thành Sách

Nội dung

- Kiểu dữ liệu trong C - Ôn lại
- Các khái niệm cơ bản
- Con trỏ this
- Tổng hợp các thuật ngữ (I)
- Tính khả kiến
- Thiết kế lớp
- Hàm khởi tạo và hàm huỷ
- Định nghĩa lại toán tử
- Thành viên có tính "static"
- Thành viên có tính "const"
- Tổng hợp các thuật ngữ (II)
- Quan hệ bạn bè (friendship)
- Tổ chức mã nguồn cho lớp
- Biểu diễn lớp bằng sơ đồ
- Tổng kết

Kiểu dữ liệu trong C

- Xét đoạn chương trình

```
typedef struct{
    int day, month, year;
} Date;
int main(int argc, char** argv) {
    int a;
    double d;
    Date c = {20, 5, 2017};

    return 0;
}
```

Bộ biên dịch cấp phát 3 vùng nhớ có tên: a, d, và c trên STACK:
a: 4 bytes → dùng sizeof(.) cho cụ thể.
d: 8 bytes
c: 12 bytes

Cả 3 vùng nhớ này đều **THỤ ĐỘNG**, CHỈ CÓ CÔNG NĂNG **LÀ CHỨA** các giá trị của kiểu được mô tả.

Kiểu dữ liệu trong C

```
typedef struct{
    int day, month, year;
} Date;
void print(Date& d){
    cout << d.day << "/" << d.month << "/" << d.year;
}
int main(int argc, char** argv) {
    int a;
    double d;
    Date c = {20, 5, 2017};
    print(c);
    return 0;
}
```


Vì thụ động, nên khi cần xử lý dữ liệu, thực hiện:

a) Tạo ra hàm

b) Gọi hàm và truyền dữ liệu vào → **Ví dụ như hàm: "print" ở trên**

Kiểu dữ liệu trong C

```
typedef struct{
    int day, month, year;
} Date;
void print(Date& d){
    cout << d.day << "/" << d.month << "/" << d.year;
}
int main(int argc, char** argv) {
    int a;
    double d;
    Date c = {20, 5, 2017};
    print(c);
    return 0;
}
```


Vì thụ động, nên khi cần xử lý dữ liệu:

a) Tạo ra hàm

b) Gọi hàm và truyền dữ liệu vào → **Ví dụ như hàm: print ở trên**

Nhược điểm của tính THỤ ĐỘNG này là gì?

Kiểu dữ liệu trong C

- **Nhược điểm là gì?**
  - **Khó biết được dữ liệu có thể được xử lý bởi hàm nào.**
  - **Khó đảm bảo ràng buộc trên dữ liệu, ví dụ:**
    - **Ngày: 1 → 31 (tuỳ tháng, tối đa)**
    - **Tháng: 1 → 12**

Chỉ cần tính chất về "**đóng gói**" (**enscapsulation**) của lập trình hướng đối tượng (OOP) đã giải quyết những vấn đề trên.

Hơn nữa, ngoài tính "**đóng gói**", OOP còn cung cấp những tính năng hay khác nữa mà ngôn ngữ C không có.

Khái niệm cơ bản

- (Q.1) **Lớp (class):**
  - Là một kiểu dữ liệu do người lập trình tạo ra.
  - **Quan niệm:** Lớp như cái khuôn để từ đó tạo ra các đối tượng như nói sau.

- (Q.2) **Đối tượng (object, instance):**
  - Là một biến tạo ra từ kiểu lớp.
  - Ví dụ:
    - Giả sử đã có lớp `MyClass`
    - Dòng:
      ```
      MyClass c;
      ```
    - Sẽ tạo ra một đối tượng, đặt tên là "c", nghĩa là một vùng nhớ có tên là "c".

Khái niệm cơ bản

- (Q.3) **Mô tả lớp có gì khác mô tả một cấu trúc trong C (struct)**
  - Khi mô tả kiểu này, cần mô tả
    - Các **dữ liệu** mà một đối tượng của lớp có.
    - Các **hàm (phương thức)** có thể thực thi với đối tượng của lớp.
      - Những hành động mà một đối tượng của lớp đó có thể thực hiện → tính chủ động của đối tượng (không chỉ là vùng nhớ thụ động)

Với kiểu struct (của C):
Mô tả kiểu này không có mô tả hàm/phương thức như kiểu lớp!

Khái niệm cơ bản

- (Q.4) **Tạo đối tượng như thế nào?**
  - Giống như struct. Giả sử có lớp `MyClass`
  - (1) Tạo tĩnh trên STACK:

```
MyClass obj;
MyClass obj(…);
//có thể truyền tham số - xem Phần Constructor
```

  - (2) Tạo động trên HEAP:

```
MyClass *ptr = new MyClass();
MyClass *ptr = new MyClass(…);
//có thể truyền tham số - xem Phần Constructor
//dùng ptr tại đây
delete ptr;
```

Khái niệm cơ bản

- (Q.4) **Tạo đối tượng như thế nào?**
  - Giống như struct. Giả sử có lớp `MyClass`
  - (1) Tạo tĩnh trên STACK, mảng tĩnh

```
MyClass obj[SIZE];
```


  - (2) Tạo động trên HEAP:

```
MyClass *ptr = new MyClass[SIZE];
//.. sử dụng ptr
```



```
delete []ptr;
```

Khái niệm cơ bản

- (Q.5) **Truy xuất dữ liệu và gọi phương thức ntn?**
  - Giống như struct. Giả sử có lớp `MyClass`
  - (1) đối tượng trên STACK:

```
MyClass obj;
...
obj.[tên-thành-viên]; //xem ví dụ.
```

  - (2) đối tượng trên HEAP:

```
MyClass *ptr = new MyClass();
...
Obj->[tên-thành-viên]; //xem ví dụ.
```


Lưu ý:
Toán tử chấm "." và toán tử mũi tên "->"

Khái niệm cơ bản: Minh hoạ (I)

- Tạo ra một kiểu "Date", theo yêu cầu:
  - Một đối tượng của "Date" có phải chứa được dữ liệu về **ngày**, **tháng**, và **năm**.
  - Một đối tượng của "Date" có thể ***đón nhận lời yêu cầu*** "**print**". Một khi nó (đối tượng) nhận được yêu cầu này, nó in ra màn hình ngày, tháng, và năm mà nó đang giữ.
    - Đối tượng có tính **CHỦ ĐỘNG** hơn so với các biến kiểu cấu trúc (của C)

Khái niệm cơ bản: Minh hoạ (I)

```
class Date{
public:
    int day, month, year;

    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```

Một lớp (class) "Date"

Chú ý: kết thúc bằng dấu ;

Mô tả phương thức cho đối tượng của lớp "Date":
khai báo + định nghĩa hàm

Mô tả dữ liệu cho đối tượng của lớp "Date"

Khái niệm cơ bản: Minh hoạ (I)

```
class Date{
public:
    int day, month, year;

    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```


Từ khoá "public" nghĩa là gì?

Cho phép bất kỳ nơi nào, miễn sao có tham chiếu đến đối tượng kiểu "Date", là có thể dùng được các dữ liệu và phương thức theo sau "public", i.e., **day**, **month**, **year** và **print**

Xem phần "Tính khả kiến" – theo sau.

Khái niệm cơ bản: Minh hoạ (I)

Dùng lớp "Date" như thế nào?
Xem hàm main() sau:

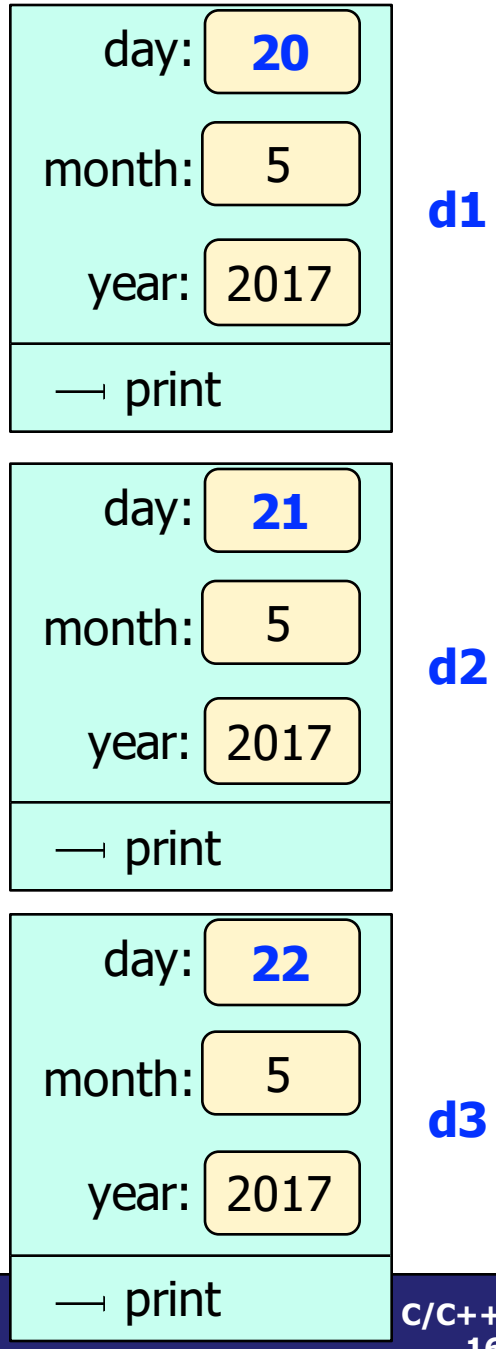
```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

Tạo ra 3 đối tượng có tên là d1, d2, và d3,

và khởi gán giá trị ban đầu.



Lưu ý:

Đối tượng = dữ liệu + hàm

Lưu ý

Trong trường hợp như lớp "Date", phương thức "print" không phải liên kết động (vì: non-virtual) nên con trỏ hàm không cần đi kèm trong bộ nhớ cấp cho đối tượng, xem hình.

Tuy nhiên, slide này trình bày mô hình như vậy để giúp sinh viên dễ dàng nắm bắt khái niệm *gói cả dữ liệu và hàm.* Có thể xem đây là mô hình đơn giản và ở mức cao (luận lý).

Người học nên xem thêm các tài liệu khác để nắm rõ hơn về mô hình đối tượng (Object Model):

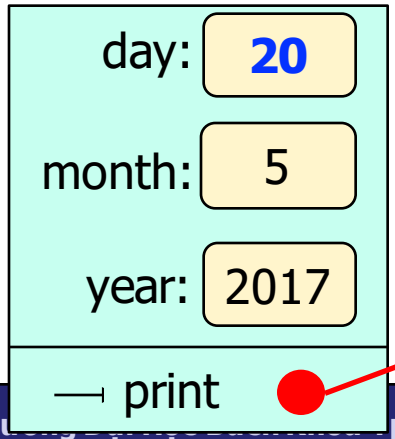
[1] Stanley B. Lippman, "Inside the C++ Object Model," Addison Wesley, 1996.
[2] http://spockwangs.github.io/2011/01/31/cpp-object-model.html

Tuy vậy, chỉ nên đọc sau khi nắm bắt chắc các khái niệm và cách dùng OOP trong slide này.

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

