

Lời nói đầu

Tài liệu này được viết với mục tiêu:

- Dùng như một giáo trình môn học Lập trình Hướng đối tượng với ngôn ngữ C++. Tại các trường chuyên Công nghệ Thông tin, môn học này được học sau môn Lập trình với ngôn ngữ C, nên giả định rằng bạn đọc đã biết sử dụng ngôn ngữ C.
- Nội dung của tài liệu tập trung vào Lập trình Hướng đối tượng, không phải trình bày ngôn ngữ C++. Vì vậy, tài liệu bỏ qua một số vấn đề của C++: template, container, thư viện STL.
- Thể hiện bằng ngôn ngữ ANSI C++ chuẩn, dù có ghi chú về C++11 nhưng chưa dùng đến chuẩn C++11. Các ví dụ và đáp án bài tập được viết với phong cách lập trình đặc thù của C++ chuẩn, dễ tiếp cận với bạn đọc đã nắm vững ngôn ngữ C.
- Dùng như một bộ bài tập môn học Lập trình Hướng đối tượng. Các bài tập được sắp xếp hợp lý, trực quan, đa dạng, số lượng đủ nhiều để bạn đọc rèn luyện kỹ năng (65 lớp). Các bài tập đều có đáp án cẩn thận, chi tiết.

Tôi xin tri ân đến các bà đã nuôi dạy tôi, các thầy cô đã tận tâm chỉ dạy tôi. Chỉ có cống hiến hết mình cho tri thức tôi mới thấy mình đền đáp được công ơn đó.

Tôi đặc biệt gửi lời cảm ơn chân thành đến anh Huỳnh Văn Đức, anh Lê Gia Minh; tôi đã được làm việc chung và học tập các anh rất nhiều khi các anh giảng dạy môn Lập trình Hướng đối tượng tại Đại Học Kỹ thuật Công nghệ Thành phố Hồ Chí Minh.

Tôi xin cảm ơn gia đình đã hy sinh rất nhiều để tôi có được khoảng thời gian cần thiết thực hiện được tài liệu này.

Mặc dù đã dành rất nhiều thời gian và công sức cho tài liệu này, phải hiệu chỉnh chi tiết và nhiều lần, nhưng tài liệu không thể nào tránh được những sai sót và hạn chế. Tôi thật sự mong nhận được các ý kiến góp ý từ bạn đọc để tài liệu có thể hoàn thiện hơn.

Các bạn đồng nghiệp nếu có sử dụng giáo trình này, xin gửi cho tôi ý kiến đóng góp phản hồi, giúp giáo trình được hoàn thiện thêm, phục vụ cho công tác giảng dạy chung.

Phiên bản

Cập nhật ngày: 20/10/2016

Thông tin liên lạc

Mọi ý kiến và câu hỏi có liên quan xin vui lòng gửi về:

Dương Thiên Tứ

91/29 Trần Tấn, P. Tân Sơn Nhì, Q. Tân Phú, Thành phố Hồ Chí Minh

Facebook: <https://www.facebook.com/tu.duongthien>

E-mail: thientu2000@yahoo.com

Trung tâm: CODESCHOOL – <http://www.codeschool.vn>

Fanpage: <https://www.facebook.com/codeschool.vn>

Khai báo và định nghĩa lớp

Class – Declaration & Definition

I. Khái niệm

1. Lập trình hướng đối tượng

Lập trình hướng thủ tục (POP – Procedure-Oriented Programming) đặc trưng bởi cách tiếp cận:

- Thiết kế từ trên xuống (top-down design): phân rã vấn đề thành các thủ tục nhỏ, tập trung vào chức năng của chương trình.
- Dữ liệu + thuật toán \Rightarrow chương trình: tổ chức thực hiện các thủ tục theo một lưu đồ nào đó để giải quyết vấn đề.

Tuy nhiên, khi các chương trình trở nên lớn và phức tạp hơn, lập trình hướng thủ tục có các điểm yếu:

- Các hàm có thể truy xuất không giới hạn đến dữ liệu toàn cục (global), vì vậy khó kiến trúc và thay đổi chương trình.
- Sự tách biệt giữa dữ liệu và các hàm gây khó khăn khi mô phỏng thế giới thật, nơi các đối tượng có thuộc tính và hành vi liên quan với nhau.

Vì vậy xuất hiện một cách tiếp cận lập trình mới được gọi là lập trình hướng đối tượng (OOP – Object-Oriented Programming). OOP phân rã vấn đề cần giải quyết thành các lớp/đối tượng, xây dựng thuộc tính (dữ liệu) và hành vi (phương thức) gắn liền với các đối tượng này. Chương trình cho các đối tượng tương tác với nhau theo một kịch bản nào đó để giải quyết vấn đề.

OOP có ưu điểm:

- Thừa kế những tính năng tốt nhất của lập trình hướng thủ tục và thêm vào một số khái niệm mới.
- Cung cấp một cách suy nghĩ, tổ chức, phát triển chương trình mới dựa trên các đối tượng, gắn gũi với thế giới thật.
- Khả năng đóng gói giúp che giấu thông tin làm hệ thống an toàn và tin cậy hơn.
- Khả năng thừa kế cho phép tái sử dụng dễ dàng, hệ thống có tính mở cao.

OOP kết hợp ba kỹ thuật chính, thường gọi là tam giác P.I.E:

- Encapsulation (đóng gói): dữ liệu và phương thức được liên kết trong một đơn vị gọi là lớp (class). Không thể truy cập dữ liệu từ bên ngoài mà phải thông qua các phương thức được đóng gói trong lớp đó.
- Inheritance (thừa kế): dữ liệu và phương thức của một lớp có thể được thừa kế để tạo lớp mới, hình thành một cây phân cấp các lớp. Điều này cung cấp khả năng tái sử dụng, bổ sung và hiệu chỉnh một lớp mà không cần sửa đổi nó.
- Polymorphism (đa hình): có thể khái quát hóa các lớp cụ thể có liên quan với nhau thành lớp chung để đáp ứng một thông điệp chung. Tính đa hình là đặc điểm đáp ứng thông điệp chung bằng các hình thức khác nhau tùy theo lớp cụ thể được gọi.

2. Đối tượng (object)

Bước đầu tiên hướng tới việc giải quyết một vấn đề là phân tích. Trong OOP, phân tích bao gồm xác định và mô tả các đối tượng và xác định mối quan hệ giữa chúng. Mô tả đối tượng là nhằm rút ra các đặc điểm chung để trừu tượng hóa chúng thành lớp.

Một *đối tượng* (object) thể hiện một thực thể (vật lý hay khái niệm) trong thế giới thực. Một đối tượng có 3 khía cạnh: *định danh* (identity), *trạng thái* (state), *hành vi* (behavior).

- *Identity*: định danh thể hiện sự *tồn tại duy nhất* của đối tượng. Đối tượng có một địa chỉ duy nhất được cấp phát trong bộ nhớ liên kết với nó, có một tên duy nhất được khai báo bởi người lập trình hoặc hệ thống.

- *State*: trạng thái của đối tượng, bao gồm một tập các *thuộc tính* (attribute) của đối tượng và trị của chúng tại một thời điểm. Các thuộc tính là tên của dữ liệu dùng mô tả trạng thái của đối tượng.

Trị của các thuộc tính, tức trạng thái của đối tượng, có thể thay đổi trong quá trình thực thi chương trình.

- *Behavior*: hành vi của một đối tượng, chỉ định các *tác vụ* (operation) mà đối tượng có thể thực hiện. Các tác vụ được cài đặt thành các *phương thức* (method) của đối tượng. Có thể dùng các tác vụ này để xem xét hoặc thay đổi trạng thái của đối tượng.

Đóng gói thành lớp Car
kết quả của trừu tượng hóa

Car
- dateWhenBuild: int
- capacity: int
- chassisNumber: string
+ run()
+ brake()
+ turnoff()

instantiation

Thể hiện sportCar thuộc lớp Car dùng trong chương trình

sportCar : Car
- dateWhenBuild = 2005
- capacity = 300
- chassisNumber = "12143"
+ run()
+ brake()
+ turnoff()

abstraction:
trạng thái + hành vi



Một đối tượng Car
trong thế giới thực

3. Lớp (class) và thể hiện (instance) của lớp

Một lớp mô tả một tập hợp các đối tượng có chung kiểu. Có thể hiểu lớp là *kiểu chung* của một nhóm đối tượng, là kết quả của sự trừu tượng hóa nhóm đối tượng đó thành một kiểu chung.

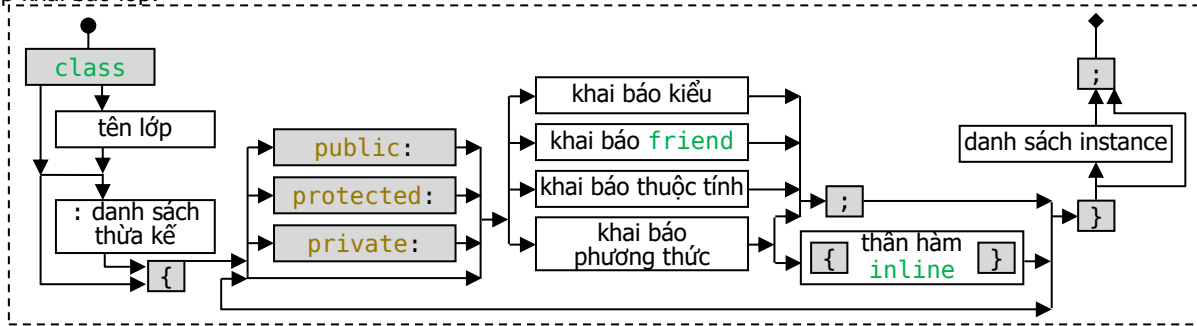
- Thuộc tính được khai báo như dữ liệu thành viên (data member) của lớp.

- Hành vi được khai báo rồi cài đặt như phương thức thành viên (method member) của lớp. Các phương thức cũng giống như hàm (có tên, danh sách đối số, trị trả về, ...), nhưng liên kết với một đối tượng chỉ định, nghĩa là *chỉ gọi thông qua đối tượng*. Theo cách gọi của lập trình hướng thủ tục, ta thường gọi phương thức thành viên là hàm thành viên.

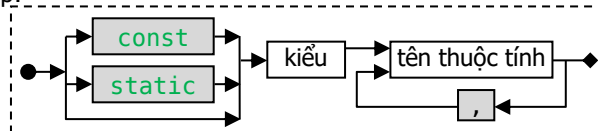
Lớp là khuôn mẫu để sinh ra các *thể hiện* (instance) của lớp. Một lớp định nghĩa các thuộc tính và tác vụ được hỗ trợ bởi các thể hiện thuộc lớp. Như vậy hai thể hiện của cùng một lớp sẽ:

- Có cùng các thuộc tính, nhưng trị của các thuộc tính có thể khác nhau, nghĩa là trạng thái của chúng khác nhau.
- Có cùng các hành vi, nhưng cùng một hành vi sẽ cho kết quả khác nhau do kết quả tùy thuộc vào trạng thái của từng đối tượng.

Cú pháp khai báo lớp:



Cú pháp khai báo thuộc tính của lớp:



Trong định nghĩa của lớp ta phân định *phạm vi truy xuất* các thành viên của lớp bằng các bộ từ truy xuất (access modifier): private, public hoặc protected.

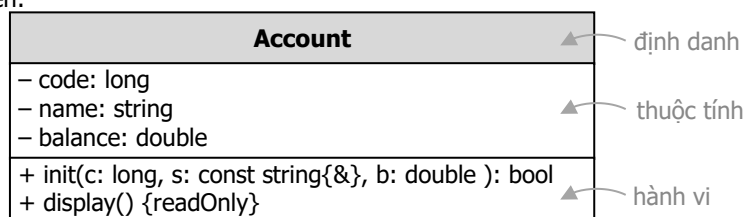
Ví dụ định nghĩa một lớp trong tập tin tiêu đề (header file):

```
// account.h
// khai báo lớp Account
#ifdef _ACCOUNT_           // tránh khai báo (include) nhiều lần
#define _ACCOUNT_
using std::string;

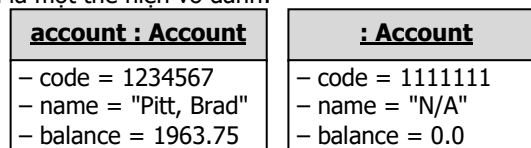
class Account
{
public:                    // Thành viên thuộc về giao diện công khai:
    bool init( long = 111111, const string & = "N/A", double = 0.0 );
    void display() const;
private:                 // Thành viên thuộc giao diện riêng tư, được bảo vệ:
    long code;           // mã tài khoản
    string name;         // tên chủ tài khoản
    double balance;      // tiền trong tài khoản
};
#endif                   // _ACCOUNT_
```

Lớp và các thành viên của nó có thể được mô tả một cách đồ họa bằng cách dùng UML (Unified Modeling Language):

Ký hiệu lớp và các thành viên:



Ký hiệu các thể hiện của lớp, bên phải là một thể hiện vô danh:



Ta trừu tượng hóa nhóm đối tượng chung của thế giới thật thành lớp, rồi tạo các thể hiện của lớp trong chương trình để giải quyết vấn đề. Cần chú ý rằng: ta thường dùng từ "đối tượng" để nói đến các "thể hiện" (instance) này. Một chương trình chạy là một tập các *đối tượng* tương tác với nhau.

4. Trừu tượng hóa dữ liệu (data abstraction) và đóng gói (encapsulation)

Các khái niệm chủ yếu của lập trình hướng đối tượng, đối tượng và lớp, được thiết kế theo các nguyên tắc quan trọng sau:

- *Trừu tượng hóa* (abstraction):

Trong bước phân tích để giải quyết vấn đề bằng OOP, ta gom nhóm, mô tả các đối tượng và phát hiện mối quan hệ tác động giữa chúng. Kết quả việc mô tả đối tượng là trừu tượng hóa từng nhóm đối tượng chung thành lớp.

Một lớp được xem như một kiểu dữ liệu trừu tượng (ADT – abstract data type) do người dùng định nghĩa, một thể hiện của lớp xem như một biến có kiểu dữ liệu là lớp mới tạo. Sự trừu tượng hóa dữ liệu giúp người dùng kiểu dữ liệu trừu tượng mới không

phải quan tâm đến những chi tiết cài đặt bên trong kiểu dữ liệu đó.

- **Đóng gói** (encapsulation) hay **ẩn giấu thông tin** (information hiding):

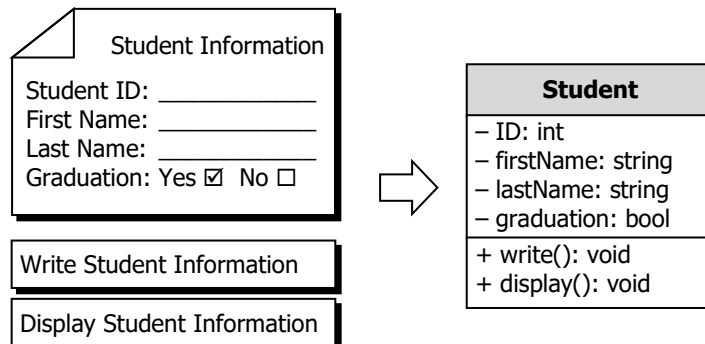
Dữ liệu thành viên của lớp thường được che giấu bên trong phần riêng tư (private) của lớp, không được truy xuất từ bên ngoài. Phương thức thành viên của lớp thường được bộc lộ ra bên ngoài bằng cách khai báo trong phần công khai (public) của lớp. Chú ý là ứng dụng không cần biết cấu trúc dữ liệu bên trong lớp cũng như không cần biết chi tiết cài đặt các phương thức của lớp. Như vậy những thay đổi bên trong lớp có thể thực hiện mà không ảnh hưởng đến ứng dụng.

II. Tạo lớp

1. Encapsulation

Encapsulation (đóng gói) là kỹ thuật gom chung thuộc tính (dữ liệu) và hành vi liên quan (phương thức tác động trên dữ liệu đó) của một nhóm đối tượng vào thành lớp. Để truy xuất thuộc tính và hành vi của lớp phải tạo ra một thể hiện của lớp đó.

Chúng ta trừu tượng hóa đối tượng bằng cách định nghĩa một lớp. Định nghĩa của lớp đã đóng gói thuộc tính và hành vi của các đối tượng thuộc lớp đó.



Định nghĩa lớp Student đóng gói thuộc tính và hành vi của đối tượng thể hiện một sinh viên

Mục đích chủ yếu của đóng gói là ẩn giấu thông tin, bảo vệ dữ liệu tránh khỏi sự truy xuất tự do từ người dùng lớp.

Lớp được phát triển từ structure, tạo một lớp là tạo một kiểu dữ liệu mới. Do đó, lớp có thể *khai báo* và *cài đặt lồng* trong một hàm hoặc cài đặt lồng trong một lớp khác.

2. Phân định phạm vi truy xuất

Khi tổ chức các tập tin cho dự án, ta tách biệt giao diện và cài đặt, gọi là lập trình theo khối (modular programming):

- Khai báo của lớp, gọi là phần giao diện của lớp (class interface) thường đặt trong tập tin .h (header file).

- Định nghĩa của lớp, gọi là phần cài đặt cho lớp (class implementation) thường đặt trong tập tin khác (.cpp).

Khi khai báo lớp, ta cũng phân định phạm vi truy xuất của các thành viên thuộc lớp, bằng cách dùng các *bổ từ truy xuất* (access modifier), còn gọi là tính "thấy được" (visibility) của các thành viên thuộc lớp:

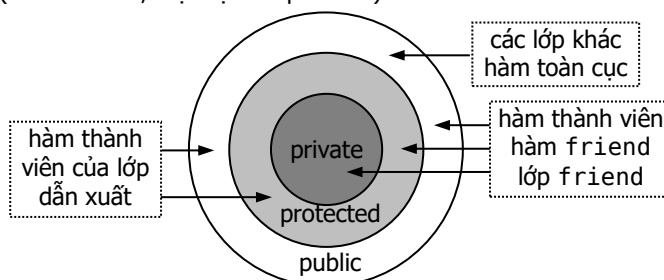
- private: phần "riêng tư" của lớp, thường là thuộc tính của lớp, chỉ có thể được truy xuất bởi các phương thức của lớp. Nếu một thành viên không thuộc phần nào thì mặc định là private (với structure, mặc định là public).

- public: phần "công khai" của lớp, thường là các phương thức công cộng của lớp, có thể truy xuất cả từ bên ngoài lớp. Muốn gọi một hành vi của đối tượng thuộc một lớp, ta truyền thông điệp đến đối tượng, nghĩa là gọi các phương thức được bộc lộ trong phần public.

Phần public tạo thành giao diện công cộng của lớp với bên ngoài, thường gọi là *contract* (giao kết).

- protected: phần "bảo vệ" của lớp, tương tự như phần private, nhưng sẽ có ý nghĩa khi thừa kế, sẽ được thảo luận sau trong phần thừa kế.

Cần có quyền truy xuất khi gọi các phương thức thành viên của một lớp. Quyền truy xuất được mô tả trong bảng bên dưới, một số chi tiết trong bảng sẽ được thảo luận sau.

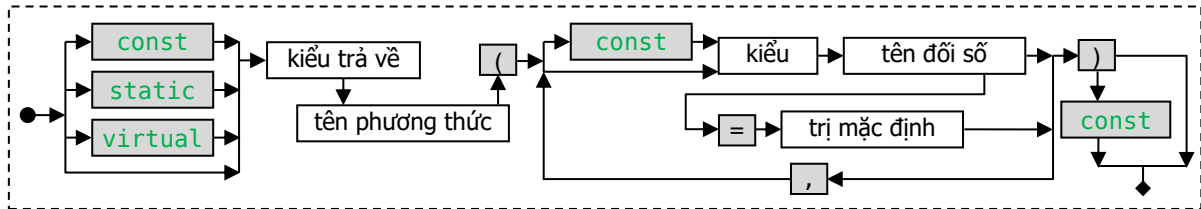


Quyền truy xuất nhìn từ lớp A			Lớp A		Lớp dẫn xuất từ A		Lớp bên ngoài	
			Phương thức hằng	Phương thức thường	Phương thức hằng	Phương thức thường	Phương thức hoặc lớp friend	Phương thức khác
Lớp A	Thuộc tính	public	đọc	đọc/ghi	đọc	đọc/ghi	đọc/ghi	đọc/ghi
		protected	đọc	đọc/ghi	đọc	đọc/ghi	đọc/ghi	
		private	đọc	đọc/ghi			đọc/ghi	
	Phương thức hằng	public	gọi	gọi	gọi	gọi	gọi	gọi
		protected	gọi	gọi	gọi	gọi	gọi	
		private	gọi	gọi			gọi	
	Phương thức thường	public		gọi		gọi	gọi	gọi
		protected		gọi		gọi	gọi	
		private		gọi			gọi	

3. Định nghĩa các phương thức thành viên

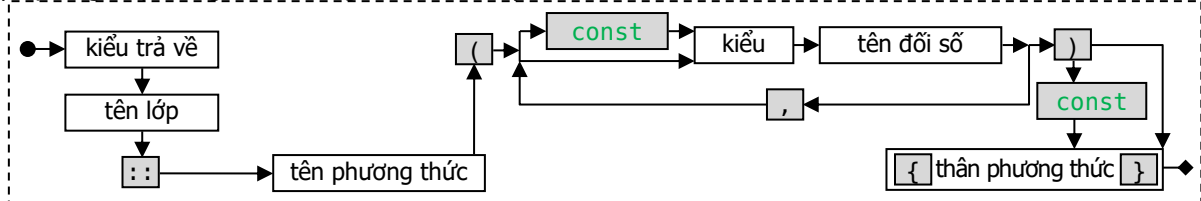
Khi định nghĩa các phương thức của một lớp *bên trong định nghĩa lớp*, ta định nghĩa giống như định nghĩa các hàm toàn cục, không cần tên lớp kèm theo.

Cú pháp khai báo phương thức trong khai báo lớp:



Khi định nghĩa các phương thức của một lớp *bên ngoài định nghĩa lớp*, phải cung cấp *tên lớp ngay trước tên phương thức*, cách biệt nó với tên phương thức bằng *toán tử phân định phạm vi* (scope resolution) "::".

Cú pháp định nghĩa phương thức bên ngoài khai báo lớp:



Bên trong một phương thức thành viên, có thể *truy cập trực tiếp* tất cả các thành viên khác của lớp (dữ liệu và phương thức) bằng cách gọi tên của chúng.

```
// account.cpp
// định nghĩa lớp Account
#include "account.h"           // bao gồm tập tin tiêu đề chứa định nghĩa lớp
#include <iostream>
#include <iomanip>
using namespace std;

// cài đặt phương thức init() khởi tạo dữ liệu thành viên của lớp
// về sau sẽ thay bằng constructor
bool Account::init( long c, const string & s, double b )
{
    if ( s.empty() )
        return false;
    code = c;
    name = s;
    balance = b;
    return true;
}

// cài đặt phương thức display() hiển thị dữ liệu thành viên
void Account::display() const
{
    cout << fixed << setprecision( 2 )
         << "-----\n"
         << "Account number : " << code << '\n'
         << "Account holder : " << name << '\n'
         << "Account balance: " << balance << '\n'
         << "-----\n"
         << endl;
}
```

Định nghĩa một lớp không tự động cấp phát bộ nhớ cho dữ liệu thành viên của lớp. Khi sinh một thể hiện của lớp, mới có cấp phát bộ nhớ thật sự.

4. Dùng các thể hiện của lớp

Một đối tượng liên lạc với đối tượng khác bằng cách *truyền thông điệp* (message passing) để yêu cầu đối tượng khác thực hiện một *hành vi* nào đó. Theo cách nói của lập trình hướng thủ tục, truyền thông điệp đến một đối tượng tương tự một lời gọi hàm thành viên của đối tượng đó.

Từ phương thức của một đối tượng X, ta có thể gửi thông điệp đến một đối tượng Y khác. X và Y có thể là những thể hiện của cùng một lớp hoặc thuộc các lớp khác nhau. X và Y cũng có thể cùng một đối tượng (gọi hàm thành viên của chính mình).

Thông điệp cũng có thể được gửi từ phương thức toàn cục. Ví dụ: từ hàm main() ta gọi phương thức của một đối tượng.

Đối tượng nhận thông điệp sẽ đáp ứng bằng cách triệu gọi phương thức thành viên của nó tương ứng với tác vụ được yêu cầu.

Một thông điệp chỉ định:

- Đối tượng nhận thông điệp.
- Tác vụ yêu cầu đối tượng nhận thực hiện.
- Danh sách đối số của tác vụ nếu có.

Thông điệp truy xuất thành viên của lớp bằng cách dùng toán tử truy xuất thành viên:

- Truy xuất thành viên (dữ liệu hoặc phương thức) thông qua đối tượng được thực hiện bằng toán tử truy xuất thành viên ".".
- Truy xuất thành viên thông qua *con trỏ chỉ đến đối tượng*, được thực hiện bằng toán tử truy xuất thành viên "->".

```
int main()
{
```

```

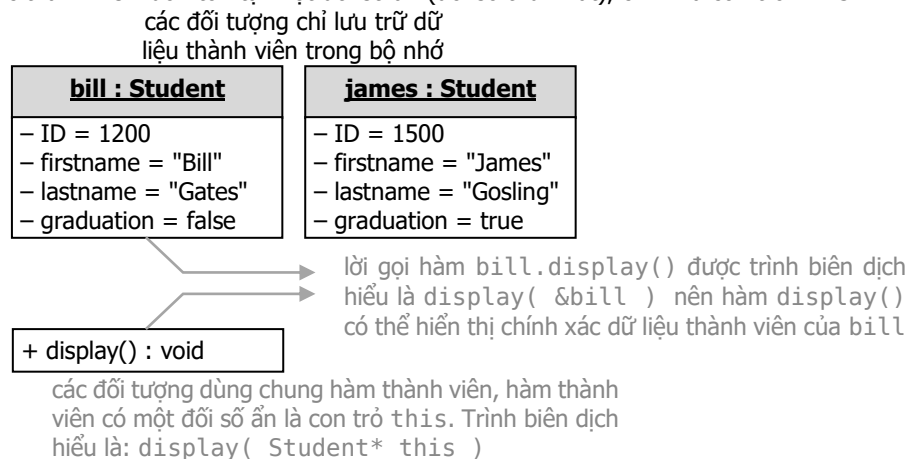
Account account; // trước tiên cần tạo một thể hiện của lớp
Account* p = &account; // cũng có thể dùng con trỏ chỉ đến thể hiện của lớp
// truyền đến đối tượng account thông điệp init() yêu cầu khởi tạo đối tượng
// đối tượng account triệu gọi tác vụ init() để đáp ứng thông điệp.
account.init( 1234567, "Pitt, Brad", 1963.75 );
// truyền đến đối tượng account thông điệp display(),
// đối tượng account triệu gọi tác vụ display() để đáp ứng thông điệp.
account.display // dùng toán tử "."
p->display(); // dùng toán tử "->"
return 0;
}

```

5. Con trỏ this

Bên trong lớp, một phương thức thành viên có thể truy xuất trực tiếp đến thành viên khác của đối tượng hiện hành mà không cần thông qua tên của đối tượng. Phương thức biết được đối tượng hiện hành nào đang làm việc với nó, vì khi phương thức được gọi, một đối số ẩn chứa địa chỉ của đối tượng hiện hành được truyền đến phương thức. Địa chỉ này chứa sẵn trong một con trỏ hằng gọi là con trỏ this.

Như vậy, phương thức thành viên luôn tồn tại một *đối số ẩn* (đối số thứ nhất), chính là con trỏ this.



Con trỏ this là một từ khóa, cũng là một con trỏ hằng chỉ đến đối tượng hiện hành, cho phép tham chiếu đối tượng hiện hành khi cần.

```

#include <iostream>
using namespace std;

class Student
{
public:
    Student( int id, string fname, string lname, bool g )
    : ID( id ), firstName( fname ), lastName( lname ), graduation( g ) { }

    void display() const
    {
        // dùng con trỏ ẩn this để truy xuất đến dữ liệu thành viên. Tuy nhiên không cần thiết.
        cout << "Student: [" << this->ID << "] "
              << this->firstName << " " << this->lastName
              << "\nGraduated: " << this->isGraduation() << endl;
    }
    // cần thiết dùng this để phân biệt dữ liệu thành viên và đối số truyền cùng tên
    void setGraduation( bool graduation ) { this->graduation = graduation; }
private:
    int ID;
    bool graduation;
    string firstName, lastName;
    string isGraduation() const
    {
        return graduation ? "yes" : "no"; // trình biên dịch hiểu là this->graduation
    }
};

```

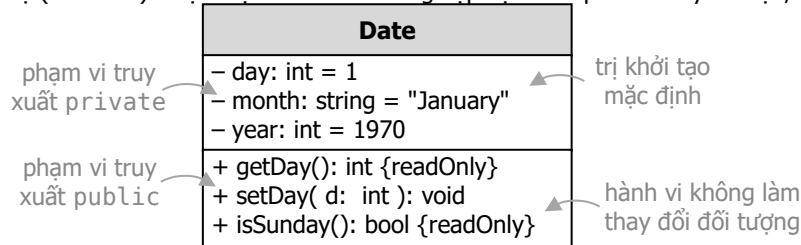
Khi cần truy xuất đối tượng hiện hành, ta dùng *this, nghĩa là áp dụng toán tử dereference với con trỏ this. Điều này thường xảy ra khi phải trả đối tượng hiện hành trở về bằng trị hoặc bằng tham chiếu.

Khi sinh đối tượng con, ta có thể truyền con trỏ this như đối số cho constructor đối tượng đó, để đối tượng con có thể thao tác ngược lại đối tượng sinh ra nó.

III. Các phương thức thành viên

Đối tượng có thể có bốn kiểu hành vi cơ bản: tạo, hủy, truy vấn (queries) và cập nhật (updates). Cần xây dựng các nhóm phương thức cài đặt các hành vi trên:

- constructor: các phương thức tạo, còn gọi là *hàm dựng*, dùng khởi tạo một thể hiện của lớp. Có nhiều constructor để khởi tạo các thể hiện bằng nhiều cách khác nhau.
- destructor: phương thức hủy, còn gọi là *hàm hủy*, dùng giải phóng bộ nhớ cấp phát cho đối tượng khi tiến hành hủy đối tượng.
- các phương thức truy vấn: dùng để truy vấn (xem) dữ liệu của đối tượng. Các phương thức này thường dùng xem trạng thái của đối tượng mà không làm thay đổi đối tượng nên còn gọi là các phương thức non-mutation (hoặc inspectors).
- các phương thức cập nhật: dùng để thay đổi trạng thái (thuộc tính, dữ liệu) của đối tượng, còn gọi là các phương thức mutation.
- các phương thức nghiệp vụ (business): thực hiện các thao tác nghiệp vụ của lớp như xử lý dữ liệu, tính toán, ...



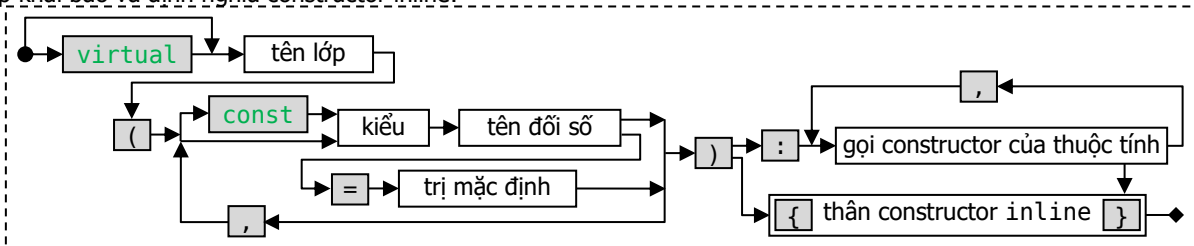
1. Constructor (hàm dựng)

Constructor, thường gọi là ctor, là phương thức đặc biệt dùng khởi tạo thể hiện của lớp. Khi một thể hiện (tức một đối tượng) của lớp được tạo, một constructor nào đó của lớp sẽ được gọi. Điều này giúp tránh quên khởi tạo đối tượng trước khi sử dụng. Constructor cần:

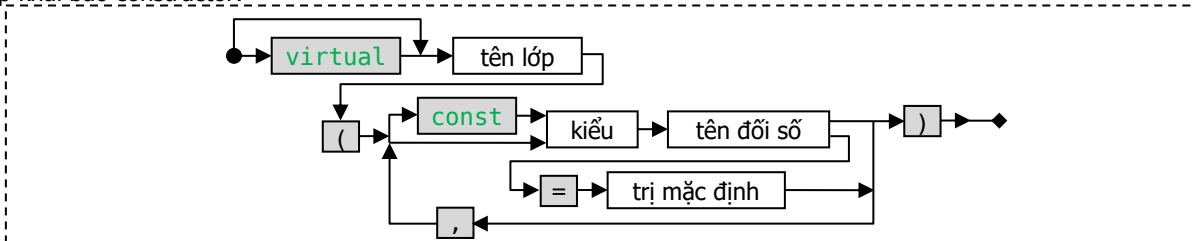
- Có *tên trùng với tên lớp* và *không có trị trả về*.
- Có nhiều constructor với danh sách đối số khác nhau (nạp chồng constructor), cho phép cung cấp nhiều kiểu khởi tạo khác nhau tùy theo danh sách đối số cung cấp khi tạo đối tượng.

Constructor thường có phạm vi truy xuất là public. Tuy nhiên trong một số trường hợp, phạm vi truy xuất của constructor có thể là private hoặc protected. Khi đó có thể dùng phương thức static (gọi là named constructor) để sinh thể hiện.

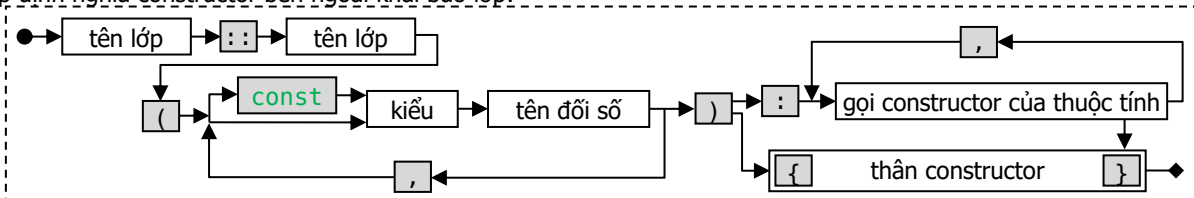
Cú pháp khai báo và định nghĩa constructor inline:



Cú pháp khai báo constructor:



Cú pháp định nghĩa constructor bên ngoài khai báo lớp:



Constructor không có kiểu trả về nên nó không thể trả về mã lỗi. Nếu khởi tạo cho đối tượng thất bại, đối tượng trở thành "zombie", không điều khiển được đối tượng mặc dù chúng vẫn chiếm giữ vùng nhớ trong heap. Giải pháp là ném exception nếu constructor có lỗi, khi đó vùng nhớ liên kết với đối tượng đang khởi tạo sẽ được giải phóng.

Constructor thường được nạp chồng và có đối số mặc định.

a) Nạp chồng hàm (function overloading)

Nạp chồng hàm cũng được xem như một dạng của đa hình, gọi là đa hình hàm (function polymorphism) hoặc đa hình thời gian dịch (compile time polymorphism), đa hình tĩnh (static polymorphism). Các hàm nạp chồng có *cùng một tên hàm* nhưng khác số lượng đối số và kiểu đối số (phần signature của hàm). Các hàm nạp chồng có thể có kiểu trả về khác nhau¹.

Cùng một thông điệp (gọi phương thức cùng tên) nhưng với danh sách đối số khác, ta thấy các phương thức được gọi đáp ứng bằng các hành vi khác nhau. Nói cách khác, các phương thức nạp chồng thể hiện hành vi khác nhau tùy theo loại dữ liệu truyền

¹ Signature của một hàm bao gồm tên hàm, danh sách đối số, các bổ từ (ví dụ const), nhưng KHÔNG bao gồm kiểu trả về của hàm. Hai hàm có cùng tên và danh sách đối số nhưng khác kiểu trả về sẽ sinh lỗi biên dịch.

đến nó.

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

class Account
{
public:
    // nạp chồng constructor cho phép khởi tạo các thể hiện bằng nhiều cách khác nhau
    Account( long, const string &, double );
    Account( const string & );
private:
    long code;
    string name;
    double balance;
};

Account::Account( long c, const string & s, double b )
: code( c ), name( s ), balance( b )
{ }

Account::Account( const string & s )
: code( 111111 ), name( s ), balance( 0.0 )
{ }

int main()
{
    Account giro( 1234567, "Mouse, Mickey", -1200.45 ), save( "Luke, Lucky" );
    Account depot;    // lỗi, do không định nghĩa default constructor
    return 0;
}
```

b) Phương thức có đối số mặc định (default arguments)

Phương thức có đối số mặc định là phương thức có toàn bộ hoặc một số đối số trong danh sách đối số được khởi tạo mặc định. Như vậy, nếu không truyền đối số đến các đối số mặc định cho phương thức đó, phương thức sẽ dùng trị mặc định của đối số. Các đối số mặc định thường được khai báo trong prototype. Khi gọi hàm:

- phải cung cấp các đối số không mặc định
- không cung cấp hoặc cung cấp trị khác cho các đối số mặc định.

```
class Point {
public:
    Point( int = 0, int = 0 );
    void moveTo( int = 0, int = 0 );
private:
    int x, y;
};

Point::Point( int a, int b ) : x( a ), y( b ) { }

void Point::moveTo( int dx, int dy )
{
    x += dx;
    y += dy;
}

int main()
{
    Point p;                // tương đương Point( 0, 0 )
    p.moveTo();              // không di chuyển, tương đương moveTo( 0, 0 )
    p.moveTo( 12 );          // di chuyển song song với trục hoành, tương đương moveTo( 12, 0 )
    p.moveTo( 36, 18 );      // tịnh tiến, đối số nhận được sẽ chồng lên đối số mặc định
    return 0;
}
```

Point
- x: int - y: int
«constructor» + Point(a: int, b: int) «business» + moveTo(dx: int, dy: int)

Để tránh tình trạng không rõ ràng (ambiguous), các đối số mặc định được *khai báo cuối danh sách đối số*. Nói cách khác, khi khai báo trị mặc định cho một đối số, các đối số theo sau phải là đối số mặc định. Khi gọi hàm, nếu đã bỏ qua một đối số mặc định, phải bỏ qua tất cả những đối số sau nó.

```
double capital( double balance, double rate = 3.5, int year = 1 );
double capital( double balance = 0, double rate = 3.5, int year );    // không hợp lệ
```

Phương thức có đối số mặc định dùng trong trường hợp đối số của phương thức thường xuyên được truyền cùng một trị. Thành viên dữ liệu của lớp không được *khởi tạo trực tiếp* (ngoài trừ dữ liệu const static nguyên), giải pháp là dùng constructor với các đối số mặc định. C++11 cho phép khởi tạo trực tiếp thành viên dữ liệu của lớp.

c) Các loại constructor

Do nhu cầu khởi tạo các đối tượng bằng nhiều cách khác nhau, constructor thường được nạp chồng, chúng có các loại sau:

- Constructor mặc định (default constructor): là constructor không có đối số (0-argument constructor). Thường được gọi khi khai báo đối tượng mà không cung cấp đối số nào.

Constructor mặc định được tạo trong các trường hợp sau:

+ Trình biên dịch cung cấp một constructor mặc định (compiler-generated default constructor) nếu ta không định nghĩa một constructor không đối số nào. Tuy nhiên, nếu lớp có các constructor khác, nhưng lại không định nghĩa constructor mặc định, khi ta khai báo đối tượng mà không cung cấp đối số trình biên dịch sẽ báo lỗi không tìm thấy constructor mặc định. Vì vậy, nên tạo một constructor mặc định (rỗng) trước khi viết các constructor khác.

C++11 cung cấp khái niệm *explicitly defaulted constructor* cho việc khai báo một constructor rỗng như trên và khái niệm *explicitly deleted constructor* khi không muốn lớp có bất kỳ constructor nào và cũng không muốn trình biên dịch tạo ra constructor mặc định.

+ Constructor không đối số do ta khai báo và định nghĩa.

+ Constructor với tất cả các đối số đều *khởi tạo với trị mặc định*.

- Constructor sao chép (copy constructor): còn gọi là X-X-ref, là constructor được gọi khi ta tạo đối tượng mới từ bản sao của một đối tượng có sẵn. Nói cách khác, đối số của copy constructor là tham chiếu đến đối tượng của chính lớp đó. Trình biên dịch cũng cung cấp một copy constructor mặc định, sẽ được thảo luận chi tiết sau.

Đối số của copy constructor phải là tham chiếu, không được *truyền đối số bằng trị* đến copy constructor.

```
class Date
{
public:
    // constructor với các đối số đều mặc định, có thể dùng như default constructor
    Date( int d = 1, int m = 1, int y = 1970 )
    : day( d ), month( m ), year( y )
    { }

    // copy constructor
    Date( const Date & o )
    : day( o.day ), month( o.month ), year( o.year )
    { }

private:
    int day, month, year;
};

// hàm toàn cục
Date today( Date d )
{ return d; }

int main()
{
    Date d1( 20, 4, 1976 );
    Date d2( d1 );           // gọi tường minh copy constructor
    Date d3 = d1;           // khởi tạo, không phải phép gán, copy constructor được gọi tại đây
    today( d1 );            // gọi default constructor 2 lần
    return 0;
}
```

↑ danh sách khởi tạo, chỉ dùng với constructor

← copy constructor được gọi tự động khi sao chép tham số hình thức sang tham số thực

← copy constructor được gọi tự động khi sao chép biến cục bộ sang đối tượng tạm

Date
– day: int – month: int – year: int
«constructor» + Date(d: int, m: int, y: int) + Date(o: const Date {&})

Dữ liệu thành viên có thể khởi tạo bằng danh sách khởi tạo (initialization list, constructor initializer hay ctor-initializer) hoặc gán trong thân constructor. Tuy nhiên dữ liệu thành viên hằng và tham chiếu chỉ có thể khởi tạo bằng danh sách khởi tạo.

Dữ liệu thành viên được khởi tạo theo thứ tự khai báo chúng trong lớp, không theo thứ tự trong danh sách khởi tạo.

```
class X
{
public:
    X( int );           // conversion constructor
private:
    int iv;             // biến nguyên
    const int ic;       // hằng nguyên
};

// dữ liệu thành viên hằng, khởi tạo bắt buộc trong danh sách khởi tạo
X::X( int value ) : ic( value )
{
    iv = value;         // gán trong thân ctor hoặc khởi tạo trong danh sách khởi tạo
}
```

C++11 hỗ trợ thêm kiểu `initializer_list<T>` như đối số của constructor.

- Constructor chuyển kiểu (conversion constructor): là constructor chỉ có một đối số, được dùng để hình thành một đối tượng từ một kiểu dữ liệu khác chuyển đến nó. Điều này khác copy constructor, copy constructor hình thành một đối tượng từ một đối tượng cùng kiểu nên không thực hiện việc chuyển kiểu.

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

```

class Dollard
{
public:
    // conversion constructor
    Dollard( double x )      // double ⇒ Dollard
    {
        x *= 100.0;          // làm tròn
        cent = ( long )( x >= 0.0 ? x + 0.5 : x - 0.5 );
    }

    long getWholePart() const { return cent / 100; }
    int getCents() const { return ( int )( cent % 100 ); }
    const Dollard & operator+=( const Dollard & );
    string toString() const; // xuất Dollard như string
private:
    long cent;
};

inline string Dollard::toString() const
{
    stringstream ss;
    long temp = cent;
    if ( temp < 0 ) { ss << '-'; temp = -temp; }
    ss << temp/100 << '.'
        << setfill( '0' ) << setw( 2 ) << temp % 100;
    return ss.str();
}

const Dollard & Dollard::operator+=( const Dollard & right )
{
    cent += right.cent;
    return *this;
}

ostream & operator<<( ostream & os, const Dollard & right )
{
    os << right.toString() << " USD" << endl;
    return os;
}

int main()
{
    Dollard usd( 10.7 );      // conversion ctor: double ⇒ Dollard
    usd = 25.2;               // chuyển kiểu không tường minh: double ⇒ Dollard
    usd += 42.5;              // chuyển kiểu không tường minh: double ⇒ Dollard
    cout << usd;
    usd = Dollard( 99.9 );    // chuyển kiểu tường minh: dùng conversion ctor
    usd = ( Dollard )12.3;    // chuyển kiểu tường minh: ép kiểu theo cách của C
    cout << usd;
    return 0;
}

```

Dollard
- cent: int
+ getWholePart(): long {readOnly}
+ getCents(): int {readOnly}
+ operator+=(const Dollard{&}): const Dollard{&}
+ toString(): string {readOnly}
«friend»
+ operator<<(ostream{&}, const Dollard{&}): ostream{&}

Ngoài conversion constructor, việc chuyển kiểu có thể thực hiện bằng cách nạp chồng toán tử ép kiểu hoặc dùng các toán tử ép kiểu `static_cast`, `dynamic_cast` (sẽ thảo luận sau).

Đôi khi ta không kiểm soát được việc đổi tượng *chuyển kiểu không tường minh* một cách tự động như trong ví dụ trên. Ta có thể ngăn chặn điều này bằng cách dùng từ khóa `explicit` khi khai báo conversion constructor.