```
class Date{
public:
    int day, month, year;
    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```

d1

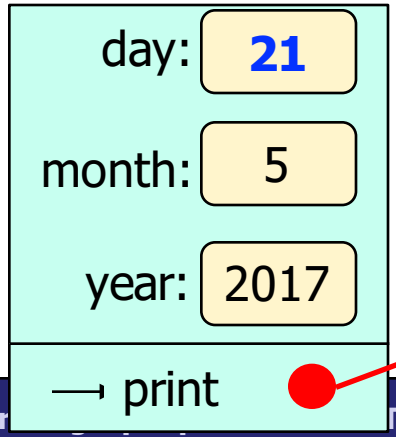
Lưu ý:

Khi gọi hàm print của d1 thì:
day = **20**;
month = 5;
Và year = 2017

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

```
class Date{
public:
    int day, month, year;
    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```

d2

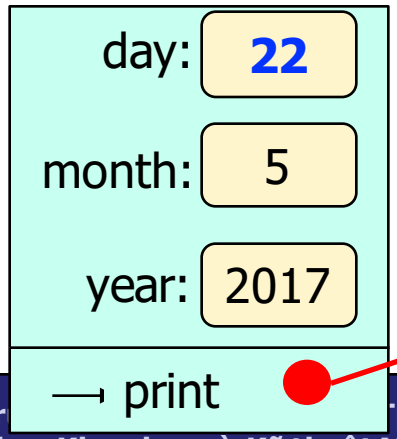
Lưu ý:

Khi gọi hàm print của d2 thì:
day = **21**;
month = 5;
Và year = 2017

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

```
class Date{
public:
    int day, month, year;
    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```



d3

Lưu ý:

Khi gọi hàm print của d3 thì:
day = **22**;
month = 5;
Và year = 2017

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

Kết quả xuất ra màn hình:

20/5/2017
21/5/2017
22/5/2017

Khái niệm cơ bản: Minh hoạ (I)

```
int main(int argc, char** argv) {
    Date d1 = {20, 5, 2017};
    Date d2 = {21, 5, 2017};
    Date d3 = {22, 5, 2017};

    d1.print();
    d2.print();
    d3.print();

    return 0;
}
```

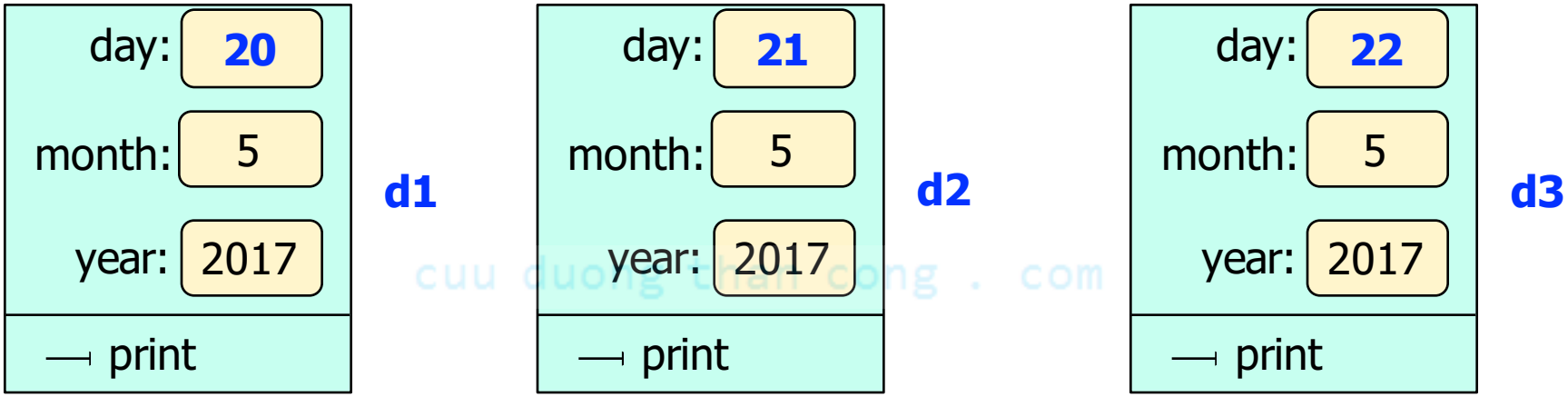
```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
};
```

Vì từ khoá "public" → day, month, year và print có thể truy cập được bất kỳ đâu.

→ Truy cập được trong hàm main()

Con trỏ "this"

- Đối tượng = dữ liệu + hàm. Ví dụ:



- Địa chỉ đến (byte đầu tiên của) đối tượng được lưu trong biến "this" (là từ khoá). Địa chỉ này chỉ có thể dùng được bên trong các hàm thành viên của đối tượng.
  - Bên trong **print** của **d1**: this chỉ đến vùng **d1**.
  - Bên trong **print** của **d2**: this chỉ đến vùng **d2**, ...

Con trỏ "this": Minh hoạ (I)

```
class Date{
public:
    int day, month, year;

    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```


Hàm print có thể viết như sau:

```
class Date{
public:
    int day, month, year;

    void print(){
        cout << this->day << "/" << this->month << "/" << this->year << endl;
    }
};
```

Con trỏ "this": Minh hoạ (II)

```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
    void setDay(int day){

    }

};
```

Tại đây có hai biến cùng tên: day.

- **day**: của thông số
- Và, **day** là thành viên của lớp.

Mặc nhiên, **day** của thông số là ưu tiên. Do đó, để gán **day** của thông số vào **day** thành viên thì dùng "this", như sau:

```
this->day = day;
```

Con trỏ "this": Minh hoạ (III)

```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
    void setDay(int day){
        this->day = day;
    }
    Date& getDate(){

    }


};
```

Giả sử: cần trả về tham khảo đến đối tượng đang chứa hàm **getDate**
Làm như thế nào?

Dùng con trỏ "this", như sau:

```
return *this;
```

Con trỏ "this": Minh hoạ (IV)

```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
    void setDay(int day){
        this->day = day;
    }
    Date& getDate(){
        return *this;
    }
    Date* getDatePtr(){

    }
};
```

Giả sử: cần trả về con trỏ đến đối tượng đang chứa hàm **getDatePtr**
Làm như thế nào?

Dùng con trỏ "this", như sau:

```
return this;
```

Con trỏ "this": Câu hỏi

```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
    void setDay(int day){
        this->day = day;
    }
    Date& getDate(){
        return *this;
    }
    Date* getDatePtr(){
        return this;
    }
};
```

```
int main(){
    Date d;
    cout << "ptr1:" << hex << &d << endl;
    cout << "ptr2:" << hex <<
            d.getDatePtr() << endl;
    return 0;
}
```


Giá trị in ra chỗ ptr1 và ptr2 có bằng nhau?

Thuật ngữ (I)

```
class Date{
public:
    int day, month, year;

    void print(){
        //đã rút gọn
    }
    void setDay(int day){
        this->day = day;
    }
    Date& getDate(){
        return *this;
    }
    Date* getDatePtr(){
        return this;
    }
};
```

Tính khả kiến

Thuật ngữ:

a) Biến thành viên (member variable)
b) Thuộc tính (atrribute)
c) Trường dữ liệu (field)

Thuật ngữ:

a) Phương thức, ít gọi là hàm (method)
b) Hành xử (behavior)
c) Hành động (operation)

Thuật ngữ (I)

Lớp (class): Date

```
class Date{
public:
    //đã rút gọn
};
```

```
int main(){
    Date d;
    Date *ptr = new Date();

    d.print();
    ptr->print();

    delete ptr;
    return 0;
}
```

Đối tượng: d (trên STACK)

Đối tượng: *ptr (trên HEAP)

Thuật ngữ:
a) Truyền thông điệp
b) Gọi phương thức
c) Gọi hàm

Lưu ý:

ptr : luôn luôn nằm trên STACK
*ptr: nằm trên HEAP (ở ví dụ này)

Tính khả kiến (visibility)

- Tính khả kiến
  - Là tính chất cho biết biến và hàm thành viên của lớp được nhìn thấy và dùng được (còn gọi là truy cập được) ở đâu.
  - Có 3 mức khả kiến:
    - public
    - protected
    - private

Tính khả kiến (visibility)

- **public**:
  - Thuộc tính hay phương thức có tính khả kiến là "public" thì
    - Chúng có thể được nhìn thấy và truy xuất được bởi bất kỳ đâu.
    - Nghĩa là, vị trị gọi phương thức hay truy xuất biến không nhất thiết chỉ là các phương thức thành viên của lớp đó, **có thể bất kỳ đâu**!

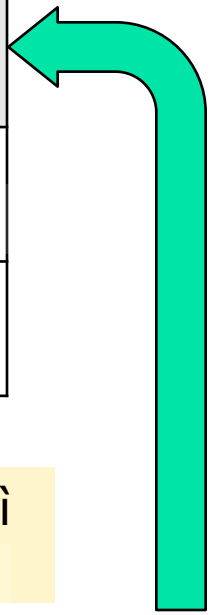
Tính khả kiến (visibility)

- **private**:
  - Thuộc tính hay phương thức có tính khả kiến là "private" thì,
    - Chúng **CHỈ CÓ THỂ** được nhìn thấy và truy xuất được ở các phương thức thành viên của lớp đó.

- **protected**:
  - Thuộc tính hay phương thức có tính khả kiến là "protected" thì,
    - Chúng có thể được nhìn thấy và truy xuất được:
      - (i) ở các phương thức thành viên của lớp đó và
      - (ii) **ở các phương thức của các lớp dẫn ra từ lớp đó** – xem phần thừa kế.

Tính khả kiến (visibility)

Tính khả kiến →	public	protected	private
Các phương thức của lớp **ClassX**	yes	yes	yes
Các phương thức của lớp con của lớp **ClassX**	yes	yes	**no**
Các phương thức và hàm không thuộc hai dạng trên	yes	**no**	**no**



Luôn luôn "yes": biến/hàm thành viên có tính "public" thì có thể truy cập ở bất kỳ đâu, không riêng gì lớp chứa nó.

Luôn luôn "yes": các phương thức của lớp ClassX thì luôn luôn truy cập được biến/hàm thành viên của lớp đó, bất kể chúng khai báo có tính khả kiến là gì.

Tính khả kiến (visibility): Minh hoạ (I)