```

#include <iostream>
using namespace std;

class Dollard
{
public:
    explicit Dollard( double x )
    {
        x *= 100.0;          // làm tròn
        data = ( long )( x >= 0.0 ? x + 0.5 : x - 0.5 );
    }
    long getData() const { return data; }
private:
    long data;
};

```

```

void deposit( const Dollard & d )
{
    cout << "Deposit: " << d.getData() << endl;
}

int main()
{
    ::deposit( 10.7 ); // lỗi, do chuyển kiểu không tường minh
    ::deposit( Dollard( 12.52 ) ); // phải chuyển kiểu tường minh
    return 0;
}

```

C++11 cung cấp *delegating constructor* (hàm dựng ủy nhiệm), cho phép constructor này có thể gọi constructor khác cùng lớp.

2. Destructor (hàm hủy)

Destructor, thường gọi là dtor, là phương thức đặc biệt dùng để thực hiện việc dọn dẹp cần thiết trước khi một đối tượng bị hủy. Destructor được tự động gọi để giải phóng tài nguyên đối tượng chiếm giữ, không được gọi tường minh destructor.

Một đối tượng bị hủy khi:

- Đối tượng cục bộ (trừ static) ra khỏi tầm vực (scope, khối khai báo đối tượng).
- Đối tượng toàn cục và static, khi chương trình kết thúc.
- Đối tượng được cấp phát động bị xóa bằng cách dùng toán tử delete.

Destructor cần:

- Có tên trùng với tên lớp với dấu "~" đặt trước. Không có trị trả về và không có đối số.
- Chỉ có một destructor, *không có nạp chồng* destructor.

Nếu không cung cấp destructor, trình biên dịch sẽ dùng destructor mặc định.

```

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

class Article
{
public:
    Article( const string & s, double p );
    virtual ~Article();
private:
    long code;
    string name;
    double price;
    static int countObj;
};

// định nghĩa dữ liệu thành viên static
int Article::countObj = 0;

Article::Article( const string & s, double p )
: name( s ), price( p )
{
    code = countObj;
    ++countObj;
    cout << "[" << code << "]" << name << " is created.\n"
        << "This is the #" << countObj << " article!" << endl;
}

// định nghĩa destructor
Article::~Article()
{
    cout << "[" << code << "]" << name << " is destroyed.\n"
        << "There are still " << --countObj << " article!"
        << endl;
}

```

Nếu destructor bị lỗi, không được ném exception; thay vào đó ta có thể ghi nhận vào tập tin log nếu có.

3. Phương thức truy xuất (access function) và phương thức công cụ (utility function)

Dữ liệu thành viên của một lớp thường đặt trong phần private của lớp. Để truy xuất đến dữ liệu thành viên, có thể đặt chúng trong phần public, nhưng cách này vi phạm nguyên tắc đóng gói dữ liệu.

Các phương thức truy xuất (accessor hoặc getter/setter) cho phép dữ liệu thành viên với phạm vi truy xuất private được đọc và thao tác một cách có điều khiển:

- Kiểm tra ngăn chặn truy xuất không hợp lệ ngay từ lúc đầu.
- Ẩn giấu các cài đặt thực của lớp như truy xuất các cấu trúc dữ liệu phức tạp. Điều này cho phép có thể nâng cấp lớp mà không ảnh hưởng đến ứng dụng.

```

#include <iostream>

```

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

class Article
{
public:
    Article( const string & = "", double = 0.0 );
    virtual ~Article();
    // các phương thức truy xuất (accessors)
    // getters
    long getCode() const { return code; }
    const string & getName() const { return name; }
    double getPrice() const { return price; }
    // setters
    void setName( const string & s )
    {
        if ( s.size() < 1 )
            throw string( "Empty string!" ); // tránh tên rỗng
        name = s;
    }
    void setPrice( double p )
    {
        price = p > 0.0 ? p : 0.0; // tránh giá là số âm
    }
private:
    long code;
    string name;
    double price;
    static int countObj; // số đối tượng (static)
    void setCode() { code = countObj; }
};

int Article::countObj = 0; // khởi gán dữ liệu thành viên static

Article::Article( const string & s, double p )
: name( s )
{
    setPrice( p );
    setCode();
    ++countObj;
    cout << "[" << code << "]" << " " << name << " is created.\n"
        << "This is the #" << countObj << " article!" << endl;
}

Article::~~Article()
{
    cout << "[" << code << "]" << " " << name << " is destroyed.\n"
        << "There are still " << --countObj << " article!"
        << endl;
}
```

Article
- code: long - name: string - price: double - countObj: int = 0
«constructor» + Article(const string{&}, double) «destructor» + ~Article() «accessor» - setCode() + setName(s: const string{&}) + setPrice(p: double) + getCode(): long {readOnly} + getName(): const string{&}{readOnly} + getPrice(): double {readOnly}

Một số phương thức nghiệp vụ dùng trong nội bộ lớp, chúng *chỉ được gọi bởi các phương thức thành viên* khác của lớp như là một phương thức hỗ trợ (helper) hay phương thức công cụ (utility). Vì vậy chúng được khai báo trong phần private.

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

class SalesPerson
{
public:
    SalesPerson();
    void getSalesFromUser(); // nhập doanh thu từ bàn phím
    void setSales( int, double ); // đặt doanh thu tháng chỉ định
    void printAnnualSales() const; // in tổng doanh thu
private:
    double totalAnnualSales() const; // phương thức công cụ
    double sales[12]; // doanh thu của 12 tháng
};

SalesPerson::SalesPerson()
{
    for ( int i = 0; i < 12; ++i )
```

SalesPerson
- sales: double[12]
«utility» - totalAnnualSales(): double {readOnly} «business» + getSalesFromUser() + setSales(m: int, a: double) + printAnnualSales() {readOnly}

```

    sales[i] = 0.0;
}

void SalesPerson::getSalesFromUser()
{
    double salesFigure;
    for ( int i = 1; i <= 12; ++i )
    {
        cout << "Enter sales amount for month " << i << ": ";
        cin >> salesFigure;
        setSales( i, salesFigure );
    }
}

void SalesPerson::setSales( int month, double amount )
{
    if ( month < 1 && month > 12 && amount <= 0 )
        throw string( "Invalid month or sales figure" );
    sales[month - 1] = amount;
}

void SalesPerson::printAnnualSales() const
{
    cout << setprecision( 2 ) << fixed << "\nThe total annual sales are: $"
        << totalAnnualSales() << endl;    // gọi phương thức công cụ
}

double SalesPerson::totalAnnualSales() const
{
    double total = 0.0;
    for ( int i = 0; i < 12; ++i ) total += sales[i];
    return total;
}

```

4. Phương thức thành viên inline

Trong một lớp có thể có các phương thức thực hiện các tác vụ đơn giản như đọc ghi dữ liệu thành viên. Việc triệu gọi các phương thức đơn giản này nhiều lần sẽ ảnh hưởng đến thời gian chạy do tốn thời gian gọi hàm (overhead time).

Để cải thiện, chúng ta khai báo chúng như các phương thức inline (phương thức nội tuyến). Khi gặp phương thức inline, trình biên dịch thay *lời gọi đến phương thức* bằng *phần thân của phương thức* (inline code). Như vậy kích thước chương trình sẽ tăng, bù lại thời gian chạy được cải thiện.

Xử lý inline của phương thức đệ quy tùy thuộc vào trình biên dịch, vì vậy nói chung không nên khai báo inline cho phương thức đệ quy. Phương thức inline cũng được viết để thay thế cho các macro theo kiểu C.

Các phương thức inline có thể được định nghĩa tường minh (explicit) hoặc không tường minh (implicit):

- inline tường minh: phương thức được khai báo bên trong lớp và cài đặt bên ngoài lớp. Khi cài đặt, đặt từ khóa inline trước tiêu đề hàm. Cài đặt này phải được thực hiện trong tập tin tiêu đề.
- inline không tường minh: định nghĩa (cài đặt) phương thức *ngay trong lớp*, không cần từ khóa inline.

```

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

class Account
{
public:
    // Constructors: implicit inline
    Account( long c = 1111111L, const string & s = "N/A", double b = 0.0 )
        : code( c ), name( s ), balance( b )
    { }
    // Destructor rỗng: implicit inline
    virtual ~Account(){ }
    void display() const;
private:
    long code;
    string name;
    double balance;
};

// display(): explicit inline
inline void Account::display() const
{
    cout << fixed << setprecision(2)
        << "-----\n"

```

```

<< "Account number : " << code << '\n'
<< "Account holder : " << name << '\n'
<< "Account balance: " << balance << '\n'
<< "-----\n"
<< endl;
}

```

5. Phương thức friend

Đôi khi ta muốn một phương thức toàn cục hay một phương thức thành viên của *một lớp khác* có thể truy xuất được dữ liệu thành viên private của lớp đang xây dựng. Điều này thực hiện được nhờ khai báo friend (bạn), sẽ tạm thời bỏ qua tính đóng gói (encapsulation) của lớp đang xây dựng, cho phép lớp khác truy xuất phần private của lớp đang xây dựng.

Phương thức friend là một phương thức:

- được khai báo friend *trong lớp đang xây dựng*,
- có quyền truy xuất được phần private (dữ liệu và phương thức) giống như các phương thức thành viên khác của lớp,
- nhưng *không phải là phương thức thành viên của lớp đang xây dựng* mà như một hàm thuộc toàn cục hoặc một phương thức thuộc lớp khác.

Một phương thức có thể khai báo friend với nhiều lớp, khi đó nó có quyền truy cập đến phần private của các lớp nó là friend. Tuy nhiên không nên khai báo friend tùy tiện vì nó phá vỡ tính đóng gói khi xây dựng lớp.

Không có ràng buộc về vị trí khai báo phương thức friend trong lớp, nó thuộc phạm vi truy xuất nào cũng được.

a) Phương thức friend là phương thức toàn cục

```

class Node
{
    // phương thức friend khai báo trong lớp nhưng không phải là thành viên lớp
    friend void setPrev( Node*, Node* );
public:
    Node* succ();
    Node* pred();
    Node();
protected:
    void setNext( Node* );
private:
    Node* prev;
    Node* next;
};

// Phương thức thành viên, truy xuất dữ liệu thành viên next
void Node::setNext( Node* p ) { next = p; }

// Phương thức friend được cài đặt như hàm toàn cục, truy xuất được dữ liệu thành viên private của Node
void setPrev( Node* p, Node* n ) { n->prev = p; }

```

Phương thức friend cho phép ta định nghĩa nhiều phương thức nạp chồng toán tử đặc biệt: toán tử << và >>, toán tử hai ngôi có tính giao hoán, ... Vấn đề này sẽ được trình bày trong phần nạp chồng toán tử (operator overloading).

b) Phương thức friend là thành viên lớp khác và lớp friend

Ta có thể chỉ định một phương thức thành viên thuộc lớp khác trở thành friend của lớp đang xây dựng. Nói cách khác, ta cho phép một phương thức thành viên thuộc lớp khác truy xuất vào phần private của lớp đang xây dựng:

```

class C;           // khai báo forward2 do trình biên dịch chưa
class B           // biết đến lớp C nhưng cần C khi cài đặt lớp B
{
public:
    int foo( C & );
};

class C
{
    // chỉ định phương thức foo của lớp B là "bạn" của lớp C
    // cho phép foo của B có thể truy xuất phần private của lớp C
    friend int B::foo( C & );
};

```

Ta cũng có thể chỉ định *một lớp khác* trở thành "bạn" của lớp đang xây dựng. Nghĩa là tất cả các hàm thành viên của lớp khác đó đều là "bạn" của lớp đang xây dựng.

```

class A
{
    // mọi phương thức của lớp B đều có thể truy xuất phần private của lớp A

```

² Nếu hai lớp tham chiếu lẫn nhau, khai báo tạm và đơn giản một lớp trước rồi cung cấp khai báo hoàn chỉnh sau. Một dạng khác là khai báo lớp hỗ trợ ẩn (không cho bên ngoài sử dụng lớp này) trong phần private, gọi là kiểu mờ (opaque type). Sau đó, dùng con trỏ kiểu lớp này, gọi là *pimpl* (pointer to implementation) để thực hiện các tác vụ của lớp đang xây dựng.


```
friend class B;
};
```

6. Đối tượng hằng và phương thức thành viên hằng

Từ khóa `const` được dùng để tạo các đối tượng read-only (chỉ đọc, đối tượng hằng) hoặc các phương thức read-only (phương thức hằng). Trên một đối tượng read-only, chỉ có thể gọi các phương thức read-only.

Một đối tượng hằng phải được khởi gán khi định nghĩa nó và không được thay đổi bởi chương trình về sau.

Một phương thức hằng không được thay đổi dữ liệu thành viên trong thân của nó.

Có thể tạo hai phiên bản của một phương thức: phiên bản read-only sẽ được gọi từ các đối tượng read-only; một phiên bản thường, gọi từ đối tượng không hằng.

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

class Time
{
public:
    Time( int = 0, int = 0, int = 0 );
    // các setter
    void setTime( int, int, int );
    void setHour( int );
    void setMinute( int );
    void setSecond( int );
    // các getter (thường khai báo const)
    int getHour() const { return hour; }
    int getMinute() const { return minute; }
    int getSecond() const { return second; }
    // các phương thức xuất (thường khai báo const)
    void printUniversal() const; // xuất Time dạng HH:MM:SS
    void printStandard(); // xuất Time dạng chuẩn HH:MM:SS AM hoặc PM
private:
    int hour; // 0 - 23
    int minute; // 0 - 59
    int second; // 0 - 59
};

Time::Time( int hour, int minute, int second )
{ setTime( hour, minute, second ); }

void Time::setTime( int hour, int minute, int second )
{
    setHour( hour );
    setMinute( minute );
    setSecond( second );
}

void Time::setHour( int h )
{ hour = ( h >= 0 && h < 24 ) ? h : 0; }

void Time::setMinute( int m )
{ minute = ( m >= 0 && m < 60 ) ? m : 0; }

void Time::setSecond( int s )
{ second = ( s >= 0 && s < 60 ) ? s : 0; }

void Time::printUniversal() const
{
    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
}

void Time::printStandard() // phương thức không hằng
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << ":" << setfill( '0' ) << setw( 2 ) << minute
         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
}

int main()
{
```

Time

```
- hour: int
- minute: int
- second: int

+ setTime( int, int, int )
+ setHour( int )
+ setMinute( int )
+ setSecond( int )
+ getHour(): int {readOnly}
+ getMinute(): int {readOnly}
+ getSecond(): int {readOnly}
+ printUniversal() {readOnly}
+ printStandard() {readOnly}
```

```

Time wakeUp( 6, 45, 0 ); // đối tượng non-constant
const Time noon( 12 ); // đối tượng constant, được khởi gán khi định nghĩa
// OBJECT MEMBER FUNCTION
wakeUp.setHour( 18 ); // non-const non-const
noon.setHour( 12 ); // const non-const báo lỗi
wakeUp.getHour(); // non-const const
noon.getMinute(); // const const
noon.printUniversal(); // const const
noon.printStandard(); // const non-const báo lỗi
return 0;
}

```

Chương trình làm việc trực tiếp với phần cứng thường có dữ liệu thành viên được điều khiển bởi các tiến trình bên ngoài trực tiếp điều khiển chương trình đó. Ví dụ một chương trình có thể chứa biến được cập nhật từ đồng hồ hệ thống.

Từ khóa volatile, ít dùng, dùng tạo các đối tượng có thể thay đổi không chỉ bởi chương trình mà còn bởi chương trình khác hoặc bởi các sự kiện bên ngoài (ví dụ ngắt phần cứng).

```

volatile unsigned long clock_ticks;
volatile Task *curr_task;

```

Ta cũng có thể định nghĩa một phương thức thành viên là volatile, giống như cách định nghĩa phương thức hằng const. Chỉ các phương thức thành viên volatile có thể được gọi trên các đối tượng volatile.

7. Thành viên static

Khi sử dụng trong các phương thức toàn cục thông thường, từ khóa static dùng khai báo các biến tĩnh:

- tồn tại duy nhất trong suốt quá trình chạy chương trình do lưu trong data segment,
- khi khai báo trong hàm foo() chẳng hạn, ta dùng chung biến static này cho tất cả các lần chạy hàm foo().

```

#include <iostream>
void counter()
{
    static int count = 0; // biến static, khởi tạo 0
    std::cout << "Count #" << ++count << std::endl;
}

int main()
{
    for (int i = 0; i < 5; ++i )
        counter(); // in ra trị count tăng dần, thay vì 0
    return 0;
}

```

Đối với lớp, static dùng khai báo thành viên dữ liệu dùng chung cho mọi thể hiện của lớp:

- tồn tại duy nhất trong suốt thời gian lớp tồn tại, có mặt trước mọi thể hiện của lớp.
- dùng chung cho tất cả các thể hiện (instance) của lớp.

Sau khi khai báo trong lớp, phải định nghĩa và khởi gán dữ liệu thành viên static một cách độc lập với mọi thể hiện của lớp. C++11 cho phép khởi gán luôn khi khai báo trong lớp cho thành viên dữ liệu static.

```

#include <iostream>
using namespace std;

class Student
{
public:
    Student( const string & s ) : name( s ) { membership++; }
    virtual ~Student() { membership--; }
    void print() const { cout << name << endl; }
private:
    string name;
    // khai báo số như một dữ liệu thành viên static
    // thường đặt trong phần private để tránh truy cập "riêng"
    static int membership;
};

// định nghĩa và khởi gán dữ liệu thành viên static
int Student::membership = 0;

```

Dữ liệu thành viên static dùng chung cho các thể hiện của lớp, các phương thức thành viên khác có thể truy cập dữ liệu này. Tuy nhiên, nếu dữ liệu thành viên static có phạm vi truy xuất private, bên ngoài lớp muốn truy cập đến nó phải thông qua phương thức thành viên của lớp.

Do dữ liệu thành viên static độc lập với mọi đối tượng, phương thức truy xuất đến dữ liệu thành viên static cũng phải độc lập với mọi đối tượng. Phương thức thành viên static được dùng với mục đích này.

Dùng từ khóa static trước tiêu đề hàm để khai báo phương thức thành viên static. Khi định nghĩa phương thức không cần từ khóa static. Một phương thức thành viên static có thể được gọi thông qua bất kỳ đối tượng nào của lớp hoặc *thông qua tên lớp*, dùng toán tử phân định phạm vi "::".

```

#include <iostream>
using namespace std;

class Person
{
public:
    Person( const string &, const string & );
    virtual ~Person();
    const string & getFirstName() const
    { return firstName; }
    const string & getLastName() const
    { return lastName; }
    // phương thức thành viên static
    static int getCount(); // phương thức static truy xuất dữ liệu static (không const)
private:
    string firstName;
    string lastName;
    // dữ liệu thành viên static lưu số thể hiện của lớp
    static int count;
};

// định nghĩa và khởi tạo dữ liệu thành viên static, tầm vực tập tin
// không nên đặt trong tập tin header vì có thể gây nên lặp lại định nghĩa
int Person::count = 0;

// định nghĩa phương thức thành viên static, không cần có từ khóa static
int Person::getCount()
{ return count; }

Person::Person( const string & first, const string & last )
{
    firstName = first;
    lastName = last;
    count++; // thay đổi biến static, tăng số đối tượng sinh ra
}

Person::~~Person()
{ count--; } // thay đổi biến static, giảm số đối tượng sau khi hủy

int main()
{
    // gọi phương thức static thông qua tên lớp, dù chưa khai báo đối tượng nào
    cout << "Before instantiate: " << Person::getCount() << " person(s)" << endl;

    Person *p1 = new Person( "Gates", "Bill" );
    Person *p2 = new Person( "Gosling", "James" );
    // gọi phương thức static thông qua đối tượng như phương thức thành viên thông thường
    cout << "After instantiate : " << p1->getCount() << " person(s)" << endl;

    delete p1; p1 = NULL;
    delete p2; p2 = NULL;
    // không còn đối tượng, gọi phương thức static lần nữa thông qua tên lớp
    cout << "After destroy : " << Person::getCount() << " person(s)" << endl;
    return 0;
}

```

Person
- firstName: string - lastName: string - count: int = 0
+ getFirstName(): const string{&} {readOnly} + getLastName(): const string{&} {readOnly} + getCount(): int

Một phương thức thành viên static không dùng con trỏ this, vì phương thức thành viên đó không thuộc một đối tượng cụ thể nào của lớp. Vì vậy mọi tham chiếu đến con trỏ this trong phương thức thành viên static, ví dụ truy xuất một dữ liệu thành viên không phải static, sẽ gây lỗi biên dịch. Cũng với lý do như vậy, phương thức thành viên static không được khai báo hằng (const).

Nạp chồng toán tử

Operator Overloading

I. Khái niệm

Một lớp mới là một kiểu dữ liệu trừu tượng mới (ADT – Abstract Data Type). Để có thể dùng đối tượng của ADT mới với các toán tử trong biểu thức một cách bình thường như các kiểu dữ liệu có sẵn (built-in data type) khác, ta cần nạp chồng toán tử. Như vậy toán tử được nạp chồng sẽ "thao tác" được thêm trên một kiểu dữ liệu mới.

Nói cách khác, ADT mới được liên kết với một tập các toán tử để có thể thao tác một cách tự nhiên như các kiểu dữ liệu có sẵn.

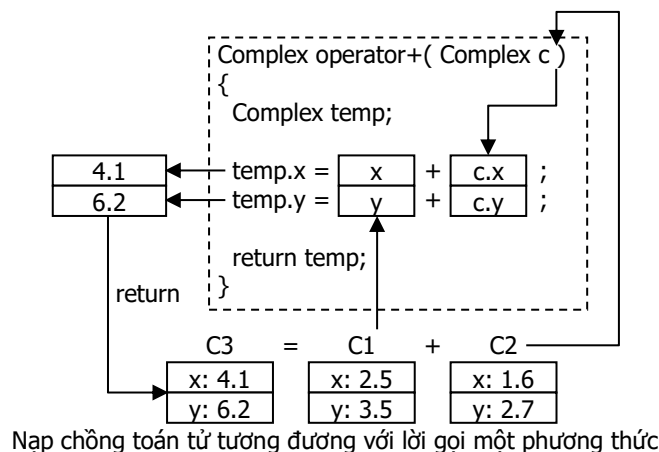
```
Complex X( 1, 2 ), Y( 3, 4 );
// để cộng X và Y rồi hiển thị, thay vì gọi các phương thức một cách phức tạp và không quen thuộc như sau
X.add( Y ).display();
// sau khi nạp chồng toán tử cộng + và toán tử chèn << trong lớp Complex ta có thể
// cộng và xuất hai đối tượng của lớp Complex cũng đơn giản như cộng hai số double
cout << X + Y;
double M = 3.2, N = 5.7;
cout << M + N;
```

Khi nạp chồng toán tử, cần chú ý:

- Nạp chồng toán tử để mang lại thuận tiện và an toàn cho người sử dụng lớp, không phải là yêu cầu bắt buộc khi phát triển lớp.
- *Tôn trọng ý nghĩa của toán tử gốc.* Các toán tử gán =, lấy địa chỉ &, toán tử dấu phẩy, có tác vụ được định nghĩa trước với từng kiểu dữ liệu, tác vụ các toán tử này thường thay đổi tùy theo định nghĩa nạp chồng của chúng ta. Tác vụ của toán tử có thể thay đổi nhưng không nên sai lệch nhiều so với tác vụ ban đầu của toán tử, ví dụ cộng hai đối tượng lớp MyString (mô tả chuỗi) có nghĩa là nối chuỗi (không phải cộng trị).
- *Không thể định nghĩa lại toán tử của các kiểu cơ bản.* Ví dụ, không hợp lệ nếu muốn nạp chồng lại toán tử / để cung cấp thêm khả năng kiểm tra phép chia cho 0 của số nguyên và số thực.
- *Không thể thay đổi thứ tự ưu tiên và thứ tự thực hiện của toán tử gốc.*
 - + Thứ tự ưu tiên (precedence) cho biết thứ tự lựa chọn thực hiện các toán tử khác nhau trong một biểu thức dùng nhiều toán tử. Ví dụ toán tử * được thực hiện trước toán tử +.
 - + Thứ tự thực hiện (order), còn gọi là thứ tự liên kết (associativity), cho biết toán tử nào sẽ thực hiện trước nếu có nhiều toán tử cùng cấp ưu tiên trong một biểu thức, thường là từ trái sang phải hay từ phải sang trái.
- *Không thể thay đổi số lượng toán hạng (arity)* cần cho một toán tử.
- *Không được tạo toán tử mới*, chỉ được nạp chồng các toán tử đã có sẵn.
- Hàm nạp chồng toán tử không thể khai báo đối số với trị mặc định.

Cơ sở để có thể nạp chồng toán tử: C++ thực hiện một toán tử tương đương với một lời gọi phương thức.

```
Complex X( 1, 2 ), Y( 3, 4 );
X + Y; // tương đương với gọi phương thức X.operator+( Y )
```



Một số toán tử không nạp chồng được:

Toán tử	Tên
.	Toán tử truy xuất thành viên
.*	Dereference con trỏ chỉ đến thành viên thông qua đối tượng
::	Toán tử phân định phạm vi (scope resolution)
?:	Toán tử điều kiện (3 ngôi - ternary)
sizeof	Toán tử lấy kích thước của toán hạng
typeid	Toán tử lấy kiểu
static_cast	Các toán tử ép kiểu
dynamic_cast	
const_cast	
reinterpret_cast	

Chỉ có hai toán tử: = và & được trình biên dịch cung cấp mặc định cho lớp đang xây dựng. Ngoài hai toán tử nói trên, nếu muốn sử dụng một toán tử khác cho các đối tượng của lớp, cần phải nạp chồng toán tử đó. Bảng sau trình bày các toán tử có thể nạp chồng:

Toán tử	Tên	Số toán hạng	Cách nạp chồng
,	Dấu phẩy (toán tử thứ tự)	một toán hạng	<i>không nên nạp chồng</i>
!	NOT logic	một toán hạng	<code>bool operator!() const;</code> (<i>nên thay bằng toán tử ép kiểu <code>bool</code> hay <code>void*</code></i>)
!=	So sánh khác	hai toán hạng	<code>friend bool operator!=(const T&, const T&);</code>
%	Lấy phần dư	hai toán hạng	<code>friend const T operator%(const T&, const T&);</code>
%=	Lấy phần dư / Gán	hai toán hạng	<code>T& operator%=(const T&);</code>
&	Bitwise AND (xử lý bit)	hai toán hạng	<i>không nên nạp chồng</i>
&	Lấy địa chỉ	một toán hạng	<i>không nên nạp chồng</i>
&&	AND logic	hai toán hạng	<i>không nên nạp chồng</i>
&=	Bitwise AND / Gán	hai toán hạng	<code>T& operator&=(const T&);</code>
()	Gọi hàm	–	<i>như hàm thành viên, cho phép đổi số mặc định³</i>
*	Nhân	hai toán hạng	<code>friend const T operator*(const T&, const T&);</code>
*	Nội dung nơi con trỏ chỉ đến (dereference, indirect)	một toán hạng	<code>E& operator*() const;</code>
=	Nhân / Gán	hai toán hạng	<code>T& operator=(const T&);</code>
+	Cộng	hai toán hạng	<code>friend const T operator+(const T&, const T&);</code>
+	Cộng một ngôi	một toán hạng	<code>const T operator+() const;</code>
++	Tăng 1	một toán hạng	<code>T& operator++();</code> và <code>T& operator++(int);</code>
+=	Cộng / Gán	hai toán hạng	<code>T& operator+=(const T&);</code>
–	Trừ	hai toán hạng	<code>friend const T operator-(const T&, const T&);</code>
–	Trừ một ngôi (đổi dấu)	một toán hạng	<code>const T operator-() const;</code>
—	Giảm 1	một toán hạng	<code>T& operator--();</code> và <code>T& operator--(int);</code>
-=	Trừ / gán	hai toán hạng	<code>T& operator-=(const T&);</code>
->	Chọn thành viên	hai toán hạng	<code>E* operator->() const;</code>
->*	Dereference con trỏ chỉ đến hàm thành viên	một toán hạng	<i>không nên nạp chồng</i>
/	Chia	hai toán hạng	<code>friend const T operator/(const T&, const T&);</code>
/=	Chia / Gán	hai toán hạng	<code>T& operator/=(const T&);</code>
<	Nhỏ hơn	hai toán hạng	<code>friend bool operator<(const T&, const T&);</code>
<<	Dịch trái	hai toán hạng	<code>friend const T operator<<(const T&, const T&);</code>
<<	Toán tử chèn	hai toán hạng	<code>friend ostream& operator<<(ostream&, const T&);</code>
<<=	Dịch trái / Gán	hai toán hạng	<code>T& operator<<=(const T&);</code>
<=	Nhỏ hơn hoặc bằng	hai toán hạng	<code>friend bool operator<=(const T&, const T&);</code>
=	Gán	hai toán hạng	<code>T& operator=(const T&);</code>
==	So sánh bằng	hai toán hạng	<code>friend bool operator==(const T&, const T&);</code>
>	Lớn hơn	hai toán hạng	<code>friend bool operator>(const T&, const T&);</code>
>=	Lớn hơn hoặc bằng	hai toán hạng	<code>friend bool operator>=(const T&, const T&);</code>
>>	Dịch phải	hai toán hạng	<code>friend const T operator>>(const T&, const T&);</code>
>>	Toán tử trích	hai toán hạng	<code>friend istream& operator>>(istream&, T&);</code>
>>=	Dịch phải / Gán	hai toán hạng	<code>T& operator>>=(const T&);</code>
[]	Chỉ số mảng	–	<code>E& operator[](int);</code> hoặc <code>const E& operator[](int) const;</code>
^	Bitwise XOR	hai toán hạng	<code>friend const T operator^(const T&, const T&);</code>
^=	Bitwise XOR / Gán	hai toán hạng	<code>T& operator^=(const T&);</code>
	Bitwise OR	hai toán hạng	<code>friend const T operator (const T&, const T&);</code>
=	Bitwise OR / Gán	hai toán hạng	<code>T& operator =(const T&);</code>
	OR logic	hai toán hạng	<i>không nên nạp chồng</i>
~	Bù 1 (toán tử đảo bit)	một toán hạng	<code>const T operator~() const;</code>
delete	Hủy cấp phát (delete[])	–	<code>void operator delete(void* ptr) noexcept;</code>
new	Cấp phát (new[])	–	<code>void* operator new(size_t size);</code>

Chú ý có hai cách dùng toán tử tăng 1 và giảm 1: prefix và postfix. Ta cũng nên nạp chồng đồng thời cả hai toán tử của các cặp toán tử có ý nghĩa ngược nhau, ví dụ cặp toán tử so sánh == và !=.

II. Cú pháp

Định nghĩa một toán tử nạp chồng là định nghĩa một phương thức (operator function), tên của phương thức là `operator@`, với từ khóa `operator`, ký hiệu `@` theo sau thể hiện toán tử cần nạp chồng. Số đối số trong danh sách đối số của toán tử nạp chồng tùy theo:

- Số toán hạng tham gia khi thực hiện toán tử.

- Cách định nghĩa một toán tử nạp chồng. Có hai cách:

+ Định nghĩa toán tử nạp chồng như một hàm *không phải thành viên* (hàm toàn cục – global function), có số đối số bằng với

³ Không dùng đối số mặc định với các phương thức nạp chồng toán tử, ngoại trừ phương thức nạp chồng toán tử gọi hàm.

số toán hạng. Nạp chồng các toán tử như một hàm không phải thành viên cho phép cài đặt các biểu thức có tính giao hoán, sẽ thảo luận sau trong phần phương thức friend.

+ Định nghĩa toán tử nạp chồng như một *hàm thành viên*, có *số đối số ít hơn 1 đối số* so với số toán hạng cần cho toán tử, vì đối tượng gọi hàm là đối số ẩn, chính là con trỏ *this*.

```
// dùng toán tử nạp chồng hai ngôi như hàm thành viên
if ( Box1 > Box2 ) // ...
toán hạng bên trái là đối tượng gọi phương thức, cũng là đối số ẩn *this
toán hạng bên phải là đối số của phương thức nạp chồng

bool Box::operator>( const Box & aBox ) const
{
    return ( this->Volume() > aBox.Volume() );
}
```

Nếu muốn không cho phép sử dụng một toán tử nào đó với lớp đang xây dựng, ta khai báo toán tử đó trong phần private và không cài đặt chúng.

1. Toán tử một ngôi (unary)

Toán tử một ngôi chỉ thao tác trên một toán hạng. Hàm nạp chồng toán tử nên là hàm thành viên để bảo đảm tính đóng gói. Toán tử một ngôi bao gồm:

a) Toán tử một ngôi tiền tố đơn giản

Toán tử một ngôi tiền tố đơn giản là các toán tử *!, +, -* (đổi dấu). Chúng thường được nạp chồng như một hàm thành viên không đổi số. Ngoài ra, chúng đều *không thay đổi toán hạng* theo sau mà chỉ *trả về một đối tượng mới vô danh* (nameless temporary object).

```
// nạp chồng toán tử - một ngôi (đổi dấu) như hàm thành viên (không có đối số)
Complex Complex::operator-() const
{
    return Complex( -re, -im );
}
// gọi toán tử nạp chồng trên
Complex Z( 2, 5 );
-Z;
```

b) Toán tử tăng 1 (increment) và giảm 1 (decrement) – prefix và postfix

Nạp chồng các toán tử ++ và -- là một trường hợp đặc biệt, cần phải phân biệt sự khác nhau giữa hai trường hợp prefix và postfix: prefix trả về *tham chiếu* đến nó sau khi đã tăng, postfix trả về *đối tượng* chứa trị cũ của nó.

```
// prefix                                // postfix
a = 5;                                  reg = a;
x = ++a;                                a++;
                                         x = reg;
// x = 6  a = 6                          // x = 5  a = 6
```

Vì toán tử ++ có thứ tự ưu tiên cao hơn toán tử =, nên trong trường hợp postfix, một biến thanh ghi được sử dụng để lưu trữ a trước khi thực hiện ++, trị lưu này sẽ được gán cho x sau. Điều này sẽ được mô phỏng khi nạp chồng các toán tử trên.

```
// prefix increment operator
const Complex & Complex::operator++()
{
    ++re; ++im;
    return *this;
}
// postfix increment operator, đối số kiểu int chỉ là đối số "giả" vô danh để phân biệt với prefix
Complex Complex::operator++( int )
{
    Complex temp = *this;           // biến "dummy"
    re++; im++;
    return temp;
}
// cách cài đặt khác, phải định nghĩa trước copy constructor và toán tử prefix increment
Complex Complex::operator++( int )
{
    Complex temp( *this );           // biến "dummy"
    ++( *this );
    return temp;
}
// gọi toán tử nạp chồng trên
Complex Z( 2, 5 );
++Z;                                // gọi operator++() – pre-increment
Z++;                                // gọi operator++( int ) – post-increment
```


c) Toán tử ép kiểu (type casting)

Nạp chồng toán tử ép kiểu như ngôn ngữ C cho phép chuyển một kiểu bất kỳ thành một kiểu khác: kiểu cơ bản, đối tượng lớp, con trỏ, tham chiếu, nhưng không phải là một mảng hoặc tên hàm.

Phương thức nạp chồng toán tử ép kiểu *không chỉ định kiểu trả về*, do tên của phương thức tự chỉ ra kiểu trả về. Tên phương thức nạp chồng do đó có thể chứa nhiều từ khóa như unsigned short hoặc const float*.

Phương thức nạp chồng toán tử ép kiểu phải là một hàm thành viên không phải static và không có đối số.

Thường nạp chồng toán tử ép kiểu để chuyển một đối tượng UDT (User-define Data Type) thành một kiểu cơ bản, đơn giản. Khi cần chuyển một kiểu cơ bản thành kiểu UDT, thiết kế conversion constructor tương ứng thuộc lớp UDT đó.

Tránh cung cấp cùng lúc cả hai phương thức: conversion constructor B::B(A) và nạp chồng toán tử ép kiểu A::operator B() khi muốn chuyển đổi kiểu A thành kiểu B, khiến trình biên dịch không xác định được phương thức cần phải gọi.

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

class Time
{
public:
    Time( int m = 0, int s = 0, int h = 0 )
        : hths( h * 3600 + m * 60 + s )
    { }
    operator int() const           // ép kiểu Time thành int cơ bản
    { return hths; }               // nghĩa là trả về số int chứa trong Time
private:
    int hths;                     // thời gian với đơn vị giây
};

int main()
{
    Time t;
    t = 100;                     // tương tự dùng conversion constructor: t = Time( 100 )
    cout << ( int )t << endl;    // ép kiểu int tưởng mình, dùng nạp chồng toán tử ép kiểu
    cout << t;                   // ép kiểu int không tưởng mình, tương tự trên
    return 0;
}
```

Khi chuyển đổi một kiểu UDT thành kiểu con trỏ hay tham chiếu, cần đề phòng việc thay đổi dữ liệu của lớp thông qua con trỏ hoặc tham chiếu bằng từ khóa const.

```
operator const Type * () const;
operator const Type & () const;
```

Không có toán tử ép kiểu trả về kiểu bool, thay vào đó ta nạp chồng toán tử ép kiểu trả về kiểu void*, trình biên dịch sẽ chuyển kiểu không tưởng mình con trỏ trả về thành kiểu bool.

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

class Number {
public:
    Number( int n ) : num( n ) { }
    operator const void *() const // hoặc operator void const *() const
    { return isPrime() ? this : NULL; } // kiểm tra số nguyên tố
private:
    bool isPrime() const
    {
        if ( num < 2 ) return false;
        for ( long i = 2; i <= sqrt( num ); ++i )
            if ( num % i == 0 ) return false;
        return true;
    }
    long num;
};

int main()
{
    Number n( 523582853 );
    cout << boolalpha << ( bool )n << endl;
    return 0;
}
```

C++ cung cấp toán tử static_cast<> thay cho ép kiểu như ngôn ngữ C. Vì vậy, thường tránh nạp chồng toán tử ép kiểu vì toán tử này có thể làm xuất hiện nhiều trường hợp chuyển kiểu không tưởng mình khó kiểm soát được.

2. Toán tử hai ngôi (binary)

Phương thức chỉ có một đối số, là toán hạng bên phải, toán hạng bên trái (đối tượng gọi phương thức) chính là đối số ẩn `*this`. Ví dụ sau trình bày cú pháp nạp chồng toán tử hai ngôi như một hàm thành viên:

```
// ví dụ nạp chồng toán tử hai ngôi như hàm thành viên (một đối số)
Complex Complex::operator+( const Complex & right ) const
{
    return Complex( re + right.re, im + right.im );
}

// tùy trường hợp, trả về đối tượng mới hoặc tham chiếu đến toán hạng bên trái
const Complex & Complex::operator+=( const Complex & right )
{
    im += right.im;
    re += right.re;
    return *this;
}

// gọi toán tử nạp chồng này
Complex X( 1, 2 ), Y( 3, 4 ), W( 5, 6 );
X + Y;           // gọi operator+()
W += X;          // gọi operator+=()
```

Chú ý toán tử `operator=` chỉ cho phép nạp chồng như một hàm thành viên và cần kiểm tra trường hợp tự gán.

3. Đối số và trị trả về của phương thức nạp chồng toán tử

Mặc dù có thể truyền các đối số và trả về trị với bất kỳ cách nào ta muốn, nhưng nên bảo đảm các quy tắc sau:

- Với bất kỳ đối số nào của một phương thức, nếu ta chỉ cần đọc đối số mà *không cần thay đổi nó*, vì vậy nên *truyền như một tham chiếu hằng* (const reference). Ví dụ các toán tử số học, so sánh, ... không thay đổi toán hạng của nó, khi đó phương thức nạp chồng các toán tử này giống như một hàm thành viên hằng (const member function).

```
// sinh ra đối tượng là tổng của hai toán hạng
Complex Complex::operator+( const Complex & right ) const
```

- Kiểu của trị trả về được chọn tùy theo ý nghĩa ta mong đợi ở toán tử. Nếu kết quả của toán tử là *sinh ra một đối tượng mới* thì cần *trả về một đối tượng*.

```
// sinh ra đối tượng là tổng của hai toán hạng
Complex Complex::operator+( const Complex & right ) const
```

- Tất cả toán tử gán *làm thay đổi đối tượng bên trái*. Như vậy trị trả về cho tất cả toán tử gán là một *tham chiếu đến đối tượng bên trái*. Từ khóa `const` để ngăn ngừa tác động ngược vào thành phần dữ liệu riêng tư của lớp thông qua phương thức này.

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

Ví dụ: Các toán tử tăng 1 và giảm 1 làm thay đổi đối tượng gọi nên các phương thức nạp chồng tương ứng không là hàm thành viên hằng.

- Kiểu prefix trả về trị của đối tượng sau khi đã thay đổi. Như vậy với prefix ta trả về một *tham chiếu hằng* đến `*this`.
- Kiểu postfix lưu đối tượng, thay đổi đối tượng rồi gửi trả về đối tượng được lưu *như một đối tượng mới*.

```
// prefix increment operator
const Complex & Complex::operator++()
// postfix increment operator
Complex Complex::operator++( int )
```

- Nạp chồng các toán tử so sánh thường trả về kiểu `bool`.

III. Nạp chồng toán tử như hàm friend (friend functions)

1. Toán tử có tính giao hoán

Nạp chồng toán tử bằng hàm thành viên không có tính giao hoán (symmetry), nhất là khi có sự chuyển đổi kiểu không tường minh trong biểu thức. Xem ví dụ:

```
class Vector
{
    double x, y;
public:
    Vector( double = 0, double = 0 );
    Vector operator+( const Vector & right ) const;
    Vector operator*( double k ) const;
    const Vector & operator=( const Vector & right );
};

// cài đặt các phương thức nạp chồng toán tử lớp Vector
// ...
int main()
{
    Vector a, b( 1.2, 3.4 ), c( 7.33, 8.0 );
    double k = 2.1;
    a = b + c; // tương đương a = b.operator+( c )
    a = b + k; // OK, tương đương a = b.operator+( Vector( k ) )
```

```

a = k + b; // Lỗi, k không phải là một đối tượng
b = a * k; // OK, tương đương b = a.operator*( k )
b = k * a; // Lỗi, k không phải là một đối tượng
return 0;
}

```

Vấn đề trên được giải quyết như sau:

- Nạp chồng toán tử bằng một hàm tự do; nghĩa là hàm toàn cục, không phải là hàm thành viên.
- Nếu cần truy xuất thành phần dữ liệu private của lớp là toán hạng của toán tử nạp chồng này, nạp chồng toán tử bằng một hàm friend với lớp. Dùng hàm friend là đi ngược lại tính đóng gói của lớp, vì vậy chỉ sử dụng hàm friend khi cần thiết.

2. Nạp chồng toán tử dùng hàm friend

Các lưu ý khi nạp chồng toán tử như hàm toàn cục hay hàm friend:

- Toán tử nạp chồng là hai ngôi và hai toán hạng có *tính giao hoán*.
- Toán tử nạp chồng được gọi từ một lớp khác.
- Nếu phương thức nạp chồng toán tử không truy xuất phần private của lớp hiện hành, phương thức này được có thể cài đặt như một hàm toàn cục (không cần khai báo trong lớp) thay vì cài đặt như hàm friend.

Hàm friend tuy *khai báo trong lớp* nhưng *không phải là phương thức thành viên* của lớp. Vì vậy hàm friend không dùng con trỏ this. Khi nạp chồng toán tử dùng hàm friend, các toán hạng phải khai báo đầy đủ trong danh sách đối số (ví dụ toán tử *hai* toán hạng thì hàm nạp chồng phải có *hai* đối số...).

Khi khai báo hàm friend cần từ khóa friend; khi cài đặt hàm friend, không cần từ khóa friend.

```

class Vector
{
    friend Vector operator+( const Vector & V1, const Vector & V2 );
    friend Vector operator-( const Vector & V1, const Vector & V2 );
    friend Vector operator*( const Vector & V, double k );
    friend Vector operator*( double k, const Vector & V );
public:
    Vector() { }
    Vector( double a, double b = 0 ){ x = a; y = b; }
    // ...
private:
    double x, y;
};

Vector operator+( const Vector & V1, const Vector & V2 )
{ return Vector( V1.x + V2.x, V1.y + V2.y ); }

Vector operator-( const Vector & V1, const Vector & V2 )
{ return Vector( V1.x - V2.x, V1.y - V2.y ); }

Vector operator*( const Vector & V, double k )
{ return Vector( V.x * k + V.y * k ); }

Vector operator*( double k, const Vector & V )
{ return Vector( V.x * k + V.y * k ); }

int main()
{
    Vector V1( 2.0, 3.0 ), V2( 4.0, 5.0 );
    V1 = V1 + V2;
    V1 = V1 + 3.3;           // giải quyết được vấn đề giao hoán
    V1 = 2.1 + V2;
    return 0;
}

```

không phải là phương thức của lớp nhưng là hàm friend nên vẫn truy xuất được dữ liệu "riêng tư" của lớp một cách trực tiếp

Một số toán tử: gán ('='), gọi hàm ('()'), chỉ số ('[]'), truy xuất thành viên ('->') phải được nạp chồng như hàm thành viên.

3. Nạp chồng toán tử chèn (insertion) và toán tử trích (extraction)

```

cout.operator<<( "An integer: " )

```

cout << "An integer: " << 23 << "and a double: " << 23.45

Toán tử chèn << và toán tử trích >> có chủ thể gọi hàm (toán hạng thứ nhất) *thuộc lớp khác* (ostream và istream) nên khi nạp chồng thường dùng hàm friend. Tuy nhiên, nếu các toán tử này không tham chiếu đến thành viên private của lớp, chúng có thể nạp chồng như một hàm toàn cục, không nhất thiết phải là hàm friend.

Lớp ostream nạp chồng một loạt toán tử chèn, lớp istream cũng nạp chồng một loạt toán tử trích.

Đối tượng thuộc lớp con ostream và istream sẽ *thay đổi* sau khi thực hiện toán tử nên được *truyền như tham chiếu*. Tham chiếu (toán hạng bên trái) trả về bảo đảm việc gọi toán tử liên tiếp nhiều lần.

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

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

// dùng cout << r, tương đương với gọi hàm operator<<( cout, r )
ostream & operator<<( ostream & os, const Vector & r )
{
    return ( os << '(' << r.x << ',' << r.y << ')' );
}

istream & operator>>( istream & is, Vector & r )
{
    return ( is >> r.x >> r.y );
}

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

int main()
{
    Vector a, b;
    // gọi hàm operator>>( operator>>( cin, a ), b );
    cin >> a >> b; // nhập 2 3 4 5
    cout << "a + b = " << a + b << endl; // (6, 8)
    return 0;
}
```

IV. Các toán tử chỉ nạp chồng như hàm thành viên

1. Toán tử gán và toán tử chỉ số (subscript operator)

Toán tử gán mặc định được trình biên dịch cung cấp nếu ta không nạp chồng toán tử gán. Tuy nhiên toán tử gán mặc định chỉ thực hiện việc sao chép bề mặt (shallow copy) sẽ gây lỗi nếu lớp có thành phần dữ liệu động.

Ví dụ dưới minh họa nạp chồng toán tử gán cho lớp có thành phần dữ liệu động, các vấn đề về toán tử kiểu này sẽ được thảo luận trong chương sau.

Toán tử gán được thực hiện từ phải sang trái và trả về một *tham chiếu* để có thể *gán tiếp tục*, ví dụ $a = b = c$. Tham chiếu trả về là tham chiếu đến đối tượng bên trái vừa gán xong.

Phương thức nạp chồng toán tử chỉ số (hai ngôi) trả về một *tham chiếu* đến phần tử nó chỉ đến. Điều này cho phép toán tử chỉ số có thể nằm ở hai bên của phép gán: vừa là L-value⁴, vừa là R-value. Khi nằm bên trái phép gán, toán tử chỉ số cho phép thay đổi thành viên dữ liệu private của đối tượng. Vì vậy, toán tử chỉ số cũng thuộc loại toán tử đột biến.

Ngoài ra cần chú ý các quy tắc: chỉ số trao cho toán tử phải là số nguyên (integer), trị trả về có kiểu của dữ liệu cần lấy.

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

class DoubleVector
{
public:
    DoubleVector( int n = 10 )
    {
```

⁴L-value và R-value được phân biệt tùy theo vị trí của chúng trong phép gán: bên trái (L) hay bên phải (R) phép gán. R-value phải là một "chỗ chứa" được cấp phát trước.

```

    p = new double[size = n ];
    for ( int i = 0; i < size; ++i ) p[i] = i;
};

// dùng delete[] khi constructor có dùng new[]
~DoubleVector() { delete []p; p = NULL; }
double & operator[]( int i );
/* cần thận hơn, có thể định nghĩa tách riêng thành hai toán tử
const double & operator[]( int i ) { return p[i]; }
double operator[]( int i ) const { return p[i]; }
*/
int ubound() { return size; }
protected:
double* p; // thành phần dữ liệu động
size_t size;
private:
DoubleVector( const DoubleVector & v )
{
    p = new double[ size = v.size ];
    for ( int i = 0; i < size; ++i ) p[i] = v.p[i];
}

DoubleVector& operator=( const DoubleVector & v );
friend ostream& operator<<( ostream & os, const DoubleVector & r )
{
    for ( int i = 0; i < r.size; ++i ) os << r.p[i] << ' ';
    return ( os << endl );
}
};

double & DoubleVector::operator[]( int i )
{
    if ( i < 0 || i >= size ) throw out_of_range( "" );
    return p[i]; // trả về tham chiếu đến p[i]
}

DoubleVector & DoubleVector::operator=( const DoubleVector & r )
{
    if ( this != &r ) // chỉ trả về *this nếu tự gán
    {
        if ( r.size != size ) throw string( "Size is not equals!" );
        delete[] p;
        p = new double[r.size];
        for ( int i = 0; i < size; ++i )
            p[i] = r.p[i];
    }
    return *this;
}

int main()
{
    DoubleVector x( 5 ), y( 5 );
    for ( int i = 0; i < x.ubound(); ++i )
        x[i] = i + i/10.0; // gọi x.operator[]( i )
    y = x; // lỗi, gọi phương thức private: y.operator=( x )
    cout << y; // lớp không cho phép gán đối tượng
    return 0;
}

```

C++11 thêm phương thức nạp chồng toán tử gán di chuyển (move assignment operator), đối tượng bên phải là đối tượng tạm và sẽ bị hủy sau khi gán; phân biệt với nạp chồng toán tử gán sao chép (copy assignment operator).

Ngoài ra, giống như với constructor, C++11 cung cấp khái niệm *explicitly defaulted assignment operator* cho việc định nghĩa nạp chồng toán tử gán rỗng và khái niệm *explicitly deleted assignment operator* khi không muốn lớp có nạp chồng toán tử gán.

2. Toán tử gọi hàm (function call operator) và đối tượng hàm (function object)

Toán tử gọi hàm có thể xem là toán tử hai ngôi với đối tượng gọi như toán hạng thứ nhất và danh sách đối số như toán hạng thứ hai.

```

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

class Word

```

```

{
    friend ostream & operator<<( ostream & os, const Word & r )
    {
        return os << r.str << endl;
    }
public:
    Word( const char* );
    // cho phép dùng đối số mặc định với toán tử gọi hàm
    Word operator()( int = 0, int = 0 );
private:
    char* str;
    int len;
};

Word::Word( const char* s ) {
    len = strlen( s );
    str = new char[ len + 1 ];
    strcpy( str, s );
}

// toán tử gọi hàm trả về một chuỗi con, chứa n ký tự, bắt đầu từ vị trí i
Word Word::operator()( int i, int n )
{
    i = ( i < 0 ) ? 0 : ( ( i > len ) ? len : i );
    n = ( n < 0 ) ? 0 : ( ( n > len - i ) ? ( len - i ) : n );
    char* s = new char[ n + 1 ];
    strncpy( s, str + i, n );
    s[n] = '\0';
    return Word( s );
}

int main()
{
    Word w = "Tester String";           // w là một đối tượng hàm
    cout << w( 0, 4 );                  // dùng w để gọi hàm, kết quả: Test
    return 0;
}

```

Thể hiện của *một lớp có định nghĩa nạp chồng toán tử gọi hàm* gọi là một *đối tượng hàm*. Đối tượng hàm được dùng để:

- gọi hàm, giống như một hàm thông thường.
- truyền như đối số đến một hàm khác, rất thường dùng với thư viện <algorithm>.

```

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

class Politician
{
    friend ostream & operator<<( ostream & os, const Politician & );
public:
    Politician( const string & n = "", const string & add = "" )
        : name( n ), office( add ) { }
    void operator()( const Politician & t ) { cout << t; }
private:
    string name;
    string office;
};

ostream & operator<<( ostream & os, const Politician & r )
{
    os << r.name << " [" << r.office << "]" << endl;
    return os;
}

int main()
{
    vector<Politician> vPolitician;
    vPolitician.push_back( Politician( "Barack Obama", "White House" ) );
    vPolitician.push_back( Politician( "Mitt Romney", "Nursing Home" ) );
    // truyền một "đối tượng hàm" như đối số đến phương thức của thư viện <algorithm>
    for_each( vPolitician.begin(), vPolitician.end(), Politician() );
}

```



```

    return 0;
}

```

3. Toán tử truy xuất thành viên (class member access)

C++ cho phép nạp chồng hai toán tử truy xuất thành viên: * và ->; để hỗ trợ các lớp mà đối tượng của chúng được dùng như con trỏ (pointerlike) chỉ đến đối tượng khác. Đối tượng thuộc các lớp này thường dùng làm bộ lặp (iterator) trong các tập hợp (collection).

Ví dụ: tạo một lớp X, thường sẽ là đối tượng chứa trong collection.

```

class X
{
friend ostream & operator<<( ostream & os, X & r )
{
    return os << r.value() << ' ';
}
public:
    X ( int a = 0 ) : n( a ) { }

    X& operator=( int i )
    {
        n = i;
        return *this;
    }

    int & value() { return n; }
private:
    int n;
};

```

Tạo một lớp X_ptr, là con trỏ (cursor, hay iterator – bộ lặp) chỉ đến đối tượng lớp X, chứa trong collection. Để hoàn thành vai trò pointerlike này, cần nạp chồng một số toán tử sau cho lớp X_ptr, phục vụ cho "số học con trỏ":

*, ->: cho truy xuất thành viên

++, +, -: cho số học con trỏ

```

class X_ptr
{
    X* ptr;
public:
    X_ptr( X* p ) : ptr( p ) {}

    X* operator->() { return ptr; }

    X& operator*() { return *ptr; }

    X_ptr operator+( int k )
    { return X_ptr( ptr + k ); }

    int operator-( const X_ptr & r )
    { return ptr - r.ptr; }

    X_ptr& operator++()
    {
        ptr++;
        return *this;
    }

    X_ptr operator++( int )
    {
        X_ptr dump( ptr );
        ptr++;
        return dump;
    }
};

```

Kiểm tra hoạt động của lớp X_ptr, chú ý vai trò con trỏ của nó với lớp chứa X.

```

int main()
{
    int i;
    X a[3] = { X( 1 ), X( 2 ), X( 3 ) };
    X_ptr p = a;
    for ( i = 0; i < 3; ++i )           // phép cộng con trỏ, dùng toán tử truy xuất thành viên
        ( p + i )->value() += i;       // cập nhật trị cho đối tượng được chỉ bởi con trỏ
    for ( i = 0; i < 3; ++i )
        cout << *p++;                 // tăng con trỏ lên 1, rồi dùng toán tử dereference
}

```

```
cout << "\nNumber of element: "
    << p - a << endl;    // phép trừ hai con trỏ, trả về số phần tử giữa hai con trỏ
return 0;
}
```

4. Toán tử new và delete

Nạp chồng toán tử new và delete cho một lớp thường do nhu cầu cần quản lý hiệu quả việc cấp phát và thu hồi bộ nhớ, ví dụ như tạo "pool" quản lý số thực thể của lớp, lưu trữ thực thể (caching) để dùng lại. Đôi khi toán tử new và delete toàn cục cũng được nạp chồng để gỡ rối (debug) khi cấp phát bộ nhớ.

```
void * operator new( size_t );
void * operator new[]( size_t );

void operator delete( void * );
void operator delete[]( void * );
```

Ví dụ:

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

class Point3D {
    friend ostream & operator<<( ostream &, const Point3D & );
public:
    Point3D() { x = y = z = 0; }
    Point3D( int i, int j, int k ) : x( i ), y( j ), z( k ) { }
    void set( int i, int j, int k ) { x = i; y = j; z = k; }

    void *operator new( size_t size );
    void operator delete( void *p );

    void *operator new[]( size_t size );
    void operator delete[]( void *p );
private:
    int x, y, z;
};

ostream & operator<<( ostream & os, const Point3D & r ) {
    os << r.x << ", " << r.y << ", " << r.z << endl;
    return os;
}

void *Point3D::operator new( size_t size )
{
    cout << "Point3D::operator new" << endl;
    void* p = malloc( size );
    if ( !p ) throw bad_alloc();
    return p;
}

void Point3D::operator delete( void *p )
{
    cout << "Point3D::operator delete" << endl;
    free( p );
}

void *Point3D::operator new[]( size_t size )
{
    cout << "Point3D::operator new[]" << endl;
    void* p = malloc(size);
    if ( !p ) throw bad_alloc();
    return p;
}

void Point3D::operator delete[]( void *p )
{
    cout << "Point3D::operator delete[]" << endl;
    free( p );
}

int main()
{
    Point3D *p1, *p2;
```

```
try {
    p1 = new Point3D ( 10, 20, 30 );
} catch ( bad_alloc ba ) {
    cout << "Allocation Point3D object error" << endl;
    return 1;
}
cout << *p1;
delete p1;

try {
    p2 = new Point3D[4];
} catch ( bad_alloc ba ) {
    cout << "Allocation Point3D array error" << endl;
    return 1;
}
p2[1].set( 17, 23, 31 );
p2[3].set( -18, 24, -46 );
cout << "Contents of Point3D array: " << endl;
for( int i = 0; i < 4; ++i )
    cout << "array[" << i << "]: " << p2[i];
delete [] p2;
return 0;
}
```

Con trỏ và tham chiếu

Pointer – Reference

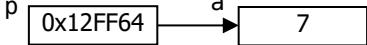
Trong C, thao tác phức tạp và dễ nhầm lẫn nhất là dùng con trỏ. Trong C++, khái niệm tham chiếu và con trỏ⁵ dễ bị hiểu lẫn lộn và dùng sai. Chương này vừa ôn tập vừa giải thích các khái niệm cần nắm vững khi dùng con trỏ và tham chiếu trong C++, đặc biệt khi truyền đối số đến một phương thức hoặc một hàm.

I. Con trỏ (Pointer)

1. Khái niệm

Con trỏ là một biến đặc biệt *lưu địa chỉ* dùng để tham chiếu đến một biến khác hoặc truy xuất một vùng nhớ nào đó. Thao tác tạo một con trỏ chỉ đến một biến được xem như lấy một đối tượng đặc biệt dùng quản lý cho biến đó. Như vậy con trỏ sẽ không có ý nghĩa nếu không *khởi gán* bằng cách chỉ đến một biến khác hoặc một vùng nhớ nào đó.

```
int a = 7, v[20];
int* p, *q;
p = &a;           // p chứa địa chỉ của a, như vậy p chỉ đến a
q = &v[5];        // q chứa địa chỉ của v[5], q quản lý mảng con bắt đầu từ v[5]
q = v + 5;        // tương đương dòng lệnh trên, nhưng về phải dùng địa chỉ
*p = 5;           // a bây giờ là 5, đã bị thay đổi GIÁN TIẾP thông qua con trỏ p
cout << *p + *(q + 2); // tương đương a + v[7]
```



Nhận xét: có thể truy xuất **GIÁN TIẾP** đến biến a thông qua con trỏ p chỉ đến nó (chứa địa chỉ của nó). Muốn thực hiện điều này ta phải dùng hai toán tử:

&: toán tử lấy địa chỉ của biến, đối tượng, ... để "đặt" địa chỉ đó vào con trỏ.

*: (dereference) toán tử lấy nội dung của biến, đối tượng, ... nơi con trỏ chỉ đến.

Dereference (giải quy) một con trỏ là thao tác *lấy nội dung* của đối tượng nơi con trỏ chỉ đến.

```
#include <iostream>

class Fraction
{
public:
    int num, denom;
    void display()
    {
        std::cout << num << '/' << denom << std::endl;
    }
};

int main()
{
    Fraction u = { 2, 3 };           // phân số 2/3
    Fraction* fp = &u;              // dùng con trỏ fp quản lý đối tượng u lớp Fraction
    (*fp).num = (*fp).denom + 1;     // ít dùng cú pháp gọi thành viên này, phân số 4/3
    u.display();                     // gọi phương thức display() TRỰC TIẾP
    fp->display();                     // gọi phương thức display() GIÁN TIẾP
    u.num = u.denom + 1;             // thay đổi u TRỰC TIẾP
    // thay đổi u GIÁN TIẾP thông qua con trỏ fp bằng cách dùng toán tử ->
    fp->num = fp->denom + 2;          // phân số 5/3
    u.display();
    return 0;
}
```

Toán tử dereference * còn gọi là *toán tử gián tiếp* (indirect operator), do ta dùng toán tử này để *truy xuất gián tiếp* đối tượng nơi con trỏ chỉ đến.

2. Số học con trỏ (Pointer Arithmetic) và con trỏ với mảng

Số học con trỏ thể hiện rõ nhất khi dùng con trỏ truy xuất một mảng hay chuỗi.

- Khi dùng con trỏ chỉ đến đầu một mảng, ta có thể *gán chỉ số cho con trỏ* để truy xuất mảng đó thay cho tên mảng.

- Phép cộng: khi tăng con trỏ một đơn vị, con trỏ sẽ dịch chuyển một đoạn bằng *kích thước của đối tượng* mà con trỏ chỉ đến. Như vậy, trong trường hợp dùng mảng, con trỏ sẽ chỉ đến phần tử kế tiếp trong mảng.

- Phép trừ: với hai con trỏ chỉ đến hai phần tử trong mảng, con trỏ chỉ đến phần tử có chỉ số lớn hơn trừ cho con trỏ chỉ đến phần tử có chỉ số nhỏ hơn sẽ cho biết *số phần tử* nằm giữa hai con trỏ.

- So sánh hai con trỏ: có thể được thực hiện với hai con trỏ cùng kiểu.

```
#include <iostream>
using namespace std;
int main()
{
    int a[10], i;
    int *p, *q;
```

⁵ C++ gọi các kiểu của một kiểu có sẵn là kiểu dẫn xuất (derived type): con trỏ, tham chiếu, mảng, hằng, hàm.

```

p = q = a;
// có thể gán chỉ số cho con trỏ p khi p chỉ đến mảng a
for ( i = 0; i < 10; ++i )
    p[i] = i;
// phép cộng con trỏ, xuất các phần tử mảng
for ( i = 0; i < 10; ++i )
    cout << *p++ << ' ';
// phép trừ con trỏ cho biết số phần tử của mảng
cout << "\nHas been " << p - q << " element(s)" << endl;
// so sánh con trỏ, xuất ngược các phần tử mảng
for ( --p; p >= q; --p )
    cout << *p << ' ';
return 0;
}

```

Cần nhắc lại một số đặc điểm của mảng `int a[]` trong C++:

Tên mảng `a` chính là địa chỉ (không phải con trỏ) phần tử đầu tiên của mảng, tương đương `&a[0]`.

`(a + i)` chính là địa chỉ phần tử `a[i]` của mảng, vậy `*(a + i)` tương đương `a[i]`⁶

3. Cấp phát động (Dynamic Storage Allocation)

Toán tử `new` dùng để cấp phát cho đối tượng *một vùng nhớ trên heap* và khởi tạo cho đối tượng đó. Nếu thành công, kết quả nhận được là con trỏ quản lý đối tượng, con trỏ này lại nằm trong stack.

C++ chỉ định các trình biên dịch dùng một phiên bản của `new`, sẽ trả về 0 khi cấp phát gặp lỗi:

```

#include <new> // khai báo đối tượng nothrow, kiểu nothrow_t
double *ptr = new( nothrow ) double[ 50000000 ];

```

Toán tử `delete` dùng để giải phóng vùng nhớ cấp phát cho đối tượng này.

```

class Vector
{
public:
    Vector( double a, double b ) : x( a ), y( b ) { }
private:
    double x, y;
};

void foo( int n )
{
    Vector a( 2.2, 6.7 ); // biến cục bộ a tạo trên stack
    // đối tượng thuộc lớp Vector được cấp phát động và khởi tạo trên heap
    // và được quản lý bởi con trỏ p trên stack
    Vector* p = new Vector( 1.2, 3.4 );
    // cấp phát vùng nhớ trong heap cho con trỏ plong bằng toán tử new, rồi khởi gán trị lưu tại vùng nhớ đó
    long* plong = new long(1234567);
    // ...
    delete plong; // giải phóng vùng nhớ cấp phát cho plong trong heap
    delete p; // giải phóng vùng nhớ cấp phát cho đối tượng do p chỉ đến
} // biến cục bộ p, b và a lần lượt bị hủy tại đây

```

Cũng có thể dùng toán tử `new[]` để cấp phát cho một mảng, nhưng không thể đồng thời khởi tạo cho các phần tử của mảng được. Dùng toán tử `delete[]` để giải phóng vùng nhớ cấp cho một mảng.

```

int size = 100;
Person* p = new Person[size];
// nếu dùng delete p để giải phóng mảng sẽ không biết số đối tượng
// nên không gọi được destructor cho tất cả các đối tượng thuộc mảng
delete[] p;

```

4. Một số con trỏ đặc biệt

a) Con trỏ NULL

Một địa chỉ hợp lệ chứa trong con trỏ phải khác 0. Địa chỉ bằng 0 được dùng để chỉ một lỗi. Với con trỏ, hằng số `NULL` được định nghĩa như 0 trong các tập tin header chuẩn. Một con trỏ chứa trị `NULL` gọi là con trỏ `NULL`, biểu thị rằng con trỏ này không chỉ đến một biến nào cả. Không thể dùng toán tử dereference (`*`) với con trỏ `NULL`.

Sau khi giải phóng vùng nhớ cấp phát do một con trỏ chỉ đến bằng toán tử `delete`, con trỏ trở thành con trỏ lạc (stray pointer). Nên đặt con trỏ lạc thành con trỏ `NULL`, vì dùng toán tử `delete` trên con trỏ lạc sẽ gây lỗi, nhưng dùng con trỏ `NULL` thì an toàn. Con trỏ lạc và con trỏ `NULL` đều phải khởi gán lại trước khi sử dụng tiếp.

b) Con trỏ kiểu `void*`

Một con trỏ kiểu `void*` *không xác lập với một kiểu nào*, như vậy con trỏ `void*` được tham chiếu đến như là một *con trỏ không kiểu* (typeless pointer).

⁶ Phát biểu sau được xem như "thành ngữ của C", C++ thừa kế đặc điểm đó:

`a[i] ≡ *(a + i) ≡ *(i + a) ≡ i[a]`

Không thể thực hiện số học con trỏ trên con trỏ kiểu `void*` vì không xác định được kích thước dữ liệu mà nó chỉ đến (nhưng thực hiện được với con trỏ `void**`). Không thể deference con trỏ kiểu `void*` vì không xác định được kiểu dữ liệu của nó. Chỉ có thể ép kiểu tưởng mình nó thành kiểu con trỏ khác.

Ép kiểu con trỏ có kiểu bất kỳ thành con trỏ có kiểu `void*` rồi truyền nó như đối số đến toán tử `<<` sẽ trả về địa chỉ vùng nhớ chứa trong con trỏ đó dưới dạng hex.

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

int main()
{
    char* p = "OOP with C++";
    void* q;
    // con trỏ void* có thể chỉ đến dữ liệu có kiểu bất kỳ, ý tưởng:
    // lưu trữ một con trỏ kiểu bất kỳ trong con trỏ void*
    q = p;
    // con trỏ void* q quản lý chuỗi "OOP with C++", có thể gán p bằng NULL
    p = NULL;
    // phải ép kiểu con trỏ void* để sử dụng được dữ liệu nơi nó chỉ đến
    p = ( char* )q;
    cout << p << endl;           // in chuỗi "OOP with C++"
    cout << ( void* )p << endl;   // in địa chỉ chứa trong p
    return 0;
}
```

Con trỏ `void*` dùng xây dựng các phương thức, template, nạp chồng toán tử new, ... trong đó cần con trỏ *chỉ đến một kiểu chung* do chưa xác định được kích thước kiểu dữ liệu chỉ đến.

c) Con trỏ hàm (function pointer - functor)

Khác với con trỏ thông thường dùng chỉ đến dữ liệu, con trỏ hàm là một loại con trỏ dùng chỉ đến code. Trong C++, *tên một hàm là con trỏ hằng chỉ đến hàm đó*. Có nhiều cách dùng con trỏ hàm: dùng gọi hàm thay cho tên hàm, dùng như đối số truyền đến hàm khác, lưu vào mảng để truy xuất theo chỉ số một hàm trong tập các hàm giống nhau.

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

int add( int a, int b )    // định nghĩa một số hàm có đặc điểm giống nhau
{ return a + b; }          // nhận hai đối số int và trả về kiểu int

int mul( int a, int b )
{ return a * b; }

// khai báo con trỏ hàm nhận hai đối số int và trả về kiểu int
int ( *pFunc )( int, int );
// khai báo và gán một mảng các con trỏ hàm (các tên hàm)
int ( *apFunc[] )( int, int ) = { add, mul };
// khai báo một hàm nhận đối số là một con trỏ hàm (tên hàm)
int caller( int, int, int ( *p )( int, int ) );

int main()
{
    // gán cho con trỏ hàm một địa chỉ hàm để nó ủy nhiệm đến khi gọi hàm
    pFunc = &add;
    // dereference một con trỏ hàm nghĩa là ủy nhiệm lời gọi đến hàm cần gọi, trong trường hợp này là hàm add()
    cout << ( *pFunc )( 4, 5 ) << endl;

    // nên dùng cú pháp sau, linh động và dễ sử dụng hơn, ủy nhiệm đến hàm mul(),
    pFunc = mul;           // tên một hàm xem như là một con trỏ hàm!
    // dùng con trỏ hàm như một hàm! (ủy nhiệm gọi hàm), tên con trỏ thay cho tên hàm
    cout << pFunc( 4, 5 ) << endl;

    // dùng một con trỏ hàm trong mảng các con trỏ hàm
    cout << apFunc[0]( 4, 5 ) << endl;
    // một phần tử của mảng con trỏ hàm cũng là một con trỏ!
    cout << ( *( apFunc + 1 ) )( 4, 5 ) << endl;

    // chuyển lời gọi hàm mul() đến hàm caller(), truyền con trỏ hàm như đối số
    cout << caller( 4, 5, mul ) << endl;
    return 0;
}

int caller( int a, int b, int ( *p )( int, int ) )
{
```



```
return p( a, b );
}
```

Dùng typedef sẽ làm cho chương trình dễ đọc hơn nhiều, xem lại ví dụ trên:

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

int add( int a, int b ) { return a + b; }
int mul( int a, int b ) { return a * b; }
typedef int ( *VPF )( int, int );
VPF v[] = { add, mul };           // mảng các con trỏ hàm
int caller( int, int, VPF );     // truyền con trỏ hàm như đối số

int main()
{
    VPF pFunc = mul;             // dùng tên hàm như con trỏ hàm
    cout << pFunc( 4, 5 ) << endl; // dùng con trỏ hàm giống như một hàm
    cout << v[0]( 4, 5 ) << endl;  // dùng con trỏ hàm trong mảng các con trỏ hàm
    cout << caller( 4, 5, mul ) << endl; // gọi hàm với đối số là con trỏ hàm
    return 0;
}

int caller( int a, int b, VPF p )
{
    return p( a, b );
}
```

Con trỏ chỉ đến phương thức thành viên (pointer to member function) cũng được sử dụng trong trường hợp lớp có nhiều phương thức thành viên có đặc điểm giống nhau:

- Truy xuất con trỏ chỉ đến phương thức thành viên thông qua con trỏ chỉ đến đối tượng, dùng toán tử ->*
- Truy xuất con trỏ chỉ đến phương thức thành viên thông qua đối tượng, dùng toán tử .*

Khi truy xuất con trỏ chỉ đến dữ liệu thành viên cũng sử dụng hai toán tử trên theo cách tương tự.

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

class Greetings {
public:
    void hello( const string & name ) const
    { cout << "Hello, " << name << endl; }
    void byebye( const string & name ) const
    { cout << "Byebye, " << name << endl; }
};

typedef void ( Greetings::*VPF )( string ) const;

void greet( const Greetings* p, string name, VPF pFunc = &Greetings::hello )
{
    ( p->*pFunc )( name );
}

int main()
{
    Greetings* p = new Greetings();
    greet( p, "Albert Einstein", &Greetings::byebye );
    return 0;
}
```

Con trỏ hàm thường được dùng để thực hiện các hàm callback, cơ chế kết nối động (dynamic binding), các ứng dụng hướng sự kiện, cơ chế ủy nhiệm (delegate), ...

II. Tham chiếu (Reference)

1. Kiểu tham chiếu

```
int a = 7;
// khai báo tham chiếu, ký hiệu &
// không phải là toán tử lấy địa chỉ
int& aa = a;
cout << aa << endl; // đọc GIÁN TIẾP trị của a thông qua "tên khác" aa
aa = 5;             // a bây giờ là 5, bị thay đổi GIÁN TIẾP thông qua aa
```

aa ----> a
 chỉ là khai báo một "tên khác"

aa là một *tên khác* của a, một bí danh (alias) của a (a gọi là *referent* của aa). Ta truy xuất đến aa cũng giống như truy xuất đến a. Nhận xét: có thể truy xuất **GIÁN TIẾP** đến biến a thông qua bí danh của nó là biến tham chiếu aa.

Tham chiếu cũng truy xuất **GIÁN TIẾP** đến một biến như con trỏ nhưng cú pháp ít phức tạp hơn. C++ có khuynh hướng sử

dùng tham chiếu nhiều hơn con trỏ.

Tham chiếu không phải là một biến *riêng biệt*, biến tham chiếu aa thật sự đồng *nhất* với biến a. Áp dụng toán tử & lên tham chiếu giống như áp dụng toán tử & lên biến mà nó tham chiếu đến, kết quả cho cùng một địa chỉ là địa chỉ của biến. Việc định nghĩa một tham chiếu không tốn thêm bộ nhớ như con trỏ.

```
cout << &aa << ' ' << &a; // kết quả cho cùng một địa chỉ, ví dụ 0x25FDA8
```

Như vậy, tham chiếu chính là *một tên khác của một biến đã tồn tại*, nhưng dùng tham chiếu có nghĩa là ta đã truy xuất đến biến đó một cách *gián tiếp*. Vì bản chất tham chiếu khác con trỏ nên tham chiếu có rất điểm khác con trỏ:

- Tham chiếu cần được khởi gán tại thời điểm nó được tạo ra, sau đó không thể thay đổi tham chiếu chỉ đến một đối tượng khác. Nghĩa là không dùng một tham chiếu cho nhiều biến khác nhau, không gán lại tham chiếu.
- Không có biến tham chiếu NULL. Không có tham chiếu đến một con trỏ.
- Không có đặc tính cộng trừ như con trỏ. Không thể gán chỉ số cho tham chiếu.
- Không thể tạo một mảng các tham chiếu.
- Con trỏ được dereference bằng toán tử * hoặc -> một cách tường minh. Tham chiếu được dereference tự động, không dùng toán tử. Vì vậy dùng tham chiếu có vẻ "tự nhiên" hơn, giống với dùng biến trực tiếp.

// con trỏ	// tham chiếu	// trực tiếp
int x;	int x;	int x;
int* p = &x;	int& y = x;	
*p = 10;	y = 10;	x = 10;
cout << *p;	cout << y;	cout << x;

2. Chú ý khi dùng tham chiếu

Biến tham chiếu hoàn toàn không có ý nghĩa cho đến khi nó được khởi tạo bằng cách gán (attach) với một referent nào đó. Gán cho aa trị của b, nghĩa là gán cho a trị của b, không phải thay đổi tham chiếu aa cho nó chỉ (kết nối - bound) đến b.

```
int a = 2, b = 3;
int& aa = a; // khởi gán tham chiếu aa là một alias của a
int c = aa; // gán c với tham chiếu aa, tương đương gán c với a (mà aa là alias)
aa = b; // thay đổi a với trị của b, không phải gán aa là alias của b
int& bb = 10; // SAI. Tham chiếu phải là alias của biến hoặc đối tượng
```

Có thể tham chiếu đến một đối tượng cấp phát động.

```
#include <iostream>
class Point
{
    int x, y;
public:
    Point( int a = 0, int b = 0 )
    { x = a; y = b; }
    void show()
    { std::cout << "(" << x << ", " << y << ")\n"; }
};

int main()
{
    // dùng tham chiếu của một đối tượng cấp phát động
    Point& p = *new Point( 3, 4 );
    p.show();
    // dùng con trỏ chỉ đến đối tượng cấp phát động
    Point* q = new Point( 5, 6 );
    q->show();
    return 0;
}
```

III. Truyền đối số cho một hàm

1. Truyền bằng trị (Pass By Value)

```
void DoubleIt( int x )
{ x *= 2; }

int main()
{
    int a = 8;
    DoubleIt( a );
    cout << a << endl; // kết quả: 8, a không đổi sau khi gọi hàm DoubleIt
    return 0;
}
```

Khi dùng truyền bằng trị biến a đến hàm DoubleIt(), đối số thực a sẽ được COPY đến đối số hình thức x: int x = a;

Hàm DoubleIt() sẽ thao tác trên biến cục bộ x này giống như thao tác trên một bản sao của a.

Hàm DoubleIt() không làm thay đổi được biến a do biến a ở tầm vực (scope) khác: thuộc về hàm main().

Truyền một mảng đến hàm tương đương với truyền con trỏ, hàm nhận đối số là mảng có thể thay đổi các phần tử của mảng.

Truyền con trỏ hằng (const con trỏ truyền) để bảo vệ mảng tránh thay đổi nếu cần.

2. Truyền bằng tham chiếu thông qua con trỏ (Pointer Based Pass By Reference)

Ta hiểu là truyền bằng trị một đối số kiểu con trỏ.

```
void DoubleIt( int* x )
{ *x *= 2; }

int main()
{
    int a = 8;
    DoubleIt( &a );
    cout << a << endl; // kết quả: 16, a thay đổi sau khi gọi hàm DoubleIt
    return 0;
}
```

Địa chỉ của đối số thực a được **COPY** đến đối số hình thức là con trỏ *x như sau: `int* x = &a;`

Như vậy, rõ ràng x là con trỏ chỉ đến biến a.

Hàm DoubleIt() không thể truy xuất **TRỰC TIẾP** biến a của hàm main(); nhưng vẫn có thể truy xuất **GIÁN TIẾP** và làm thay đổi đối số thực a thông qua con trỏ x chỉ đến nó.

Ta nhận thấy cú pháp sử dụng phức tạp và dễ nhầm lẫn cho cả người gọi hàm lẫn người cài đặt hàm, do dùng nhiều toán tử * và &. Tuy nhiên, đây là cách ngôn ngữ C dùng.

Ta thường dùng con trỏ để truy xuất đọc và ghi đến một đối tượng. Khi dùng chỉ với tác vụ đọc, ta có thể dùng từ khóa `const` để gán "nhãn chống ghi" cho đối tượng do con trỏ chỉ đến. Con trỏ trong trường hợp này gọi là con trỏ chỉ đọc (read-only pointer).

```
#include <iostream>
using namespace std;
// mô phỏng các hàm xử lý chuỗi của C trả về số ký tự có trong chuỗi
size_t _strlen( const char *str )
{
    const char* p = str;
    while ( *p ) ++p;
    return ( p - str );
}
// so sánh len ký tự của chuỗi s1 với s2
int _strncmp( const char *s1, const char *s2, size_t len )
{
    if ( len > _strlen( s1 ) ) len = _strlen( s1 );
    if ( len > _strlen( s2 ) ) len = _strlen( s2 );
    for ( int i = 0; i < len; ++i )
        if ( s1[i] != s2[i] ) return 1;
    return 0;
}
// tìm chuỗi s2 trong chuỗi s1
char* _strstr( const char *s1, const char *s2 )
{
    size_t len = _strlen( s2 );
    for ( ; *s1 != '\0'; ++s1 )
        if ( _strncmp( s1, s2, len ) == 0 )
            return ( char * )s1;
    return NULL;
}
// trả về vị trí các chuỗi s2 trong s1
int main()
{
    char* s = "hom qua qua khong qua";
    char* p = s;
    while ( ( p = _strstr( p, "qua" ) ) != NULL )
    {
        std::cout << p - s << ' ';
        p++;
    }
    return 0;
}
```

3. Truyền bằng tham chiếu (Pass By Reference)

Ta hiểu là truyền bằng trị một đối số kiểu tham chiếu.

```
void DoubleIt( int& x ){ x *= 2; }

int main()
{
    int a = 8;
    DoubleIt( a );
    cout << a << endl; // kết quả: 16, a thay đổi sau khi gọi hàm DoubleIt
}
```

```

    return 0;
}

```

Địa chỉ của đối số thực a được **COPY** đến đối số hình thức là tham chiếu &x như sau: `int& x = a;`

Như vậy, x là một bí danh hay một *tên khác* của đối số thực a.

Hàm `DoubleIt()` không thể truy xuất **TRỰC TIẾP** biến a của hàm `main()`; nhưng vẫn có thể truy xuất **GIÁN TIẾP** và làm *thay đổi* đối số thực a thông qua bí danh x của nó.

Ta nhận thấy cú pháp sử dụng đơn giản cho người gọi hàm, người cài đặt hàm chỉ phải quan tâm đến danh sách đối số.

Ví dụ khác: Tìm phần tử nhỏ nhất trong *một phần* của mảng

```

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

void order( int & p, int & q )
{
    if ( p > q )
    { int temp = p; p = q; q = temp; }
}

// truyền tham chiếu của con trỏ để toán tử new thay đổi nó
void createArray( int * & a, int size ) {
    srand( time( NULL ) );
    a = new int[ size ];
    for ( size_t i = 0; i < size; ++i )
        cout << ( a[i] = rand() % 201 - 100 ) << ' ';
    cout << endl;
}

int getMin( int* a, int size, int left, int right )
{
    if ( left < 0 ) left = 0;
    if ( right > size ) right = size;
    int min = a[left];
    for ( size_t i = left; i < right; ++i )
        order( min, a[i] );
    return min;
}

int main()
{
    int* a;
    createArray( a, 6 );
    cout << "Minimum = " << getMin( a, 6, 0, 6 ) << endl;
    return 0;
}

```

Truyền bằng tham chiếu có thể thay đổi gián tiếp đối số truyền cho hàm. Một số hàm thật sự cần điều này, ví dụ hàm trả về nhiều hơn một trị (returning multiple values). Nguy cơ là có thể vô tình thay đổi trị của đối số truyền cho hàm, ta giải quyết vấn đề đó bằng cách dùng tham chiếu hằng.

4. Áp dụng trong class

a) Tham chiếu hằng (Const Reference)

```

class Big
{
public:
    void Foo() { ... }
private:
    // dữ liệu thành viên
};

class Client
{
public:
    void valMethod( Big a ) { ... a.Foo() ... }

    void refMethod( Big & b ) { ... b.Foo() ... }

    // "nhãn chống ghi" cho c
    void crefMethod( const Big & c ) { ... c.Foo() ... }
    ...
private:

```

```
// dữ liệu thành viên
};
```

Phương thức thành viên `valMethod()` truyền đối số bằng trị. Đối số hình thức `a` là một bản **COPY** của đối tượng `Big`. Truyền bằng trị như vậy có nhiều vấn đề: nó, phương thức thành viên trở nên không an toàn.

Phương thức thành viên `CrefMethod()` cũng truyền đối số bằng tham chiếu. Nhưng đối số hình thức là *tham chiếu đã được bảo vệ* bởi từ khóa `const`. Bất kỳ mọi thay đổi có ảnh hưởng đến đối số `c` trong phương thức thành viên này đều được cảnh báo bằng lỗi biên dịch. Ta gọi: phương thức được truyền các *đối số tham chiếu chỉ đọc* (read-only)

Ta cũng gọi: Phương thức `CrefMethod()` được truyền một *tham chiếu hằng*. `const Big& c` không có nghĩa đối tượng kiểu `Big` mà `c` tham chiếu đến là một hằng. Nó chỉ cho thấy là đối tượng đó *không thể thay đổi* thông qua bí danh `c`.

Truyền bằng tham chiếu hằng có ưu thế của truyền bằng trị lẫn truyền bằng tham chiếu: ta muốn truyền bằng tham chiếu cho gọn nhẹ (không có sao chép đối số kích thước lớn) và cũng không muốn vô tình thay đổi đối số truyền cho phương thức.

```
#include <iostream>
using std::cout;
// truyền const reference thay cho by-value
void f( double a, double & b, const double & c ) { }
```

↑
không được thay đổi c
trong hàm thành viên f

```
int main()
{
    int i = 1, j = 2, k = 3;
    f( i, j, k );
    double d = 4.0;
    f( d + 1, d + 2, d + 3 );
    return 0;
}
```

OK. Đối số thực có thể chuyển kiểu thành `double` Lỗi. Không thể chuyển kiểu tham chiếu (không `const`)

OK. Đối số thực không cần là L-value Lỗi. Đối số thực phải là L-value

Lập trình viên C++:

- Phân biệt `const X*` (con trỏ đến đối tượng `X` hằng, đối tượng `X` không được thay đổi) với `Point const *` (con trỏ hằng chỉ đến đối tượng `X`, đối tượng `X` có thể thay đổi).
- Không phân biệt `const X &` với `X const &`, đều có nghĩa làm không thay đổi đối tượng `X` thông qua tham chiếu.

b) Phương thức thành viên hằng (Const Member Function)

Một số phương thức thành viên, thường là các phương thức xuất, thể hiện trạng thái của lớp, ... không làm thay đổi thành viên của lớp. Ta gọi chúng là các phương thức không đột biến (non-mutating) hoặc phương thức thành viên chỉ đọc (read-only). Các phương thức này thường được khai báo với từ khóa `const` ở cuối, cho thấy đây là một phương thức thành viên hằng. Trong phương thức thành viên hằng, chỉ được gọi các phương thức thành viên hằng khác.