```
class Foo{
private:
        int value;
protected:
        int reserved_value;
public:
        Foo(int value){
                this->value = value;
        }
        void print(){
                cout << this->value << endl;
        }
};
```

Cho dù value được khai báo với tính private, nhưng nó vẫn truy cập được trong các hàm thành viên: Foo và print.

Tóm lại: hàm thành viên luôn luôn truy cập được các biến thành viên và hàm thành viên khác, không quan tâm tính khả kiến là gì.

Tính khả kiến (visibility): Minh hoạ (I)

```
class Bar{
public:
        void func(){
                Foo obj(99);
                obj.print();
        }
};
```


Hàm func trong lớp Bar:

(a) gọi hàm khởi tạo – sẽ trình bày sau. Nó gọi được vì hàm này của lớp Foo có tính public.

(b) gọi hàm print. Gọi được vì print của Foo có tính public.

(c) TUY NHIÊN: nếu nó truy cập vào biến value và reserved_value của Foo thì lỗi – không truy cập được, do value có tính private, còn reserved_value thì có tính protected.

Tính khả kiến (visibility): Minh hoạ (I)

```
int main(){
        Bar bar;
        bar.func();

        Foo foo(100);
        foo.print();

        return 0;
}
```


Hàm main chương trình:

(a) gọi hàm khởi tạo – sẽ trình bày sau – để tạo đối tượng bar và foo (chữ thường). Nó gọi được vì các hàm này có tính public.

(b) gọi hàm func trên đối tượng bar và gọi hàm print trên foo. Gọi được vì func và print có tính public.

(c) TUY NHIÊN: nếu nó truy cập vào biến value và reserved_value của Foo thì lỗi – không truy cập được, do value có tính private, còn reserved_value thì có tính protected.

Thiết kế các lớp

- Mỗi lớp biểu diễn một tập hợp các đối tượng có liên quan chặt chẽ, có sự giống nhau về mặt gì đó.
  - Ví dụ:
    - Lớp Student:
      - Các sinh viên trong nhóm đều chịu sự quản lý như nhau bởi nhà trường, họ đều có danh sách các môn đã học, các môn sẽ học, có điểm trung bình HK, điểm TB tích luỹ, v.v.

- Do đó,
  - Cần dùng tính khả kiến để đảm bảo chỉ cung cấp ra bên ngoài (dùng tính public) những dữ liệu và phương thức được lựa chọn cẩn thận.
  - Đồng thời che dấu (tính private) những chi tiết về hiện thực cũng như những phương thức và dữ liệu nội bộ.
  - Có những trường dữ liệu và phương thức có thể chia sẽ với lớp con (dùng tính protected)

Thiết kế các lớp

- Gợi ý:
  - Thường che dấu dữ liệu của lớp → dùng tính **private**
  - Chỉ cho phép bên ngoài truy xuất dữ liệu thông qua những phương thức được thiết kế sẵn.
  - Đối với mỗi thuộc tính:
    - Bổ sung một phương thức cho phép bên ngoài đọc giá trị của thuộc tính → gọi là **getter, có tính public**
    - Bổ sung một phương thức cho phép bên ngoài ghi giá trị mới vào thuộc tính → gọi là **setter, có tính public**
  - **Getter**:
    - Lấy giá trị thuộc tính → định dạng → xuất ra ngoài
  - **Setter**:
    - Lấy giá trị được truyền vào → kiểm tra có hợp lệ → (a) báo lỗi hoặc (b) gán vào thuộc tính

Thiết kế các lớp: Minh hoạ (I)

- Với lớp Date:
  - day, month, year: nên là private
  - Với mỗi thuộc tính: có cặp setter và getter (public)
  - Có các phương thức tiện tích khác cho bên ngoài (tính public)
    - Lấy Date dưới dạng một chuỗi theo định dạng
    - Cho phép điều chỉnh định dạng xuất thàng chuỗi: dd/mm/yyyy, mm/dd/yy, v.v
    - In Date in màn hình (chỉ hữu dụng cho Debug)
    - So sánh giữa 2 đối tượng Date
    - Tính số ngày nằm giữa hai đối tượng Date
    - Các hàm khởi tạo tiện tích khác – Xem phần Hàm khởi tạo.
      - Khi tạo đối tượng có thể truyền ngày, tháng, và năm
      - Khi tạo không truyền gì thì lấy ngày, tháng, năm hiện tại của máy tính.

Thiết kế các lớp: Minh hoạ (I)

```
class Date{
private:
    int day, month, year;
public:
    int getDay(){
        return day;
    }
    void setDay(int new_day){
        day = new_day;
    }
    int getMonth(){
        return month;
    }
    void setMonth(int new_month){
        month = new_month;
    }
    //... Xem slide kế tiếp
};
```

Setter và Getter cho thuộc tính day

Setter và Getter cho thuộc tính month

Thiết kế các lớp: Minh hoạ (I)

```
class Date{
private:
    int day, month, year;
public:
    //... Xem slide kế trước
    int getYear(){
        return year;
    }
    void setYear(int new_year){
        year = new_year;
    }

    void print(){
        cout << day << "/" << month << "/" << year << endl;
    }
};
```

Setter và Getter cho thuộc tính year

Thiết kế các lớp: Bài tập (I)

- Bổ sung các phương thức khác như so sánh hai đối tượng Date, tính số ngày giữa hai Date, v.v. – xem slide trước.
- Viết chương trình ngắn sử dụng các phương thức đã tạo cho Date.

Hàm khởi tạo: Dẫn nhập

```
int main(int argc, char** argv) {
    Date d;
    d.setDay(20);
    d.setMonth(5);
    d.setYear(2017);
    d.print();

    return 0;
}
```

Khai báo một đối tượng "d".
Gọi các setter để gán: ngày, tháng, và năm
Gọi hàm "print" để in ra đối tượng "c" theo định dạng: dd/mm/yyyy

Việc tạo đối tượng "d" quá dài dòng, cần gọi đến 3 Setter!

Dùng hàm khởi tạo

Hàm khởi tạo (constructor)

- Đặc điểm cú pháp:
  - Không có kiểu trả về
  - Tên hàm cũng chính là tên lớp
  - Có thể có nhiều hàm khởi tạo, chỉ cần khác chữ ký (overloading)

- Đặc điểm ngữ nghĩa:
  - Chỉ được gọi **1 và chỉ 1** lần duy nhất – chính là lúc tạo ra đối tượng.

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Khởi tạo mặc nhiên**
    a) Chỉ có hiệu lực khi không mô tả tường minh bất kỳ hàm khởi tạo nào.
    b) Không có thông số → khi tạo đối tượng không truyền bất cứ đối số nào.

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Khởi tạo mặc nhiên**
    - a) **Ví dụ:**
      - a) Nếu `MyClass` là một lớp, thì dòng lệnh sau
        ```
        MyClass a;
        ```
        Sẽ tạo đối tượng "a".
      - b) Bộ thực thi sẽ tìm hàm khởi tạo không có thông số do người lập trình định nghĩa.
        - Nếu không thấy, nhưng thì lại thấy có hàm khởi tạo khác → **báo lỗi**.
        - Nếu không thấy bất kỳ hàm khởi tạo nào → gọi hàm khởi tạo mặc nhiên, khởi tạo các thuộc tính vào trạng thái mặc nhiên.

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Khởi tạo copy (copy constructor):**
    - Có prototype là **một trong** các dạng sau:

```
MyClass( const MyClass& other );
MyClass( MyClass& other );
MyClass( volatile const MyClass& other );
MyClass( volatile MyClass& other );
```

(http://www.cplusplus.com/articles/y8hv0pDG/)

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Khởi tạo copy (copy constructor):**
    - Hàm copy contructor này có thể được gọi khi:
      a) Tạo đối tượng và truyền vào đối số là đối tượng

      Ví dụ:

      MyClass a;

      MyClass b(a); //truyền "a" vào hàm khởi tạo khi tạo ra "b"

      b) Tạo đối tượng và khởi gán bằng đối tượng khác

      Ví dụ:

      MyClass a;

      MyClass b = a; //khởi gán "a" cho "b"

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Khởi tạo copy (copy constructor):**
    - Hàm copy contructor này có thể được gọi khi:
      c) Truyền đối tượng bằng trị vào hàm

      Ví dụ:

```
void foo(MyClass x){ …}
//…
MyClass a;
foo(a);  //Hàm copy constructor sẽ được gọi để tạo ”x” từ “a”
```

      d) Hàm trả về giá trị

      Ví dụ:

```
MyClass a = bar();
//Với bar() là hàm trả về đối tượng kiểu MyClass
```

Hàm khởi tạo (constructor)

- Các loại hàm khởi tạo
  - **Các hàm do người lập trình định nghĩa khác:**
    - Một hàm khởi tạo do người lập trình định nghĩa sẽ được gọi khi tạo ra tạo đối tượng có kèm theo các thông số trùng khớp với chữ ký của hàm khởi tạo này.
    - Ví dụ: khi lớp MyClass có hàm khởi tạo là:


```
MyClass(int a, double* ptr);
```

- Hàm khởi tạo đó sẽ được gọi với đoạn code sau:

```
double list[] = {10.5, 20.5, 40.5};
MyClass obj(1, list);
//Tại đây có đối tượng obj
```

Hàm huỷ (destructor)

- Đặc điểm cú pháp:
  - Không có kiểu trả về
  - Tên hàm có dạng: `~Class`. Ở đó, `Class` là tên lớp
  - Không có thông số
  - Nếu, cho phép kế thừa thì nên có thêm từ khoá "virtual" đứng ở đầu – xem phần kế thừa.
- Đặc điểm ngữ nghĩa:
  - Chỉ được gọi **1 và chỉ 1** lần duy nhất – chính là lúc cần huỷ đối tượng. Cụ thể, là khi nào?
    a) Khi ra khỏi tầm vực (kể cả kết thúc hàm) → huỷ các đối tượng trong tầm vực vừa ra.
    b) Khi người lập trình chủ động gọi delete.

Hàm khởi tạo & Hàm huỷ

- Công dụng:
  - Hàm khởi tạo:
    - Dùng để khởi tạo tình trạng (trạng thái) ban đầu cho đối tượng khi được tạo ra.
  - Hàm huỷ:
    - Làm công việc dọn dẹp để chuẩn bị cho đối tượng bị huỷ.

Hàm khởi tạo & Hàm huỷ

- Công dụng:
  - Thường gặp I: **Đơn giản**
    - Hàm khởi tạo: khởi gán các biến, đưa đối tượng về trạng thái ban đầu nào đó
    - Hàm huỷ: không cần

  - Thường gặp II:
    - Một biến thành viên có kiểu con trỏ, **cần được cấp phát động**
    - Hàm khởi tạo: gọi đến "new" để xin bộ nhớ, và khởi động giá trị
    - Hàm huỷ: gọi đến "delete" để giải phóng bộ nhớ.

Hàm khởi tạo & Hàm huỷ

- Công dụng:
  - Thường gặp III:
    - Các hàm thành viên khác đọc ghi dữ liệu xuống một file cụ thể
    - Hàm khởi tạo: mở file, chuẩn bị cho việc ghi/đọc.
    - Hàm huỷ: đóng file.
  - Thường gặp IV:
    - Hàm khởi tạo cần khởi tạo **trên cùng dòng** các biến thành viên hay các đối tượng của lớp cha.
    - Trường hợp biến thành viên có tính "const" → xem khởi động trong phần theo sau.
    - Trường hợp cần khởi động đối tượng của lớp cha → xem phần thừa kế.

Hàm khởi tạo & hàm huỷ: Minh hoạ

- Yêu cầu:
  - Xây dựng một lớp hỗ trợ tính toán với ma trận, cụ thể:
    - Cho phép dễ dàng tạo ra đối tượng biểu diễn ma trận → hàm khởi tạo
    - Cho phép các hàm cho phép xử lý ma trận dễ dàng.

Hàm khởi tạo & hàm huỷ: Minh hoạ

- Phân tích:
  - Tên lớp: Matrix
  - Dữ liệu:
    - Số hàng, số cột
    - Con trỏ đến dữ liệu của ma trận, nên **dùng cấp phát động**.
  - Hàm khởi tạo:
    - Không có tham số: tạo ma trận rỗng, cẩn thận với con trỏ dữ liệu
    - Hàm khởi tạo copy: cấp phát bộ nhớ cho ma trận mới và copy dữ liệu từ đối tượng truyền vào.
      - **Thành viên được cấp phát động → BẮT BUỘC phải định nghĩa lại hàm này theo ý trên.**
    - Hàm khởi tạo khác: cho phép khởi động từ mảng truyền vào
  - Hàm huỷ:
    - Giải phóng bộ nhớ, nếu đã xin được.

Hàm khởi tạo & hàm huỷ: Minh hoạ

- Phân tích:
  - Các hàm thành viên khác:
    - Hàm "print" để kiểm tra
    - Hàm chuyển dữ liệu ma trận thành một chuỗi.
    - Hàm lấy giá trị tại vị trí hàng và cột truyền vào
  - Các toán tử: (xem thêm phần operator overloading)
    - Phép gán copy (copy assignment): kiểm tra self-assignment, cấp phát bộ nhớ cho ma trận mới và copy dữ liệu từ đối tượng truyền vào.
      - **Thành viên được cấp phát động → BẮT BUỘC phải định nghĩa lại hàm này theo ý trên.**
    - Các phép toán khác cho ma trận: +, -, *, / (inverse)
    - Các phép toán khác giữa ma trận và các số vô hướng.

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
   //Xem slide sau
private:
    double *m_data_ptr;
    int m_nrows, m_ncols;
};
```

Các biến thành viên (thuộc tính):
Số hàng, số cột và con trỏ đến dữ liệu

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
    Matrix(){
        this->m_nrows = 0;
        this->m_ncols = 0;
        this->m_data_ptr = NULL;
    }
    Matrix(int nrows, int ncols){
        this->m_nrows = nrows;
        this->m_ncols = ncols;
        this->m_data_ptr = new double[nrows*ncols]();
    }
   //Xem slide sau
   //...
};
```

Tạo ma trận rỗng.

Cho phép đặc tả số hàng và số cột khi tạo ma trận

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
    Matrix(const Matrix& right){
      if(right.m_data_ptr != NULL){
          //object in parameters has its data
          this->m_nrows = right.m_nrows;
          this->m_ncols = right.m_ncols;
          int size = right.m_nrows*right.m_ncols;
          this->m_data_ptr = new double[size];
          //copy from right.m_data_ptr to this->m_data_ptr
          std::memcpy(this->m_data_ptr,
                  right.m_data_ptr,
                  size*sizeof(double));
      }
    }
   //Xem slide sau
   //...
};
```

Hàm khởi tạo copy:
Cho phép COPY từ một đối tượng có sẵn

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
    Matrix(int nrows, int ncols, double* data_ptr, int count){
        this->m_nrows = nrows;
        this->m_ncols = ncols;
        this->m_data_ptr = new double[nrows*ncols]();
        int size = nrows*ncols;
        size = (size < count? size: count);
        //copy from right.m_data_ptr to this->m_data_ptr
        std::memcpy(this->m_data_ptr,
                data_ptr,
                size*sizeof(double));
    }
   //Xem slide sau
   //...
};
```

Cho phép COPY dữ liệu từ array truyền vào

Số phần tử thực sự được copy là số nhỏ nhất giữa số phần tử của ma trận và giá trị biến count.

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
    ~Matrix(){
       if(this->m_data_ptr != NULL) delete this->m_data_ptr;
    }

   //Xem slide sau
   //…
};
```

Hàm huỷ

Luôn luôn kiểm tra con trỏ != NULL trước khi gọi delete!

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
class Matrix{
public:
    void print(){
        if(this->m_data_ptr == NULL){
            cout << "Empty matrix" << endl;
        };
        for(int rows=0; rows < this->m_nrows; ++rows){
            for(int cols=0; cols < this->m_ncols; ++cols){
              int index = this->m_ncols*rows + cols;
                cout << fixed << setw(5) << setprecision(2)
                    << this->m_data_ptr[index]
                      << string(' ', 2);
            }//cols
            cout << endl;
        }//rows
    }//print
    //Xem slide sau
    //...
};
```

Hàm in dữ liệu

Hàm khởi tạo & hàm huỷ: Minh hoạ

```
int main(int argc, char** argv) {
    double data[] = {1,2,3,4,5,6,7};
    Matrix m1(2,3, data, 7);
    m1.print();
    {
            Matrix m2(m1);
            m2.print();

            Matrix m3;
            m3.print();
    }
    Matrix m4(3,4);
    m4.print();

    return 0;
};
```

Hàm khởi nào?

Hàm khởi nào?

Hàm khởi nào?

Kết thúc một tầm vực: hàm huỷ nào được gọi?

Hàm khởi nào?

Hàm khởi tạo & hàm huỷ: Bài tập

- Hiện thực đầy đủ các phương thức cho ma trận như đã gợi ý ở các slide trước.

Định nghĩa lại các toán tử (operator overloading)

- Nhiều toán tử có sẵn trong C++ có thể định nghĩa lại (overloading) để chúng có thể chấp nhận các toán hạng là đối tượng.

- Các toán tử có thể định nghĩa là:


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

Định nghĩa lại các toán tử (operator overloading)

- Trong số đó, toán tử ”=“ (toán tử gán) là đặc biệt chú ý.
  - Thường gặp:
    - Nếu một biến thành viên được **cấp phát động**, thì :
      - Cần phải định nghĩa lại **toán tử gán**
      - và cần phải định nghĩa lại **hàm khởi tạo copy**.

    - Xem ví dụ về ma trận: hàm khởi tạo copy được định nghĩa lại để nó **không dùng chung** bộ nhớ với đối tượng khác!

Định nghĩa lại các toán tử (operator overloading)

Các prototype phổ biến của toán tử gán

(1) MyClass& operator=(const MyClass& rhs);
(2) MyClass& operator=(MyClass& rhs);
(3) MyClass& operator=(MyClass rhs);
(4) const MyClass& operator=(const MyClass& rhs);
(5) const MyClass& operator=(MyClass& rhs);
(6) const MyClass& operator=(MyClass rhs);
(7) MyClass operator=(const MyClass& rhs);
(8) MyClass operator=(MyClass& rhs);
(9) MyClass operator=(MyClass rhs);

(nguồn: http://www.cplusplus.com/articles/y8hv0pDG/)

Định nghĩa lại các toán tử: Minh hoạ