```
#include <iostream>
class Stack
{
    int s[100], top;
public:
    Stack() { top = -1; }
    bool isEmpty() const;    // phương thức thành viên hằng
};
```

- Phát sinh nhiều vấn đề phức tạp nếu đối tượng `Big` có thành phần dữ liệu động (sẽ thảo luận trong một chương sau).

- Quá trình **COPY** sẽ chậm (tăng overhead) và chiếm nhiều bộ nhớ nếu đối tượng `Big` có kích thước lớn.

Phương thức thành viên `refMethod()` truyền đối số bằng tham chiếu. Việc quản lý đối tượng `Big` thông qua bí danh `b` của nó trở nên nhẹ nhàng hơn. Tuy nhiên nguy cơ tiềm ẩn là đối tượng được tham chiếu có thể bị thay đổi thông qua bí danh `b` của

```
bool Stack::isEmpty() const
{
    return ( -1 == top );
}
```


↑
isEmpty() không thể thay đổi *this, nghĩa là không thay đổi các thành viên của lớp

```
int main()
{
    Stack p;
    std::cout << p.isEmpty();
    return 0;
}
```

Do `const` trong trường hợp dùng với mục tiêu không làm thay đổi `*this`, nên không dùng với phương thức không phải là phương thức thành viên của lớp.


Phương thức thành viên hằng không làm thay đổi thành viên của lớp, tuy nhiên ràng buộc này không quá nghiêm ngặt, trong

phương thức thành viên hằng vẫn có thể dùng toán tử `const_cast<>` để thay đổi con trỏ hằng `this`:

```
#include <iostream>
using namespace std;
class X {
    int times;
    int value;
public:
    X( int t, int v ) : times( t ), value( v ) { }
    void display() const
    {
         chuyển đổi con trỏ hằng this
        thành con trỏ X* bình thường
        const_cast<X*>( this )->times++;
        cout << value << '[' << times << ']' << endl;
    }
};

int main()
{
    X a( 0, 2 );
    for ( int i = 0; i < 5; ++i )
        a.display();
    return 0;
}
```

Trong phương thức thành viên hằng, ta cũng có thể thay đổi các thành viên "có thể thay đổi được" (mutable), nghĩa là các thành viên có dùng từ khóa `mutable`, đây là các thành viên nếu cần thay đổi là có thể thay đổi được:

```
#include <iostream>
class X {
    mutable int times;  times là biến có thể
    thay đổi được
    int value;
public:
    X( int t, int v ) : times( t ), value( v ) { }
    void display() const
    {
        times++;
        std::cout << value << '[' << times << ']' << std::endl;
    }
};
```

c) Trả về một tham chiếu

Trả về một tham chiếu sẽ *không tạo đối tượng mới* mà tạo nên *một tên mới đến một đối tượng có sẵn*. Đối tượng có sẵn có thể là chính đối tượng gọi phương thức (`*this`) hay đối tượng được tạo trên heap (bằng cấp phát) từ trong phương thức.

Cần nhớ rằng đối tượng được tham chiếu trong trả về phải tồn tại sau khi thoát khỏi phương thức, việc trả về một tham chiếu của đối tượng cục bộ tạo trên stack (bằng khai báo cục bộ) trong phương thức sẽ gây lỗi do tham chiếu sẽ bị hủy theo đối tượng cục bộ khi phương thức kết thúc.

Không dùng một tham chiếu để nhận tham chiếu trả về, vì sẽ gặp nhiều lỗi tiềm ẩn khi giải phóng vùng nhớ cho đối tượng được tham chiếu, thay vào đó, ta có thể: khai báo một đối tượng, dùng nó để nhận tham chiếu trả về từ phương thức hoặc truyền nó bằng tham chiếu đến phương thức.

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

class Point
{
public:
    double x, y;
    Point( double x, double y )
        { this->x = x; this->y = y; }

    string toString()
    {
        stringstream ss;
        ss << "(" << x << ", " << y << " )\n";
        return ss.str();
    }

    Point upByVal( double dist )
        { y += dist; return *this; }

    Point& upByRef( double dist )
```

```

    { y += dist; return *this; }

    Point & mid( const Point & p ) const
    {
        return *new Point( ( x + p.x )/2, ( y + p.y )/2 );
    }
};

```

không thể tham chiếu đến một đối tượng tạm cục bộ trên stack;
mà cần tham chiếu đến một đối tượng được cấp phát trên heap

```

// trả về bằng trị, một đối tượng mới (một R-value)
Point leftmostVal( Point & p, Point & q )
{ return ( p.x < q.x ) ? p : q; }

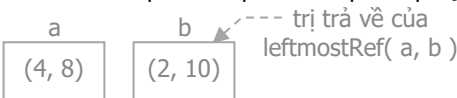
```



```

// trả về bằng tham chiếu đến một đối tượng cũ (một L-value)
Point & leftmostRef( Point & p, Point & q )
{ return ( p.x < q.x ) ? p : q; }

```



```

int main()
{
    Point a( 4, 8 ), b( 2, 10 ), c( 1, 5 ), d( 1, 5 );
    cout << leftmostVal( a, b ).toString();           // trả về ( 2, 10 )
    cout << leftmostRef( a, b ).toString();           // trả về ( 2, 10 )
    leftmostRef( a, b ) = c;                          // OK. Vì leftmostRef() là một L-value
    cout << b.toString();                             // trả về ( 1, 5 )
    a = leftmostRef( a, b );                          // OK. Chuyển kiểu Point& thành Point
    cout << a.toString();                             // trả về ( 1, 5 )

    // một đối tượng tạm là bản sao của a được tạo ra
    a.upByVal( 1 ).upByVal( 2 );
    cout << a.toString();                             // trả về ( 1, 6 )
    b.upByRef( 1 ).upByRef( 2 );
    // trả về ( 1, 8 ) vẫn tham chiếu đến đối tượng b
    cout << b.toString();
    cout << a.mid( b ).toString();                    // trả về ( 1, 7 )
    return 0;
}

```

Trong ví dụ trên ta nhận thấy *phương thức trả về một tham chiếu* có thể nằm ở *hai bên của phép gán* (vừa là L-value vừa là R-value). Nếu phương thức trả về tham chiếu là phương thức thành viên của một lớp, đối tượng được tham chiếu (thường là chính đối tượng gọi *this) có thể *bị ghi ngược trở lại* từ bên ngoài lớp.

```

#include <iostream>
using namespace std;
class Matrix
{
    double m[100][100];
    int deg;
public:
    Matrix( int n = 0, int val = 1 );
    // phương thức value() trả về một tham chiếu đến một vị trí trong biến thành viên mảng m
    double& value( int i, int j )
    { return m[i][j]; }
    Matrix operator*( int k );
    void display();
};

Matrix::Matrix( int n, int val ) : deg( n )
{
    for ( int i = 0; i < deg; ++i )
        for ( int j = 0; j < deg; ++j )
            value( i, j ) = val;
}

Matrix Matrix::operator*( int k )

```



```

{
    Matrix a( deg );
    for ( int i = 0; i < a.deg; ++i )
        for ( int j = 0; j < a.deg; ++j )
            a.value( i, j ) = k * value( i, j );
    return a;
}

```

↑ dùng hàm value() ở hai bên của phép gán

```

void Matrix::display()
{
    for ( int i = 0; i < deg; ++i )
    {
        for ( int j = 0; j < deg; ++j )
            cout << value( i, j ) << '\t';
        cout << endl;
    }
}

int main()
{
    Matrix a( 4, 2 );
    ( a * 3 ).display();
    return 0;
}

```

Có thể hạn chế tính "hai chiều" này bằng từ khóa `const` để có phương thức thành viên an toàn. Ta gọi: phương thức thành viên trả về một tham chiếu chỉ đọc (read-only reference).

```

class Point
{
public:
    double x, y;
    Point( double x, double y )
    { this->x = x; this->y = y; }
    const Point & upByRef( double dist )
    { y += dist; return *this; }
};

// ...
Point a( 4, 8 );
a.upByRef( 1 ); // OK. ( 4, 9 )
a.upByRef( 1 ).upByRef( 2 ); // Lỗi, a.upByRef( 1 ) trả về một tham chiếu hằng

```

Trả về một tham chiếu được dùng nhiều khi nạp chồng toán tử, vì dùng một toán tử nạp chồng thực chất là gọi một phương thức thành viên.

C++11 cung cấp khái niệm mới gọi là *rvalue reference*, dùng `&&` thay cho tham chiếu thay đổi được L-value &.

5. So sánh các kiểu truyền tham số

Truyền bằng ...	trị (không tham chiếu)	tham chiếu	tham chiếu hằng
Đối số hình thức là ...	biến cục bộ của hàm được gọi, sao chép trị đối số thực	tham chiếu đến đối số thực	tham chiếu hằng đến đối số thực hoặc đối tượng tạm
Hàm được gọi có thể làm thay đổi đối số hình thức?	Có	Có	Không
Đối số hình thức khi thay đổi sẽ ảnh hưởng đến đối số thực?	Không	Có	Không cho phép thay đổi đối số hình thức
Đối số thực có thể là ...	R-value	L-value (thay đổi được)	R-value
Đối số thực có thể chuyển kiểu trùng với đối số hình thức?	Có	Không (một số ngoại lệ)	Có (tạo biến tạm)

Lớp có thành phần dữ liệu cấp phát động

Dynamic Members

Bất kỳ lớp nào cũng có sẵn bốn phương thức do trình biên dịch tạo: hàm dựng mặc định (default constructor), hàm hủy (destructor), hàm dựng sao chép (copy constructor) và toán tử gán (assignment operator).

Một lớp có thể có *thành phần dữ liệu động* (dynamic members) như con trỏ, sẽ được cấp phát vùng nhớ khi đối tượng thuộc lớp sinh ra và hoạt động. Sự có mặt của thành phần dữ liệu động này mang đến những nguy cơ tiềm ẩn, nhất là khi có sự *sao chép đối tượng*. Khi các phương thức có sẵn trên được dùng để sao chép đối tượng, tác vụ sao chép là *sao chép bề mặt*, chính là nguyên nhân gây ra lỗi. Để giải quyết, phần này hướng dẫn xây dựng bốn phương thức cần thiết khi tạo lớp có thành phần dữ liệu động: hàm dựng (constructor), hàm hủy (destructor), hàm dựng sao chép (copy constructor) và nạp chồng toán tử gán (assignment operator).

C++11 dùng các con trỏ thông minh (smart pointer): `unique_ptr<>` và `shared_ptr<>` giúp quản lý bộ nhớ tự động, rất thuận tiện và an toàn khi xây dựng các lớp có thành phần dữ liệu động.

I. Constructor

1. Khái niệm

Constructor là thành phần đặc biệt không thể thiếu của một lớp, được *gọi tự động* khi tạo đối tượng. Constructor có chức năng chính là *cấp phát vùng nhớ* và *khởi gán* giá trị ban đầu cho các thành phần dữ liệu của đối tượng.

Một lớp có thể có một hoặc nhiều constructor, gọi là nạp chồng hàm dựng (constructor overloaded). Constructor có thể không có đối số (default constructor - hàm dựng mặc định) hoặc có các đối số. Các đối số của constructor có thể được gán trước, gọi là constructor có đối số mặc định.

Quy định viết constructor:

- Tên của constructor phải *cùng tên* với tên lớp.
- Constructor *không trả về* bất cứ trị nào, kể cả trả về void.

Không thể bắt lỗi constructor khi không tạo được đối tượng. Vấn đề này sẽ được giải quyết bằng ném exception.

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

class Point
{
private:
    int x, y, color;
    char* label;           // thành phần dữ liệu động
public:
    // Khai báo constructor có đối số mặc định
    Point( int = 0, int = 0, int = 1, char* = "" ) throw();

    void display() const {
        cout << label << "( " << x << ", " << y
            << ", " << color << " )" << endl;
    }
};

// Định nghĩa constructor dùng danh sách khởi tạo (base/member initialization list)
// cấp phát và khởi gán cho thành phần dữ liệu động
Point::Point( int x1, int y1, int col, char* s ) throw()
: x( x1 ), y( y1 ), color( col )
{
    label = new char[strlen( s ) + 1];
    strcpy( label, s );           // 1 cho ký tự kết thúc chuỗi '\0'
}
```

Constructor được *tự động gọi* khi:

- Tạo đối tượng tường minh (có đối số), dùng constructor có danh sách đối số tương ứng.
- Tạo đối tượng không tường minh (không đối số), dùng constructor mặc định.

Với constructor chỉ có *một đối số*, có thể gọi constructor đó bằng phép gán.

```
// hai cách sau tương đương nhau, gọi constructor một đối số
Year birth( 1945 );
Year birth = 1945;
// gọi constructor mặc định, lớp Year cần viết trước constructor mặc định
Year birth;
// SAI. Đây là khai báo một hàm trả về đối tượng kiểu Year
Year birth();
```

Trong constructor, khi dùng toán tử new để cấp phát mà không đủ bộ nhớ, xử lý new (new handler) mặc định sẽ được gọi. Nó sẽ ném exception và kết thúc chương trình. Như vậy không cần phải thiết kế mã xử lý lỗi mỗi lần gọi toán tử new. Với trình biên dịch cũ, new trả về NULL nếu không đủ bộ nhớ để cấp phát.

2. Constructor mặc định phát sinh và constructor mặc định khai báo

Nếu không định nghĩa constructor một cách tường minh (có hoặc không có đối số) thì trình biên dịch sẽ tạo ra một constructor mặc định phát sinh. Tuy nhiên constructor mặc định phát sinh này không thực hiện cấp phát và không thực hiện việc khởi tạo các biến.

```
#include <iostream>

class Complex
{
    double real, image;
public:
    // dùng constructor có tất cả đối số mặc định như constructor mặc định
    Complex( double a = 0.0, double b = 0.0 )
    : real( a ), image( b ) { }

    void display()
    {
        std::cout << real << " + " << image << "i" << std::endl;
    }
};

int main()
{
    // gọi constructor mặc định cho 5 phần tử của mảng quản lý bởi con trỏ a
    Complex *a = new Complex[5];
    // gọi constructor mặc định cho 1 phần tử của mảng b
    Complex *b = new Complex;
    for ( int i = 0; i < 5; ++i )
        ( a + i )->display();
    b->display();
    return 0;
}
```

Ngược lại, nếu đã định nghĩa *ít nhất* một constructor thì sẽ không tạo ra constructor mặc định phát sinh nữa. Khi tạo đối tượng không dùng đối số, constructor mặc định khai báo (do ta tạo) sẽ được gọi, nếu không tìm được trình biên dịch sẽ báo lỗi. Vì vậy cần:

- Tạo một constructor mặc định khai báo, nghĩa là viết một constructor không đối số.
- Hoặc tạo constructor *có các đối số đều mặc định*, khi tất cả các đối số đều được dùng mặc định thì constructor đó trở thành constructor mặc định.

3. Dùng constructor khi khai báo đối tượng

a) Biến đối tượng và mảng đối tượng

Constructor sẽ được *tự động gọi* và được lựa chọn gọi tùy theo danh sách đối số của các constructor trong lớp của đối tượng được khai báo, nghĩa là tùy theo signature của constructor.

Khi khai báo biến đối tượng ta dùng danh sách đối số của constructor tương ứng khởi tạo cho các thuộc tính của đối tượng.

```
Date birthday( 19, 5, 1890 ); // dùng constructor có 3 đối số
```

Khi khai báo mảng các đối tượng (còn gọi là mảng lớp – class arrays) không thể dùng đối số để khởi tạo hàng loạt. Tuy nhiên có thể khởi tạo từng phần tử của mảng các đối tượng với constructor tường minh.

```
Date holidays[2]; // dùng constructor mặc định cho mỗi đối tượng
// khởi tạo cụ thể cho mỗi đối tượng
Date holidays[] = { Date( 1, 5 ), Date( 2, 9 ) };
```

b) Con trỏ chỉ đến đối tượng

Khi khai báo và cấp phát một *con trỏ chỉ đến đối tượng*, ta dùng toán tử new với constructor tương ứng và danh sách đối số cụ thể để khởi tạo cho các thuộc tính của *đối tượng nơi con trỏ chỉ đến*, ta gọi là khởi tạo đối tượng thông qua con trỏ.

Với mảng các con trỏ chỉ đến đối tượng, ta cũng khởi tạo theo cách tương tự.

```
Date* holidays = new Date( 30, 4 );
Date* holidays[] = { new Date( 1, 5 ), new Date( 2, 9 ) };
```

c) Đối tượng tạm

Đối tượng tạm, còn gọi là đối tượng vô danh (nameless objects), thường xuất hiện khi tạo và trả một đối tượng trở về từ một hàm. Để thể hiện đối tượng tạm người ta dùng cách viết:

```
ClassName( arguments_list )
```

Ví dụ:

```
Point Point::operator+ ( const Point & right )
{
    return Point( x+right.x, y+right.y, color + right.color );
}
```

II. Destructor

1. Khái niệm

Destructor được gọi *tự động* khi:

- Thoát khỏi hàm hoặc phương thức, khi đó các biến cục bộ sẽ bị hủy. Nếu biến cục bộ là đối tượng của một lớp thì destructor của lớp đó sẽ được gọi.
- Khi kết thúc việc thực hiện các toán tử nạp chồng có toán hạng là đối tượng, điều này tương đương với lời gọi hàm.
- Khi giải phóng đối tượng được cấp phát bằng toán tử delete.

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

class Point
{
private:
    int x, y, color;
    string label;
public:
    // ...
    Point( int a = 0, int b = 0, int c = 1, string s = "" )
    : x( a ), y( b ), color( c ), label( s ) { }

    ~Point()
    {
        cout << "Point::~destructor()" << endl;
    }

    void display()
    {
        cout << label << "( " << x << ", " << y
            << ", " << color << " )" << endl;
    }
};

int main()
{
    Point* a = new Point( 3, 5 );
    {
        Point b( 4, 7 );
    } // destructor của đối tượng b được gọi, do b ra ngoài tầm vực
    delete a; // destructor của đối tượng a được gọi, do con trỏ quản lý a bị hủy
}
```

Chú ý với toán tử delete:

- Không dùng delete hai lần với cùng một đối tượng.
- Không dùng delete để giải phóng biến cấp phát tĩnh.

2. Destructor mặc định và destructor tường minh

Nếu không có destructor tường minh trình biên dịch sẽ cung cấp một destructor mặc định. Nếu lớp *không* dùng thành phần dữ liệu động thì destructor mặc định đủ giải quyết vấn đề giải phóng bộ nhớ.

Khi lớp có thành phần dữ liệu động, cần viết destructor tường minh. Mỗi lớp chỉ có một destructor tường minh (không nạp chồng).

Nhiệm vụ chủ yếu của destructor tường minh là giải phóng vùng nhớ cho thành phần dữ liệu động của lớp.

Cần phải có destructor trong các lớp có thành phần dữ liệu động, nhằm bảo đảm đủ các cặp new/delete tương ứng. Điều này giúp đối tượng của lớp giải phóng bộ nhớ chiếm dụng khi ra khỏi phạm vi (scope) của nó, tránh tình trạng bộ nhớ heap bị phân mảnh (memory leakage) do chiếm dụng không hợp lý.

Quy định viết destructor tường minh:

- Tên của destructor phải *cùng tên* với tên lớp, có dấu ~ đầu tên.
- Destructor *không trả về* bất cứ trị nào, ngay cả void.
- Destructor không có danh sách đối số nên không có destructor nạp chồng.
- Destructor phải public

III. Shallow copy và Deep copy

Với đối tượng thuộc lớp có thành phần dữ liệu động, việc sao chép (copy) đối tượng có thể gây lỗi tiềm ẩn.

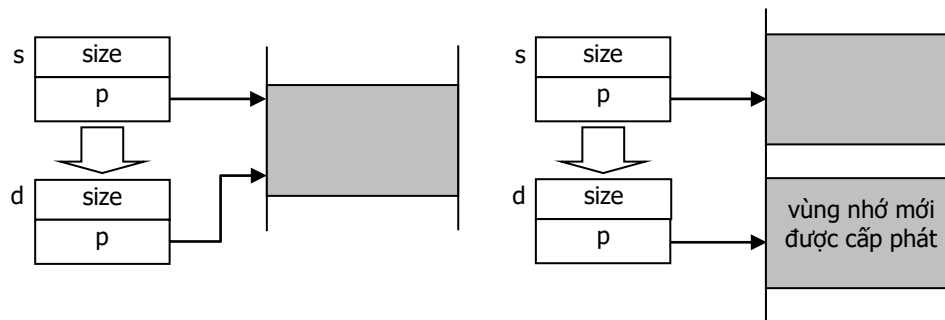
Sao chép một đối tượng sẽ xảy ra khi:

- Khởi tạo một đối tượng từ một đối tượng có sẵn, không tường minh.
- Thực hiện toán tử gán một đối tượng cho một đối tượng khác, tường minh.
- Truyền bằng trị một đối tượng đến một hàm, nghĩa là có sự sao chép từ đối số thực đến đối số hình thức, không tường minh.
- Trả về một đối tượng từ hàm, nghĩa là có sự sao chép từ đối tượng cục bộ đến đối tượng tạm để trả về, không tường minh.

Chú ý sử dụng toán tử nạp chồng tương đương với lời gọi hàm thành viên, nghĩa là cũng có sự trả về một đối tượng.

Khi thực hiện các quá trình sao chép nói trên, sự sao chép bề mặt được sử dụng nên sẽ nảy sinh vấn đề cả hai đối tượng (gốc và sao chép) *dùng chung thành phần dữ liệu động* nếu có, làm cho các đối tượng này không tách biệt nhau.

Ta dùng cơ chế sao chép sâu để giải quyết vấn đề này.



shallow copy

deep copy

Sao chép bề mặt (shallow copy): khi sao chép đối tượng s sang đối tượng d, mọi thành phần dữ liệu của đối tượng s sẽ được sao chép sang đối tượng d. Chú ý rằng *địa chỉ chứa trong con trỏ p* (thành phần dữ liệu động) của s cũng được sao chép giống vậy đến con trỏ p của d. Kết quả là cả hai đối tượng s và d *dùng chung một vùng nhớ* vì p của chúng *chứa chung một địa chỉ*, như vậy đưa đến lỗi tiềm ẩn:

- Thay đổi thành phần dữ liệu động của một đối tượng sẽ làm đối tượng kia thay đổi theo.
- Hủy một đối tượng bằng destructor sẽ ảnh hưởng đến đối tượng kia. Vùng nhớ cấp phát chung bị hủy theo một đối tượng làm con trỏ của đối tượng còn lại bị lạc (hay bị "treo") (stray pointer hay dangling pointer).

Sao chép sâu (deep copy): giải pháp cho vấn đề dùng chung vùng nhớ là phải thực hiện việc sao chép sâu gồm các bước sau:

- Sao chép các thành phần dữ liệu tĩnh.
- Cấp phát vùng nhớ mới cho các thành phần dữ liệu động.
- Sao chép dữ liệu từ vùng nhớ cũ sang vùng nhớ mới.

Như vậy hai đối tượng sẽ tách rời nhau ra.

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

class DoubleVector
{
    friend ostream & operator<<( ostream &, const DoubleVector & );
private:
    double* p; // thành phần dữ liệu động của lớp
    size_t size;
public:
    DoubleVector( int = 10 ) throw( bad_alloc );
    ~DoubleVector()
    { delete p; } // giải phóng cấp phát trong destructor
    int ubound() const { return size - 1; }
    double & operator[]( int i ) throw( out_of_range );
};

DoubleVector::DoubleVector( int n ) throw( bad_alloc )
: size( n ) {
    if ( n < 0 ) throw bad_alloc();
    p = new double[ size ]; // cấp phát trong constructor
}

double & DoubleVector::operator[]( int i ) throw( out_of_range )
{
    if ( i < 0 || i >= size )
        throw out_of_range( "::operator[]" );
    return p[i];
}

ostream & operator<<( ostream& os, const DoubleVector& right )
{
    for ( int i = 0; i < right.size; ++i )
        os << right.p[i] << ' ';
    return os << endl;
}

DoubleVector & foo( DoubleVector & dv )
{
    for ( int i = 0; i <= dv.ubound(); ++i )
        dv[i] = i + i/10.0;
    return dv;
}

int main()
```

nếu không truyền bằng tham chiếu, copy constructor mặc định sẽ được gọi tự động ở đây, do đối số thực s được **COPY** đến đối số hình thức d (có sự sao chép đối tượng)

nếu không trả về một tham chiếu, copy constructor sẽ được gọi tự động ở đây, do đối tượng cục bộ dv được **COPY** đến đối tượng tạm để trả về

```
{
    DoubleVector s( 10 ), b;
    foo( s ); // gọi toán tử gán mặc định ở đây và
    b = s;   ← không tách biệt được s với b do
    return 0; // dùng cơ chế sao chép bề mặt
}
```

Đối với các lớp có thành phần dữ liệu động: constructor thực hiện việc cấp phát, destructor thực hiện việc giải phóng, copy constructor và toán tử gán *phải được xây dựng trên cơ chế deep copy* vì copy constructor mặc định và toán tử gán mặc định chỉ thực hiện cơ chế shallow copy). Mục đích là tạo một đối tượng hoàn toàn độc lập với đối tượng cũ bằng cơ chế deep copy.

IV. Hàm dựng sao chép (copy constructor)

1. Khái niệm

Copy constructor được *tự động* gọi trong các tình huống sau:

- Cần tạo đối tượng mới có nội dung tương tự đối tượng đã có.

```
Complex c1( 2, 3 );
Complex c2 = c1;
Complex c3( c2 ); // thực chất chính là truyền bằng trị đối tượng đến constructor
```

- Khi truyền một đối tượng đến một hàm theo cơ chế *truyền bằng trị* (pass by value). Khi đó sẽ có sự sao chép từ đối số thực đến đối số hình thức.

Để tránh dùng copy constructor trong trường hợp này, ta thường dùng *tham chiếu* để truyền đối tượng. Khi đó ta làm việc trực tiếp với đối tượng nguồn, không phải với bản sao của nó.

- Trả về một đối tượng từ hàm (trả về bằng trị), khi đó một đối tượng mới sẽ được sao chép từ một đối tượng cục bộ bên trong hàm hay từ chính đối tượng gọi hàm (*this) ra bên ngoài.

```
Complex Complex::operator+( const Complex & right )
{
    return Complex( real + right.real, image + right.image );
}

Complex Complex::operator=( const Complex & right )
{
    if ( this != &right )
    {
        real = right.real;
        image = right.image;
    }
    return *this;
}
```

Copy constructor khác phép gán (gọi toán tử gán) ở các điểm cơ bản sau:

- Phép gán thực hiện việc sao chép nội dung từ đối tượng này sang đối tượng khác, do đó cả hai đối tượng của phép gán *đều đã tồn tại*.

```
Complex c1( 2, 3 );
Complex c2;
c2 = c1; // gọi toán tử gán
```

- Copy constructor đồng thời thực hiện hai nhiệm vụ: cấp phát mới đối tượng và sao chép nội dung từ đối tượng đã có vào đối tượng mới.

```
Complex c1( 2, 3 );
Complex c2 = c1; // gọi copy constructor, viết khác: Complex c2( c1 );
```

2. Hàm dựng sao chép tường minh

Nếu không định nghĩa copy constructor một cách tường minh thì trình biên dịch cung cấp một copy constructor mặc định, còn gọi là copy constructor chuẩn. Copy constructor mặc định chỉ đơn thuần tạo đối tượng mới và các giá trị thành phần trong đối tượng mới bằng các giá trị thành phần trong đối tượng cũ (shallow copy). Đối với các lớp không có các thành phần dữ liệu động (con trỏ hoặc tham chiếu) chỉ cần dùng copy constructor mặc định là đủ.

Khi lớp có thành phần dữ liệu động thì copy constructor mặc định chưa đủ đáp ứng yêu cầu vì gây ra tình trạng dùng chung vùng nhớ với đối tượng cũ. Do đó gây ra lỗi tiềm ẩn.

Khai báo và định nghĩa copy constructor tường minh theo cú pháp:

```
ClassName ( const ClassName & );
```

- Dấu & bắt buộc dùng. Nghĩa là phải truyền bằng tham chiếu mà không phải bằng trị, vì cơ chế truyền bằng trị (có COPY) sẽ gây ra đệ quy khi trình biên dịch xử lý kiểu dữ liệu mới.

- Do truyền bằng tham chiếu, nên có thêm từ khóa const nhằm ngăn ngừa việc thay đổi đối số truyền cho hàm.

Copy constructor dùng sao chép dữ liệu từ đối tượng được truyền đến đối tượng gọi (*this) được xây dựng theo các bước của cơ chế deep copy như sau:

- Sao chép thành phần dữ liệu tĩnh.
- Cấp phát vùng nhớ mới cho các thành phần dữ liệu động.
- Chuyển dữ liệu từ nơi cấp phát cũ sang nơi cấp phát mới.

Ví dụ: copy constructor cho lớp DoubleVector trên:

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

```

class DoubleVector
{
    // xem ví dụ phần III
private:
    double* p;
    int size;
public:
    // copy constructor
    DoubleVector( const DoubleVector & ) throw( bad_alloc );
    // xem ví dụ phần III
};

// Copy constructor dùng deep copy
DoubleVector::DoubleVector( const DoubleVector & dv ) ← copy constructor sẽ được gọi tự động ở đây (tạo đối tượng từ đối tượng)
    throw( bad_alloc ): size( dv.size )
{
    p = new double[size]; ← cấp phát vùng nhớ mới cho thành phần dữ liệu động
    memcpy( p, dv.p, size * sizeof( double ) ); ← sao chép dữ liệu tĩnh bằng danh sách khởi tạo chỉ có ở constructor, kể cả copy constructor
    // sao chép dữ liệu từ nơi cấp phát cũ sang nơi cấp phát mới
}

// Hàm toàn cục dùng kiểm tra copy constructor
DoubleVector boo( DoubleVector dv ) ← copy constructor sẽ được gọi tự động ở đây (truyền bằng trị)
{
    for( int i = 0; i < dv.ubound(); ++i )
        dv[i] = i + i/10.0;
    return dv; ← copy constructor sẽ được gọi tự động ở đây (trả về đối tượng tạm)
}

int main()
{
    DoubleVector a( 10 );
    DoubleVector b = a;
    boo( a );
    return 0;
}

```

V. Toán tử gán (Assignment Operator)

1. Khái niệm

Toán tử gán dùng để gán một đối tượng này cho một đối tượng khác. Theo nguyên tắc, toán tử gán chỉ được áp dụng cho các đối tượng *cùng kiểu* và *đã tồn tại*.

Nếu không định nghĩa toán tử gán tường minh, trình biên dịch sẽ dùng toán tử gán mặc định. Thực chất là sao chép các thành phần dữ liệu từ đối tượng này sang đối tượng kia theo cơ chế shallow copy.

Đối với lớp có các thành phần dữ liệu động thì phải viết toán tử gán tường minh (dùng cơ chế deep copy) để tránh việc hai đối tượng cũ và mới dùng chung thành phần dữ liệu động đó.

2. Toán tử gán tường minh

Toán tử gán tường minh dùng sao chép dữ liệu từ đối tượng được truyền như đối số đến đối tượng gọi (**this*) *đã có sẵn* (toán hạng bên trái), đồng thời trả về tham chiếu chỉ đến toán hạng bên trái.

Cú pháp:

```
[const] ClassName & operator=( const ClassName & )
```

Xây dựng toán tử gán tường minh theo các bước sau:

- Khi thực thi toán tử gán có giải phóng vùng nhớ. Vì vậy để tránh tự hủy, cần kiểm tra để tránh việc tự gán bằng *so sánh địa chỉ*, không phải so sánh đối tượng (vì lại phải viết nạp chồng toán tử so sánh).
- Giải phóng vùng nhớ của đối tượng *bên trái* phép gán (viết giống destructor). Cần nhớ là đối tượng bên trái này *đã tồn tại*.
- Sao chép các thành phần dữ liệu theo các bước của cơ chế deep copy (viết giống copy constructor).
- Phải có kiểu trả về để có thể gán liên tiếp, ví dụ *a = b = c*. Kiểu trả về thường là một tham chiếu đến đối tượng bên trái (return **this*).

Ví dụ:

```

#include <cstring>
// ...
class Person
{
    char *name;
    int age;
public:
    // nạp chồng toán tử gán
    const Person & operator=( const Person & );
}

```



```
};

const Person & Person::operator=( const Person & right )
{
    // so sánh 2 địa chỉ xem có phải trường hợp tự gán hay không, nhằm tránh tự hủy
    if ( this == &right ) return *this;
    // giải phóng vùng nhớ thành phần dữ liệu động của đối tượng bên trái
    delete []name;
    // chuyển dữ liệu từ đối tượng về phải sang theo cơ chế deep copy
    age = right.age;
    name = new char[ strlen( right.name ) + 1 ];
    strcpy( name, right.name );
    // trả về một tham chiếu đến đối tượng về trái, nghĩa là *this
    return *this;
}
```

3. Phân biệt toán tử gán và hàm dựng sao chép

Chú ý không phải mọi câu lệnh có dấu "=" đều gọi toán tử gán. Cần phân biệt 3 trường hợp sau:

- Câu lệnh "=" new" cấp phát vùng nhớ cho một đối tượng mới bằng toán tử new, khởi tạo cho đối tượng bằng cách gọi constructor mặc định hoặc có tham số, trả về một con trỏ quản lý đối tượng đó.

```
Point *p = new Point( 3, 5, 14 );
```

- Câu lệnh khai báo và khởi tạo một đối tượng từ một đối tượng có sẵn sẽ gọi copy constructor.

```
Point p1 = *p;
// tương đương Point p1( *p );
```

- Câu lệnh gán sẽ gọi toán tử gán.

```
Point p2;        // gọi constructor mặc định
p2 = *p;         // gọi toán tử gán nạp chồng
```

Để kiểm tra hoạt động của copy constructor và toán tử gán tường minh, sinh ra các cặp đối tượng gốc và sao chép, thay đổi một đối tượng để xem cặp đối tượng đó có dùng chung thành phần dữ liệu nào không.

Đôi khi ta cũng muốn ngăn cản người dùng sao chép hoặc gán các đối tượng của lớp. Để thực hiện điều này, đưa khai báo copy constructor và nạp chồng toán tử gán vào phần private hoặc protected, không cần cài đặt chúng.

Lớp bao và lớp thành phần

Composed class – Embedded class

Có hai cách tiếp cận khi xây dựng một lớp mới:

- **Tổng hợp**: (composition) kết hợp một hay nhiều lớp đã có để tạo lớp mới, đối tượng của các lớp đã có trở thành *thành viên* của lớp mới. Ta nhận được lớp bao hay lớp phức hợp (composed class).

- **Thừa kế**: (inheritance) thừa kế các lớp đã có để tạo lớp mới, nghĩa là lớp mới được tạo ra *trên cơ sở* lớp đã có. Ta nhận được lớp dẫn xuất (derived class).

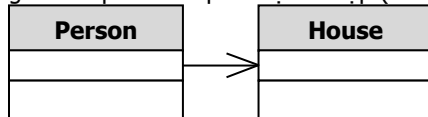
Thực tế, một lớp được tạo ra bằng cách phối hợp cả hai cách trên.

I. Khái niệm

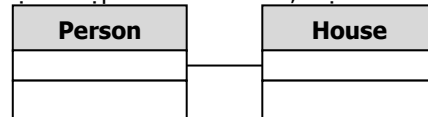
Một lớp có các dữ liệu thành viên là các đối tượng thuộc các lớp khác gọi là lớp bao (hay lớp phức hợp – composed class).

Một lớp có các đối tượng là dữ liệu thành viên của lớp bao được gọi là lớp thành phần (hay lớp nhúng – embedded class), các đối tượng được nhúng trong lớp bao gọi là đối tượng con (subobject).

Quan hệ giữa 2 lớp trên là quan hệ kết hợp (association). Quan hệ kết hợp có thể hai chiều, một chiều hoặc đệ quy.



House được dùng trong lớp Person



House có trong lớp Person và ngược lại

// quan hệ liên kết một chiều

```
class Person {
    House* theHouse;
public:
    Person();
    ~Person;
};

class House {
public:
    House();
    ~House();
};
```

// quan hệ liên kết hai chiều

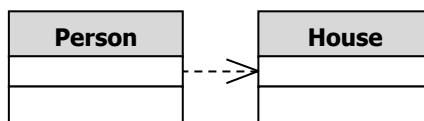
```
class Person {
    House* theHouse;
public:
    Person();
    ~Person;
};

class House {
    Person* the Person;
public:
    House();
    ~House();
};
```

II. Quan hệ giữa lớp bao và lớp thành phần

1. Quan hệ phụ thuộc (dependency)

Quan hệ phụ thuộc cũng là quan hệ kết hợp giữa hai lớp nhưng chỉ là quan hệ một chiều, chỉ ra một lớp phụ thuộc vào lớp khác. Quan hệ này cho thấy một lớp tham chiếu đến một lớp khác. Do vậy, nếu lớp được tham chiếu thay đổi sẽ ảnh hưởng đến lớp sử dụng nó. Subobject dùng trong trường hợp này thường là biến toàn cục, biến cục bộ của hàm thành viên hoặc đối số của hàm thành viên.



Person có tham chiếu đến House khi tạo lớp

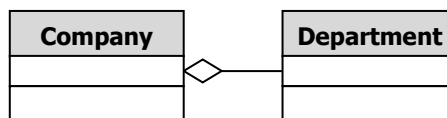
```
class Person {
public:
    void buyHouse( House* theHouse );
};
```

2. Quan hệ tụ hợp (aggregation)

Quan hệ tụ hợp là hình thức mạnh của quan hệ kết hợp. Quan hệ kết hợp giữa hai lớp cho thấy chúng cùng mức, không có lớp nào quan trọng hơn. Quan hệ tụ hợp là quan hệ giữa toàn thể và bộ phận trong đó một lớp biểu diễn cái lớn hơn (tổng thể, lớp bao), còn lớp kia biểu diễn cái nhỏ hơn (bộ phận, lớp thành phần).

Có 2 loại quan hệ tụ hợp:

- Nếu tổng thể và thành phần được hình thành và hủy bỏ vào các thời điểm khác nhau, ta gọi là quan hệ tụ hợp bởi tham chiếu hoặc quan hệ kết tập.

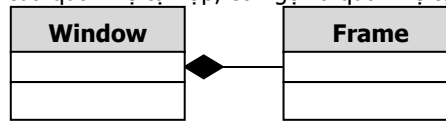


Company có một hay nhiều Department

```
class Company {
```

```
vector<Department*> departments;
public:
    Company();
    ~Company();
    bool addDepartment();
};
```

- Nếu tổng thể và thành phần được hình thành và hủy bỏ cùng thời điểm, ta gọi là quan hệ gộp hoặc quan hệ hợp thành (composition), đây là hình thức mạnh hơn của quan hệ tự hợp, còn gọi là quan hệ tự hợp bởi trị.



Window có một hay nhiều Frame

```
class Window
{
    Frame AboutFrame;
public:
    Window();
    ~Window();
};
```

III. Các phương thức của lớp bao

1. Constructor

Khi xây dựng constructor của lớp bao, ngoài việc khởi gán dữ liệu riêng của lớp bao, còn phải khởi gán cho dữ liệu là các subobject. Các phương thức của lớp bao *không thể truy cập* được đến dữ liệu nội tại của subobject, do đó phải sử dụng các phương thức của lớp thành phần tương ứng để khởi gán cho các subobject:

- Bằng constructor hoặc copy constructor của lớp thành phần. Trình biên dịch căn cứ vào danh sách đối số, chọn constructor thích hợp trong lớp thành phần tương ứng để khởi gán cho các subobject. Công việc này được thực hiện trong danh sách khởi tạo, thông qua *tên của các subobject* (không phải tên lớp thành phần) tương ứng. Các subobject không liệt kê trong danh sách khởi tạo sẽ được khởi tạo tự động bằng constructor mặc định (không tham số) của lớp thành phần tương ứng.

```
composed_class ( argument_values )
: sub_object_i ( argument_values_i ), ...
{
    // các phát biểu khởi gán các thành phần dữ liệu riêng của lớp bao,
    // không kể dữ liệu là các subobject.
}
```

- Bảng toán tử gán của lớp thành phần (xem ví dụ).

```
#include <iostream>
#include <cmath>

class Point // mô tả một điểm trong hệ tọa độ Cartesian
{
public:
    Point( double xval = 0.0, double yval = 0.0 )
    : x( xval ), y( yval ) { }
    // các accessor của lớp thành phần
    double getX() const { return x; }
    double getY() const { return y; }
private:
    double x, y;
};

class Line // mô tả một đường xác định bởi hai điểm hoặc tọa độ hai điểm
{
public:
    Line( const Point & p1, const Point & p2 );
    Line( double plx, double ply, double p2x, double p2y );
    operator double() const;
private:
    Point pt1, pt2; // subobject
    double length;
};

// dùng tên của subobject để khởi tạo
Line::Line( const Point & p1, const Point & p2 )
: pt1( p1 ), pt2( p2 )
{
    // khởi tạo từ copy constructor
    // mặc định của lớp thành phần
    length = (double)( *this );
}
```

```

}

/* hoặc
Line::Line( const Point& p1, const Point& p2 )
{
    pt1 = p1; ← khởi tạo từ toán tử gán mặc
    pt2 = p2;   định lớp thành phần
    length = (double)(*this);
}
*/

Line::Line( double p1x, double p1y, double p2x, double p2y )
: pt1( p1x, p1y ), pt2( p2x, p2y )
{
    ← khởi tạo từ constructor lớp thành phần
    length = ( double )( *this );
}

Line::operator double() const
{
    // lớp bao dùng các accessor của lớp thành phần
    double dx = pt2.getX() - pt1.getX();
    double dy = pt2.getY() - pt1.getY();
    return sqrt( dx * dx + dy * dy );
}

int main()
{
    Point p( 3, 2 ), q( -1, 4 );
    Line pq( p, q ); // hoặc Line pq( Point( 3, 2 ), Point( -1, 4 ) );
    std::cout << (double)pq << std::endl;
    Line pq1( 3, 2, -1, 4 );
    std::cout << (double)pq1 << std::endl;
    return 0;
}

```

Các constructor được gọi theo *thứ tự khai báo* của subobject trong lớp bao, không phải theo thứ tự của các subobject trong danh sách khởi tạo. Thứ tự gọi các destructor thì ngược lại.

Đối số trao cho constructor của các subobject không nhất thiết phải là trị cụ thể mà có thể là biểu thức. Như vậy, có khả năng quy định đặc trưng của đối tượng vào lúc chạy chương trình.

Một số đối số của constructor lớp bao có thể không cần dùng trong thân hàm mà có mặt để dùng trong danh sách khởi tạo, dùng khi khởi gán các subobject. Nghĩa là đối số của constructor lớp bao gồm: các đối số dùng khởi tạo dữ liệu thành viên của lớp bao và các đối số dùng khởi tạo các subobject của lớp bao.

Lớp bao chứa mảng các đối tượng thuộc lớp thành phần: constructor của các đối tượng trong mảng luôn được gọi một cách tự động, lớp của các đối tượng này phải có sẵn constructor mặc định.

Lớp bao chứa con trỏ chỉ đến đối tượng thuộc lớp thành phần: constructor lớp bao chấp nhận con trỏ chỉ đến đối tượng thuộc các lớp dẫn xuất từ lớp thành phần vì chúng cũng được xem là đối tượng thuộc lớp thành phần. Phần này sẽ xem xét trong các chương sau.

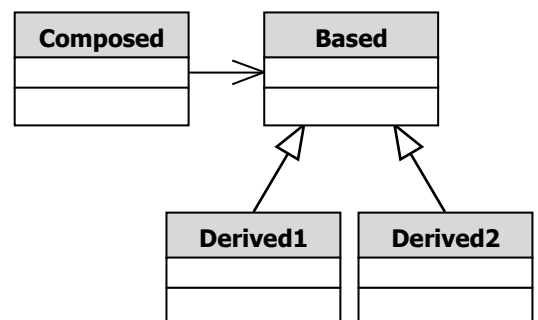
// có thể "cắm" vào đối tượng lớp Composed các subobject kiểu khác nhau dẫn xuất từ kiểu Base

```

class Base { };
class Derived1 : public Base { };
class Derived2 : public Base { };
class Composed
{
    Base* p0;
public:
    Composed ( Base *p ) { p0 = p; }
};

int main()
{
    Derived1 d1;
    Derived2 d2;
    // khởi gán cho constructor của Compose các dẫn xuất từ Base
    Composed c1( &d1 );
    Composed c2( &d2 );
    return 0;
};

```



2. Copy constructor, destructor và nạp chồng toán tử gán của lớp bao

Nếu lớp bao không chứa thành phần dữ liệu động thì không nhất thiết xây dựng các phương thức copy constructor, destructor và nạp chồng toán tử gán tường minh, chỉ sử dụng các phương thức tương ứng mặc định là đủ.

Nếu lớp bao chứa các thành phần dữ liệu động thì cần thiết phải xây dựng các hàm trên theo nguyên tắc sau:

- Khi lớp bao thực hiện toán tử gán, ngoài dữ liệu thành phần các subobject cũng được gán. Vì vậy toán tử gán của các lớp thành phần phải được xây dựng tường minh hoặc mặc định, tùy theo lớp thành phần có dữ liệu động hoặc không.
- Destructor của lớp bao cũng cần được xây dựng nhằm hủy bỏ các thành phần dữ liệu động có trong lớp bao, còn các thành phần dữ liệu động có trong lớp thành phần sẽ được tự động hủy bỏ nhờ các destructor tường minh có trong lớp thành phần.
- Copy constructor trong lớp bao được xây dựng theo hai cách sau:

Cách 1: Copy constructor trong lớp bao được xây dựng dựa trên toán tử gán tường minh của các lớp thành phần.

```
const composed_class ( const composed_class & A )
{
    sub_object_1 = A.sub_object_1;
    sub_object_2 = A.sub_object_2;
    ...
    sub_object_k = A.sub_object_k;
    // các phát biểu để khởi gán các thuộc tính của A cho các thuộc tính của đối tượng mới this
}
```

Cách 2: Copy constructor trong lớp bao được xây dựng dựa trên copy constructor tường minh của các lớp thành phần, dùng danh sách khởi tạo.

```
const composed_class ( const composed_class & A ) :
    sub_object_1 ( A.sub_object_1 ),
    sub_object_2 ( A.sub_object_2 ),
    ...
    sub_object_k ( A.sub_object_k )
{
    // các phát biểu để khởi gán các thuộc tính của A cho các thuộc tính
    // của đối tượng mới this, không kể các subobject trong lớp bao.
}
```

3. Truy cập đến dữ liệu riêng của lớp thành phần

Dùng thành viên của các lớp thành phần để thực hiện các chức năng của lớp bao gọi là ủy nhiệm (delegation). Kết quả là hiệu suất (performance) hoạt động của lớp bao bị ảnh hưởng do việc khởi tạo và hủy đối tượng của các lớp thành phần.

Lớp bao tuy chứa các subobject thuộc lớp thành phần nhưng KHÔNG được phép truy cập đến dữ liệu riêng của các subobject này. Để truy xuất đến cần phải:

- Về phía lớp bao: các phương thức của lớp bao dùng các phương thức public của lớp thành phần để truy xuất đến các thành phần dữ liệu của subobject cần dùng.
- Về phía lớp thành phần: cần xây dựng sẵn các phương thức truy xuất (các accesor) public để có thể lấy các giá trị từ dữ liệu riêng ra, cho phép lớp bao gọi các phương thức truy xuất này.

Một phương án khác là dẫn xuất kiểu private từ một lớp cơ sở. Khi đó, quan hệ giữa lớp cơ sở và lớp dẫn xuất không phải là quan hệ IS-A. Lớp dẫn xuất giống như lớp bao chứa một thể hiện của lớp cơ sở, ngoài ra lớp dẫn xuất có thể truy xuất thành viên protected của lớp cơ sở. Tuy nhiên, nếu cần nhiều hơn một thể hiện của lớp cơ sở, phải dùng cách composition.

Thừa kế

Inheritance

Sử dụng lại (reusability) là một đặc điểm quan trọng của OOP, thừa kế là cơ chế giúp tạo ra một lớp mới từ những thuộc tính và phương thức của những lớp đã tồn tại.

I. Khái niệm

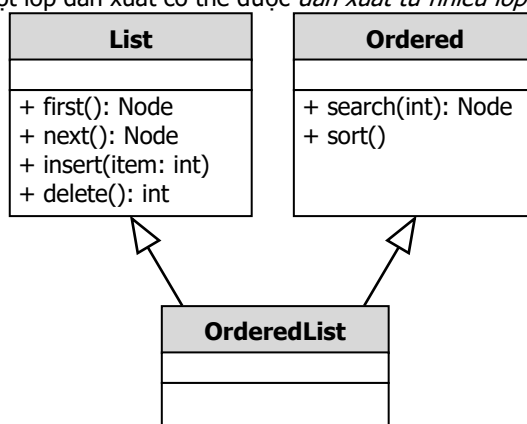
Thừa kế cho phép ta định nghĩa thêm một lớp mới, gọi là *lớp dẫn xuất* (derived class, hoặc lớp con - subclass) trên cơ sở của lớp đã tồn tại, gọi là *lớp cơ sở* (base class, hoặc lớp cha - superclass).

Lớp dẫn xuất sẽ:

- Thừa kế một số thành viên (thuộc tính, phương thức) của lớp cơ sở.
- Được phép mở rộng lớp cơ sở bằng cách thêm vào các thành viên mới.
- Được phép định nghĩa lại (overridden) những phương thức mà lớp cơ sở chưa làm tốt hoặc không còn phù hợp.

Có ba cách thực hiện thừa kế trong chương trình, thường dùng phối hợp nhau:

- Thừa kế đơn (simple inheritance): một lớp là lớp cơ sở của một hoặc nhiều lớp dẫn xuất khác.
- Thừa kế nhiều mức (multiple level inheritance): khi một lớp dẫn xuất lại trở thành lớp cơ sở của một lớp dẫn xuất khác ta có sự thừa kế nhiều mức. Tập các lớp cơ sở trực tiếp hoặc gián tiếp của một lớp gọi là các lớp tổ tiên (ancestor classes).
- Đa thừa kế (multiple inheritance): một lớp dẫn xuất có thể được *dẫn xuất từ nhiều lớp cơ sở*, gọi là đa thừa kế.



Để xác định mối quan hệ thừa kế, ta chú ý kiểm tra:

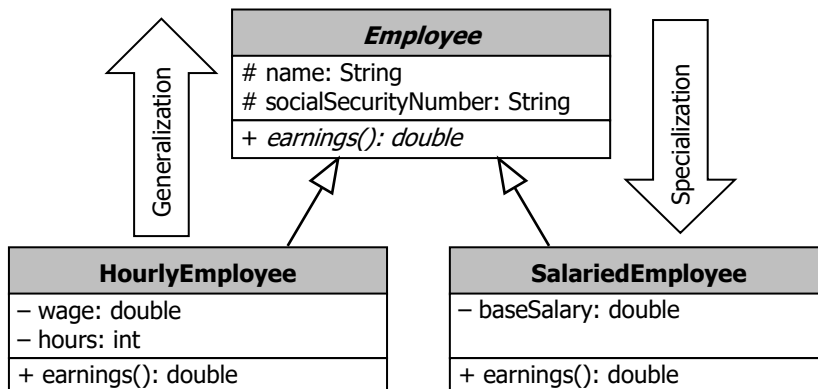
- Quan hệ *liên kết* giữa hai lớp là một quan hệ HAS-A (có một...), nghĩa là lớp này có tham chiếu đến lớp khác khi xây dựng lớp. Lớp này chứa thể hiện của lớp kia như một thành viên trong lớp.
- Quan hệ *thừa kế* giữa hai lớp là một quan hệ IS-A (là một..., là một loại của...), nghĩa là lớp này thừa kế lớp khác khi xây dựng lớp.

Trong một lớp có thể xuất hiện cả hai loại quan hệ này.

Thừa kế có một số lợi ích quan trọng cho người phát triển phần mềm:

- Trừu tượng hóa dữ liệu (data abstraction): các tính chất và hành vi chung có thể xử lý trong lớp cơ sở. Sau đó cụ thể hóa thành một hệ phân cấp các lớp dẫn xuất. Điều này đưa đến việc dễ quản lý các đối tượng phức tạp có quan hệ với nhau.
- Khả năng dùng lại (reusability): Các lớp đã định nghĩa và đã kiểm tra có thể được dùng lại và phối hợp với nhau để thực hiện tác vụ mới. Khi dùng lại, không cần biết đến cài đặt cụ thể trong lớp cơ sở, chỉ cần biết giao diện của lớp.

Khái quát hóa (generalization) và chuyên biệt hóa (specialization) là hai cách nhìn ngược nhau của sự phân cấp lớp (hierarchy inheritance):



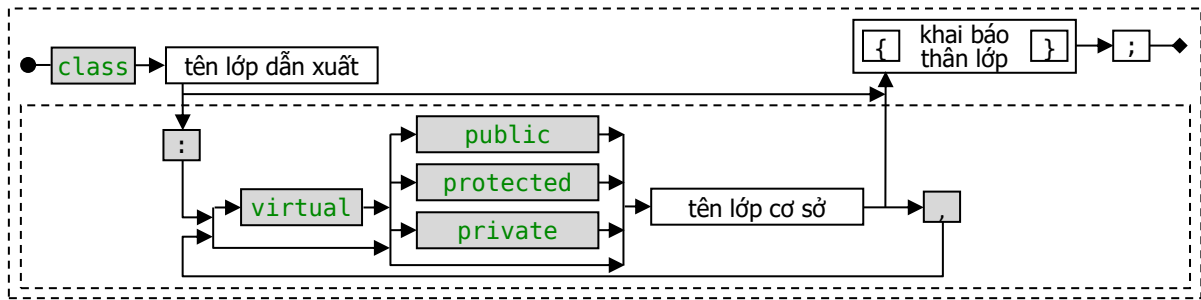
- Khái quát hóa cho phép từ tập các lớp con có những đặc điểm chung khái quát lên thành lớp cha chung mang tính khái niệm, mô tả các thuộc tính và hành vi chung cho các lớp con. Lớp cha chung này giúp quản lý các lớp con đơn giản và hiệu quả, nhất là được hỗ trợ bởi khả năng đa hình, sẽ được đề cập trong chương sau.

- Chuyên biệt hóa cho phép thu thập thêm các đặc trưng cụ thể của tập các đối tượng mới. Từ lớp cha đang tồn tại, các đặc tính này giúp mở rộng thành lớp mới, lớp con.

C++11 cho phép đánh dấu một lớp là `final` nhằm ngăn cản không cho dẫn xuất các lớp khác từ lớp đó.

II. Định nghĩa lớp dẫn xuất

Trong định nghĩa lớp dẫn xuất phải liệt kê tên của các lớp cơ sở trực tiếp của nó. Cú pháp khai báo lớp dẫn xuất như sau:



Chỉ định kiểu truy xuất (access-specifier) bao gồm: `public`, `protected` và `private`. Nếu bỏ qua chỉ định kiểu truy xuất, thì ngầm hiểu là dẫn xuất kiểu `private`. Ví dụ: Lớp D đa thừa kế, dẫn xuất kiểu `public` từ lớp B1 và dẫn xuất kiểu `private` từ lớp B2, ta viết: `class D : public B1, B2`

1. Thừa kế thuộc tính

Lớp dẫn xuất có các thuộc tính sau:

- Các thuộc tính thừa kế từ lớp cơ sở.
- Các thuộc tính bổ sung, mới được khai báo trong lớp dẫn xuất.

Khi xây dựng các thuộc tính cho lớp dẫn xuất, cho phép đặt trùng tên thuộc tính (cùng kiểu hoặc khác kiểu) của lớp cơ sở. Trong phạm vi lớp dẫn xuất, mọi truy cập đến thuộc tính trùng tên này sẽ dẫn đến truy xuất thuộc tính của lớp dẫn xuất, nếu muốn truy cập đến thuộc tính cùng tên trong lớp cơ sở phải dùng toán tử phân định phạm vi.

2. Thừa kế phương thức

Lớp dẫn xuất có các phương thức sau:

- Các phương thức thừa kế từ lớp cơ sở, *ngoại trừ* các phương thức đặc biệt: constructor, destructor, toán tử gán.
- Đa số các toán tử nạp chồng là hàm thành viên được thừa kế. Các toán tử định nghĩa thêm và các phương thức `friend` không được thừa kế.
- Các phương thức mới được xây dựng trong lớp dẫn xuất. Các phương thức này có thể truy cập đến các thuộc tính (thừa kế hoặc bổ sung) của lớp dẫn xuất, đồng thời cũng sử dụng (triệu gọi - invoke) các phương thức (thừa kế hoặc bổ sung) của lớp dẫn xuất. Các phương thức này cũng có thể *gọi phương thức lớp cơ sở* nhờ toán tử phân định phạm vi.
- Có thể cài đặt phương thức lớp dẫn xuất *trùng tên* với phương thức lớp cơ sở. Điều này gọi là định nghĩa lại (redefined), từ chối thừa kế (overridden) hay che hàm thành viên của lớp cơ sở. Đây là cơ sở cho việc cài đặt tính đa hình. Định nghĩa lại khác cơ bản với nạp chồng hàm (function overloading):
 - + Định nghĩa lại chỉ được xét khi nói đến thừa kế, kết quả là hai hàm thành viên giống nhau về tên, danh sách đối số và trị trả về; nhưng khác nhau về vị trí, nằm trong hai lớp có quan hệ thừa kế.
 - + Nạp chồng hàm diễn ra trong một lớp. Hai hàm thành viên giống nhau về tên; khác nhau về danh sách đối số.

Một số chú ý khi định nghĩa lại một phương thức:

- Định nghĩa lại một phương thức nạp chồng (overloaded): khi định nghĩa lại một phương thức nạp chồng, trình biên dịch sẽ ẩn (hiding) các phương thức nạp chồng còn lại, nghĩa là lớp dẫn xuất sẽ không gọi được các phương thức nạp chồng còn lại. Để tránh tình huống này, nên định nghĩa lại *tất cả* các phương thức nạp chồng.
- Định nghĩa lại phương thức `private` hoặc `protected`: các phương thức này thường là phương thức công cụ, được gọi bởi các phương thức `public` của lớp cơ sở. Khi định nghĩa lại chúng trong lớp dẫn xuất, phương thức `public` thừa kế được trong lớp dẫn xuất sẽ gọi các phiên bản mới này.
- Định nghĩa lại phương thức `static`: do phương thức `static` là phương thức cấp lớp, việc định nghĩa lại chúng trong lớp dẫn xuất thực chất là tạo ra một phương thức `static` mới cho lớp dẫn xuất.

C++11 cho phép đánh dấu phương thức `final`, các phương thức này không cho phép định nghĩa lại.

Lớp cơ sở:

```
class Student
{
public:
    // fresh: mới, sophomore: năm hai, junior: năm ba, senior: năm cuối, graduation: tốt nghiệp
    enum Year { fresh, soph, junior, senior, grad };
    Student( const string &, int, double, Year );
    virtual string toString() const;
protected:
    string name;
    int studentID;
    double gpa;           // điểm trung bình (grade point average)
    Year year;
};

Student::Student( const string & nm, int id, double g, Year y )
    : name( nm ), studentID( id ), gpa( g ), year( y )
{ }

string Student::toString() const
{
    stringstream ss;
    ss << "\nStudent: " << name << ", " << studentID
```



```

    << " , " << year << " , " << gpa << endl;
    return ss.str();
}

```

Lớp dẫn xuất:

```

class GradStudent : public Student
{
public:
    enum Support { ta, ra, fellowship, other };
    GradStudent( const string &, int, double,
        Year, Support, const string &, const string & );
    string toString() const;
protected:
    Support s;          toString() được phủ quyết
    string dept;        thừa kế (overridden) hoặc định
                       nghĩa lại (redefined)
    string thesis;
};

GradStudent::GradStudent( const string & nm, int id, double g,
    Year x, Support t, const string & d, const string & th )
    : Student( nm, id, g, x ), s( t ), dept( d ), thesis( th )
{ }
    // gọi constructor của lớp cơ sở

string GradStudent::toString() const
{
    stringstream ss;
    ss << Student::toString() // gọi toString của lớp cơ sở
    << dept << " , thesis: \"\" << thesis << \"\" << endl;
    return ss.str();
}

```

Sử dụng 2 lớp:

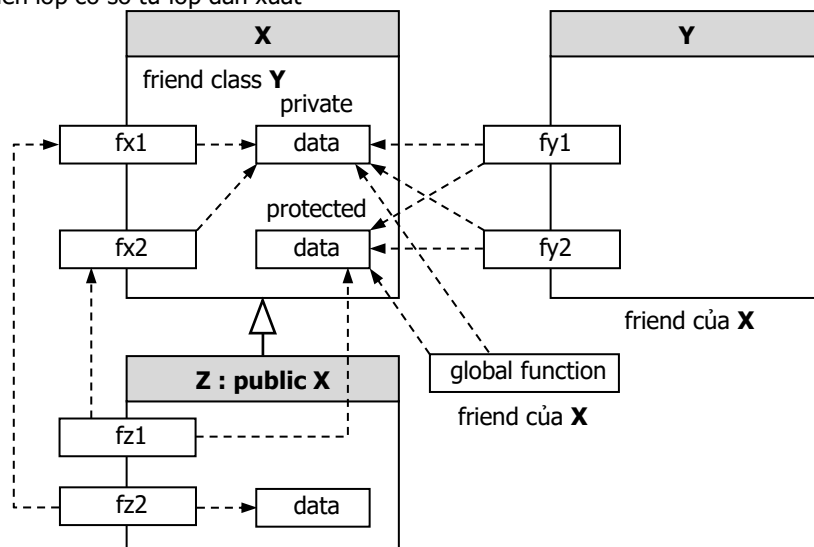
```

int main()
{
    Student s( "Bill Gates", 101, 6.3, Student::junior );
    GradStudent g( "James Gosling", 201, 9.8, Student::grad, GradStudent::other,
        "Computer Science", "Java language" );
    cout << s.toString();
    cout << g.toString();
    return 0;
}

```

III. Điều khiển truy xuất trong thừa kế

1. Truy xuất các thành viên lớp cơ sở từ lớp dẫn xuất



Mỗi tên không liên nét là các truy xuất cho phép thực hiện giữa các lớp

Mặc dù lớp dẫn xuất thừa kế tất cả các thành viên của lớp cơ sở, nhưng lớp dẫn xuất không thể truy nhập đến tất cả những thành viên này:

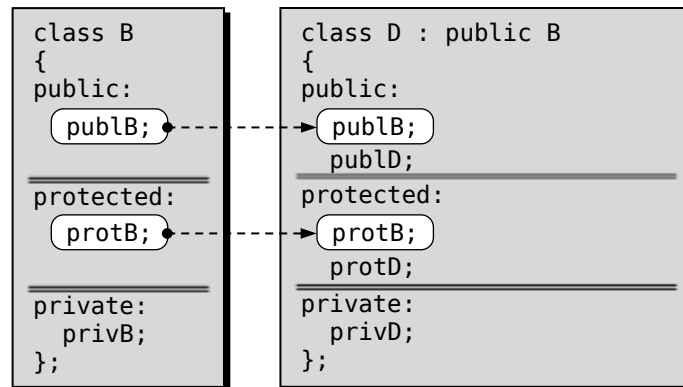
- Thành viên private của lớp cơ sở chỉ được truy xuất bởi các phương thức thành viên chung lớp cơ sở và bởi các phương thức friend, *không truy xuất trực tiếp* được từ lớp dẫn xuất. Nếu không, thừa kế sẽ phá vỡ tính đóng gói (encapsulation); bởi vì chỉ cần thừa kế một lớp, người dùng có thể truy xuất đến các chi tiết thực thi lớp đó.
- Thành viên protected của lớp cơ sở có thể được truy xuất bởi phương thức thành viên chung lớp cơ sở, bởi các phương thức friend và *từ các lớp dẫn xuất*. Như vậy thành viên protected là một mở rộng của thành viên private, nghĩa chúng là không

truy xuất được từ bên ngoài lớp, nhưng *có thể truy xuất bởi các lớp dẫn xuất*.

2. Các kiểu dẫn xuất

Kết quả dẫn xuất được điều khiển bởi các chỉ định truy xuất, là các từ khóa mô tả kiểu dẫn xuất:

a) Dẫn xuất kiểu public



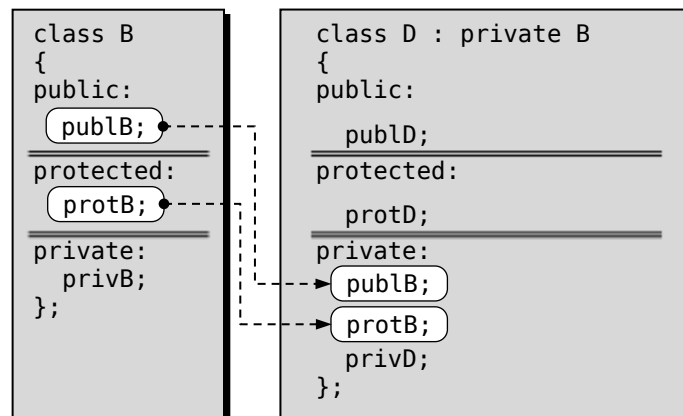
Dẫn xuất kiểu public thường được sử dụng, thể hiện một quan hệ kiểu con, còn gọi là quan hệ "là một" (IS-A) / "là một loại của" (is-a-kind-of), hoặc còn gọi là sự *thừa kế giao diện* (interface inheritance).

```
class D : public B { ... };
```

D là một loại của B, D là một kiểu con của B. Một đối tượng D có thể được dùng như một đối tượng B.

Ngoài thành viên private, các thành viên còn lại của lớp cơ sở được thừa kế trong lớp dẫn xuất với phạm vi truy xuất tương ứng trong lớp cơ sở.

b) Dẫn xuất kiểu private



Dẫn xuất kiểu private sử dụng khi ta nhận thấy một số kiểu dữ liệu lớp dẫn xuất đã được định nghĩa trong lớp cơ sở. Lớp cơ sở chỉ đơn giản cung cấp phần bổ sung cho lớp dẫn xuất. Nói cách khác, lớp dẫn xuất chứa MỘT thực thể lớp cơ sở. Điều này làm cho quan hệ giữa lớp dẫn xuất và lớp cơ sở giống như quan hệ kết hợp (composition).

Khi thừa kế kiểu private ta tạo ra một lớp dẫn xuất chứa tất cả dữ liệu và phương thức của lớp cơ sở, nhưng những phương thức này đều được ẩn giấu, trở thành một phần cài đặt bên trong của lớp dẫn xuất. Người dùng lớp dẫn xuất không truy cập được đến các phương thức bên trong này.

```
class D: private B { ... };
```

hoặc khai báo mặc định:

```
class D: B { ... };
```

Khi dẫn xuất private, tất cả các thành viên public của lớp cơ sở trở thành private trong lớp dẫn xuất. Nếu muốn sử dụng một thành viên nào đó, phải gọi tên thành viên đó (không tham số và không trả về) trong phần public của lớp dẫn xuất:

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

class Pet
{
public:
    Pet( const string & s ) : name( s ) {}
    void eat() const { cout << "Pet::eat" << endl; }
    void speak() const { cout << "Pet::speak" << endl; }
    void sleep() const { cout << "Pet::sleep" << endl; }
protected:
    string name;
};

class GoldFish : private Pet
{
    public:
        void eat() const { Pet::eat(); }
        void speak() const { Pet::speak(); }
        void sleep() const { Pet::sleep(); }
};
```

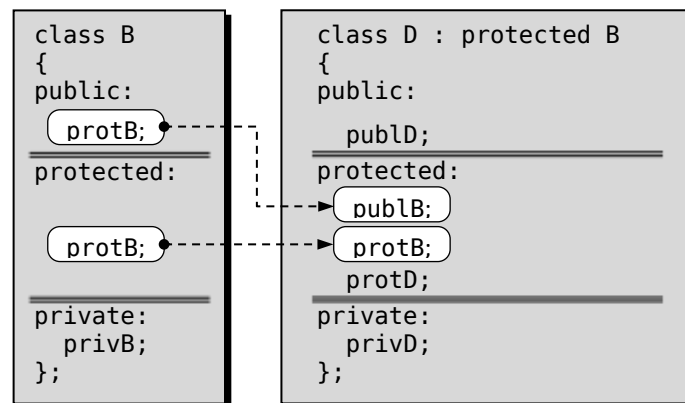
```

public:
    using Pet::name;          // "công khai" (public access) một dữ liệu thành viên thừa kế
private
    GoldFish( const string & s ) : Pet( s ) { }
    using Pet::eat;           // "công khai" một phương thức thành viên thừa kế private
    void sleep() const        // gọi phương thức sleep lớp Pet
    { Pet::sleep(); cout << name << endl; }
};

int main()
{
    GoldFish bob( "Nemo" );
    cout << bob.name << endl;
    bob.eat();                // do phương thức eat đã "công khai" nên gọi được
    return 0;
}

```

c) Dẫn xuất kiểu protected

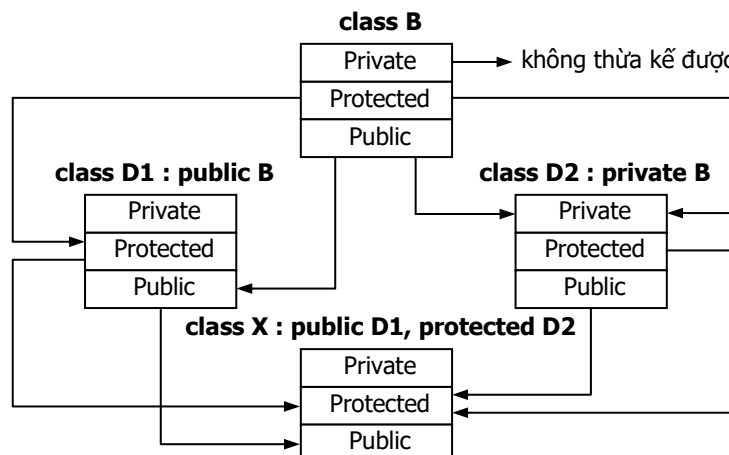


```
class D: protected B { ... };
```

Dẫn xuất kiểu protected đưa các thành viên lớp cha thành protected ở lớp con.

Dẫn xuất kiểu protected và private không được xem như mối quan hệ IS-A mà giống quan hệ HAS-A hơn, nên để tránh nhầm lẫn, người ta thường tránh các dẫn xuất kiểu này.

Hình sau mô tả các thành viên lớp cơ sở (với phạm vi truy xuất khác nhau) được thừa kế trong lớp dẫn xuất, sử dụng các kiểu dẫn xuất khác nhau:



Kết quả thừa kế với các kiểu dẫn xuất khác nhau

IV. Các phương thức của lớp dẫn xuất

1. Constructor của lớp dẫn xuất

Constructor của đối tượng được tự động gọi theo thứ tự các lớp cơ sở (trong khai báo đa thừa kế), rồi đến các lớp dẫn xuất, cuối cùng là lớp dẫn xuất sinh đối tượng (gọi là lớp most-derived). Destructor được tự động gọi theo thứ tự ngược lại, từ lớp dẫn xuất sinh đối tượng, ngược lên các lớp dẫn xuất trên, rồi đến lớp cơ sở.

Lớp dẫn xuất không thừa kế các constructor, destructor và toán tử gán của lớp cơ sở nên ta phải tự xây dựng lấy. Hơn nữa, các constructor của lớp dẫn xuất phải chứa thông tin khởi tạo, làm đối số để chuyển cho các constructor của lớp cơ sở trong danh sách khởi tạo.

Trong lớp dẫn xuất có thể có ba nhóm thuộc tính sau:

- Các thuộc tính là đối tượng thuộc lớp khác (subobject): khởi gán thông qua danh sách khởi tạo, dùng *tên của subobject*.
- Các thuộc tính kế thừa từ lớp cơ sở: khởi gán thông qua danh sách khởi tạo, dùng *tên của lớp cơ sở* tương ứng.

Các trị cần truyền cho constructor của lớp cơ sở hoặc lớp thành phần được chuyển từ danh sách trị-đối số của constructor lớp

dẫn xuất.

- Các thuộc tính riêng của lớp dẫn xuất: các thuộc tính này có thể truy nhập trực tiếp bên trong lớp dẫn xuất. Vì vậy được khởi gán bằng các phát biểu trực tiếp trong thân constructor hoặc trong danh sách khởi tạo.

```
<D_class> :: <D_class>( [argument_values list] ) :  
<B_class_1>( <argument_values list 1> ),  
[<B_class_k>( <argument_values list k> )],  
...  
[<subobject_1>( <argument_values list 1> )],  
[<subobject_n>( <argument_values list n> )],  
{  
    // khởi gán các thuộc tính mới của lớp dẫn xuất  
}
```

← danh sách khởi tạo
(initialization list)

Nếu lớp cơ sở hoặc lớp thành phần không cần khởi tạo với constructor có đối số thì chúng cũng không cần có mặt trong danh sách khởi tạo, nhưng vẫn được khởi tạo bằng constructor mặc định. Ví dụ:

```
class Base  
{  
public:  
    int pubB;  
    Base( int b = 0 ) : pubB( b ) {}  
};  
  
class Sub  
{  
public:  
    double pubS;  
    Sub( double s = 0.0 ) { pubS = s; }  
};  
  
class Derived : public Base  
{  
    int* a;  
    Sub objS;  
public:  
    Derived( int, double, int );  
};  
  
Derived::Derived( int b1, double s1, int n ) : Base( b1 ), objS( s1 )  
{  
    a = new int[n];  
}
```

gọi constructor
lớp cơ sở tên của
subobject

```
/* hoặc:  
Derived::Derived( int b1, double s1, int n ) : Base( b1 ), objS( s1 ), a( new int[n] )  
{ }  
*/
```

2. Hàm hủy của lớp dẫn xuất

Khi hủy đối tượng thuộc lớp dẫn xuất, các subobject và các đối tượng thừa kế được từ lớp cơ sở chứa trong nó sẽ tự giải phóng với destructor tương ứng. Vì vậy khi xây dựng destructor của lớp dẫn xuất, chỉ cần quan tâm đến các thuộc tính khai báo thêm trong lớp dẫn xuất, giải phóng vùng nhớ các thuộc tính này chiếm nếu có.

3. Toán tử gán của lớp dẫn xuất

Khi lớp dẫn xuất có thành viên dữ liệu động, kể cả thành viên được thừa kế từ lớp cơ sở, thì không được dùng toán tử gán mặc định mà phải xây dựng toán tử gán tường minh cho nó. Ta cần làm như sau:

- Xây dựng toán tử gán cho các lớp cơ sở và các lớp thành phần nếu có.
- Trong định nghĩa toán tử gán của lớp dẫn xuất, dùng toán tử gán của lớp cơ sở để thực hiện phép gán trên các đối tượng này.
- Trong định nghĩa toán tử gán của lớp dẫn xuất, còn có các phép gán các subobject (dùng toán tử gán của các lớp thành phần), và phép gán các thuộc tính bổ sung của lớp dẫn xuất. Ví dụ:

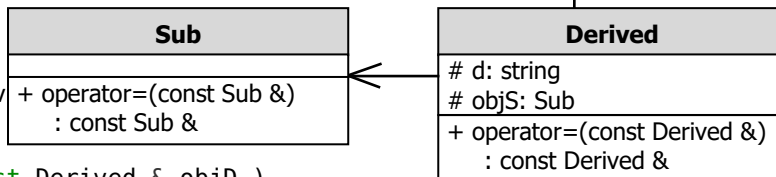
```
#include <string>  
using std::string;  
  
class Base  
{  
public:  
    const Base & operator=( const Base & )  
    { /* code */ }  
};  
  
class Sub  
{
```

```

public:
    const Sub & operator=( const Sub & )
    { /* code */ }
};

class Derived : public Base
{
    string d;
    Sub objS;
public:
    const Derived & operator=( const Derived & objD )
    {
        d = objD.d; // các phát biểu gán các thuộc tính riêng của lớp dẫn xuất
        objS = objD.objS; // các phát biểu gán các subobject
        Base::operator=(objD); // gán các thuộc tính mà Derived thừa kế từ Base
        return *this;
    }
}

```



4. Copy constructor của lớp dẫn xuất

Khi lớp dẫn xuất có thành viên dữ liệu động, kể cả thành viên được thừa kế từ lớp cơ sở, thì không được dùng copy constructor mặc định mà phải xây dựng copy constructor tường minh cho nó.

Ta cần thực hiện một trong các cách sau:

- Sử dụng copy constructor của lớp cơ sở và toán tử gán của lớp thành phần để định nghĩa copy constructor của lớp dẫn xuất.

```

Derived::Derived( const Derived & objD )
: Base( objD ), d( objD.d ), objS( objD.objS )
{
    // gọi copy constructor của lớp cơ sở
}

```

- Sử dụng toán tử gán của lớp cơ sở và của lớp thành phần để định nghĩa copy constructor của lớp dẫn xuất.

```

Derived::Derived( const Derived & objD )
: d( objD.d ), objS( objD.objS )
{
    *( Base* )( this ) = *( Base* )( &objD ); // gọi toán tử gán lớp cơ sở
}

```

- Sử dụng toán tử gán của lớp dẫn xuất, toán tử này phải được xây dựng trước như phần trên trình bày. Sau đó copy constructor của lớp dẫn xuất viết theo mẫu sau:

```

Derived::Derived( const Derived & objD )
{
    *this = objD;
}

```

Chú ý rằng cách này không thực hiện được nếu toán tử gán của lớp dẫn xuất có hủy vùng nhớ cấp phát động của đối tượng bên trái.

V. Đa thừa kế và lớp cơ sở ảo

1. Đa thừa kế (Multiple Inheritance)

Thừa kế từ nhiều lớp cơ sở gọi là đa thừa kế. Lớp dẫn xuất sẽ thừa kế tất cả thuộc tính và phương thức (thừa kế được) từ các lớp cơ sở. Quan hệ IS-A vẫn là quan hệ của lớp dẫn xuất với các lớp cơ sở.

Khi định nghĩa một lớp dẫn xuất từ đa thừa kế (Multiple-Derived class), chỉ định truy xuất (private, protected, public) phải được định nghĩa đầy đủ, tách biệt cho từng lớp cơ sở. Nếu chỉ định truy xuất không có thì mặc định là thừa kế private.

```

class MotorHome : public Car, Home // dẫn xuất Home là private, không phải public
{ . . . };

```

Đa thừa kế có khuynh hướng thu hẹp cây phân cấp, là một vấn đề phức tạp trong C++. Vài tình huống đa thừa kế như sau:

- Muốn lớp mới bao gồm các đặc điểm của hai lớp cơ sở khác nhau, thường có thể tránh bằng thiết kế tốt hơn.
- Dùng dẫn xuất public để tạo quan hệ IS-A và dẫn xuất private (hoặc protected) để tạo quan hệ HAS-A, xây dựng các lớp từ các đơn thể (component-based class). Như trình bày trong phần trên, nên tránh dẫn xuất private.
- Tạo các lớp *mix-in*. Nhiều lớp trên cây thừa kế, ngoài dẫn xuất chính thức trong cây thừa kế, còn dẫn xuất từ các lớp mix-in để thêm nhanh các phương thức cần thiết. Java có khái niệm *interface* tương tự các lớp mix-in này.

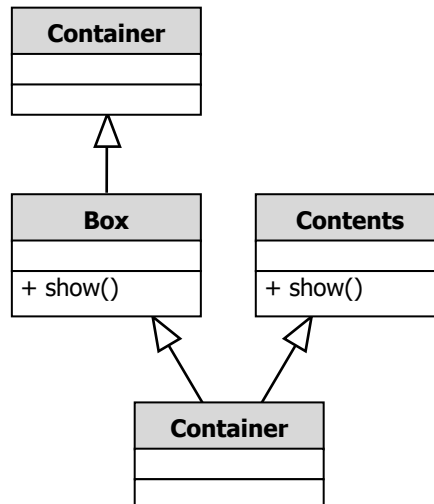
2. Những vấn đề của đa thừa kế

a) Trùng tên phương thức

Vấn đề trùng tên phương thức nảy sinh khi các lớp cơ sở có các phương thức có tên giống nhau.

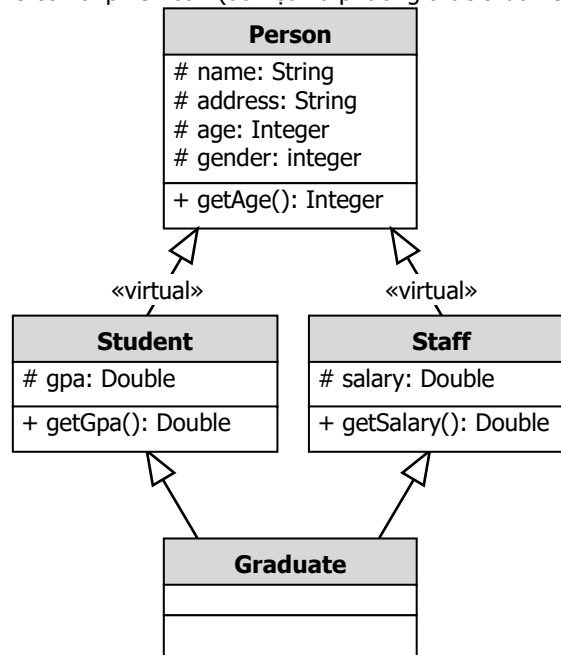
Để giải quyết, ta cần dùng toán tử phân định phạm vi :: chỉ cụ thể tên lớp cơ sở của phương thức được gọi nhằm tránh tình trạng mơ hồ (ambiguous) khi trình biên dịch chọn phương thức để gọi.

```
myPackage.Box::show(); // gọi phương thức show() thừa kế từ lớp cha Box
myPackage.Contents::show(); // gọi phương thức show() thừa kế từ lớp cha Contents
```



b) Lớp cơ sở ảo (virtual base class)

Xét trường hợp thừa kế mô tả trong sơ đồ của ví dụ bên dưới, gọi là thừa kế "hình thoi" (diamond inheritance) hoặc thừa kế nhiều đường (multi path inheritance): lớp Graduate đa thừa kế hai lớp Student và Staff, hai lớp này lại cùng thừa kế lớp Person; như vậy trong lớp Graduate có hai phiên bản (dữ liệu và phương thức thừa kế) của lớp cơ sở gián tiếp Person.



Vấn đề nảy sinh:

```
Graduate* g = new Graduate();
int s;
s = g->getAge(); // không được, có hai phiên bản getAge()
s = g->Person::getAge(); // không được, có hai phiên bản Person
// cách sau được, nhưng cho kết quả là các trị truy xuất tại vị trí khác nhau trong bộ nhớ (không nhất quán)
s = g->Staff::age(); // dùng dữ liệu lớp Person mà Staff thừa kế được
s = g->Student::age(); // dùng dữ liệu lớp Person mà Student thừa kế được
```

Giải pháp cho vấn đề này là dùng lớp cơ sở ảo (virtual base class): một lớp cơ sở trực tiếp được khai báo là `virtual` khi các lớp dẫn xuất được định nghĩa, từ khóa `virtual` được dùng ngay trước tên lớp cơ sở (virtual và public có thể đổi thứ tự). Lớp Person trở thành lớp cơ sở ảo của lớp Student và lớp Staff.

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

class Person
{
protected:
    string name;
    string address;
    int age;
    bool gender;
```

```

public:
    Person( const string & s1 = "", const string & s2 = "", int a = 0, bool s = true )
        : name( s1 ), address( s2 ), age( a ), gender( s )
    {}
    int getAge() const { return age; }
};

class Student : virtual public Person
{
protected:
    double gpa;
public:
    Student( double d = 0.0 ) : gpa( d ) {}
    double getGPA() const { return gpa; }
};

class Staff : virtual public Person
{
protected:
    double salary;
public:
    Staff( double d = 0.0 ) : salary( d ) {}
    double getSalary() const { return salary; }
};

class Graduate : public Student, public Staff
{
public:
    Graduate( const string & s1 = "", const string & s2 = "", int a = 0,
              int s = 1, double g = 0.0, double sal = 0.0 )
        : Person( s1, s2, a, s ), Student( g ), Staff( sal )
    {}
};

int main ()
{
    Graduate p( "Arnold", "Hollywood", 30 );
    cout << p.getAge() << endl;
    return 0;
}

```

Cần chú ý:

- Một lớp cơ sở ảo vẫn duy trì cho các lớp dẫn xuất từ nó, các lớp dẫn xuất từ Student vẫn xem lớp Person như một lớp cơ sở ảo.

Không thể thay đổi khai báo của một lớp cơ sở *gián tiếp* thành virtual.

- Khi tạo đối tượng từ một lớp dẫn xuất đa thừa kế, constructor của các lớp cơ sở sẽ được gọi trước tiên. Trong trường hợp có lớp cơ sở ảo, constructor của *lớp cơ sở ảo gần nhất* trong cây phân cấp sẽ được thực thi *trước* constructor của các lớp cơ sở không ảo.

```

Graduate( const string & s1 = "", const string & s2 = "", int a = 0, bool s = true )
    : Person( s1, s2, a, s )
{}

```


Đa hình

Polymorphism

1. Tương thích giữa lớp dẫn xuất và lớp cơ sở

Vì đối tượng thuộc lớp dẫn xuất chứa bên trong nó *giao diện* của lớp cơ sở, nên một đối tượng lớp dẫn xuất có thể được dùng như một đối tượng có kiểu của chính nó hoặc như một đối tượng *có kiểu là lớp cơ sở* dẫn xuất ra nó. Hơn nữa, có thể truy cập đến đối tượng thuộc lớp dẫn xuất thông qua con trỏ hoặc tham chiếu đến lớp cơ sở. Ta gọi đây là sự tương thích giữa lớp dẫn xuất và lớp cơ sở. Điều này cho phép quản lý các đối tượng của các lớp dẫn xuất khác nhau như là các đối tượng của một lớp cơ sở chung.

1. Tương thích giữa đối tượng thuộc lớp dẫn xuất và đối tượng thuộc lớp cơ sở

Có thể gán một đối tượng thuộc lớp dẫn xuất cho một đối tượng thuộc lớp cơ sở. Đối tượng kết quả của phép gán sẽ bị xén bớt (sliced), nghĩa là chỉ chứa các thành viên của lớp cơ sở, không chứa các thành viên do lớp dẫn xuất bổ sung.

Trong phép gán trên, trình biên dịch đã dùng phép chuyển kiểu không tường minh (implicit type conversion) để chuyển đối tượng từ kiểu lớp dẫn xuất lên kiểu lớp cơ sở, gọi là chuyển kiểu lên (upcasting).

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

class Instrument
{
protected:
    string artist;
public:
    enum note { middleC, Csharp, Cflat };
    Instrument( const string & s = "" ) : artist( s ) {}
    void play( note ) const
    { cout << "Instrument::play " << artist << endl; }
};

// nhạc cụ thổi Wind "là một loại" Instrument
class Wind : public Instrument
{
public:
    Wind( const string & s = "" ) : Instrument( s ) {}
    void play( note ) const // cài đặt lại phương thức play
    { cout << "Wind::play " << artist << endl; }
};

int main()
{
    Wind saxophone( "Kenny G" );
    Instrument clarinet;
    clarinet = saxophone; // upcasting Wind thành Instrument
    clarinet.play( Instrument::Csharp ); // sẽ gọi Instrument::play
    return 0;
}
```

2. Tương thích giữa con trỏ lớp dẫn xuất và con trỏ lớp cơ sở

Hiện tượng xén bớt đối tượng khi chuyển kiểu lên một đối tượng lớp dẫn xuất như trên thường gây lỗi. Thay vào đó, khi chuyển kiểu lên, ta thường chuyển *con trỏ* kiểu lớp dẫn xuất thành con trỏ kiểu lớp cơ sở để sử dụng khả năng đa hình.

Một con trỏ kiểu lớp cơ sở có thể dùng chỉ đến đối tượng của lớp dẫn xuất, nghĩa là:

- Con trỏ của lớp cơ sở có thể dùng để chứa địa chỉ các đối tượng của lớp dẫn xuất.
- Có thể gán một con trỏ lớp dẫn xuất vào một con trỏ lớp cơ sở.

Khi đó, từ con trỏ lớp cơ sở chỉ có thể truy xuất giao diện lớp cơ sở được thừa kế tại lớp dẫn xuất, không truy xuất được các thành viên được bổ sung tại lớp dẫn xuất.

```
int main()
{
    Instrument* instr[2]; // mảng các con trỏ Instrument*
    Wind trumpet( "Louis Amstrong" );
    instr[0] = new Wind( "Kenny G" ); // chứa một con trỏ Wind*
    instr[1] = &trumpet; // chứa địa chỉ một đối tượng Wind
    for ( int i = 0; i < 2; ++i )
        instr[i]->play( Instrument::Cflat ); // gọi Instrument::play
    return 0;
}
```

Từ ví dụ trên ta cũng thấy có thể quản lý *nhiều đối tượng thuộc các lớp khác nhau* trong một *mảng chứa các con trỏ kiểu lớp cơ sở chung* của chúng.

3. Tương thích giữa tham chiếu lớp dẫn xuất và tham chiếu lớp cơ sở

Giống như sự tương thích giữa con trỏ kiểu lớp dẫn xuất và con trỏ kiểu lớp cơ sở, có thể dùng tham chiếu kiểu lớp dẫn xuất tại vị trí tham chiếu kiểu lớp cơ sở.

```
void tune( const Instrument & obj )
{
    // ...
    obj.play( Instrument::middleC );           // gọi Instrument::play
}

int main()
{
    Wind flute( "Ron Korb" );
    tune( flute );                             // OK. Vì lớp Wind thừa kế lớp Instrument
    return 0;
}
```

II. Kết nối tĩnh (static binding) và kết nối động (dynamic binding)

1. Kết nối tĩnh

Kết nối (binding) thể hiện mối quan hệ kết nối giữa phương thức được gọi với phương thức tương ứng được thực hiện, nghĩa là giải quyết việc phương thức thuộc lớp nào sẽ thực hiện.

Kết nối tĩnh (static binding), còn gọi là kết nối sớm (early binding) là kiểu kết nối được *xác định trước khi chương trình chạy*, nghĩa là trong quá trình biên dịch và liên kết. Trình biên dịch sẽ xác định cụ thể phương thức nào được sử dụng tương ứng với kiểu đối tượng nào nhận thông điệp trước khi chạy chương trình. Việc xác định này dựa theo các quy tắc sau:

- Nếu phương thức được gọi kèm theo *tên của lớp* nào thì phương thức tương ứng của lớp đó sẽ được thực hiện. Đây là cách gọi các phương thức static hoặc gọi phương thức public với tên lớp tường minh (dùng toán tử ::).
- Nếu phương thức được gọi kèm theo *đối tượng thuộc lớp* nào thì phương thức của lớp đó sẽ được thực hiện. Đây là cách gọi phương thức thông qua đối tượng của lớp.
- Nếu phương thức được gọi kèm theo con trỏ thuộc lớp nào thì phương thức của lớp đó sẽ được thực hiện. Chú ý phương thức áp dụng được xác định từ *kiểu của con trỏ*, không phải từ kiểu của đối tượng được con trỏ chỉ đến.
- Nếu phương thức được gọi không thấy trong lớp, trình biên dịch sẽ xác định kết nối bằng cách tìm phương thức cùng tên trong các lớp cơ sở, ưu tiên các lớp có quan hệ gần với lớp đang xét. Quá trình truy tìm này sẽ đi ngược lên dần trên cây phân cấp.

```
#include <iostream>
#include "student.h"           // chứa các khai báo lớp Student và GradStudent chương trước
int main()
{
    Student s( "Bill Gates", 100, 3.425, Student::fresh );
    GradStudent gs( "Linus Tovarld", 200, 3.2564, Student::grad,
                   GradStudent::ta, "Operating System", "Linux OS" );
    Student* ps = &s;          // con trỏ ps chỉ đến Student
    GradStudent* pgs;

    cout << gs.Student::toString(); // Student::toString (quy tắc 1), dù đối tượng gọi là GradStudent
    cout << gs.toString();          // GradStudent::toString (quy tắc 2)
    cout << ps->toString();         // Student::toString (quy tắc 3)

    ps = pgs = &gs;              // hai con trỏ ps và pgs đều chỉ đến GradStudent
    cout << pgs->toString();        // GradStudent::toString (quy tắc 3)
    cout << ps->toString();         // Student::toString !!! Do kiểu của con trỏ ps vẫn là Student
    return 0;
}
```

2. Phương thức ảo

Phương thức ảo là cần thiết do nó được xây dựng cho *lớp mang tính khái niệm*, vì vậy ta không biết được nó sẽ hoạt động như thế nào cho đến khi kiểu của đối tượng gọi nó được xác định trong thời gian chạy. Nói cách khác, ta khai báo một phương thức là ảo khi muốn nó trở thành phương thức đa hình.