```
Matrix& operator=(const Matrix& right){
        cout << "operator=" << endl;
        if(this == &right){
            //self-assignment
            return *this;
        }
        //DO DEEP COPY
        //(1)-> free data on this
        if(this->m_data_ptr != NULL){
            delete this->m_data_ptr;
        }
        //(2)->check if the right does not have data
        if(right.m_data_ptr == NULL){
            this->m_data_ptr = NULL;
            this->m_nrows= 0;
            this->m_ncols= 0;
            return *this;
        }
        //(3)->allocate and copy data
        this->m_nrows = right.m_nrows;
        this->m_ncols = right.m_ncols;
        int size = right.m_nrows*right.m_ncols;
        this->m_data_ptr = new double[size];
        //copy from right.m_data_ptr to this->m_data_ptr
        std::memcpy(this->m_data_ptr,
                right.m_data_ptr,
                size*sizeof(double));
    }
```

Toán tử gán cho Matrix

```
int main(int argc, char** argv) {
    double data[] = {1,2,3,4,5,6,7};
    Matrix m1(2,3, data, 7);
    m1.print();

    //... Xem slide trước

    Matrix m5;
    m5 = m1;
    m5.print();

    return 0;
}
```

Phép gán

Định nghĩa lại các toán tử: Minh hoạ

Toán tử cộng giữa 2 ma trận

```
Matrix operator+(const Matrix& right){

        Matrix rs(this->m_nrows, this->m_ncols);

        if( (this->m_nrows != right.m_nrows) ||
            (this->m_ncols != right.m_ncols))
            return rs; //better if signal error or throw exp

        int size = this->m_nrows*this->m_ncols;
        for(int idx=0; idx < size; ++idx){
            rs.m_data_ptr[idx] = this->m_data_ptr[idx] +
                    right.m_data_ptr[idx];
        }
        return rs;
}
```

Định nghĩa lại các toán tử: Minh hoạ

Sử dụng toán tử cộng giữa 2 ma trận

```
int main(int argc, char** argv) {
    double data[] = {1,2,3,4,5,6,7};
    Matrix m1(2,3, data, 7);
    m1.print();

    Matrix m2(2,3,data, 6);
    m2.print();

    Matrix m3 = m1 + m2;
    m3.print();

    return 0;
}
```

Phép cộng

Thành viên có tính "static"

- Thuộc tính và phương thức đều có thể có tính "static" – nếu có khai báo.

- Thuộc tính có tính "static":
  - Vùng nhớ cho nó không được tạo riêng cho mỗi đối tượng như trường hợp của thuộc tính "non-static"
  - Vùng nhớ cho nó được cấp 1 lần, chỉ một vùng, và dùng chung cho tất cả các đối tượng được tạo ra từ lớp đó.
    - Khi một đối tượng thay đổi giá trị của dữ liệu có tính "static"
    - Các đối tượng khác sẽ nhìn thấy sự thay đổi này – vì dùng chung.

Thành viên có tính "static"

- Phương thức có tính "static":
  - Phương thức dạng này không kết hợp với bất kỳ đối tượng cụ thể nào.
    - Tức là, trong phương thức dạng này, con trỏ "this" không có giá trị - vì nó không gắn với bất kỳ đối tượng nào.
    - Cũng có nghĩa, dùng con trỏ "this" ở hàm dạng này sẽ có lỗi.
  - **Do đó,**
    - Phương thức "static" không thể gọi các phương thức "non-static" trực tiếp. Nếu muốn, nó phải tạo đối tượng và gọi phương thức trên đối tượng.
    - Phương thức "static" có thể:
      - Truy xuất thuộc tính có tính "static"
      - Gọi các phương thức có tính "static" khác.

Thành viên có tính "static"

- Phương thức có tính "static":
  - Ví dụ thực tế: Giảng viên có thể gửi thông điệp
    - Đến một sinh viên cụ thể
      - C++: "**Đối tượng**" giảng viên lấy tham chiếu đến "**đối tượng**" sinh viên, và "**gửi thông điệp**" (gọi hàm) đến đối tượng này.

Thành viên có tính "static"

- Phương thức có tính "static":
  - Ví dụ thực tế: Giảng viên có thể gửi thông điệp
    - Hoặc, đến cả nhóm sinh viên
      - "C++: **Đối tượng**" giảng viên **KHÔNG CẦN** lấy tham chiếu đến "**đối tượng**" sinh viên, vì nó (đối tượng) không gửi thông điệp đến sinh viên cụ thể.
      - "**Đối tượng**" giảng viên dùng tên lớp (ví dụ, lớp Student) để chuyển thông điệp, thông điệp có tầm ảnh hưởng là lớp (nhóm) sinh viên.

Thành viên có tính "static": Minh hoạ (I)

- Với ứng dụng liên quan Matrix:
  - Giả sử, cần định dạng dạng hiển thị của ma trận ở màn hình hay tập tin (văn bản), các định dạng cơ bản:
    - Độ rộng của mỗi phần tử
    - Độ chính xác của mỗi phần tử
  - Cũng giả sử rằng, việc đinh dạng này là chung cho tất cả các ma trận bắt đầu từ thời điểm thiết lập định dạng.
- Câu hỏi:
  - Lưu trữ định dạng như trên như thế nào? Theo từng đối tượng hay dùng chung
    - ➔ Yêu cầu ở trên là dùng chung.
  - Thực tế: cũng có thể lưu riêng và chung (cả hai). Giá trị lưu riêng được ưu tiên hơn → có thể thiết kế như vậy.

Thành viên có tính "static": Minh hoạ (I)

```
Class Matrix{
public:
    static int width;
    static int precision;
// Các thành viên khác xem các slide trước
};

int Matrix::width = 10;
int Matrix::precision = 3;
```

Phạm vi của lớp Matrix

Khai báo một biến có tính static, có hai bước:

(1) Khai báo bên trong lớp, dùng từ khoá static

(2) Khởi động bên ngoài phạm vi lớp, dùng tên đầy đủ, sử dụng tiếp đầu ngữ: `Matrix::`, không dùng từ khoá static

Thành viên có tính "static": Minh hoạ (I)

```
int main(){
        Matrix m1;
        m1.width = 15;
        m1.precision = 5;

        Matrix m2;
        cout << m2.width << endl;
        cout << m2.precision << endl;

        return 0;
}
```

Chương trình in ra:

15
5

Vì sao?

Cả hai đối tượng "m1" và "m2" dùng chung vùng nhớ "width" và "precision".
Do vậy, khi "m1" thay đổi thì "m2" nhìn thấy giá trị thay đổi đó.

Thành viên có tính "static": Minh hoạ (I)

```
int main(){
        Matrix m1;
        Matrix::width = 15;
        Matrix::precision = 5;

        Matrix m2;
        cout << Matrix::width << endl;
        cout << Matrix::precision << endl;

        return 0;
}
```

Bộ biên dịch sẽ "warning" khi truy xuất thành viên có tính "static" như trong slide trước. Vì thực ra, các thành viên này không gắn trực tiếp vào đối tượng.

Do đó, hãy dùng cách viết như trong slide này.

Thành viên có tính "static": Minh hoạ (II)

- Với ứng dụng liên quan Matrix:
  - Giả sử các thành viên "width" và "precision" có khai báo "private" thay cho "public"
    - ➔ Hàm main cũng như các hàm không phải thành viên của Matrix không thể truy cập được.
    - ➔ Bổ sung các setter và getter cho chúng
    - ➔ Các setter và getter này cũng phải là **phương thức có tính "static".**
  - Giả sử cần cung cấp phương thức để chuyển vị ma trận. Có thể có thiết kế như sau:

Thành viên có tính "static": Minh hoạ (II)

- Với ứng dụng liên quan Matrix:
  - Giả sử cần cung cấp phương thức để chuyển vị ma trận. Có thể có thiết kế như sau:
    - (1) Bên gọi (caller) gọi "T()" trên đối tượng "m", nghĩa là, m.T(). Hàm "T()" có chuyển vị ma trận đang giữa bởi "m", rồi trả về chính "m" sau khi chuyển vị. **("m" bị thay đổi)**
      - ➔ Cách này không dùng "static"
    - (2) Bên gọi (caller) gọi "T()" trên đối tượng "m", nghĩa là, m.T(). Hàm "T()" sẽ tạo một ma trận trung giang copy từ "m", chuyển vị ma trận trung gian và trả về ma trận trung gian. **("m" không bị thay đổi)**
      - ➔ Cách này không dùng "static"

Thành viên có tính "static": Minh hoạ (II)

- Với ứng dụng liên quan Matrix:
  - Giả sử cần cung cấp phương thức để chuyển vị ma trận. Có thể có thiết kế như sau:
    - (3) Nếu bản thiết kế cần cung cấp cả hai tình huống: **"m" bị thay đổi và không bị thay đổi, thì**
      - ➔ **nên dùng static**

    - Cụ thể:
      - Phương thức "non-static": **T()** là nhằm chuyển vị và thay đổi cả ma trận đón nhận thông điệp **T()**.
      - Phương thức "static": T(const Matrix& mt)
        - Phương thức này tạo ma trận trung gian, copy từ mt. Chuyển vị và trả về ma trận trung gian.

Thành viên có tính "static": Minh hoạ (II)

```
class Matrix{
public:
        Matrix T(){
                //chuyển vị tại đây, dùng đối tượng "*this"
                return *this;
        }
        static Matrix T(const Matrix mt){
                Matrix rs = mt;

                //Chuyển vị dùng đối tượng "rs"
                return rs;
        }
        //Các thành viên khác ...
};
```


Hàm có tính static, được khai báo và hiện thực theo ví dụ

Thành viên có tính "static": Minh hoạ (II)