Phương thức ảo giống các phương thức thông thường ở nhiều mặt, chỉ khác ở các điểm sau:

- Phải là phương thức thành viên của một lớp và có từ khóa `virtual`. Không thể là thành viên static.
- Thường được cài đặt sơ bộ (hoặc rỗng) ở lớp cơ sở rồi cài đặt cụ thể lại (overridden) ở lớp dẫn xuất.
- Có thể được sử dụng trong cả hai hình thức kết nối: kết nối tĩnh và kết nối động, tùy theo tình huống cụ thể (xem phần sau).

Định nghĩa phương thức ảo bằng một trong hai cách sau:

- Thêm từ khóa `virtual` vào đầu dòng khai báo phương thức đó bên trong định nghĩa lớp cơ sở cao nhất có chứa nó.
- hoặc thêm từ khóa `virtual` vào đầu dòng khai báo phương thức đó bên trong định nghĩa của tất cả các lớp cơ sở và lớp dẫn xuất có chứa nó. Cách này ít khi dùng, vì chỉ cần chỉ định ở lớp cơ sở cao nhất là đủ, các phiên bản mới của phương thức ảo trong lớp dẫn xuất là tự động ảo.

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.14159;
```

```

class Shape
{
public:
    Shape() : x( 0.0 ), y( 0.0 ) {}
    void setPos( double nx, double ny )
    { x = nx; y = ny; } // chung cho mọi hình

    virtual double getArea() const = 0; } ← để định nghĩa lại (redefined)
    virtual string getName() const = 0; } cho các hình khác trong các
protected:                                     lớp tương ứng
    double x, y; // chung cho mọi hình
};

class Rectangle : public Shape
{
public:
    Rectangle( double h, double w ) : height( h ), width( w ) {}
    double getArea() const { return ( height * width ); }
    string getName() const { return ( " RECTANGLE " ); }
private:
    double height, width;
};

class Circle : public Shape
{
public:
    Circle( double r ) : radius( r ) {}
    double getArea() const { return ( PI * radius * radius ); }
    string getName() const { return ( " CIRCLE " ); }
private:
    double radius;
};

class Polygon : public Shape
{
public:
    Polygon( int n, double s ) : number( n ), side( s ) {}
    double getArea() const
    { return ( side * side * number / ( 4.0 * tan( PI / number ) ) ); }
    string getName() const { return ( " POLYGON " ); }
private:
    int number; // số mặt
    double side;
};

```

Kiểu trả về của một phương thức ảo phải giống như khi khai báo chúng trong lớp cơ sở, hoặc phải là một covariant (hiệp biến). Trong lớp dẫn xuất, kiểu trả về covariant là con trỏ hoặc tham chiếu đến một kiểu lớp được dẫn xuất từ kiểu trả về của phương thức ảo lớp cơ sở. Ví dụ, phương thức ảo `clone()` lớp cha trả về kiểu `Shape*`, phương thức ảo `clone()` lớp con trả về `Circle*` (`Circle` dẫn xuất từ `Shape`); chú ý covariant không nhất thiết phải tương ứng lớp chứa phương thức ảo.

3. Kết nối động và quy tắc gọi phương thức ảo

Kết nối động (dynamic binding), còn gọi là kết nối trễ (late binding) là kiểu kết nối chưa được xác định rõ ràng trong quá trình biên dịch. Chương trình chưa xác định được phương thức thuộc lớp nào sẽ được gọi cho tới khi thực sự nhận đối tượng cụ thể. Đặc tính cơ bản của kết nối động là khả năng trì hoãn thực hiện (trễ): cho đến khi thực thi mới quyết định *đối tượng thuộc lớp nào* và *phương thức thuộc lớp nào* để gọi và tham khảo đến đối tượng đó.

Phương thức ảo được dùng trong cả hai cách: kết nối tĩnh và kết nối động. Khi thực hiện, chương trình sẽ xác định phương thức ảo nào sẽ kết nối với đối tượng nhận thông điệp. Việc lựa chọn phương thức dựa trên các nguyên tắc:

- Nếu phương thức ảo được gọi kèm theo tên của lớp nào thì phương thức của lớp đó sẽ được thực hiện. Cách gọi phương thức ảo này được thực hiện theo kiểu kết nối tĩnh.
- Nếu phương thức ảo được gọi kèm theo đối tượng thuộc lớp nào thì phương thức của lớp đó sẽ được thực hiện. Kết nối tĩnh.
- Nếu phương thức ảo được gọi kèm theo con trỏ *kiểu lớp cơ sở*. Nếu con trỏ kiểu lớp cơ sở chỉ đến đối tượng thuộc lớp con nào thì phương thức thực sự của của lớp con đó sẽ được thực hiện. Trường hợp này kết nối thực hiện theo kiểu *kết nối động*. Một lần nữa chú ý rằng phương thức áp dụng được xác định từ *kiểu thật sự của đối tượng được con trỏ chỉ đến* (kiểu động), không phải từ kiểu của trỏ quản lý chúng (kiểu khai báo).
- Nếu phương thức ảo được gọi không thấy trong lớp, trình biên dịch sẽ xác định kết nối bằng cách tìm phương thức ảo cùng tên trong các lớp cơ sở, ưu tiên các lớp có quan hệ gần với lớp đang xét.

Trong ví dụ sau ta dùng mảng kiểu con trỏ kiểu lớp cơ sở `Shape` để quản lý các con trỏ kiểu lớp dẫn xuất (`Circle`, `Polygon`, `Rectangle`). Thông qua các con trỏ của mảng này ta gọi các phương thức ảo `getName()` và `getArea()`; thay vì phương thức của lớp `Shape` được gọi, kết nối động giúp phương thức ảo cài đặt cho lớp dẫn xuất cụ thể sẽ được gọi.

```

int main()
{

```

```
// con trỏ chỉ đến các đối tượng khác nhau được quản lý bằng mảng con trỏ kiểu lớp cơ sở
Shape* shps[] = { new Circle( 2.2 ), new Polygon( 14, 1.3 ), new Rectangle( 5.2, 3.0 ) };

for( int i = 0; i < 3; ++i )
    cout << shps[i]->getName() << "area = "
          << shps[i]->getArea() << endl;
return 0;
}
```

SHAPE* phương thức ảo

Như vậy ta nhận thấy cơ chế kết nối động sẽ được áp dụng khi gọi hành vi đa hình:

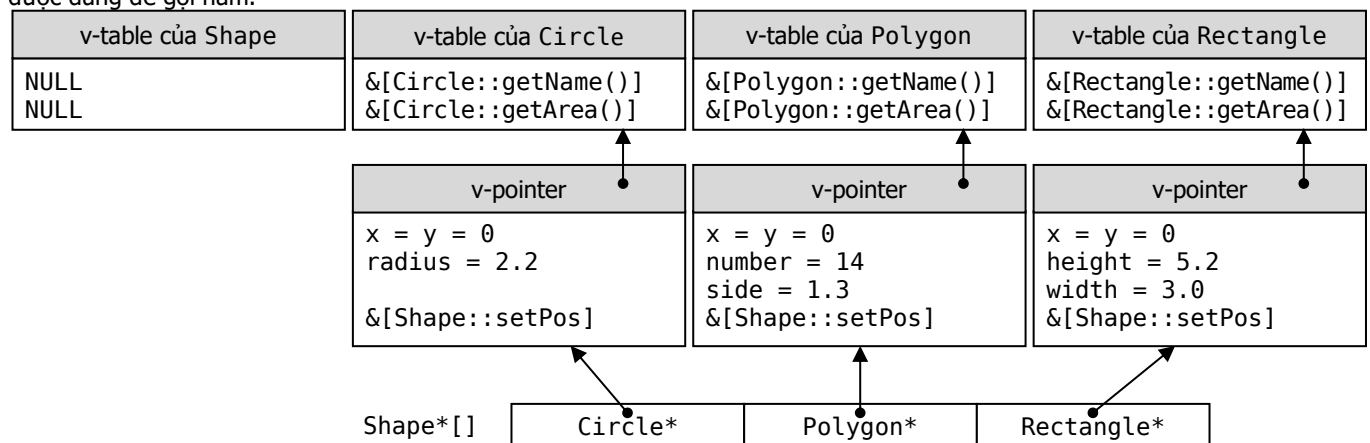
- gọi một *phương thức ảo* của một đối tượng,
- thông qua *con trỏ kiểu lớp cơ sở* đang quản lý đối tượng đó.

Trong ví dụ trên, nếu ta quản lý trực tiếp các đối tượng lớp dẫn xuất (không quản lý gián tiếp thông qua con trỏ kiểu lớp cơ sở) hoặc phương thức gọi không phải là phương thức ảo, cơ chế kết nối tĩnh được thực hiện, ta không đạt được hiệu quả đa hình. Cũng có thể gọi hành vi đa hình thông qua tham chiếu kiểu lớp cơ sở:

```
int main()
{
    const Shape& rr = Rectangle( 5.2, 3.0 );
    cout << rr.getName() << "area = " << rr.getArea() << endl;
    return 0;
}
```

4. Cơ chế gọi phương thức ảo

Trình biên dịch tạo ra một thành viên dữ liệu ẩn cho mỗi đối tượng của lớp con, gọi là con trỏ ảo vptr hay v-pointer chỉ đến một bảng phương thức ảo (v-table). Bảng phương thức ảo chứa các con trỏ hàm chỉ đến các phương thức ảo được cài đặt ở lớp con. Trình biên dịch sẽ dùng con trỏ ảo trong bảng phương thức ảo của một lớp để gọi phương thức ảo trong thời gian thực thi. Trong hình dưới, ví dụ gọi phương thức ảo `getArea()` từ đối tượng do con trỏ `Circle*` chỉ đến, con trỏ ảo v-pointer của đối tượng đó sẽ chỉ đến v-table của lớp `Circle`, từ bảng này con trỏ hàm chỉ đến phương thức `getArea()` của lớp `Circle` sẽ được dùng để gọi hàm.



Đối tượng với phương thức ảo phải duy trì một bảng v-table, vì vậy thời gian gọi phương thức sẽ tăng với phương thức ảo⁷.

5. Một số phương thức cần xây dựng ảo

C++ *không cho phép constructor ảo*, vì lúc đó chưa có đối tượng nên chưa có con trỏ ảo v-pointer. Cũng không có hàm static và hàm friend (tự do) ảo.

Phương thức ảo của một lớp có thể được khai báo là phương thức bạn (friend) của một lớp khác.

Khi xây dựng một lớp cơ sở, nhất là khi lớp cơ sở này có phương thức ảo, cần thiết định nghĩa một destructor ảo cho nó, mặc dù lớp đó không cần destructor.

Giả sử ta dùng mảng các con trỏ kiểu lớp cha để quản lý các con trỏ chỉ đến đối tượng thuộc các lớp con. Khi hủy mảng này, do kết nối tĩnh nên chỉ có destructor của lớp cha được gọi. Để các phần tử của mảng có thể hủy theo cách khác nhau (đa hình khi hủy), destructor của lớp cha (lớp cơ sở) phải luôn được khai báo như một hàm ảo.

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

class Base
{
public:
    Base()
    { cout << "Constructor of class Base\n"; }

    virtual ~Base()
    { cout << "Destructor of class Base\n"; }
};
```

⁷ Trong thực hành, khuyến khích chuyển các phương thức nghiệp vụ không static thành ảo. Các phương thức không static của Java đều là phương thức ảo.

```

class Data
{
private:
    string name;
public:
    Data( const string & n )
    {
        cout << "Constructor of class Data\n";
        name = n;
    }

    ~Data()
    {
        cout << "Destructor of class Data for "
              << "object: " << name << endl;
    }
};

class Derived : public Base
{
private:
    string data;
public:
    Derived( const string & n ) : data( n )
    { cout << "Constructor of class Derived\n"; }

    ~Derived()           // không cần destructor này
    { cout << "Destructor of class Derived\n"; }
};

int main()
{
    Base *bPtr = new Derived( "DEMO" );
    cout << "\nCall to the virtual Destructor!\n";
    delete bPtr;
    return 0;
}

```

Ví dụ trên dùng con trỏ kiểu lớp cơ sở để quản lý đối tượng lớp dẫn xuất, vì vậy destructor ảo tương ứng với lớp dẫn xuất đó sẽ được gọi khi giải phóng con trỏ quản lý. Do lớp cơ sở đã xây dựng destructor ảo, lớp dẫn xuất thừa kế destructor này nên không cần phải viết destructor cho các lớp dẫn xuất nữa, ngoại trừ lớp dẫn xuất có thêm thành phần dữ liệu động cần giải phóng khi hủy.

III. Đa hình (polymorphism)

1. Đa hình

Các lớp có liên quan với nhau được khái quát hóa thành lớp cơ sở chung, chứa ít nhất một phương thức ảo, gọi là các lớp đa hình (polymorphic⁸). Khi chưa khái quát hóa, hạn chế của kết nối tĩnh là buộc phải xử lý riêng rẽ *hành vi chung* của các đối tượng khác nhau thuộc các lớp có liên quan với nhau, làm cho code trở nên phức tạp.

Tính đa hình là khả năng xử lý các đối tượng thuộc các lớp đa hình theo một cách thức tổng quát, song khi xử lý đến từng đối tượng thì cách thức tổng quát đó trở nên cụ thể cho từng lớp cụ thể. Khả năng này có được bằng cách *thực thi thông qua phương thức ảo để có thể kết nối động*.

Các bước áp dụng đa hình nói chung như sau:

- Khi thiết kế, khái quát hóa các lớp có đặc điểm gần giống nhau thành lớp cơ sở chung, đưa vào đó các phương thức ảo. Lớp cơ sở chung có thể nâng cấp thành lớp trừu tượng bằng cách khai báo một số phương thức ảo thành phương thức thuần ảo.
- Khi cài đặt, dẫn xuất lớp cơ sở thành các lớp cụ thể, trong đó cài đặt cho các phương thức ảo, hành vi của chúng tương ứng với lớp cụ thể.
- Quản lý các thể hiện của các lớp cụ thể thông qua mảng con trỏ hoặc con trỏ kiểu lớp cơ sở. Khi gọi phương thức ảo trên các thể hiện lớp cụ thể thông qua con trỏ kiểu lớp cơ sở ta sẽ đạt được hiệu quả đa hình.

2. Lớp cơ sở trừu tượng (Abstract Base Class)

Lớp cơ sở cao nhất của các lớp đa hình thường chỉ mang tính khái niệm, vì vậy ta có khuynh hướng xây dựng lớp này thành lớp cơ sở trừu tượng.

Lớp cơ sở trừu tượng là một lớp chỉ được dùng làm *cơ sở khái niệm* cho việc tạo các lớp khác, dùng để *định nghĩa một số khái niệm tổng quát* cho các lớp dẫn xuất. Vì vậy lớp cơ sở trừu tượng KHÔNG tạo ra bất kỳ một thể hiện nào.

Lớp cơ sở trừu tượng thường chứa những phương thức ảo, tổng quát cho mọi lớp dẫn xuất. Để các phương thức này *mang tính khái niệm*, ta định nghĩa chúng như các *phương thức thuần ảo*.

Phương thức thuần ảo (pure virtual method) là một phương thức ảo mà trong định nghĩa của lớp, nó được định nghĩa là "không

⁸ Các lớp đa hình thể hiện đa hình kiểu (type polymorphism), C++ còn hỗ trợ đa hình tham số (parametric polymorphism) thông qua template.

có gì". Trong v-table, con trỏ hàm chỉ đến phương thức thuần ảo là một con trỏ NULL.

```
virtual return_type function_name( arguments ) = 0;
```

Như vậy, lớp trừu tượng là lớp có chứa phương thức thuần ảo. Các lớp dẫn xuất từ lớp trừu tượng phải khai báo lại tất cả các phương thức thuần ảo mà nó thừa kế, định nghĩa lại tất cả các phương thức thuần ảo mà nó thừa kế với cài đặt cụ thể. Khi đó lớp dẫn xuất được gọi là lớp cụ thể (concrete class).

Nếu vẫn chưa định nghĩa lại tất cả các phương thức thuần ảo, lớp dẫn xuất vẫn là lớp dẫn xuất trừu tượng (deriving abstract class). Lớp dẫn xuất từ lớp cụ thể cũng có thể là lớp trừu tượng, nếu ta khai báo thêm phương thức thuần ảo cho nó.

Bạn *có thể cài đặt* phương thức thuần ảo của lớp trừu tượng, thường làm cơ sở để các lớp dẫn xuất gọi khi cài đặt lại phương thức này. Lớp có cài đặt cho phương thức thuần ảo vẫn là lớp trừu tượng do có khai báo phương thức thuần ảo.

Phương thức thuần ảo *có thể được gọi* từ các phương thức không thuần ảo khác trong lớp trừu tượng.

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

class Shape
{
public:
    Shape( const string & s = "" ) : name( s ) {}
    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;
    virtual void draw() const = 0;
protected:
    string name; // chung cho mọi hình
};

// cài đặt cho phương thức thuần ảo, thực hiện bên ngoài khai báo lớp trừu tượng
void Shape::draw() const
{
    cout << name << endl;
}

class Rectangle : public Shape
{
public:
    Rectangle( const string & s, double w, double h )
        : Shape( s ), width( w ), height( h ) { }
    double getArea() const { return width * height; }
    double getPerimeter() const { return 2 * ( width + height ); }
    void draw() const;
private:
    double width;
    double height;
};

void Rectangle::draw() const
{
    cout << name << " (" << width << ", " << height << ")" << endl;
}

// tham chiếu kiểu lớp cơ sở trừu tượng, dùng như đối số của hàm toàn cục
void display( const Shape & shape )
{
    shape.draw();
}

int main()
{
    Shape* pShape; // con trỏ kiểu lớp cơ sở trừu tượng
    pShape = new Rectangle( "Rectangle", 3, 4 ); // dùng chỉ đối tượng lớp dẫn xuất cấp phát động
    cout << pShape->getArea() << endl; // đa hình, getArea() của lớp Rectangle được gọi
    return 0;
}
```

Mặc dù không thể tạo các thể hiện cho lớp trừu tượng, ta vẫn có thể khai báo con trỏ và tham chiếu kiểu lớp trừu tượng để sử dụng khả năng đa hình.

Lớp cơ sở không tạo ra bất kỳ một thể hiện nào cũng không nhất thiết phải chứa phương thức thuần ảo. Ví dụ lớp cơ sở có constructor với phạm vi truy xuất protected, nên constructor của nó chỉ có thể gọi từ các phương thức trong lớp dẫn xuất, lớp cơ sở chỉ góp phần khởi tạo cho việc sinh thể hiện của lớp dẫn xuất.

3. Giao diện đa hình (Polymorphic Interface)

Các phương thức thuần ảo xác định giao diện chứa các tác vụ chung cần phải cài đặt trong các lớp dẫn xuất cụ thể. Vì vậy, lớp trừu tượng được gọi là giao diện đa hình của các lớp dẫn xuất. Lớp trừu tượng có thể được sử dụng để định nghĩa một lớp thuần giao diện (pure interface), trong đó các phương thức đều thuần ảo và không có dữ liệu thành viên.

IV. Chuyển kiểu tường minh

1. Toán tử static_cast<>

Chuyển kiểu lên (upcasting) trong cây phân cấp lớp luôn thực hiện được một cách an toàn dù chuyển kiểu không tường minh. Chuyển kiểu xuống (downcasting) trong cây phân cấp chỉ có thể thực hiện tường minh bằng toán tử ép kiểu (như C) hoặc dùng toán tử static_cast<>.

```
// PassCar và TruckCar là các lớp dẫn xuất từ lớp cơ sở Car
PassCar cabrio( "Spitfire", true, 1001, "Triumph" );
Car* carPtr = &cabrio; // upcasting PassCar thành Car
[ (PassCar*) carPtr ]->display(); // OK. Đối tượng do carPtr chỉ đến có kiểu PassCar
static_cast<PassCar*>( carPtr )->display(); // OK. Đối tượng do carPtr chỉ đến có kiểu PassCar
static_cast<Truck*>( carPtr )->setAxles(); // lỗi run-time, do carPtr chỉ đến đối tượng kiểu Truck
```

Chuyển kiểu xuống, bằng toán tử ép kiểu hoặc toán tử static_cast<>, cũng chỉ an toàn khi đối tượng được chỉ đến bởi con trỏ lớp cơ sở *thực sự có kiểu là lớp dẫn xuất chỉ định* khi ép kiểu; nếu không sẽ gây lỗi thời gian chạy. Điều này cũng áp dụng với tham chiếu lớp cơ sở.

2. Toán tử dynamic_cast<>

C++ cung cấp toán tử dynamic_cast<> dùng chuyển kiểu xuống an toàn trong các lớp đa hình. Toán tử này sẽ kiểm tra trong thời gian chạy, xem yêu cầu chuyển kiểu xuống có hợp lệ hay không.

Đối số của toán tử sẽ được chuyển sang kiểu mục tiêu, trong cặp <>. Kiểu mục tiêu phải là *con trỏ hoặc tham chiếu đến một lớp đa hình* hoặc con trỏ void. Nếu kiểu mục tiêu là con trỏ, đối số của toán tử cũng phải là con trỏ. Nếu kiểu mục tiêu là tham chiếu, đối số của toán tử phải chỉ đến một đối tượng trong bộ nhớ.

```
// PassCar và TruckCar là các lớp dẫn xuất từ lớp cơ sở Car
PassCar cabrio( "Spitfire", true, 1001, "Triumph" );
Car* carPtr = &cabrio; // upcasting PassCar thành Car
PassCar* passPtr = dynamic_cast<PassCar*>( carPtr );
if ( PassCarPtr != NULL ) passPtr->display(); // chuyển kiểu xuống thành công
Truck* truckPtr = dynamic_cast<Truck*>( carPtr );
if ( truckPtr != NULL ) truckPtr->display(); // downcasting thất bại, truckPtr là con trỏ NULL
// chuyển kiểu áp dụng cho tham chiếu
Car& r_car = cabrio;
PassCar& passRef = dynamic_cast<PassCar&>( r_car ); // chuyển kiểu xuống thành công
TruckCar& truckRef = dynamic_cast<PassCar&>( r_car ); // ném ra exception kiểu bad_cast
```

Toán tử dynamic_cast còn dùng với mục đích xác định thông tin liên quan đến lớp của đối tượng, còn gọi là xác định kiểu của đối tượng trong thời gian chạy (RTTI – Run-Time Type Information).

Một cách khác để xác định RTTI là dùng toán tử typeid thuộc thư viện <typeinfo>.

```
#include <typeinfo>
PassCar cabrio( "Spitfire", true, 1001, "Triumph" );
Car* carPtr = &cabrio; // upcasting PassCar thành Car
if ( typeid( *carPtr ) == typeid( PassCar ) )
    cout << "Object of " << typeid( *carPtr ).name() << endl;
```

3. Toán tử reinterpret_cast<>

Toán tử reinterpret_cast<> được dùng khi muốn xử lý một đối tượng theo một kiểu mục tiêu nào đó, trong khi đối tượng đó không liên quan đến kiểu mục tiêu.

Ví dụ, mình họa sau ghi 100 bản ghi lưu thông tin khách hàng có kiểu dữ liệu ClientData vào tập tin nhị phân. Do đối số phương thức write của ofstream là const char*, ta phải chuyển kiểu con trỏ chỉ đến đối tượng cần lưu trữ thành kiểu const char*, nếu không trình biên dịch sẽ không biên dịch lời gọi hàm write.

Toán tử reinterpret_cast<> được thực hiện trong thời gian biên dịch, nó yêu cầu trình biên dịch *diễn dịch lại* (reinterpret) dữ liệu truyền cho toán tử thành kiểu dữ liệu mục tiêu trong cặp <>.

```
ofstream outCredit( "credit.dat", ios::binary );
if ( !outCredit )
{
    cerr << "File could not be opened." << endl;
    exit( 1 );
}
ClientData client;
// ghi 100 record (trống) đến tập tin nhị phân
for ( int i = 0; i < 100; ++i )
    // diễn dịch lại kiểu dữ liệu ClientData thành kiểu dữ liệu const char*
    outCredit.write( reinterpret_cast<const char*>( &client ), sizeof( ClientData ) );
```

Stream

Stream

I. Các lớp stream

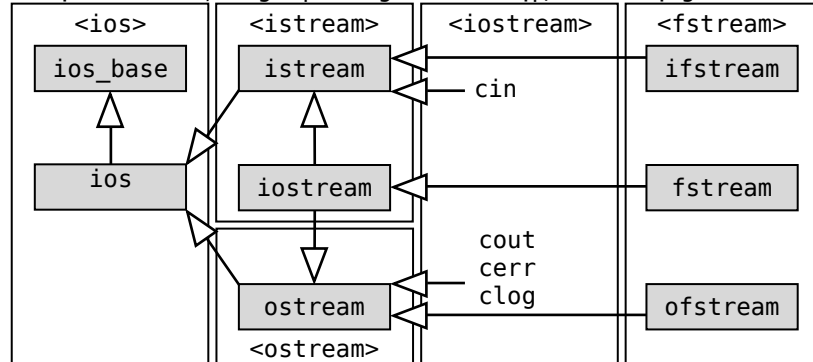
Stream được dùng để trừu tượng hóa thao tác nhập/xuất:

- Nhập có thể tiến hành từ các thực thể: bàn phím (ngõ nhập chuẩn), tập tin, socket, bộ đệm nhập.
- Xuất có thể đến các thực thể: màn hình (ngõ xuất chuẩn), tập tin, socket, bộ đệm xuất, máy in.

Stream là đối tượng "cắm" vào các thực thể đó, stream có các phương thức hỗ trợ nhập/xuất dữ liệu với các thực thể mà nó "cắm" vào.

Nói cách khác, muốn nhập/xuất với các thực thể, ta gọi các phương thức của đối tượng stream "cắm" vào thực thể đó.

C++ có một tập các lớp thể hiện các stream, cung cấp những thao tác nhập/xuất đa dạng:



iostream: nhập/xuất căn bản.

fstream: nhập/xuất tập tin.

stringstream: nhập/xuất chuỗi; dùng vận chuyển, định dạng dữ liệu trong bộ nhớ.

Ngoài ra còn có lớp streambuf quản lý vùng đệm, ta không làm việc trực tiếp với chúng.

Các đối tượng stream chuẩn thuộc thư viện <iostream> nối với các thiết bị mặc định: nhập chuẩn cin (như stdin), xuất chuẩn cout (như stdout), ngõ lỗi chuẩn cerr, clog (như stderr). cin, cout, clog là các stream có vùng đệm, cerr không có vùng đệm.

Các toán tử xuất nhập: toán tử trích (extraction) >>, toán tử chèn (insertion) <<. Các toán tử này là kết quả của việc nạp chồng các toán tử dịch bit.

II. Nạp chồng toán tử chèn và trích

Xem lại phần nạp chồng toán tử (operator overloading) và hàm bạn (friend function).

```

#include <iostream>
using namespace std;

class Star
{
public:
    friend ostream & operator<<( ostream &, const Star & );
private:
    int n;
public:
    Star( int k = 0 ) { n = k; }
};

ostream & operator<<( ostream & os, const Star & r )
{
    int i, j;
    for ( i = 1; i <= r.n; ++i )
    {
        for ( j = 1; j <= r.n + i - 1; ++j )
            os << ( ( j < r.n - i + 1 ) ? " " : "*" );
        os << endl;
    }
    return os;
}

int main()
{
    Star a( 5 );
    cout << a;
    return 0;
}
  
```

III. Các cờ định dạng stream

Định dạng được thực hiện tự động bởi lớp cơ sở ios, chi phối bởi trạng thái định dạng trong lớp này. Trạng thái định dạng gồm:

- Chiều rộng của trường.

- Ký tự lấp các khoảng trống (fill character, thường là space).
- Căn chỉnh trường.
- Cơ số (base) cho số nguyên (decimal, octal, hexadecimal).
- Dạng thức số có dấu chấm động (fixed, scientific, general).
- Có hay không có dấu + cho số dương.
- Viết hoa hay viết thường cho các số có E, 0x, các số hexadecimal.

Các trạng thái trên có thể thay đổi bằng hai cách:

- Dùng các phương thức định dạng của lớp ios: các hàm này trả về trị cũ của tham số đã thao tác, có thể lưu các trị nếu cần.

```
cout.fill( '0' );
cout.width( 10 );
```

- Dùng bộ định dạng (manipulator): phải khai báo thư viện <iomanip>

```
cout << setfill( '0' ) << setw( 10 );
```

Bảng dưới mô tả các bộ định dạng và phương thức thành viên của lớp ios tương ứng:

Bộ định dạng (manipulator)	Phương thức thành viên lớp ios
setfill(char)	char fill(char)
setw(int)	int width(int)
setprecision(int)	int precision(int)
setiosflags(long)	long setf(long)
resetiosflags(long)	long unsetf(long)
	long flags(long)
setbase(int)	
hex	
oct	
dec	
endl	
ends	
flush	
ws	

Các cờ định dạng stream gọi là fmtflags, được dùng trong các phương thức định dạng thành viên của lớp ios. Bảng sau so sánh các cờ định dạng stream (khai báo trong lớp ios) với chuỗi định dạng của C (khai báo trong stdio.h):

Cờ định dạng	Nhóm	Hiệu quả	stdio	Mặc định
	adjustfield	Thêm các ký tự chèn khi xuất để căn chỉnh:		left
left		trái	-	canh trái với số
right		phải	0	canh phải với text
internal		cho phép dùng ký tự đệm	không có	
	basefield	Chuyển đổi số integer nhập vào hay xuất số integer ở dạng d, o, h	%i	dec
dec		nhập/xuất cơ số 10	%d, %u	
oct		nhập/xuất cơ số 8	%o	
hex		nhập/xuất cơ số 16	%x	
	floatfield	Sinh ra các số dấu chấm động như fixed, scientific	%g %G	fixed
fixed		hiển thị trị có dấu chấm động ở dạng bình thường (có 6 số sau dấu chấm thập phân).	%f	
scientific		ghi chú kiểu khoa học, ví dụ: 8.888000e+1 cho số 88.88	%e %E	
boolalpha		Định dạng nhập/xuất với kiểu bool		0
showpos		hiển thị dấu + trước số dương tại ngõ xuất	+	0
showpoint		dấu chấm thập phân và các số 0 theo sau thể hiện đầy đủ khi xuất	.n (n: số các số theo sau dấu chấm)	0
showbase		cơ số được thể hiện rõ với 0x cho số hexadecimal và 0 cho số octal	#	0
skipws		bỏ các ký tự space ở đầu khi nhập	không có	1
unitbuf		súc vùng đệm xuất sau khi kết xuất	không có	0
uppercase		thay thế ký tự thường (trong các số hex, x hoặc e) thành ký tự hoa tại ngõ xuất	%X %E %G	0

IV. Các lớp File Stream

C++ cung cấp các lớp chuẩn để thao tác với tập tin, gọi là các lớp file stream. Thay vì nghĩ đến tập tin trên đĩa, chỉ cần nghĩ đến đối tượng file stream và những thao tác trên nó. Đọc dữ liệu của tập tin chính là trích dữ liệu từ file stream, ghi dữ liệu vào tập tin chính là chèn dữ liệu vào file stream.

Tập tin tiêu đề <fstream> khai báo các lớp file stream quan trọng nhất: fstream, ifstream, ofstream

```
#include <fstream>
```

Một đối tượng [io]fstream là một đối tượng [io]stream được mở rộng chức năng. Do đó ta có thể dùng tất cả các phương thức của [io]stream với các đối tượng [io]fstream.

Stream ifstream (input file stream) cho phép mở tập tin để đọc, ofstream (output file stream) cho phép mở tập tin để ghi.

1. Mở tập tin

Thao tác mở tập tin (tạo một "file handle"), chính là khai báo một đối tượng file stream. Constructor của file stream sẽ tự động thực hiện, liên kết file stream với một đối tượng filebuf (vùng đệm tập tin).

```
ofstream handle( filename, ios::mode1 | ios::mode2 )
```

Ngoài tên tập tin ta có thể cung cấp thêm cờ chế độ mở tập tin (file access mode), định nghĩa trong lớp ios. Có thể kết hợp nhiều chế độ bằng toán tử bitwise OR (|).

ios::in	mở để đọc một tập tin có sẵn
ios::out	mở để ghi, ngăn định kết hợp với ios::trunc nếu không kết hợp với ios::in, ios::app hoặc ios::app
ios::app	mở tập tin để ghi nối tiếp (append) thêm phía sau tập tin
ios::trunc	nếu tập tin đã tồn tại sẽ bị xóa (cắt đi)
ios::ate	mở và di chuyển con trỏ tập tin đến cuối tập tin ngay, không dùng cờ này, vị trí con trỏ là đầu tập tin
ios::binary	mở tập tin ở chế độ nhị phân
ios::noreplace	tập tin mở phải không tồn tại, nếu không tác vụ sẽ bị bỏ qua (không được thay thế)
ios::nocreate	tập tin mở phải tồn tại, nếu không tác vụ sẽ bị bỏ qua (không được tạo mới)

Sau khi thao tác với tập tin xong, gọi phương thức close() của file stream. Phương thức này ngắt liên kết giữa file stream với đối tượng filebuf và đóng file stream.

Dùng phương thức is_open() của file stream để xác định file stream có đang mở không. File stream đã đóng có thể mở lại với phương thức open() để dùng cho đối tượng khác.

```
handle.open( filename, ios::out | ios::app )
handle.close()
```

Lỗi có thể xảy ra khi mở tập tin, truy vấn trạng thái của file stream để nhận biết điều này:

```
if ( !file ) // hoặc if ( file.fail() )
```

2. Tập tin văn bản (text file)

Mặc định một tập tin được mở như một tập tin văn bản, có định dạng: tập hợp các ký tự ASCII chia thành nhiều dòng, mỗi dòng kết thúc bằng một ký tự sang dòng mới (new line, tùy hệ nền). Vì vậy đọc/ghi tập tin văn bản chính là thao tác đọc/ghi từng dòng với stream.

Để đọc dữ liệu từ tập tin văn bản, ta tạo file stream và trích dữ liệu từ file stream này.

```
#include <iostream>
#include <fstream> // phải có tập tin tiêu đề định nghĩa ifstream
using namespace std;

int main()
{
    ifstream file ( "test.txt" ); // khởi tạo stream với tên tập tin
    if ( !file )
    {
        cerr << "Open file error!" << endl;
        return -1;
    }
    while ( file )
    {
        string s;
        getline( file, s ); // thao tác trên tập tin chính là thao tác trên stream
        cout << s + '\n';
    }
    return 0;
}
```

Để biết trạng thái của file stream ta có hai cách:

- Đọc các cờ trạng thái.

ios::badbit	trị 1 nếu có lỗi vật lý
ios::failbit	trị 1 nếu có lỗi logic như sai định dạng
ios::eofbit	trị 1 nếu đã đến cuối stream
ios::goodbit	trị 1 nếu không có cờ trạng thái nào dựng lên

- Nhận trị trả về từ các phương thức thành viên lớp ios, chúng cũng truy cập các cờ trạng thái.

handle.bad()	trả về true nếu stream có lỗi vật lý
handle.fail()	trả về true nếu stream có lỗi logic như sai định dạng
handle.eof()	trả về true nếu đã đến cuối stream
handle.good()	trả về true nếu stream đọc tốt, trả về false nếu các trường hợp trên trả về true
ios::rdstate()	đọc trạng thái của stream, ví dụ: if (f.rdstate() & ios::failbit) ...
handle.reset()	khôi phục trạng thái good cho stream khi muốn đọc tiếp sau khi bị bad/fail
handle.clear()	đặt trạng thái lỗi hoặc xóa các bit lỗi (mặc định)

Cũng có thể dùng chính đối tượng stream, chuyển kiểu như int. Ví dụ 2 cách kiểm tra sau đây là như nhau:

```
if ( f.fail() ) ... // có lỗi
if ( !f ) ...
```

Để ghi dữ liệu vào tập tin văn bản, ta tạo ofstream và chèn dữ liệu vào stream này.

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

int main()
{
    ifstream inf ( "test.txt" );    // mặc định: chế độ cập nhật, tương đương ios::in | ios::out
    ofstream outf ( "copy.txt" );  // tạo ofstream để ghi
    if ( !inf || !outf )           // nạp chồng toán tử ! kiểm tra đối tượng stream có tồn tại không
    {
        cout << "Open file error!" << endl;
        return -1;
    }
    while ( inf && outf )
    {
        string s;
        getline( inf, s );          // getline() không nhận ký tự '\n'
        outf << s + '\n';           // thao tác trên tập tin chính là thao tác trên stream
    }
    cout << "Copy completed ..." << endl;
    return 0;
}
```

3. Tập tin nhị phân (binary file)

Tập tin nhị phân phải được mở với chế độ nhị phân:

```
ifstream f ( "test.txt", ios::binary )
```

Dữ liệu của tập tin nhị phân không có dạng thức đặc biệt nên phải đọc và ghi từng ký tự bằng các phương thức get() và put() của stream hoặc đọc ghi theo khối với các phương thức read() và write() của stream.

Khi thao tác theo khối cần một vùng đệm (buffer) để chuẩn bị dữ liệu ghi vào tập tin hoặc nhận khối dữ liệu đọc từ tập tin.

```
handle.write( char* buffer, size_t size )
handle.read( char* buffer, size_t size )
```

size là kích thước khối dữ liệu cần thao tác, tính bằng byte.