```
int main(){
        double list[] = {1,2,3,4,5,6};
        Matrix m1(2,3, list, 6);
        Matrix m2 =  Matrix::T(m1);

        m1.print(); cout << endl;
        m2.print(); cout << endl;


        cout << endl << endl;
        system("pause");

        return 0;
}
```

Gọi hàm có tính "static"

Kết quả:
Ma trận m1, kích thước: 2 x 3
Ma trận m2, kích thước: 3 x 2

Thành viên có tính "static": Bài tập

- Hiện thực các hàm chuyển vị cho ma trận dùng "static" và "non-static" theo gợi ý.
- Hiện thực các setter và getter cho các thuộc tính static là width và precision. Lưu ý, tính khả kiến của width và precision là private.
- Chỉnh lại hàm print ở các slide trước để có dùng width và precision trong định dạng đầu ra.

Thành viên có tính "const"

- Thuộc tính có tính "const":
  - Thuộc tính dạng này sẽ không cho phép sự thay đổi giá trị kể từ sau khi khởi động giá trị của nó.
    - Khởi động giá trị của thuộc tính const bằng cách **khởi động trên cùng hàm khởi tạo** – xem phần minh hoạ.
  - Ứng dụng:
    - Giúp cho người lập trình duy trì một giá trị là hằng số suốt trong thời gian sống của giá trị.

Thành viên có tính "const"

- Phương thức có tính "const":
  - Phương thức dạng này sẽ không thể thay đổi bất kỳ biến thành viên nào của lớp
    - Nghĩa là, phương thức **chỉ đọc** giá trị của các biến thành viên
    - Ngoại lệ,
      - Phương thức dạng này có thể thay đổi biến thành viên nếu biến đó **khai báo tường minh** với từ khoá **mutable** – xem phần minh hoạ.
  - Ứng dụng:
    - Trong một dự án có nhiều người tham gia, hạn chế truy xuất bằng cách dùng const như thế này sẽ tăng tính toàn vẹn dữ liệu, đảm bảo dữ liệu luôn luôn trong trạng thái đúng.

Thành viên có tính "const": Minh hoạ (I)

```
class ClassX{
private:
        int nonconst_value;
        const int const_value;
        int mutable mutable_value;
public:
        ClassX(int value):
                const_value(value),
                nonconst_value(100),
                mutable_value(200){
        }
        int process() const{
                int rs;
                rs = const_value + nonconst_value;
                //const_value = 100;
                //nonconst_value = 100;
                mutable_value = 100;
                rs += mutable_value;
                return rs;
        }
};
```

nonconst_value: thuộc tính không phải hằng, không phải mutable

const_value: thuộc tính có tính hằng

mutable_value: thuộc tính có thay đổi (mutable)

Thành viên có tính "const": Minh hoạ (I)

```
class ClassX{
private:
        int nonconst_value;
        const int const_value;
        int mutable mutable_value;
public:
        ClassX(int value):
                const_value(value),
                nonconst_value(100),
                mutable_value(200){
        }
        int process() const{
                int rs;
                rs = const_value + nonconst_value;
                //const_value = 100;
                //nonconst_value = 100;
                mutable_value = 100;
                rs += mutable_value;
                return rs;
        }
};
```

Cách khởi gán **trên** (không phải bên trong) hàm khởi tạo, chú ý dùng dấu hai chấm **":"** và dùng dấu phẩy **","** để ngăn cách các phần khởi động

Thành viên có tính "const": Minh hoạ (I)

```
class ClassX{
private:
        int nonconst_value;
        const int const_value;
        int mutable mutable_value;
public:
        ClassX(int value):
                const_value(value),
                nonconst_value(100),
                mutable_value(200){
        }
        int process() const{
                int rs;
                rs = const_value + nonconst_value;
                //const_value = 100;
                //nonconst_value = 100;
                mutable_value = 100;
                rs += mutable_value;
                return rs;
        }
};
```

Từ khoá **const** cho biết là Phương thức có tính hằng.

Thành viên có tính "const": Minh hoạ (I)

```
class ClassX{
private:
        int nonconst_value;
        const int const_value;
        int mutable mutable_value;
public:
        ClassX(int value):
                const_value(value),
                nonconst_value(100),
                mutable_value(200){
        }
        int process() const{
                int rs;
                rs = const_value + nonconst_value;
                //const_value = 100;
                //nonconst_value = 100;
                mutable_value = 100;
                rs += mutable_value;
                return rs;
        }
};
```

Từ khoá **const** cho biết là Phương thức có tính hằng.

Dù phương thức có tính hằng, nhưng **vẫn có thể thay đổi** biến thành viên có tính "mutable" – xem dòng khai báo cho biến.

Thành viên có tính "const": Minh hoạ (I)

```
class ClassX{
private:
        int nonconst_value;
        const int const_value;
        int mutable mutable_value;
public:
        ClassX(int value):
                const_value(value),
                nonconst_value(100),
                mutable_value(200){
        }
        int process() const{
                int rs;
                rs = const_value + nonconst_value;
                //const_value = 100;
                //nonconst_value = 100;
                mutable_value = 100;
                rs += mutable_value;
                return rs;
        }
};
```

Nếu hai dòng này được biên dịch → **lỗi biên dịch**.

Lý do: phương thức có tính hằng chỉ có thể thay đổi biến có tính "mutable".

Thuật ngữ (II)

- Tổng kết các khái niệm quan trọng:

Tiếng việt	Tiếng Anh	Chú thích
Lớp	class	Xem như một kiểu dữ liệu
Đối tượng	Object, instance	Xem như một biến
Thuộc tính; Biến thành viên; Trường dữ liệu; Thành viên dữ liệu	Attribute; Member variable; Data field; Data member	Dữ liệu + Phương thức được gói lại thành đối tượng (gói dữ liệu và địa chỉ hàm)
Phương thức; Hàm thành viên; Hành xử; Hành động	Method; Function member; Behavior; operation	

Thuật ngữ (II)

- Tổng kết các khái niệm quan trọng:

Tiếng việt	Tiếng Anh	Chú thích
Tính khả kiến	Visibility	Cho biết một thuộc tính & phương thức có thể dùng được ở đâu trong mã nguồn (lớp nào, phương thức nào, hàm nào?)
Ba tính khả kiến: public, private, và protected	public, private, và protected	**public**: mọi nơi **private**: của riêng **protected**: của riêng + dành cho con và cháu.
Hàm đọc dữ liệu	getter	**Nên:** • Các thuộc tính có tính khả kiến là: private • Mỗi thuộc tính nên kèm theo 01 getter và 01 setter.
Hàm ghi dữ liệu	setter	

Thuật ngữ (II)

- Tổng kết các khái niệm quan trọng:

Tiếng việt	Tiếng Anh	Chú thích
từ khoá "this"	this	this: có kiểu là địa chỉ. this: chứa địa chỉ đến byte đầu tiên của vùng nhớ của đối tượng. this: chỉ được nhìn thấy bên trong các hàm thành viên. ➔ this->data_member; this->foo(); với: data_member: thuộc tính foo(): một phương thức của đối tượng.

Thuật ngữ (II)

- Tổng kết các khái niệm quan trọng:

Tiếng việt	Tiếng Anh	Chú thích
Hàm khởi tạo copy	Copy constructor	Khi biến thành viên được cấp phát động (như ma trận ở slide trước) **CẦN PHẢI:** ➔ Định nghĩa lại cả "copy contructor" và "copy assignment"
Phép gán	Copy assignment	

Quan hệ bạn bè (friendship)

- Giả sử lớp ClassA và ClassB là hai lớp độc lập, không có quan hệ thừa kế (sẽ trình bày sau)
  - Các phương thức thành viên của ClassA không thể truy cập các thành viên (biến & hàm) có tính private và protected của ClassB và ngược lại.

- Trong dự án có một số lớp cần cho các lớp khác truy xuất các thành viên (biến & hàm) có tính private và protected của nó. Vì lý do:
  - Code cho đơn giản.
  - Tránh gọi hàm nhiều lần → chạy nhanh hơn.
  - Các lớp này được thiết kế cẩn thận và nằm trong cùng dự án nên vẫn có thể kiểm soát sự toàn vẹn dữ liệu được.

Quan hệ bạn bè (friendship)

- Các dạng friendship
  - **Hàm friend**
    - Cho phép một hàm không phải thành viên của lớp ClassX vẫn có thể truy cập các thành viên (biến & hàm) có tính private và protected của ClassX.
  - **Lớp friend**
    - Cho phép tất cả các hàm thành viên của ClassY có thể truy cập các thành viên (biến & hàm) có tính private và protected của ClassX.

Quan hệ bạn bè (friendship): Minh hoạ (I)

- **Hàm friend:**
  - Với lớp Matrix ở các slide trước, hãy bổ sung code để cho phép người lập trình có thể thực hiện các phép toán sau:
    - Cho phép người lập trình nhập và xuất ma trận bằng các toán tử: << và >>
    - Cộng/trừ/nhân/chia một số vô hướng (số thực, double) với ma trận, như ví dụ sau. Ý nghĩa của các phép toán này là thực hiện phép toán đó giữa số vô hướng với từng phần tử của ma trận

```
double data[] = {1,2,3,4,5,6};
Matrix m1(2,3, data, 6);
Matrix m2, m3, m4;
m2 = 3.5 + m1;
m3 = 3.5 - m1;
m4 = 3.5 * m1;
m5 = 3.5 / m1;
```

Quan hệ bạn bè (friendship): Minh hoạ (I)

- Phân tích:
  - Prototype của các toán tử như sau:

```
ostream &operator<<( ostream &output, const Matrix& matrix);
istream &operator>>( istream &input, Matrix& matrix);
Matrix operator+(double value, const Matrix& matrix);
Matrix operator-(double value, const Matrix& matrix);
Matrix operator*(double value, const Matrix& matrix);
Matrix operator*(double value, const Matrix& matrix);
```

  - Với dạng protoype như vậy, chúng không thể là hàm thành viên của lớp Matrix.
  - Nhưng, chúng ta truy xuất các dữ liệu: số hàng, số cột, và dữ liệu (có tính private) của Matrix
  - ➔ **Dùng Hàm friend**

Quan hệ bạn bè (friendship): Minh hoạ (I)

```
class Matrix{
public:
    friend ostream &operator<<( ostream &output,
    const Matrix& matrix);
    friend istream &operator>>( istream &input, Matrix& matrix);
    friend Matrix operator+(double value, const Matrix& matrix);
    friend Matrix operator-(double value, const Matrix& matrix);
    friend Matrix operator*(double value, const Matrix& matrix);
    friend Matrix operator*(double value, const Matrix& matrix);
// Các hàm thành viên khác, xem các slide trước.

private:
    double *m_data_ptr;
    int m_nrows, m_ncols;
};
```

Khai báo hàm (toán tử) friend

Quan hệ bạn bè (friendship): Minh hoạ (I)

```
ostream &operator<<( ostream &output, const Matrix& matrix){
        // Các lệnh xuất dữ liệu tại đây
        return output;
}
```

Định nghĩa toán tử <<


```
istream &operator>>( istream &input, Matrix& matrix){
        // Các lệnh nhập dữ liệu tại đây
        return input;
}
```

Định nghĩa toán tử >>

Quan hệ bạn bè (friendship): Minh hoạ (I)

```
Matrix operator+(double value, const Matrix& matrix){
        Matrix rs = matrix;
        //Cộng các phần tử của rs với value, gán lại vào rs.
        return rs;
}
```

```
Matrix operator-(double value, const Matrix& matrix){
        Matrix rs = matrix;
        //Trừ value cho các phần tử của rs, gán lại vào rs.
        return rs;
}
```

Định nghĩa toán tử + và –

Các toán tử * và / tương tự.

Quan hệ bạn bè (friendship): Minh hoạ (II)

- **Lớp friend:**
  - Với lớp Matrix ở các slide trước, giả sử cần thiết kế lớp xử lý dữ liệu (DataProcessor).
  - Lớp DataProcessor có nhiều hàm thành viên, chúng cần lưu dữ liệu dưới dạng ma trận, và chúng cần truy xuất dữ liệu của ma trận (số hàng, số cột, và dữ liệu các phần tử).
    - => Có thể sử dụng lớp friend để cho phép tất cả các phương thức trong DataProcessor truy xuất được các dữ liệu liệu của ma trận (có tính private).
  - Khai báo lớp DataProcessor là friend của lớp Matrix như sau.

Quan hệ bạn bè (friendship): Minh hoạ (I)

```
Class DataProcessor; //Khai báo trước cho lớp DataProcessor
class Matrix{
public:
    friend class DataProcessor;

    friend ostream &operator<<( ostream &output,
    const Matrix& matrix);
    friend istream &operator>>( istream &input, Matrix& matrix);
    friend Matrix operator+(double value, const Matrix& matrix);
    // Các hàm thành viên khác, xem các slide trước.
private:
    double *m_data_ptr;
    int m_nrows, m_ncols;
};
```


Khai báo lớp friend

Chú ý cả dòng khai báo trước cho lớp DataProcessor → để danh hiệu DataProcessor được biết là một lớp

Lớp DataProcessor: không có gì đặc biệt khi kết hợp với friend

Tổ chức mã nguồn cho các lớp

- Một lớp có nhiều hàm thành viên, phần thân những hàm này có thể để chung với phần mô tả của lớp, như ở các ví dụ trước.
- Tuy vậy, thường các dự án lớn (đặc biệt là dạng thư viện phần mềm), mỗi lớp được tổ chức thành 2 tập tin riêng biệt:
  - Tập tin header (*.h):
    - Tập tin này chứa **phần mô tả (description, declaration)** cho lớp
  - Tập tin source (*.cpp):
    - Tập tin này chứa **phần định nghĩa (definition)** cho lớp

Tổ chức mã nguồn: Minh hoạ

```
#ifndef DATE_H
#define DATE_H
class Date{
public:
   int getDay(void);
   void setDay(int newDay);
   int getMonth(void);
   void setMonth(int newMonth);
   int getYear(void);
   void setYear(int newYear);
   int compareTo(const Date& aDate);
   int distanceTo(const Date& aDate);
   Date();
   Date(const Date& oldDate);
   void date(int day, int month, int year);
private:
   int day;
   int month;
   int year;
};
#endif
```

Date.h

Date.h:
Chứa phần mô tả (khai báo) cho lớp

Tổ chức mã nguồn: Minh hoạ

```
#include "Date.h"

int Date::getDay(void){
    return day;
}
void Date::setDay(int newDay){
    day = newDay;
}
int Date::getMonth(void){
    return month;
}
void Date::setMonth(int newMonth){
    month = newMonth;
}
int Date::getYear(void){
    return year;
}
void Date::setYear(int newYear){
    year = newYear;
}
```

Date.cpp - **PART I**

Date.cpp:
Chứa phần định nghĩa (definition) cho lớp

Dùng toán tử **::** khi hàm định nghĩa bên ngoài phần mô tả của lớp.

Các hàm có tên đầy đủ là:

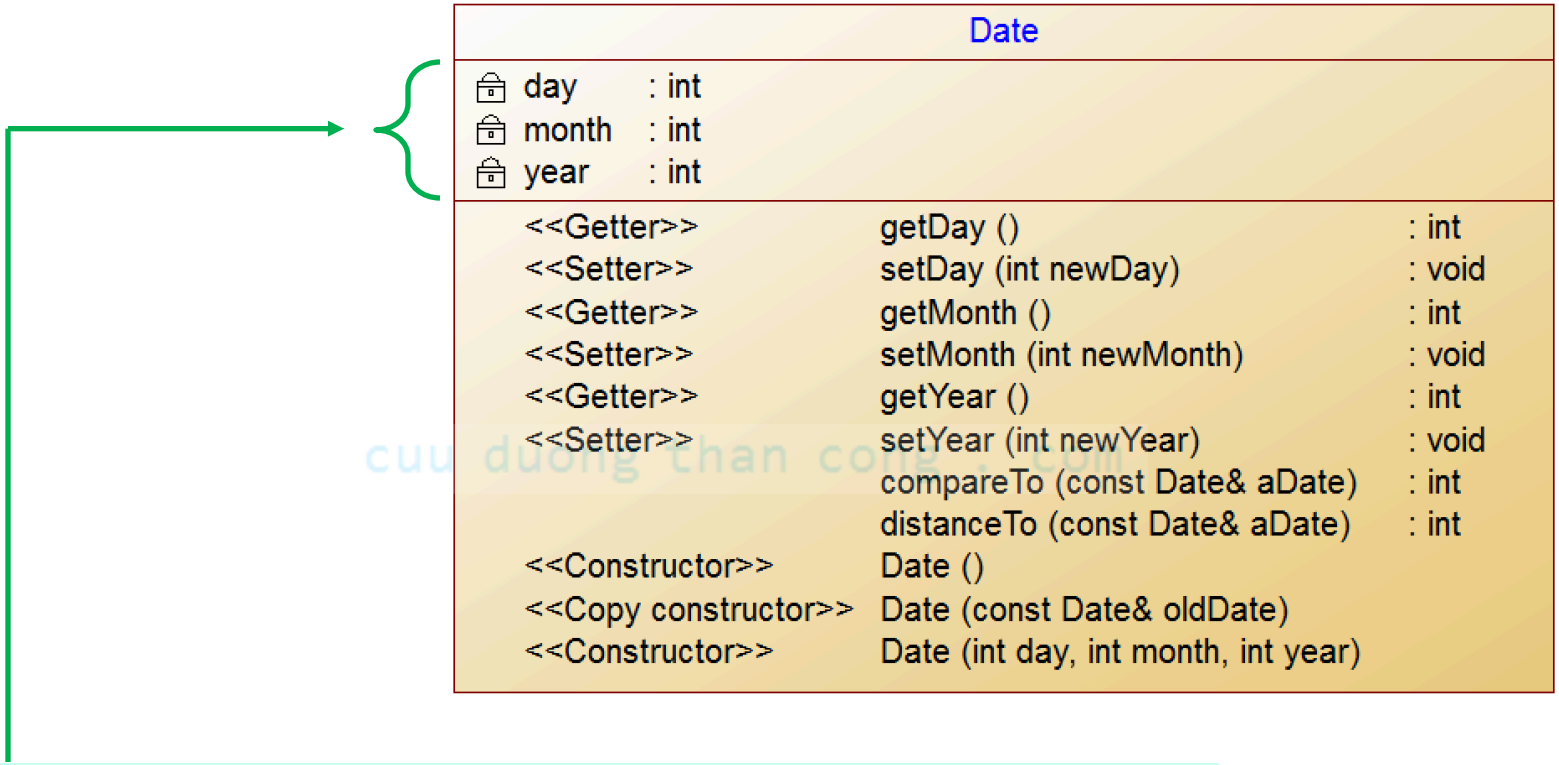
<Tên lớp>::<tên hàm>
Như ví dụ:
Date::setDay

Tổ chức mã nguồn: Minh hoạ