Các phương thức read/write chỉ chấp nhận vùng đệm char*. Với đối tượng dữ liệu phức tạp, dùng toán tử reinterpret_cast<>:

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

class Account
{
    friend ostream & operator<<( ostream &, const Account & );
private:
    int code;
    string name;
    double balance;
public:
    Account( int c = 0, const string & n = "", double b = 0 )
        : code( c ), name ( n ), balance ( b ) { }
    int getCode() const { return code; }
    ostream & write( ostream & ) const;
    istream & read( istream & );
};

ostream & Account::write( ostream & os ) const
{
    os.write( reinterpret_cast<const char *>( this ), sizeof( Account ) );
    return os;
}

istream & Account::read( istream & is )
{
    is.read( reinterpret_cast<char *>( this ), sizeof( Account ) );
    return is;
}
```

```
ostream & operator<<( ostream & os, const Account & r )
{
    os << "[" << r.code << "]" << " " << r.name << " (" << r.balance << ")";
    return os;
}

int main()
{
    Account* accounts[] = { new Account( 1, "Whitney Elizabeth Houston", 2500 ),
                             new Account( 2, "Michael Joseph Jackson", 3400 ) };
    fstream outf ( "data.bin", ios::out | ios::app | ios::binary );
    for ( int i = 0; i < 2; ++i )
        if ( ! accounts[i]->write( outf ) )
            cerr << "Error in writing!" << endl;
    outf.close();

    cout << "Account #2" << endl;
    Account temp(2);
    fstream inf ( "data.bin", ios::in | ios::binary );
    inf.seekp( ( temp.getCode() - 1 ) * sizeof( Account ) );
    if ( ! temp.read( inf ) ) cerr << "Error in reading!" << endl;
    else cout << temp << endl;
    inf.close();
    return 0;
}
```

3. Con trỏ stream (stream pointer)

Vì tập tin nhị phân được thao tác như một stream chứa các byte liên tục nhau, ta dùng con trỏ stream để truy xuất tại các vị trí ngẫu nhiên trong stream. Vị trí (position) của con trỏ stream, một số kiểu long, được trả về bằng các phương thức:

handle.tellg()	trả về vị trí con trỏ stream hiện tại khi đọc tập tin (get pointer)
handle.tellp()	trả về vị trí con trỏ stream hiện tại khi ghi tập tin (put pointer)

Đưa con trỏ stream đến một vị trí trong khi đọc/ghi tập tin:

```
// đọc (get)
handle.seekg( position )
handle.seekg( offset, direction )
// ghi (put)
handle.seekp( position )
handle.seekp( offset, direction )
```

offset là số byte cần vượt qua, *direction* được định nghĩa trong lớp ios như sau:

ios::beg	bắt đầu từ đầu stream
ios::cur	bắt đầu từ vị trí hiện tại của con trỏ stream
ios::end	bắt đầu từ cuối stream

Có thể dùng hàm ignore() để bỏ qua một số byte trên stream hoặc duyệt trên stream cho đến khi gặp một ký tự nào đó.

Bài tập

Các ví dụ minh họa có trong phần lý thuyết không đủ thể hiện được hết kỹ thuật lập trình Hướng đối tượng dùng với ngôn ngữ C++. Bộ bài tập sau được chọn lọc và sắp xếp hợp lý để mang đến cho các bạn tư duy lập trình Hướng đối tượng và kỹ năng sử dụng ngôn ngữ C++ chuẩn.

Các bạn nên tự giải bài tập, so sánh với đáp án; tham khảo và phân tích đáp án song song với việc đọc phần lý thuyết. Do tiết kiệm trang in đáp án không được kèm theo sách.

[Structure] Khai báo structure Matrix lưu một ma trận cấp phát động có *r* dòng và *c* cột, chứa các trị nguyên.

Các phương thức của structure Matrix:

- Phương thức `create(int row, int col, bool flag)`: cấp phát động cho ma trận, *row* dòng và *col* cột. Nếu cờ *flag* là `false`, trị các phần tử của ma trận đều bằng 0; nếu cờ *flag* là `true` (mặc định), trị các phần tử của ma trận lấy ngẫu nhiên trong đoạn `[-10, 10]`.

- Phương thức `print(const string & message)`: in ma trận, *message* dùng để chú thích về ma trận trước khi in.

- Phương thức `Matrix mul(const Matrix & a, const Matrix & b)`: hai ma trận *a* và *b* được truyền như đối số, trả về ma trận kết quả là tích của hai ma trận *a* và *b* này. Chú ý điều kiện nhân ma trận.

- Phương thức `void destroy(const string & message)`: giải phóng vùng nhớ được cấp phát cho ma trận, *message* dùng để báo đã giải phóng vùng nhớ cấp phát cho ma trận.

Thử chuyển structure Matrix thành lớp Matrix.

[Class] Lớp Element trừu tượng hóa một nguyên tố hóa học trong bảng tuần hoàn Mendeleev, chứa các thông tin: số hiệu nguyên tử (*number*), ký hiệu nguyên tố (*symbol*), tên nguyên tố (*name*), nguyên tử khối (*weight*).

47	79
Ag	Au
107.868	196.967
Silver	Gold

Element
- number: int
- symbol: string
- name: string
- weight: double
- trim(string): string {readOnly}
+ getName(): string {readOnly}
+ accept(): bool
+ display() {readOnly}

Lớp Element có các phương thức sau:

- Phương thức `bool accept()`: nhắc và nhập thông tin từ ngõ nhập chuẩn. Nếu nhập sai kiểu dữ liệu hoặc nhập chuỗi rỗng, yêu cầu nhập thông tin lại từ đầu. Nhập số hiệu nguyên tử bằng 0 để ngừng nhập.

- Phương thức `void display() const`: hiển thị thông tin của một nguyên tố.

Chương trình cho phép nhập đến 110 nguyên tố. Hiển thị thông tin các nguyên tố được nhập vào.

Kết quả:

```
[--- Element Information ---]
Enter element number: 79
Enter symbol: Au
Enter full name: Gold
Enter element weight: 196.967

Enter element number: 47
Enter symbol: Ag
Enter full name: Silver
Enter element weight: 107.868

Enter element number: 29
Enter symbol: Cu
Enter full name: Copper
Enter element weight: 63.546

Enter element number: 0

No   Symbol  Name      Weight
-----
79   Au      Gold      196.967
47   Ag      Silver    107.868
29   Cu      Copper    63.546
```

[Class] Lớp Triangle trừu tượng hóa một tam giác như sau:

Triangle
– a: double – b: double – c: double
«constructor» + Triangle(a: double = 1, b: double = 1, c: double = 1) «accessor» + setEdge(a: double, b: double, c: double) «business» + getKind(): enum Kind + getPerimeter(): double + getArea(): double

Phương thức setEdges thiết lập trị cho ba cạnh tam giác một cách hợp lệ: chiều dài các cạnh phải lớn hơn 0, tổng 2 cạnh bất kỳ phải lớn hơn cạnh còn lại. Nếu trị thiết lập sai thì dùng trị mặc định.

Phương thức getKind cho biết loại tam giác bằng cách trả về enum Kind mô tả các loại tam giác: tam giác thường (scalene), tam giác cân (isosceles), tam giác đều (equilateral), tam giác vuông (right) và tam giác vuông cân (right isosceles).

Phương thức getPerimeter trả về chu vi tam giác.

Phương thức getArea trả về diện tích tam giác, tính bằng công thức Heron.

Test driver:

```
#include "triangle.h"
#include <iostream>
using namespace std;

int main() {
    Triangle triangle;
    string s[] = {"scalene", "isosceles", "equilateral", "right", "right isosceles"};
    triangle.setEdges(3, 4, 5);
    cout << "This is the " << s[triangle.getKind()] << " triangle" << endl;
    return 0;
}
```

Kết quả:

This is the right triangle

[Class] Lớp Account trừu tượng hóa một tài khoản ngân hàng như sau:

Account
– balance: double – log: vector<string>
«constructor» + Account(init: double = 0.0) «accessor» + getBalance(): double {readOnly} «utility» – time2Str(time_t): string {readOnly} «business» + credit(amount: double) + debit(amount: double &) + viewLog() {readOnly} + toString(): string {readOnly}

Phương thức credit (gửi thêm tiền vào tài khoản) cần kiểm tra sao cho số tiền gửi vào lớn hơn 0. Nếu không hợp lệ, ném exception thông báo lỗi.

Phương thức debit (rút tiền khỏi tài khoản) cần kiểm tra sao cho số tiền rút ra nhỏ hơn hoặc bằng balance hiện có. Nếu số tiền rút lớn hơn balance thì chỉ rút được số tiền bằng balance hiện có. Số tiền rút cũng không được là số âm.

Mọi giao dịch gửi tiền hoặc rút tiền đều ghi nhận bằng một chuỗi văn bản trong vector log. Phương thức viewLog dùng xem thông tin giao dịch lưu trong vector log này.

Phương thức toString trả về chuỗi cho thấy số tiền có trong tài khoản hiện tại.

Test driver sẽ tạo menu như sau:

1. Deposit
 2. Withdraw
 3. View balance
 4. View transactions
 5. Quit
- Your choice?

[Class] Lớp Clock trừu tượng hóa một đồng hồ như sau:

Clock
– hour: int – minute: int – seconde: int
«constructor» + Clock(h: int, m: int, s: int) + Clock(s: const string{&}) + Clock(n: long) + Clock(o: const Clock{&}) «accessor» + setTime(h: int, m: int, s: int) + getTime(h: int{&}, m: int{&}, s: int{&}) {readOnly} «business» + incrementSeconds() + incrementMinutes() + incrementHours() + print() {readOnly} + equals(o: const Clock{&}) {readOnly} «utility» – tokenize(s: string, d: char): vector<int>

Lớp Clock có nhiều constructor nạp chồng cho phép tạo các instance bằng nhiều cách khác nhau:

- khởi tạo Clock từ các trị giờ, phút, giây chỉ định.
- khởi tạo Clock từ một chuỗi có định dạng hh:mm:ss. Phương thức công cụ tokenize dùng tách chuỗi này thành các trị nguyên giờ, phút, giây; lưu trong một vector và trả về vector này.
- khởi tạo Clock từ một trị long, chính là số giây.
- khởi tạo Clock từ một đối tượng Clock khác, đây là copy constructor.

Getter/setter áp dụng cho một bộ (hour, minute, second). Setter cần kiểm tra để các thuộc tính chứa trị thích hợp:

$0 \leq \text{hour} < 24$, $0 \leq \text{minute} < 60$, $0 \leq \text{second} < 60$. Nếu trị thiết lập sai thì dùng trị 0.

Phương thức incrementSeconds tăng second lên 1, nếu second tăng đến 60 thì second trở về 0 và minute tăng lên 1.

Phương thức incrementMinutes tăng minute lên 1, nếu minute tăng đến 60 thì minute trở về 0 và hour tăng lên 1.

Phương thức incrementHours tăng hour lên 1, nếu hour tăng đến 24 thì hour trở về 0.

Phương thức print cho kết xuất có định dạng như sau: hh:mm:ss, ví dụ: 09:24:05.

[Operator overloading] Xây dựng lớp Date sau:

Date
– day: int {short} – month: int {sort} – year: int {sort}
«constructor» + Date(d: int, m: int, y: int) «accessor» + setDate() + setDate(r: const Date{&}) + setDate(d: int, m: int, y: int) + getDay(): int {readOnly} + getMonth(): int {readOnly} + getYear(): int {readOnly} «operator» + operator++(): const Date{&} + operator++(int): Date «utility» – isLeapYear(y: int): bool {inline} – isValid(d: int, m: int, y: int): bool {inline} «business» + toString(): string{&} {readOnly} + isLess(d: const Date{&}): bool {inline}{readOnly}

Phương thức setDate không tham số thiết lập cho dữ liệu thành viên từ thời gian hiện tại.

Nạp chồng toán tử ++ (prefix và postfix) tăng đối tượng Date thành ngày kế tiếp.

Phương thức công cụ isLeapYear trả về true nếu năm của Date là năm nhuận, theo giải thuật: year là bội số của 4 và không chia hết cho 100, hoặc year là bội số của 400.

Phương thức công cụ isValid kiểm tra tính hợp lệ của ngày, tháng, năm lớp Date; trả về true nếu hợp lệ, theo giải thuật:

$1 \leq \text{day} \leq \text{số ngày nhiều nhất trong tháng}$, $1 \leq \text{month} \leq 12$, $1970 \leq \text{year}$

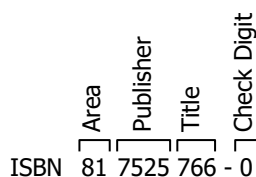
Phương thức toString trả về chuỗi chứa thông tin của đối tượng Date.

Phương thức isLess trả về true nếu đối tượng Date hiện hành gần hiện tại hơn với đối tượng Date so sánh (đối số).

[Case study] ISBN (International Standard Book Number, phát âm "is-ben") là mã số duy nhất cho sách xuất bản trên thế giới. Hiện có ISBN-10 và ISBN-13, bài này chỉ yêu cầu xử lý ISBN-10.

ISBN-10 cho phép 13 ký tự (0 – 9, ký tự nối – (hyphen) và X) chia thành 4 thành phần:

- Định danh vùng (Area): 1 đến 5 ký tự, định danh quốc gia, vùng, ngôn ngữ tham gia hệ thống ISBN.
- Định danh nhà xuất bản (Publisher): 1 đến 7 ký tự, mỗi Area có nhiều dãy (Range) và định danh Publisher phải nằm trong một dãy thuộc Area.
- Định danh sách tựa sách của nhà xuất bản (Title): 1 đến 6 ký tự, số ký tự bằng 9 – số ký tự Area – số ký tự Publisher.
- Số kiểm tra (Check Digit): chỉ chứa một ký tự (0 đến 9, X thay cho 10).



Một ISBN-10 hợp lệ phải có 10 ký tự (không tính ký tự nối) và có số kiểm tra đúng. Số kiểm tra được dùng kết hợp với các số khác trong một thuật toán kiểm tra tính hợp lệ của số ISBN như sau:

Lấy từng số của ISBN nhân với số thứ tự chỉ vị trí của nó (bắt đầu từ 1, tính từ phải sang trái, không tính dấu nối –). Tổng các tích nhận được nếu chia hết cho 11 thì số ISBN được kiểm tra là hợp lệ. Ví dụ:

ISBN	8	1	7	5	2	5	7	6	6	0
Vị trí	10	9	8	7	6	5	4	3	2	1
Tích	80	9	56	35	12	25	28	18	12	0

Tích 80 + 9 + 56 + 35 + 12 + 25 + 28 + 18 + 12 + 0 = 275

Tổng các tích là 275 chia hết cho 11, vậy số ISBN trên hợp lệ.

Một ISBN-10 hợp lệ chưa chắc là một ISBN-10 có đăng ký. Kiểm tra ISBN-10 có đăng ký bằng cách tra cứu prefix (Area và Publisher) trong tập tin chứa các prefix tại địa chỉ:

<https://cs.senecac.on.ca/~btp200/pages/assignments/prefixRanges.txt>

Một dòng của tập tin này có định dạng: [định danh Area] [cận dưới của Range] [cận trên của Range]

Do một nhóm có nhiều Range nên nhiều dòng có định danh Area giống nhau.

Xác định các thành phần của một ISBN là phức tạp vì số lượng ký tự trong mỗi thành phần biến động, trong lúc tổng số ký tự của ba thành phần là 9. Trước tiên cần kiểm tra định danh Area có trong tập tin, tiếp theo kiểm tra định danh Publisher có nằm trong một Range thuộc Area không.

Ví dụ: ISBN 8175227660 hợp lệ và đã đăng ký khi kiểm tra đến dòng: 81 7000 8499. Định danh Area 81 có tồn tại (nhưng Area 8 không tồn tại) và định danh Publisher 7522 thuộc Range [7000, 8499].

Chú ý rằng, định danh Publisher có thể có hai ký tự như 00 03, khi đó định danh như 0, 1, 2, 3 sẽ không thuộc Range.

Viết chương trình nhận các chuỗi ISBN-10 và kiểm tra xem có đăng ký hay không, nếu có, in thông tin về chuỗi ISBN-10 đó. Tổ chức các lớp:

ISBN	Prefix
<pre>- isbn: string - lines: map<string, Prefix{*}> + ISBN(const string{&}) + getInfo(): string - isISBN(): int - getArea(string{&}): Prefix{*} - isPub(string{&}, string, Prefix{*}): bool - loadFile(const char{*}): map<string, Prefix{*}></pre>	<pre>- area: string - ranges: vector<struct Range> + Prefix(const string{&}) + getArea(): string {readOnly} + getRanges(): vector<Range>{&}</pre>

- Prefix: chứa thông tin chuỗi định danh Area và vector các Range thuộc Area đó.

- ISBN: chứa cấu trúc dữ liệu thông tin lưu trong tập tin prefixRanges.txt, có các phương thức kiểm tra tính hợp lệ của chuỗi ISBN. Nếu hợp lệ, dùng các phương thức của lớp để kiểm tra và tách các định danh: Area, Publisher và Title.

Kết quả:

```
[----- ISBN Processor -----]
ISBN (0 to quit): 8175257660
Area           : 81
Publisher      : 7525
Title          : 766

ISBN (0 to quit): 817525766
Incorrect number of digits. Try again.

ISBN (0 to quit): 817525766X
Invalid check digit. Try again.

ISBN (0 to quit): 9995500000
This ISBN does not have a registered prefix

ISBN (0 to quit): 0
```


[Operator overloading] Lớp Fraction trừu tượng hóa một phân số như sau:

Fraction
- num: int - denom: int
«constructor» + Fraction(num: int = 0, denom: int = 1) «operator» + operator-(): Fraction «utility» - normalize() «friend» - operator+(x: const Fraction{&}, y: const Fraction{&}): Fraction - operator-(x: const Fraction{&}, y: const Fraction{&}): Fraction - operator*(x: const Fraction{&}, y: const Fraction{&}): Fraction - operator/(x: const Fraction{&}, y: const Fraction{&}): Fraction - operator+=(x: Fraction{&}, y: const Fraction{&}); - operator-=(x: Fraction{&}, y: const Fraction{&}); - operator*=(x: Fraction{&}, y: const Fraction{&}); - operator/=(x: Fraction{&}, y: const Fraction{&}); - operator==(x: const Fraction{&}, y: const Fraction{&}): bool - operator<(x: const Fraction{&}, y: const Fraction{&}): bool - operator>(x: const Fraction{&}, y: const Fraction{&}): bool - operator<<(os: ostream{&}, right: const Fraction{&}): ostream{&}

Phương thức công cụ normalize dùng chuẩn hóa phân số: đơn giản phân số, cả trường hợp tử số (mẫu số) là số âm. Nếu tử số cùng dấu với mẫu số, cả hai cùng dương; nếu khác dấu thì tử số có dấu âm.

Các toán tử nạp chồng có ý nghĩa như các phép toán trên phân số thật.

Test driver:

```
#include "fraction.h"
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    Fraction a(1, 4), b(3, 2), c, d;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "-b = " << -b << endl;
    c = 5 + a;
    cout << "c = 5 + a = " << c << endl;
    d = 1 - b;
    cout << "d = 1 - b = " << d << endl;
    c = 7 * a;
    cout << "c = 7 * a = " << c << endl;
    d = 2 / b;
    cout << "d = 2 / b = " << d << endl;
    c += 3;
    cout << "c += 3, c = " << c << endl;
    d *= 2;
    cout << "d *= 2, d = " << d << endl;
    cout << "a < 5: " << boolalpha << (a < 5) << endl;
    cout << "1 < b: " << boolalpha << (1 < b) << endl;
    cout << "d * b - a == c - 1: " << boolalpha << (d * b - a == c - 1) << endl;
    return 0;
}
```

Kết quả:

```
a = 1/4
b = 3/2
-b = -3/2
c = 5 + a = 21/4
d = 1 - b = -1/2
c = 7 * a = 7/4
d = 2 / b = 4/3
c += 3, c = 19/4
d *= 2, d = 8/3
a < 5: true
1 < b: true
d * b - a == c - 1: true
```

[Operator overloading] Cài đặt lớp Complex trừu tượng hóa số phức và các phép toán trên nó.

Complex
– real: double – imaginary: double
«constructor» + Complex(re: double = 0.0, im = 0.0) «operator» + operator+(right: const Complex{&}): Complex {readOnly} + operator-(right: const Complex{&}): Complex {readOnly} + operator*(right: const Complex{&}): Complex {readOnly} + operator=(right: const Complex{&}): Complex{&} + operator==(right: const Complex{&}): bool {readOnly} + operator!=(right: const Complex{&}): bool {readOnly} «friend» – operator<<(os: ostream{&}, right: const Complex{&}): ostream{&} – operator>>(is: istream{&}, right: Complex{&}): istream{&} «accessor» + setReal(re: double) + getReal(): double {readOnly} + setImaginary(im: double) + getImaginary(): double {readOnly}

Các toán tử nạp chồng có ý nghĩa như các phép toán trên số phức thật.

Biết: nếu $z = a + bi$ và $w = c + di$ là hai số phức, thì:

$$z + w = (a + c) + (b + d)i$$

$$z - w = (a - c) + (b - d)i$$

$$zw = (ac - bd) + (ad + bc)i$$

Test driver:

```
#include "complex.h"
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    Complex x(4.3, 8.2);
    Complex y(3.3, 1.1);
    Complex z;
    cout << "Enter real part and imaginary part of z: ";
    cin >> z;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;
    cout << "x + y = " << (x + y) << endl;
    cout << "x - y: " << (x - y) << endl;
    cout << "x * y: " << (x * y) << endl;
    cout << "z = x = " << (z = x) << endl;
    cout << "z == x: " << boolalpha << (z == x) << endl;
    cout << "z != y: " << boolalpha << (z != y) << endl;
    return 0;
}
```

Kết quả:

```
Enter real part and imaginary part of z: 3.2 4.6
x = 4.3 + 8.2i
y = 3.3 + 1.1i
z = 3.2 + 4.6i
x + y = 7.6 + 9.3i
x - y: 1 + 7.1i
x * y: 5.17 + 31.79i
z = x = 4.3 + 8.2i
z == x: true
z != y: true
```

[Dynamic members] Xây dựng lớp MyString sau, dựa trên thư viện <cstring>:

MyString
- s: char{*} - lenght: int
«constructor» + MyString(s: const char{*} const = "") + MyString(o: const MyString{&}) «destructor» + ~MyString() «operator» + operator=(right: const MyString{&}): const MyString{&} + operator+(right: const MyString{&}): MyString + operator+=(right: const MyString{&}): const MyString{&} + operator!(): bool {readOnly} + operator!=(right: const MyString{&}): bool {readOnly} + operator==(right: const MyString{&}): bool {readOnly} + operator<(right: const MyString{&}): bool {readOnly} + operator<(right: const MyString{&}): bool {readOnly} + operator<(right: const MyString{&}): bool {readOnly} + operator<(right: const MyString{&}): bool {readOnly} + operator[] (i: int): char{&} + operator[] (i: int):char {readOnly} + operator()(i: int, n: int = 0): MyString {readOnly} + operator char*() {readOnly} «friend» - operator<<(os: ostream{&}, right: const MyString{&}): ostream{&} - operator>>(is: istream{&}, right: MyString{&}): istream{&} - operator+(o: const char{*}, right: const MyString{&}): const MyString

operator+(const MyString &): trả về tham chiếu đến chuỗi kết quả, là kết quả của việc nối chuỗi hiện hành và chuỗi right.

operator!(): trả về trị bool cho biết chuỗi hiện hành có rỗng hay không (true: rỗng).

operator[] (i): trả về ký tự tại vị trí i của chuỗi hiện hành. Toán tử này phải vừa là lvalue, vừa là rvalue, ví dụ: s2[0] = s2[7]

operator(i, n): trả về chuỗi con n ký tự bắt đầu từ vị trí i của chuỗi hiện hành.

operator char*(): ép kiểu chuỗi hiện hành thành char*.

Test driver:

```
#include "mystring.h"
#include <iostream>
using namespace std;

int main() {
    MyString s1, s2("Computer ");
    MyString s3("Science");
    if (!s1) cout << "s1 is empty" << endl;
    else cout << "s1: \"\" << s1 << "\"\" << endl;
    cout << "s2: \"\" << s2 << "\"\" << endl;
    cout << "s3: \"\" << s3 << "\"\" << endl;
    cout << "s1 = s2 + s3: \"\" << (s1 = s2 + s3) << "\"\" << endl;
    cout << "s1(0, 8): \"\" << s1(0, 8) << "\"\" << endl;
    s2[0] = s2[7];
    cout << "s2[0] = s2[7]: \"\" << s2 << "\"\" << endl;
    cout << "s1 += \" \" + s1: \"\" << (s1 += \" \" + s1) << "\"\" << endl;
    cout << "s2 < s3 is \" << boolalpha << (s2 < s3) << endl;
    MyString *p = new MyString(s3);
    cout << "p -> s3: \"\" << *p << "\"\" << endl;
    return 0;
}
```

Kết quả:

```
s1 is empty
s2: "Computer "
s3: "Science"
s1 = s2 + s3: "Computer Science"
s1(0, 8): "Computer"
s2[0] = s2[7]: "romputer "
s1 += " " + s1: "Computer Science Computer Science"
s2 < s3 is false
p -> s3: "Science"
```

[Dynamic members] Xây dựng lớp FloatArr sau:

FloatArr
- arrPtr: float{*} - max: int - count: int
«constructor» + FloatArr(n) + FloatArr(n: int, val: float) + FloatArr(o: const FloatArr{&}) «destructor» + ~FloatArr() «operator» + operator[](i: int): float{&} + operator[](i: int): float {readOnly} + operator=(right: const FloatArr{&}): FloatArr{&} + operator+=(val: float): FloatArr{&} + operator+=(right: const FloatArr{&}): FloatArr{&} + operator==(right: const FloatArr{&}): bool {readOnly} + operator!=(right: const FloatArr{&}): bool {readOnly} «business» + length(): int {readOnly} - append(val: float): bool - append(right: FloatArr): bool + insert(val: float, pos: int): bool + insert(right: FloatArr, pos: int): bool + remove(pos: int): bool «friend» - operator<<(os: ostream{&}, right: const FloatArr{&}): ostream{&} - operator>>(is: istream{&}, right: FloatArr{&}): istream{&}

arrPtr là mảng các con trỏ kiểu float, max là số phần tử tối đa của mảng và count là số phần tử hiện có của mảng.

Phương thức FloatArr(int n, float val) là constructor cấp phát n phần tử và khởi gán mỗi phần tử với trị val.

operator[](int i) là nạp chồng toán tử lấy phần tử theo chỉ số i.

operator+=(float val) là nạp chồng toán tử +=, thêm sau mảng phần tử có trị val.

operator+=(const FloatArr& r) là nạp chồng toán tử +=, thêm sau mảng hiện tại một mảng r (r có thể là chính mảng hiện hành).

length() trả về kích thước thật của mảng (số phần tử hiện có trong mảng).

Phương thức append(float val) thêm trị val vào cuối mảng.

Phương thức append(FloatArr r) thêm mảng r vào cuối mảng (r có thể là chính mảng hiện hành).

Phương thức insert(float val, int pos) chèn trị val vào mảng tại vị trí pos.

Phương thức insert(FloatArr r) chèn mảng r vào mảng tại vị trí pos (r có thể là chính mảng hiện hành).

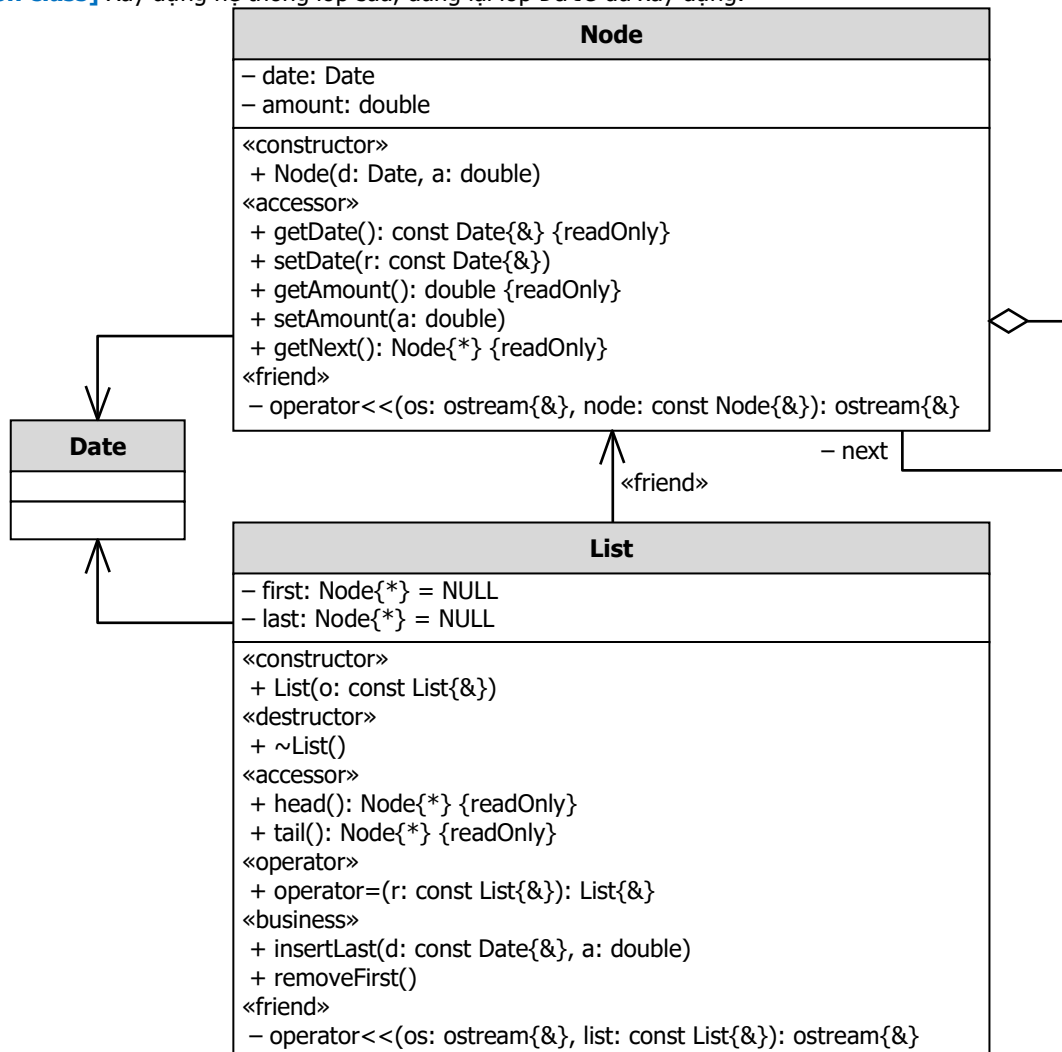
Phương thức remove(int pos) loại bỏ phần tử của mảng tại vị trí pos.

Phương thức expand(int) dùng để mở rộng mảng hiện hành (cấp phát thêm) khi mảng không đủ kích thước.

Kết quả:

```
Total of elements in v: 0
Total of elements in w: 15
v after appending:
-5 -3.3 -1.6 0.1 1.8 3.5 5.2 -5 -3.3 -1.6 0.1 1.8 3.5 5.2
cv is the copy of v:
-5 -3.3 -1.6 0.1 1.8 3.5 5.2 -5 -3.3 -1.6 0.1 1.8 3.5 5.2
w after assigning:
-5 -3.3 -1.6 0.1 1.8 3.5 5.2 -5 -3.3 -1.6 0.1 1.8 3.5 5.2
-5 -3.3 -1.6 0.1 1.8 3.5 5.2 -5 -3.3 -1.6 0.1 1.8 3.5 5.2
w after removing the odd postion:
-5 -1.6 1.8 5.2 -3.3 0.1 3.5
insert w into w at position 2
-5 -1.6 -5 -1.6 1.8 5.2 -3.3 0.1 3.5 1.8 5.2 -3.3 0.1 3.5
Enter 14 integers: 1 2 3 4 5 6 7 8 9 10 11 12 13 14
w after insert the new values
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

[Composition class] Xây dựng hệ thống lớp sau, dùng lại lớp Date đã xây dựng:



Lớp Node mô phỏng dữ liệu lưu thông tin thay đổi tài khoản, bao gồm date: ngày thay đổi và amount: số tiền giao dịch. Lớp Node có cấu trúc đệ quy, có một dữ liệu thành viên next là con trỏ Node* chỉ đến một đối tượng Node ngay sau nó.

Lớp List có:

- Phương thức List(const List&) là copy constructor, tạo một List từ một đối tượng List khác.
- Phương thức ~List là destructor, sẽ giải phóng tất cả các Node của danh sách.
- Phương thức nạp chồng toán tử gán List& operator=(const List&).
- Phương thức insertLast chèn một Node vào cuối danh sách.
- Phương thức removeFirst xóa một Node đầu danh sách.

Kết quả:

```

Hint: type invalid input to quit
Date format Day-Month-Year: 30-04-1975
Account change: 12000
Date format Day-Month-Year: 29-02-2000
Account change: 27000
Date format Day-Month-Year: 11-09-2001
Account change: 34000
Date format Day-Month-Year: q
    
```

```

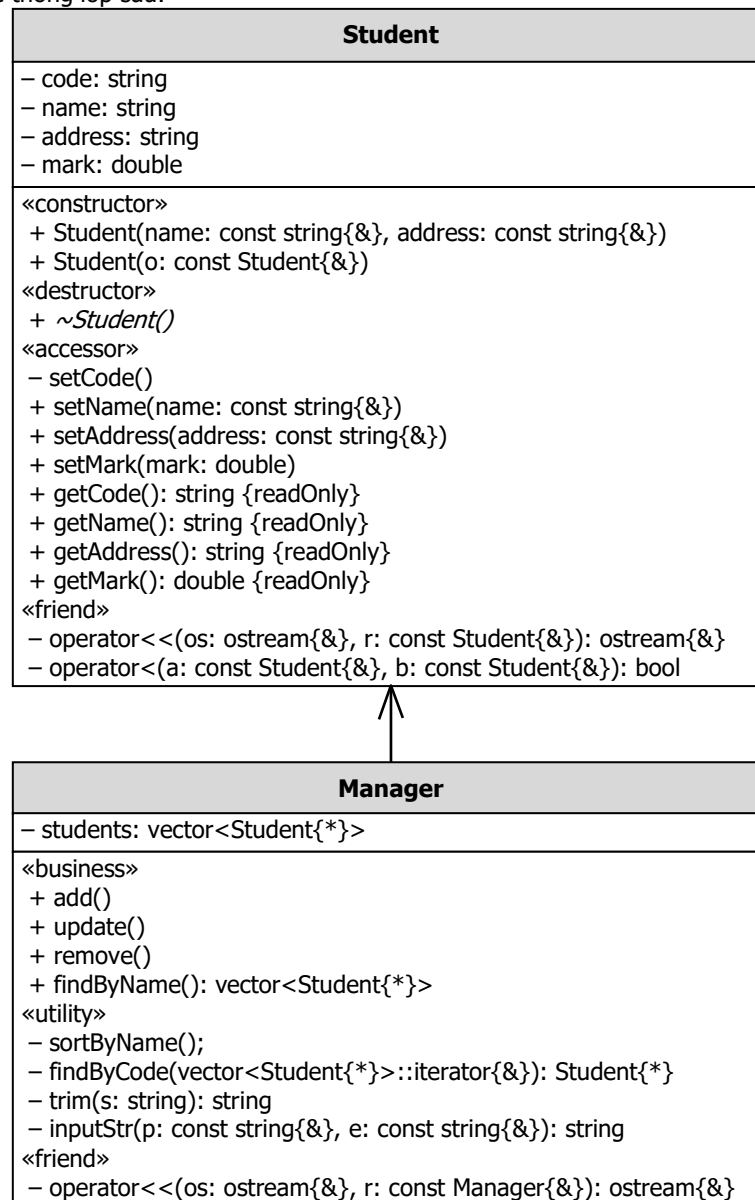
Content of the list:
[30-04-1975 Amount: 12000.00]
[29-02-2000 Amount: 27000.00]
[11-09-2001 Amount: 34000.00]
    
```

```

Removing the first node of the list:
Content of the list:
[29-02-2000 Amount: 27000.00]
[11-09-2001 Amount: 34000.00]
    
```

```

Increasing date of the first node:
Content of the list:
[01-03-2000 Amount: 27000.00]
[11-09-2001 Amount: 34000.00]
    
```

[Case study] Xây dựng hệ thống lớp sau:

Một số phương thức của lớp Student:

- Phương thức `setCode` dùng sinh mã tự động khi tạo một sinh viên, mã tự động gồm: ba ký tự đầu tên sinh viên viết hoa (ký tự space thay bằng x) và 4 ký tự số lấy từ 4 ký tự cuối trong chuỗi thời gian hiện tại (số giây tính từ 1/1/1970).

Lớp Manager thao tác trên một vector chứa các đối tượng lớp Student. Lớp Manager có thể:

- Thêm một đối tượng mới thuộc lớp Student vào vector do nó quản lý bằng phương thức `add`, đối tượng mới chỉ cần các thông tin (name, address), điểm (mark) sẽ cập nhật sau và code sẽ sinh tự động.
- Loại một đối tượng Student chỉ định bằng code khỏi vector do nó quản lý bằng phương thức `remove`.
- Cập nhật một đối tượng Student chỉ định bằng code trong vector do nó quản lý bằng phương thức `update`. Khi cập nhật, để tiện dụng, các thông tin không cần thay đổi chỉ cần nhấn Enter để bỏ qua.
- Dùng phương thức `findByName` để tìm các đối tượng Student trong danh sách sinh viên bằng cách nhập một phần của tên. Kết quả là vector lưu các đối tượng Student mà tên có một phần giống với từ khóa sẽ được chọn.
- Xuất danh sách sinh viên bằng các nạp chồng toán tử `<<`.
- Sắp xếp lại danh sách theo tên (alphabet) mỗi khi danh sách thay đổi (chèn, xóa, cập nhật). Nếu tên giống nhau thì sắp xếp theo code.

Khi nhập liệu cho một đối tượng Student mới, để kiểm tra dữ liệu nhập, dùng:

- Phương thức `trim` loại bỏ các ký tự space thừa đầu chuỗi, cuối chuỗi và giữa các từ.
- Phương thức `inputStr` cảnh báo nếu chuỗi nhập rỗng và yêu cầu nhập lại cho đến khi được chuỗi nhập hợp lệ.
- Khi cập nhật điểm cũng yêu cầu kiểm tra điểm nhập phải là số thực và hợp lệ ($0 \leq \text{mark} \leq 10$).

Test driver dùng menu hiển thị các chức năng của chương trình. Kết quả chạy mẫu như sau:

```

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
  
```

```
-----
Your choice: c
Name? Lionel Messi
Address? Argentina
Create successful!

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----

Your choice: u
Code? LI04430
Hint: use old value, press Enter
New name?
New address?
New mark? 8.6
Update successful!

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----

Your choice: r
Name? e
[LI04934] Lionel Messi (Argentina): 8.6

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----

Your choice: s
[LI04934] Lionel Messi (Argentina): 8.6

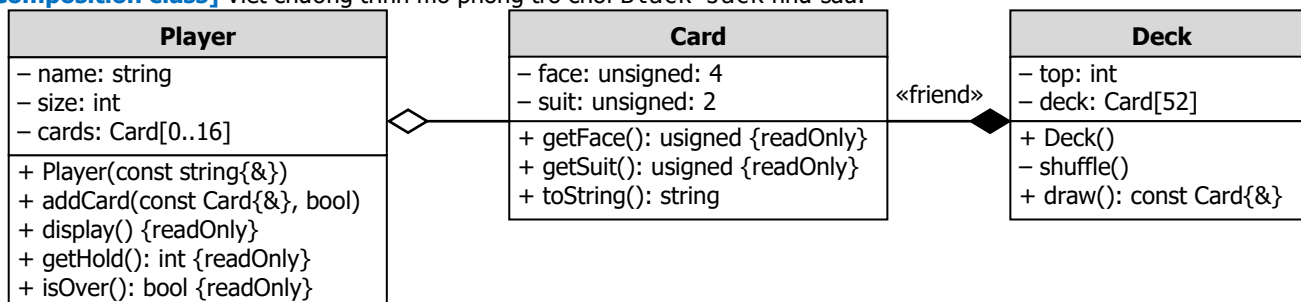
----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----

Your choice: d
Code? LI04430
Remove successful!

----- MENU -----
C Create a student
R Retrieve a student
U Update a student
D Delete a student
S Show student list
Q Quit
-----

Your choice: q
Bye bye!
```

[Composition class] Viết chương trình mô phỏng trò chơi Black Jack như sau:



Lớp Card trừu tượng hóa một lá bài tây (poker) chứa thông tin: nước (face: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King) và chất (suit: hearts, diamonds, spades, clubs).

Lớp Deck trừu tượng hóa một bộ bài có 52 lá bài. Khi khởi tạo, bộ bài được khởi gán và trộn ngẫu nhiên bằng phương thức công cụ shuffle. Lớp Deck có phương thức draw dùng để rút bài, phương thức này trả về lá bài đầu tiên trong phần còn lại của bộ bài đã trộn và đang chia.

Lớp Player trừu tượng hóa người chơi bài, mỗi người chơi bài có tên name. Lớp Player có phương thức addCard để nhận lá bài được chia từ phương thức draw thuộc lớp Deck, lưu lá bài này trong mảng các lá bài được chia cards. Phương thức addCard cũng cho phép hoặc không cho phép hiển thị lá bài được chia. Sau mỗi ván, từng người chơi sẽ mở bài (hiển thị các lá bài được chia) bằng phương thức display. Phương thức getHold dùng tính tổng điểm nhận được, bằng cách tính tổng số nút (face) trên các lá bài. Chú ý:

- Các lá bài: Jack, Queen và King đều có trị 10.
- Lá bài Ace có hai cách tính trị: bằng 1 hoặc bằng 11.

Phương thức isOver của lớp này cho biết tổng điểm nhận được có vượt 21 điểm hay không.

Hàm toàn cục main dùng lớp Player và lớp Deck để tiến hành trò chơi giữa hai người chơi như sau:

- Dealer (nhà cái) rút hai lá bài, một giấu kín và một mở bài. Người chơi cũng rút hai lá bài.
- Người chơi quyết định rút tiếp một số lá bài sao cho tổng điểm nhận được càng lớn càng tốt nhưng không vượt quá 21 điểm.
- Dealer rút một số lá bài theo chiến lược giống như người chơi. Dealer có một hệ số an toàn để dùng rút bài tiếp.
- Tổng số lá bài mỗi bên đang giữ không vượt quá 5 lá bài. Nếu rút quá 5 lá bài, cảnh báo lỗi nhưng vẫn tính điểm.

Điểm được tính như sau:

- Nếu hai bên hòa điểm hoặc cùng vượt quá 21 điểm thì dealer thắng.
- Dealer thắng: nếu dealer có tổng điểm không vượt quá 21 điểm và tổng điểm của dealer lớn hơn hoặc bằng tổng điểm của người chơi. Nếu người chơi vượt quá 21 điểm, dealer vẫn thắng theo luật hòa.
- Người chơi thắng: nếu người chơi có tổng điểm không vượt quá 21 điểm và tổng điểm của người chơi lớn hơn tổng điểm của dealer hoặc dealer có tổng điểm vượt quá 21 điểm.

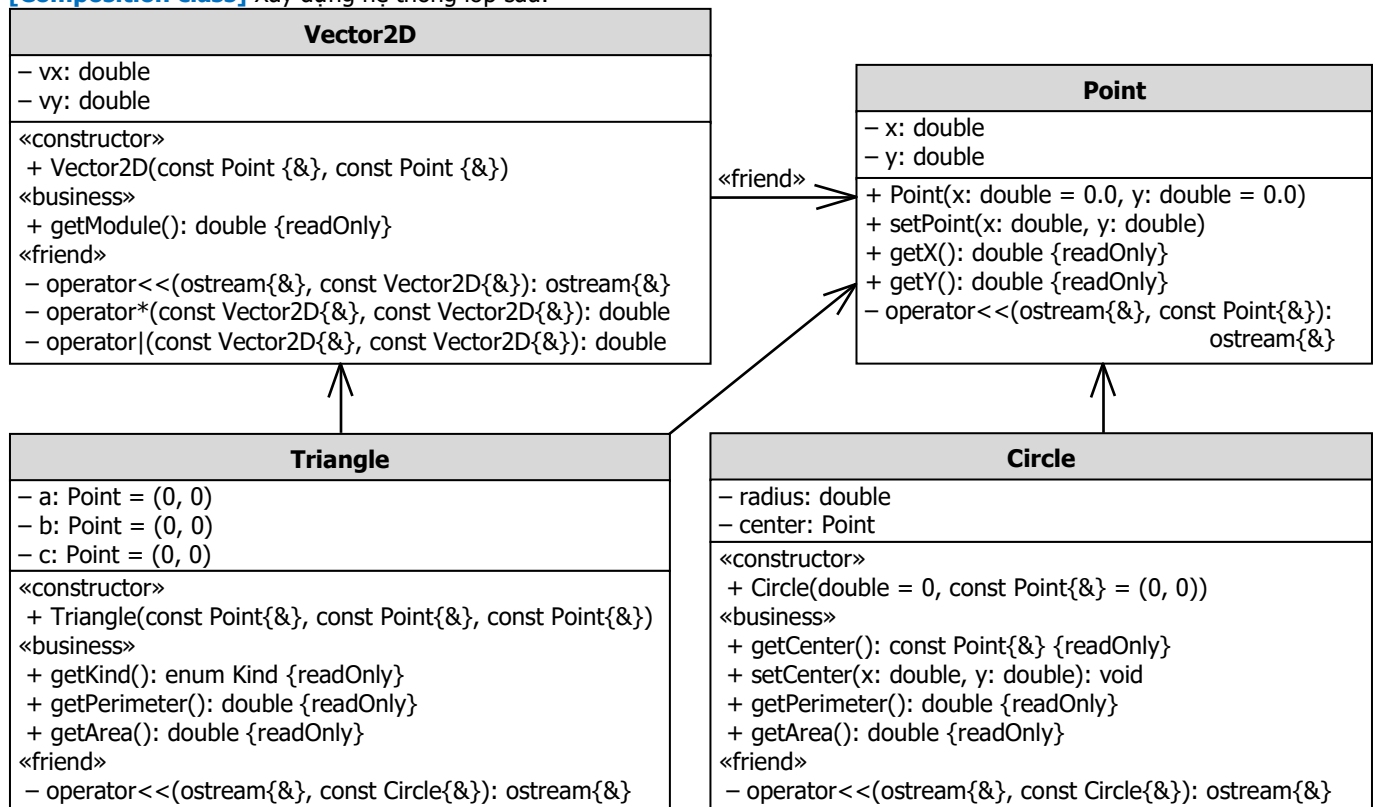
Kết quả:

```

[----- Black Jack -----]
Dealer drew: ??
Dealer drew: Jack of diamonds
You drew: four of hearts
You drew: Ace of spades
Draw? y
You drew: six of clubs
Draw? n
Dealer: ten of clubs, Jack of spades [20]
You : four of hearts, Ace of hearts, six of clubs [21]
Dealer lost, you win!
[Computer: 0 - Human: 1]
Another game? y

[----- Black Jack -----]
Dealer drew: ??
Dealer drew: four of hearts
You drew: two of diamonds
You drew: two of spades
Draw? y
You drew: ten of hearts
Draw? y
You drew: five of diamonds
Draw? n
Dealer drew: King of spades
Dealer: six of diamonds, four of hearts, King of spades [20]
You : two of diamonds, two of spades, ten of hearts, five of diamonds [19]
[Computer: 1 - Human: 1]
Another game? n
  
```


[Composition class] Xây dựng hệ thống lớp sau:



Lớp Vector2D mô tả một vector, hình thành từ 2 điểm (lớp Point 2D), có các thành phần vector vx và vy:

- Nạp chồng toán tử operator* cho kết quả là tích vô hướng của hai vector làm đối số.
- Nạp chồng toán tử operator| cho kết quả là tích hữu hướng của hai vector làm đối số.

Lớp Circle mô tả hình tròn với tâm center và bán kính radius:

- Phương thức getPerimeter trả về chu vi hình tròn, tính bằng công thức: $P = 2\pi R$
- Phương thức getArea trả về diện tích tam giác, tính bằng công thức: $S = \pi R^2$

Lớp Triangle mô tả tam giác với ba đỉnh a, b và c:

- Phương thức getKind cho biết loại tam giác bằng cách trả về enum Kind mô tả các loại tam giác: tam giác thường (scalene), tam giác cân (isosceles), tam giác đều (equilateral), tam giác vuông (right) và tam giác vuông cân (right isosceles).
- Phương thức getPerimeter trả về chu vi tam giác.
- Phương thức getArea trả về diện tích tam giác, tính hữu hướng của hai vector cạnh.

Các phương thức trong lớp Triangle dùng lớp Vector2D để tính toán, ví dụ dùng tích vô hướng của hai vector để xét hai cạnh có vuông góc với nhau không.