```
int Date::compareTo(const Date& aDate){
    // TODO : implement
}
int Date::distanceTo(const Date& aDate){
    // TODO : implement
}
Date::Date(){
}
Date::Date(const Date& oldDate){
    day = oldDate.day;
    month = oldDate.month;
    year = oldDate.year;
}
Date::Date(int day, int month, int year){
    // TODO : implement
}
```

Date.cpp - **PART II**

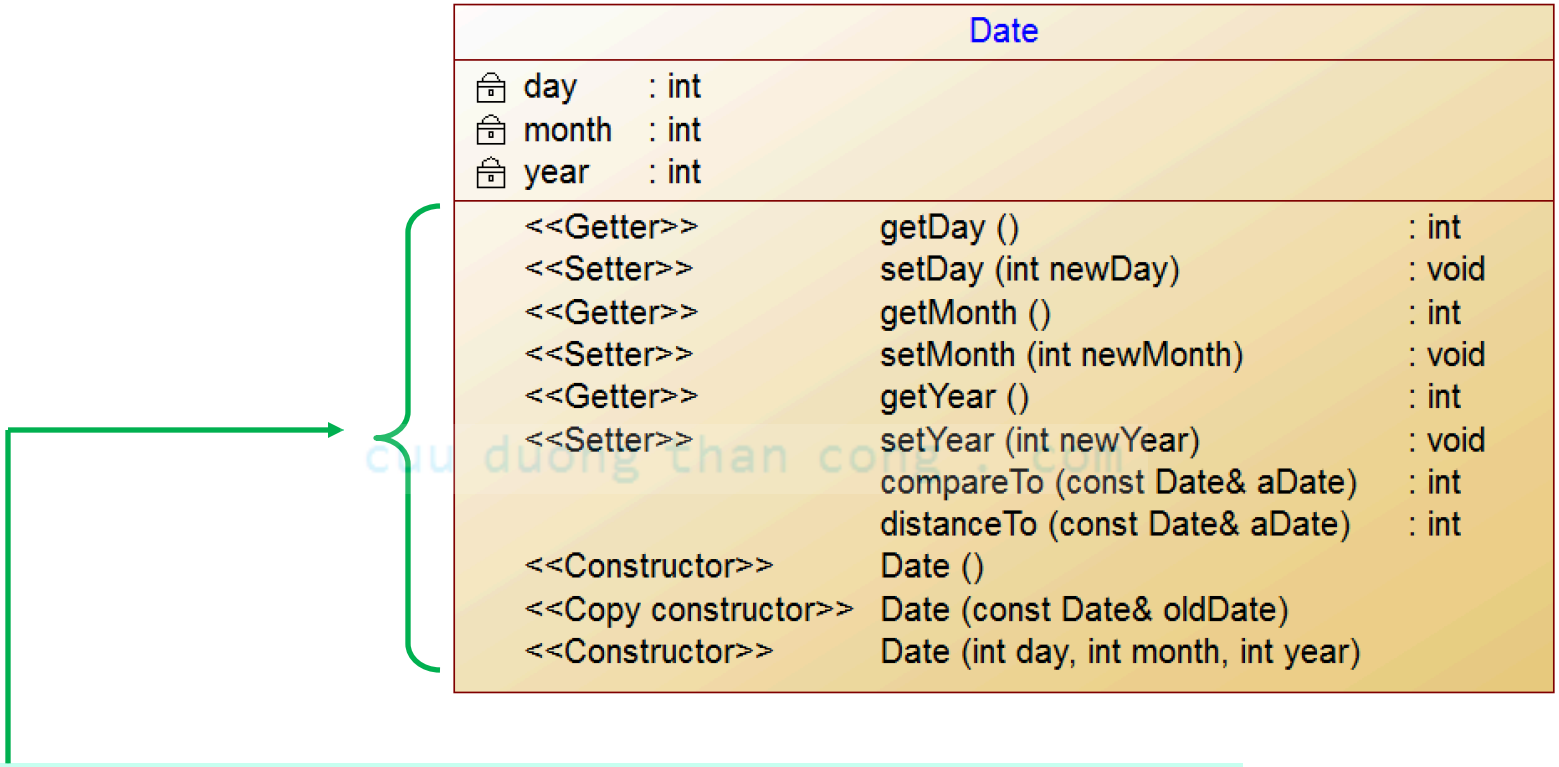
Biểu diễn lớp bằng sơ đồ



Các thuộc tính, mỗi thuộc tính kiểu kèm theo, bên phải

- Ký hiệu ổ khoá (+): tính private
- Ký hiệu chìa khoá (-): tính protected
- Để trống (#): tính public

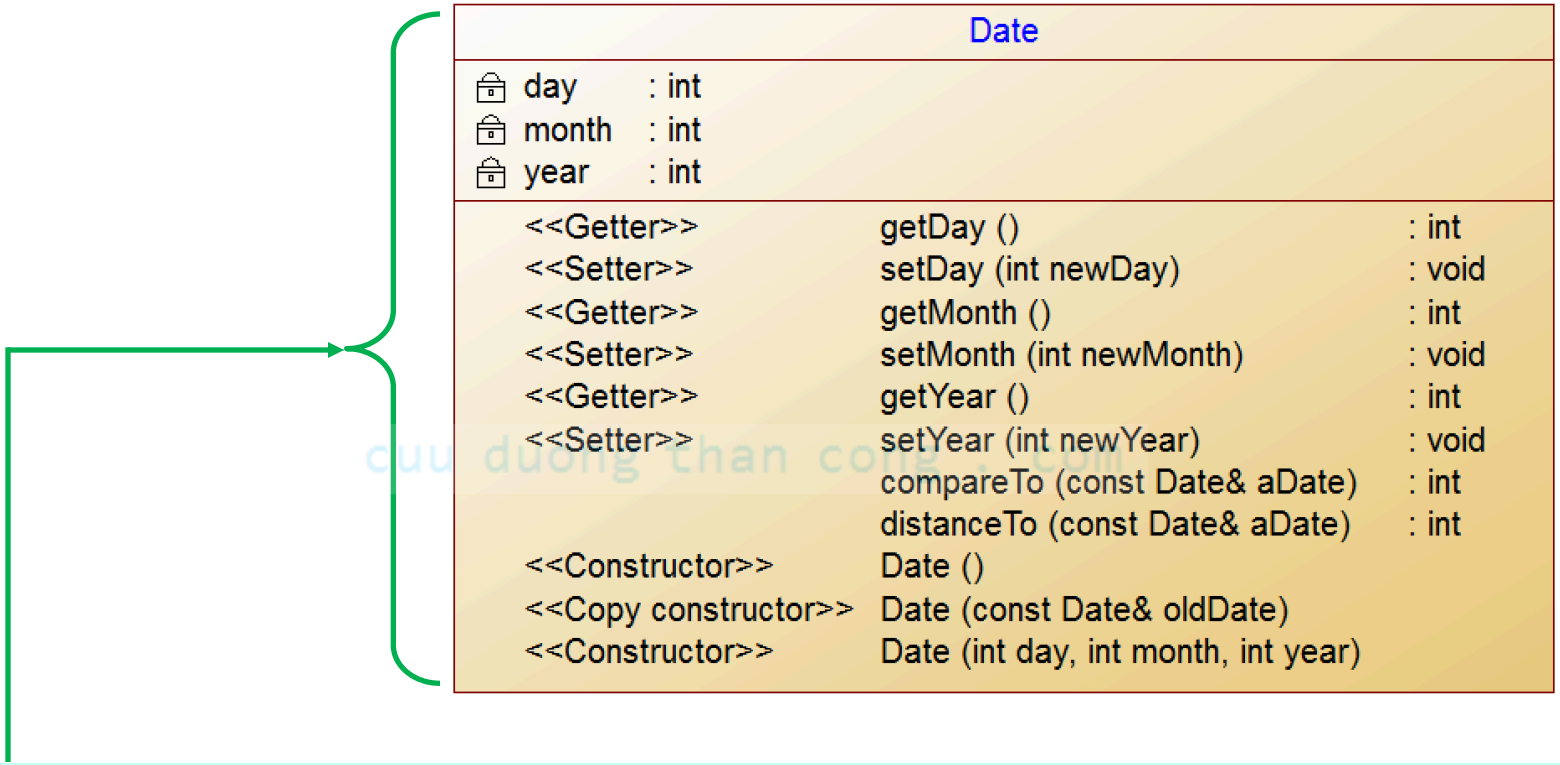
Biểu diễn lớp bằng sơ đồ



Các hàm thành viên, mỗi hàm có 3 cột

- Cột trái: thông tin bổ sung, ví dụ: setter/getter, constructor
- Cột tên hàm và danh sách thông số
- Cột kiểu trả về

Biểu diễn lớp bằng sơ đồ



Thông tin từ hình ảnh là tương đương với thông tin của tập tin header.

Từ hình ảnh có thể sinh ra hai tập tin header và source tương ứng, chỉ có một số hàm là có thể sinh ra được phần thân, như: setter/getter và constructors

Tổng kết

- Hướng đối tượng là một phương pháp để phát triển ứng dụng có nhiều ưu điểm:
  - Dễ cho thiết kế, viết mã, bảo trì và nâng cấp
  - Dễ cho quản lý dự án khi quy mô của nó lớn
- Nhược điểm:
  - Khó hiểu lúc bắt đầu
- Điểm quan trọng trong phần vừa qua:
  - (1) Quan điểm về đối tượng:
    - Đối tượng có tính chủ động.
    - Nó chứa dữ liệu riêng
    - Nó có thể đón nhận thông điệp (gọi hàm) từ các đối tượng khác.
  - (2) Thời gian sống của đối tượng:

Tổng kết

- Điểm quan trọng trong phần vừa qua:
  - (2) Thời gian sống của đối tượng:
    - Từ thời điểm nó sinh ra:
      - Hàm khởi tạo luôn luôn được gọi tại thời điểm này
    - Đến thời điểm nó bị huỷ (ra khỏi tầm vực hoặc bị delete)
      - Hàm huỷ luôn luôn được gọi tại thời điểm này.
  - (3) Trong thời gian nó sống (lưu trong bộ nhớ)
    - Các đối tượng khác có thể gửi thông điệp (gọi hàm) đến nó.
      - Nó thực thi hàm được gọi, có thể tạo ra các đối tượng khác và nhờ các đối tượng khác thực hiện một phần việc.
    - Thiết kế tốt: ít khi cho các đối tượng khác trực tiếp truy xuất dữ liệu của đối tượng đang giữ mà không thông qua các phương thức.

Tổng kết

- Điểm quan trọng trong phần vừa qua:
  - (6) Phương pháp lập trình hướng đối tượng có tính trừu tượng hoá cao (abstraction).
    - Hãy dùng tính khả kiến để che đi các phương thức hỗ trợ nội bộ và che đi dữ liệu nội bộ trong lớp.
    - Hãy chỉ chia sẽ ra bên ngoài những phương thức chọn lọc thực sự phù hợp với lớp đang được thiết kế.
      - → Người bắt đầu học về lập trình hướng đối tượng gặp khó khăn trong nguyên tắc thiết này → luyện tập dần dần.
  - (7) Biểu diễn các lớp bằng sơ đồ giúp việc nắm bắt toàn bộ dự án dễ dàng hơn.
    - Người bắt đầu có thể chưa quen.

Tổng kết

- Điểm quan trọng trong phần vừa qua:
  - (4) Quan niệm về lớp vs đối tượng
    - Lớp (class) như cái khuôn
    - Còn đối tượng như những cái bánh tạo với khuôn đó.
      - Dĩ nhiên, khuôn tròn, bánh không thể méo!
  - (5)Tổ chức mã nguồn tốt, sẽ
    - Tách riêng phần khai báo (hay mô tả) lớp vào một tập tin (*.h)
    - Tách riêng phần định nghĩa các phương thức vào tập tin hiện thực (*.cpp)
    - Module *.cpp (file cpp) phụ thuộc vào module *.h của chính lớp đó → dùng #include.