Test driver:

```

#include "circle.h"
#include "triangle.h"
#include <iostream>
using namespace std;

int main() {
    Circle circle(4.5, Point(2.1, 5.2));
    cout << "Circle: " << circle << endl;
    cout << "    Perimeter: " << circle.getPerimeter() << endl;
    cout << "    Area: " << circle.getArea() << endl;

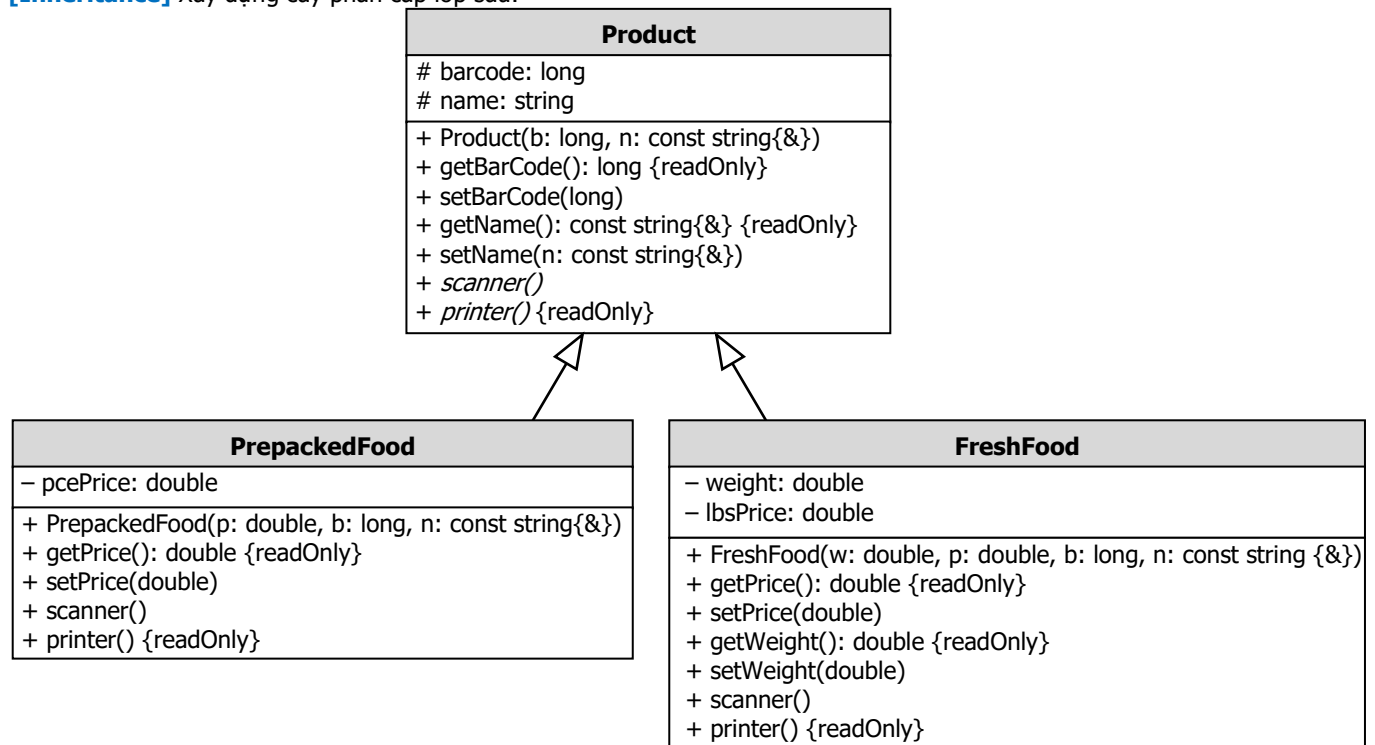
    string s[] = {"scalene", "isosceles", "equilateral", "right", "right isosceles"};
    Triangle triangle(Point(0, 7), Point(7, 0));
    cout << "Triangle: " << triangle << " -> " << s[triangle.getKind()] << " triangle" << endl;
    cout << "    Perimeter: " << triangle.getPerimeter() << endl;
    cout << "    Area: " << triangle.getArea() << endl;
    return 0;
}
  
```

Kết quả:

```

Circle: Radius: 4.5; Center: (2.1, 5.2)
    Perimeter: 28.2743
    Area: 63.6173
Triangle: [(0, 7), (7, 0), (0, 0)] -> right isosceles triangle
    Perimeter: 23.8995
    Area: 24.5
  
```

[Inheritance] Xây dựng cây phân cấp lớp sau:



Lớp Product có các phương thức trừu tượng được nạp chồng trong các lớp dẫn xuất:

- Phương thức scanner cho phép nhập thông tin của đối tượng (barcode, name, ...) từ bàn phím.
- Phương thức printer in các thông tin này.

Lớp PrepackedFood có thêm thuộc tính pcePrice (giá tính theo đơn vị).

Lớp FreshFood có thêm thuộc tính weight (trọng lượng) và lbsPrice (giá tính theo trọng lượng).

Test driver:

```

#include "product.h"
#include <vector>
using namespace std;

int main() {
    vector<Product*> products;
    products.push_back(new PrepackedFood(0.52, 12345, "Salt"));
    products.push_back(new FreshFood(1.2, 1.45, 56789, "Grapes"));
    Product* p = new PrepackedFood();
    p->scanner(); // nhận thông tin nhập
    products.push_back(p);

    cout << "\n\nList of products:";
    for (unsigned int i = 0; i < products.size(); ++i)
        products[i]->printer();
    return 0;
}
  
```

Kết quả:

```

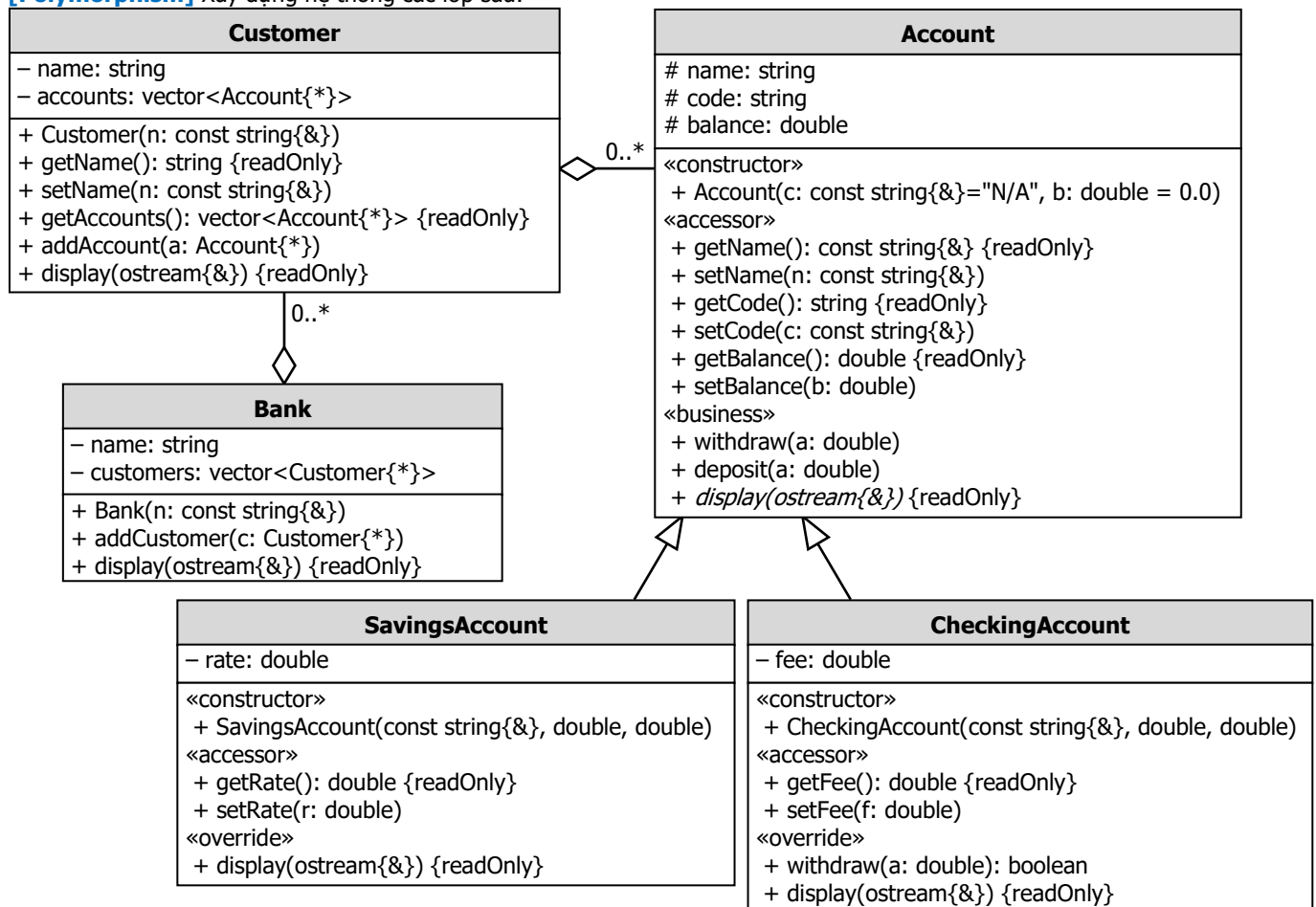
Barcode      : 13579
Name         : Sugar
Price/piece: 1.7

List of products:
[12345]
Name        : Salt
Price/piece: 0.52

[56789]
Name       : Grapes
Price/lbs  : 1.45
Weight    : 1.20
Total     : 1.74

[13579]
Name       : Sugar
Price/piece: 1.70
  
```

[Polymorphism] Xây dựng hệ thống các lớp sau:



Lớp Bank chứa danh sách các khách hàng (lớp Customer), mỗi khách hàng có nhiều tài khoản (lớp Account).

Có hai loại tài khoản:

- Tài khoản tiền gửi tiết kiệm (lớp SavingsAccount): tiền gửi tiết kiệm được tính lãi theo lãi suất (rate), công thức dùng tính lãi: $balance = balance + (rate * balance)$.
- Tài khoản tiền gửi thanh toán (lớp CheckingAccount): tiền gửi thanh toán phải tính phí (fee) khi rút tiền, nghĩa là mỗi lần rút tiền tài khoản sẽ giảm: số tiền muốn rút (amount) + phí (fee).

Kết quả:

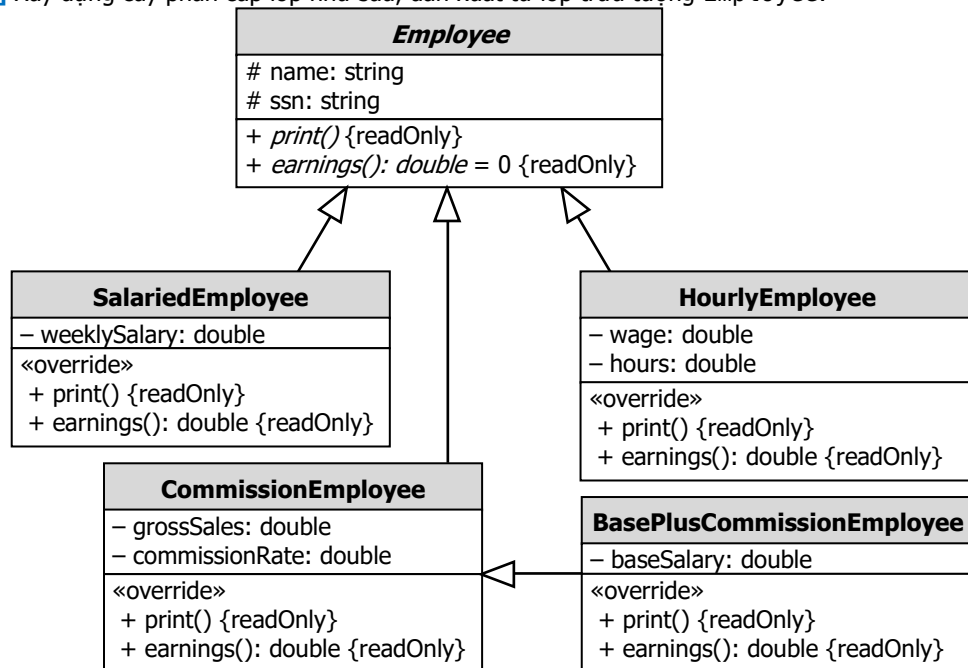
AGRIBANK

```

+ Account #1
Owner: Mickey Mouse
[SAV1234]-----
Balance      : $15000.00
Interest rate : 3.50
[CHK1357]-----
Balance      : $21000.00
Fee          : 0.50

+ Account #2
Owner: Donald Duck
[CHK5678]-----
Balance      : $24000.00
Fee          : 4.20
[SAV2468]-----
Balance      : $18000.00
Interest rate : 0.20
  
```

[Polymorphism] Xây dựng cây phân cấp lớp như sau, dẫn xuất từ lớp trừu tượng Employee:



Trong test drive, khai báo một vector employees kiểu Employee chứa cả 4 loại subclass trên (heterogenous array). Duyệt mảng và thực hiện phương thức earning và print để thấy kết quả của xử lý kết nối động (dynamic binding).

Tìm trong vector employees đối tượng thuộc lớp BasePlusCommissionEmployee (dùng dynamic_cast) và tăng baseSalary của chúng thêm 10%.

Hủy các đối tượng khi chấm dứt chương trình bằng cách dùng toán tử delete với vector employees.

Kết quả:

```

Salaried employee:
[111-11-1111] Mike Tyson
  Weekly salary: 800
  earned $800

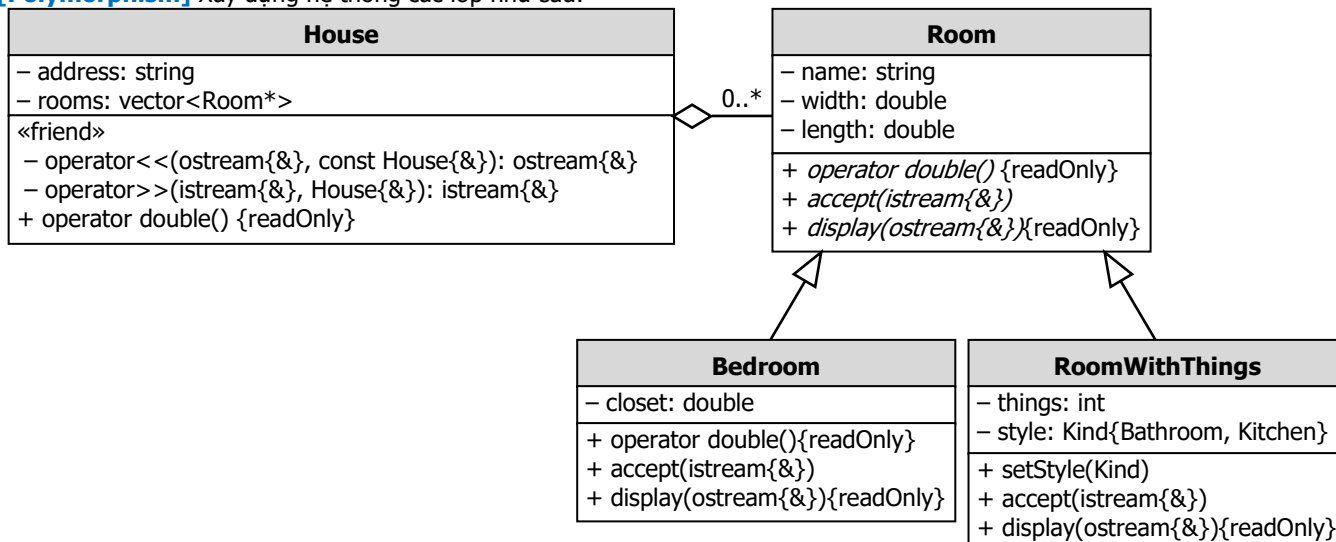
Hourly employee:
[222-22-2222] David Hayes
  Hourly wage: 28.5; Hours worked: 40
  earned $1140

Commission employee:
[333-33-3333] Muhammad Ali
  Gross sales: 10000; Commission rate: 0.06
  earned $600

Base-salaried:
Commission employee:
[444-44-4444] Wladimir Klitschko
Gross sales: 5000; Commission rate: 0.04; base salary: 300
old base salary: $300
  new base salary with 10% increase is: $330
  earned $530

Deleting object of 16SalariedEmployee
Deleting object of 14HourlyEmployee
Deleting object of 18CommissionEmployee
Deleting object of 26BasePlusCommissionEmployee
  
```

[Polymorphism] Xây dựng hệ thống các lớp như sau:



Lớp Room trừu tượng hóa một phòng trong nhà, chứa các thông tin: tên phòng (name), chiều rộng (width) và chiều dài (length). Lớp Room có các phương thức trừu tượng (virtual) để phục vụ cho đa hình:

- Nạp chồng toán tử ép kiểu double: trả về diện tích của phòng.
- accept(istream&): nhận thông tin về phòng do người dùng nhập từ stream nhập chỉ định (thường là cin).
- display(ostream&): xuất thông tin về phòng ra stream xuất chỉ định.

Lớp Bedroom dẫn xuất từ lớp Room, ngoài diện tích của phòng có tính thêm diện tích tủ chứa đồ (closet).

Lớp RoomWithThings cũng dẫn xuất từ lớp Room, có kiểu (style) là phòng tắm (Bathroom) hoặc nhà bếp (Kitchen). RoomWithThings cũng chứa thông tin về số đồ vật có trong phòng (things).

Xây dựng đầy đủ các phương thức: constructor, copy constructor và nạp chồng toán tử gán cho các lớp Room, Bedroom và RoomWithThings.

Lớp House trừu tượng hóa một nhà, chứa thông tin: địa chỉ nhà (address) và danh sách các phòng trong nhà (rooms). Các phương thức của lớp:

- Nạp chồng toán tử trích >>: hỏi người dùng địa chỉ, số lượng phòng của nhà, loại phòng; sau đó dùng phương thức đa hình accept để nhập thông tin cho các phòng, đưa chúng vào danh sách rooms.
- Nạp chồng toán tử chèn <<: duyệt danh sách rooms và dùng phương thức đa hình display hiển thị thông tin của các phòng.
- Nạp chồng toán tử ép kiểu double: duyệt danh sách rooms và gọi phương thức đa hình nạp chồng toán tử ép kiểu double để tính toán tổng diện tích của các phòng trong nhà.

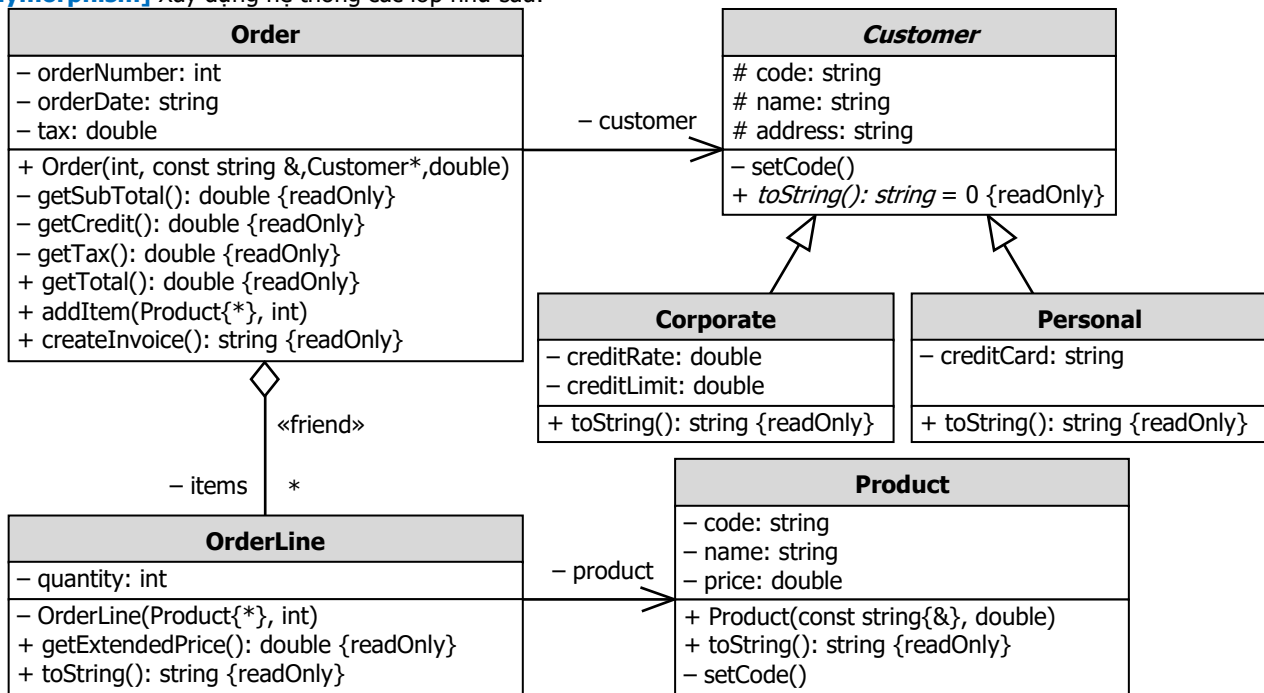
Kết quả:

```

[----- House Listing Preparation -----]
Address      : White House
No of rooms  : 4
Enter the type of room (r,b,w,k): r
Name of Room : Living
Width of Room : 3.5
Length of Room : 4.5
Enter the type of room (r,b,w,k): b
Name of Room : Master
Width of Room : 3.5
Length of Room : 3
Closet space  : 1.25
Enter the type of room (r,b,w,k): w
Name of Room : Bathroom
Width of Room : 1.2
Length of Room : 2.1
No of Pieces  : 4
Enter the type of room (r,b,w,k): k
Name of Room : Kitchen
Width of Room : 2
Length of Room : 3.5
No of Appliances: 3

[----- Room Information -----]
Address: White House
-----
Living (3.5 x 4.5)
Master (3.5 x 3), Closet space: 1.25
Bathroom (1.2 x 2.1), Number of Pieces: 4
Kitchen (2 x 3.5), Number of Appliances: 3
Total Floor Area: 37.02 sq m
  
```

[Polymorphism] Xây dựng hệ thống các lớp như sau:



Mỗi đơn hàng (Order) có số hiệu đơn hàng (`orderNumber`) và ngày lập đơn hàng (`orderDate`). Mỗi đơn hàng thuộc về một khách hàng (Customer). Mỗi khách hàng có thông tin: mã khách hàng (`code`), tên khách hàng (`name`) và địa chỉ khách hàng (`address`). Ngoài ra có thông tin riêng tùy theo loại khách hàng:

- Khách hàng cá nhân (Personal) có thông tin về thẻ tín dụng (`creditCard`).

- Khách hàng công ty (Corporate) có thông tin về chiết khấu hoa hồng (`creditRate`) tính bằng % trên tổng giá trị đơn hàng. Chiết khấu tính bằng phương thức `getSubtotal`, chiết khấu không được vượt quá giới hạn chiết khấu (`creditLimit`).

Thuế tiêu dùng (`tax`) tính bằng % trên tổng giá trị đơn hàng (đã trừ chiết khấu nếu có). Giá trị thanh toán đơn hàng được tính từ phương thức `getTotal`, bằng tổng giá trị đơn hàng + thuế.

Mỗi đơn hàng chứa một danh sách các mục đặt hàng (OrderLine), mỗi mục chứa thông tin sản phẩm (product) và số lượng đặt mua (quantity). Tổng giá trị một mục hàng được tính từ phương thức `getExpendedPrice`. Các mục đặt hàng được thêm vào đơn hàng nhờ phương thức `addItem`. Chú ý phương thức này tạo ra mục đặt hàng cho mỗi sản phẩm trước khi đưa vào mảng `items`; nói cách khác, constructor lớp OrderLine có phạm vi truy xuất private được gọi từ phương thức `addItem` của lớp friend là Order.

Mỗi sản phẩm (Product) có thông tin: mã sản phẩm (`code`), tên sản phẩm (`name`) và giá sản phẩm (`price`).

Khách hàng và sản phẩm được tạo mã tự động (phương thức `setCode`) gồm: ba ký tự đầu của tên (name) viết hoa (ký tự space thay bằng X) và 4 ký tự số lấy từ 4 ký tự cuối trong chuỗi thời gian hiện tại (số giây tính từ 1/1/1970).

Phương thức `createInvoice` dùng để xuất một đơn hàng đơn hàng cụ thể cho loại khách hàng cụ thể.

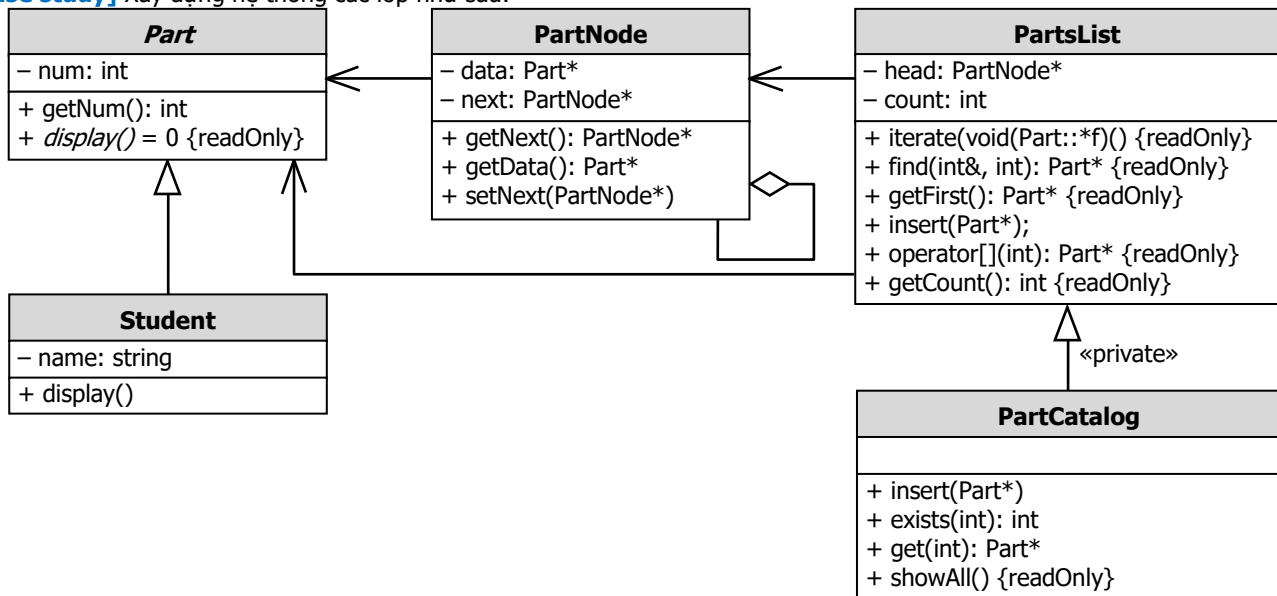
Kết quả:

```

INVOICE #100   Date: [21 Dec 2012]
Customer   : Bill Gates [BIL6014]
Credit Card: 1234-5678-9123-4567
-----
Code       Name       Price  Qty    Total Line
-----
[IPH6014] iPhone 5     15.2   4      60.8
[KIN6014] Kindle Fire 3.4   7      23.8
-----
Subtotal   : 84.6
Tax        : 2.115
Total      : 86.715

INVOICE #101   Date: [24 Dec 2012]
Customer   : Apple Inc. [APP6014]
Credit Rate: 8.5%
-----
Code       Name       Price  Qty    Total Line
-----
[IPH6014] iPhone 5     15.2   4      60.8
[KIN6014] Kindle Fire 3.4   7      23.8
-----
Subtotal   : 84.6
Credit    : 7.191
Tax        : 1.93522
Total      : 79.3442
  
```

[Case study] Xây dựng hệ thống các lớp như sau:



Dữ liệu lưu trữ được trừu tượng hóa trong lớp trừu tượng Part, lớp này có dữ liệu là num, số hiệu của một bản ghi dữ liệu. Để dùng hệ thống lớp, ta chỉ đơn giản thừa kế lớp Part và cài đặt hàm ảo display, như trong trường hợp lớp Student trên. Một danh sách liên kết đơn được trừu tượng hóa trong lớp PartsList, danh sách này được tạo từ các node được trừu tượng hóa trong lớp PartNode. Mỗi PartNode chứa: con trỏ data chỉ đến dữ liệu Part (gọi tắt, con trỏ dữ liệu) và con trỏ next chỉ đến PartNode ngay sau nó.

Lớp PartsList có các phương thức:

- Part* find(int& pos, int n) tìm node có số hiệu n, trả về con trỏ dữ liệu của node đó và vị trí của node đó tính từ đầu danh sách. Vị trí được trả về bằng tham số pos. Nếu không tìm thấy node có số hiệu chỉ định, trả về NULL và pos bằng 0.
- Part* getFirst() trả về con trỏ dữ liệu của node đầu tiên. Nếu danh sách rỗng, trả về NULL.
- void insert(Part* p) chèn node chứa con trỏ dữ liệu p vào danh sách liên kết sao cho các node của PartList có *thứ tự tăng dần* theo số hiệu num của dữ liệu lưu trữ.
- Nạp chồng toán tử lấy chỉ số operator[] (int offset), trả về con trỏ dữ liệu tại node có chỉ số offset trong danh sách liên kết, trả về NULL nếu chỉ số chỉ định không hợp lệ.
- int getCount() trả về số node có trong danh sách.
- iterate((void(Part::*f)())) nhận con trỏ chỉ đến hàm thành viên của lớp Part, cho phép thực hiện hàm này trên các node của danh sách.

Lớp PartCatalog thừa kế lớp PartsList theo kiểu private, xem PartsList như một đối tượng bên trong PartCatalog (quan hệ composition). Lớp PartCatalog có các phương thức:

- void insert(Part* p) chèn node chứa con trỏ dữ liệu p vào PartsList, báo lỗi nếu số hiệu num của node nhập đã có.
- int exists(int n) trả về vị trí tính từ đầu danh sách của node chứa con trỏ dữ liệu có số hiệu n;
- Part* get(int n) trả về con trỏ dữ liệu của node, số hiệu của dữ liệu là n.
- Phương thức showAll() duyệt tất cả các node của PartsList, sử dụng phương thức iterate của lớp PartsList.

Kết quả:

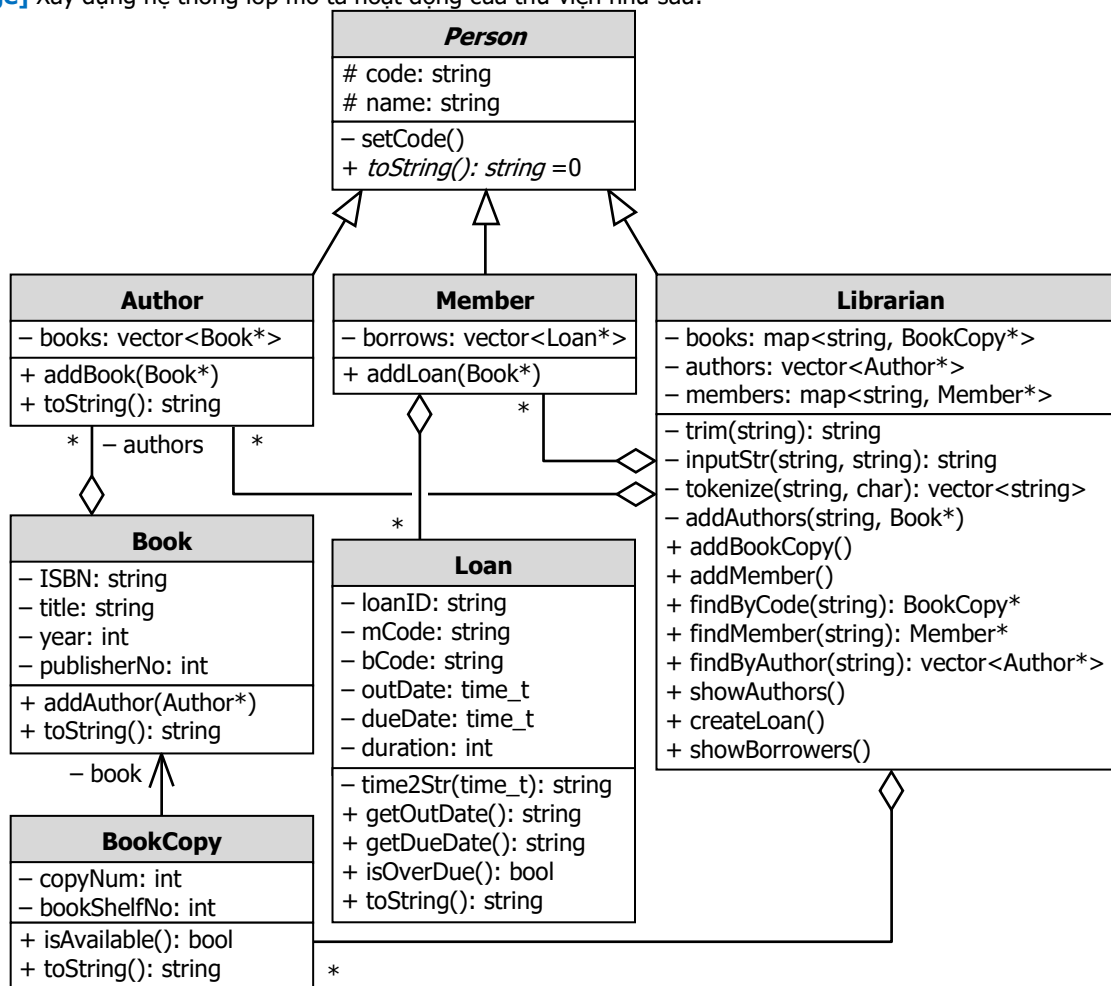
```

----- MENU -----
1. Create student
2. Show students
3. Quit
-----
Your choice: 1
Code? 101
Name? Johann Wolfgang Goethe

----- MENU -----
1. Create student
2. Show students
3. Quit
-----
Your choice: 2
[101] Johann Wolfgang Goethe

----- MENU -----
1. Create student
2. Show students
3. Quit
-----
Your choice: 3
Goodbye!
  
```


[Challenge] Xây dựng hệ thống lớp mô tả hoạt động của thư viện như sau:



Thành viên của thư viện (Member) có một danh sách các phiếu mượn (borrows), thông tin sách mượn được thêm vào danh sách này bằng phương thức addLoan.

Tác giả sách (Author) có một danh sách các sách của tác giả đó có trong thư viện (books).

Tác giả (Author), thành viên (Member) và thủ thư (Librarian) đều có thuộc tính chung là mã số (code) và tên (name), mã số được tạo tự động từ tên nhờ phương thức setCode: ba ký tự đầu của tên (viết hoa, ký tự space thay thế bằng ký tự X) ghép với 4 ký tự cuối của chuỗi thời gian hiện tại.

Một cuốn sách được trừu tượng hóa trong lớp Book, chứa các thông tin: mã ISBN, tựa sách (title), năm xuất bản (year), lần xuất bản (publisherNo). Một cuốn sách có một hoặc nhiều tác giả, lưu trong danh sách authors, một tác giả được thêm vào danh sách bằng phương thức addAuthor.

Một cuốn sách (book) của thư viện có nhiều bản sao (lớp BookCopy), mỗi bản sao có số lượng bản sao (copyNum) và số hiệu kệ chứa bản sao (bookShelfNo). Phương thức isAvailable cho biết sách còn đủ bản sao cho mượn không.

Lớp Loan lưu thông tin phiếu mượn sách, bao gồm: mã phiếu mượn (loanID, ghép từ mCode và 4 ký tự cuối của chuỗi thời gian hiện tại), mã của thành viên mượn sách (mCode), mã sách (bCode), ngày xuất sách (outDate), thời gian mượn (duration, tính bằng ngày, mặc định là 10 ngày), ngày trả (dueDate). Ngày xuất sách lấy từ thời gian hiện tại, ngày trả sách tự động tính từ ngày xuất sách và thời gian mượn; cả hai lưu bằng kiểu dữ liệu time_t, nên viết phương thức time2Str chuyển time_t thành chuỗi để dễ hiển thị. Phương thức isOverDue so sánh thời gian hiện tại với dueDate để xác định phiếu mượn có quá hạn hay không.

Lớp Librarian (thủ thư) quản lý books (danh sách sách của thư viện), authors (danh sách từng tác giả) và members (danh sách các thành viên).

Menu của hàm main gọi các phương thức của lớp Librarian (thủ thư) để quản lý thư viện:

- Tạo một thành viên mới bằng phương thức addMember.
- Thêm sách mới bằng phương thức addBookCopy, nếu đã có tựa sách đó thì tăng số lượng bản sao. Tạo một đối tượng sách (Book) mới kéo theo một đối tượng bản sao (BookCopy), vì thế constructor của hai lớp này được gọi trong phương thức addBookCopy. Khi thêm một sách mới, danh sách các tác giả (tách biệt bởi dấu phẩy) được tách ra rồi lưu vào authors (dùng phương thức công cụ addAuthors của lớp Librarian); tác giả mới được thêm vào authors còn tác giả cũ thì thêm tham chiếu của sách mới vào danh sách sách của tác giả đó (phương thức addBook của lớp Author).
- Cho một thành viên (với mã thành viên mCode) mượn một cuốn sách (với mã sách bCode), dùng phương thức createLoan. Lớp Librarian không cần lưu danh sách phiếu mượn, do đã lưu danh sách thành viên members nên có thể tham chiếu thông tin các phiếu mượn từ danh sách này.
- Phương thức showBorrowers hiển thị danh sách các thành viên đang nợ sách, với mỗi thành viên hiển thị danh sách phiếu mượn.
- Tìm theo tên tác giả, chỉ cần nhập một phần tên. Với mỗi tác giả trong danh sách kết quả, hiển thị tất cả sách của tác giả đó có trong thư viện, dùng phương thức showAuthors.

Một số chú ý khi xây dựng các lớp:

- Viết đầy đủ các getter/setter cần thiết. Bảo vệ các tham chiếu cẩn thận bằng const.
 - Các thông tin liên quan, kể cả danh sách và bảng băm, nên có kiểu dữ liệu hoặc kiểu dữ liệu thành phần là con trỏ để tránh khai báo đệ quy hoặc tình huống trình biên dịch không thể tính được kích thước kiểu dữ liệu mới.
 - Viết phương thức toString cho các lớp để dễ dàng hiển thị thông tin cho đối tượng mỗi lớp. Xem mẫu xuất bên dưới.
- Kết quả:

```

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----
Your choice: 1
Name? Pierce Brosnan
Create Successful: Pierce Brosnan [PIE0297]

```

```

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----
Your choice: 1
Name? Daniel Craig
Create Successful: Daniel Craig [DAN0306]

```

```

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----
Your choice: 2
ISBN? 1111531005
Number of copy? 10
Title? A First Book of C++
Author? Gary Bronson, Paul Deitel
Publisher number? 4
Publisher year? 2012
Bookshelf number? 101

```

```

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----
Your choice: 2
ISBN? 0132662361
Number of copy? 12
Title? C++ How to Program
Author? Paul Deitel, Walter Savitch
Publisher number? 8
Publisher year? 2012
Bookshelf number? 101

```

```

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----

```

```
Your choice: 3
Member Code? PIE0297
ISBN? 1111531005

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----

Your choice: 3
Member Code? DAN0306
ISBN? 0132662361

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----

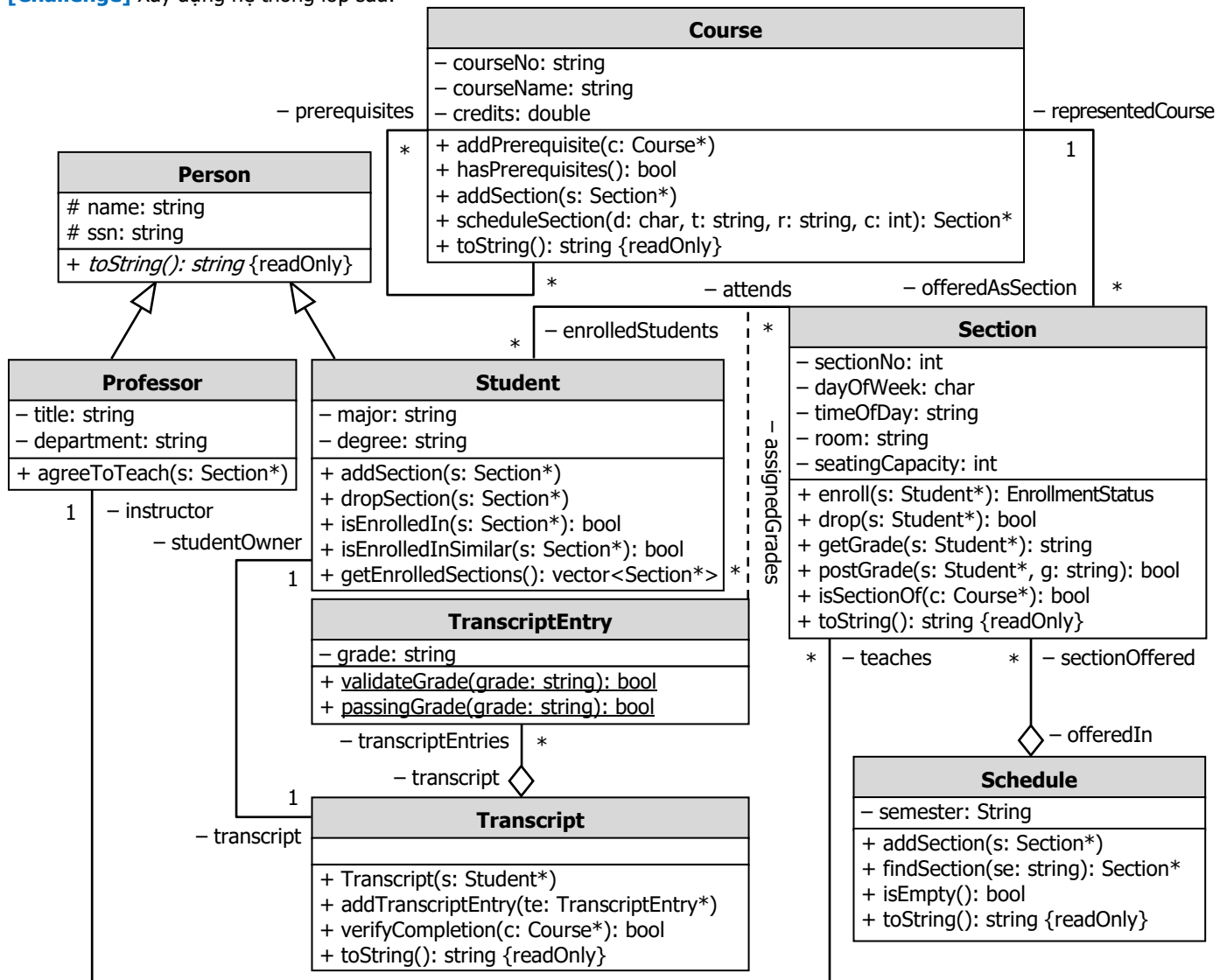
Your choice: 4
1. Pierce Brosnan [PIE0297]
(1) [1111531005] A First Book of C++
Due date: 30-10-2012 (pending)
2. Daniel Craig [DAN0306]
(1) [0132662361] C++ How to Program
Due date: 30-10-2012 (pending)

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----

Your choice: 5
Author name? Paul Deitel
Book Information
  Gary Bronson, Paul Deitel
  "A First Book of C++", Fourth Edition (2012)
  ISBN: 1111531005
Book Information
  Paul Deitel, Walter Savitch
  "C++ How to Program", Eighth Edition (2012)
  ISBN: 0132662361

----- MENU -----
1. Create member
2. Add book
3. Loan book
4. Show borrowers
5. Find by author name
6. Quit
-----

Your choice: 6
Bye bye!
```

[Challenge] Xây dựng hệ thống lớp sau:

Một môn học (Course) được dạy nhiều lần trong các khóa học (Section) khác nhau. Mỗi khóa học do một giáo viên (Professor) dạy và chiếm một mục trong thời khóa biểu (Schedule) của một học kỳ chỉ định, với thời gian và vị trí phòng học cụ thể.

Sinh viên (Student) sẽ đăng ký tham dự các khóa học (Section). Kết quả học một khóa học của sinh viên thể hiện như một mục trong học bạ (TranscriptEntry) sẽ được lưu vào học bạ (Transcript).

Mô tả chi tiết:

- Lớp Person thể hiện một cá nhân có thông tin chung là name và ssn (Social Security Number).
- Lớp Professor thể hiện một giáo viên, ngoài những thông tin cá nhân còn có: title (chức danh), department (khoa đang dạy); và thông tin liên kết khác như teaches (danh sách các khóa học đang dạy). Phương thức agreeToTeach dùng để thêm khóa học vào danh sách các khóa học đang dạy teaches, đồng thời cũng ghi nhận người dạy (instructor) cho khóa học đó trong lớp Section.

- Lớp Student thể hiện một sinh viên, ngoài những thông tin cá nhân còn có: major (chuyên ngành), degree (học vị); và các thông tin liên kết khác như transcript (học bạ), attends (các khóa học đang tham dự).

Một Student có thể tham dự nhiều khóa học (addSection), bỏ một khóa học đang tham dự (dropSection) và có phương thức kiểm tra xem có đăng ký một khóa học nào đó không (isEnrolledIn), hoặc kiểm tra xem đã đăng ký môn học dạy trong khóa học hiện hành trước đây chưa (isEnrolledInSimilar). Mỗi sinh viên được tạo ra sẽ có một học bạ (Transcript); nói cách khác, constructor của Transcript được gọi từ constructor của Student.

- Lớp Course thể hiện một môn học, một Course có courseNo (mã môn học), courseName (tên môn học), credits (số tín chỉ); và các thông tin liên kết như: prerequisites (danh sách các môn học tiên quyết, bắt buộc phải qua môn trước khi học môn này), offeredAsSection (danh sách các khóa học đang dạy môn học này).

Có thể thêm một môn học tiên quyết vào danh sách môn học tiên quyết (addPrerequisite), thêm một khóa học vào danh sách các khóa học dạy môn này (addSection). Phương thức scheduleSection sinh ra một khóa học với thông tin cụ thể và đưa nó vào danh sách offeredAsSection; nói cách khác, ta không gọi trực tiếp constructor của lớp Section mà gọi thông qua phương thức này.

Lớp Course cũng có phương thức hasPrerequisites để kiểm tra xem môn học chỉ định có môn học tiên quyết hay không.

- Lớp Section thể hiện một khóa học, chứa các thông tin: mã khóa học có định dạng courseCode-sectionNo, courseCode là mã môn học, sectionNo là số lần dạy môn học này (tính từ offeredAsSection), dayOfWeek (thứ trong tuần của ngày học), timeOfDay (thời gian học), room (số hiệu phòng), seatingCapacity (số lượng học viên tối đa); và các thông tin liên kết như: instructor (giáo viên đảm nhận khóa học), representedCourse (môn dạy), offeredIn (thuộc về thời khóa biểu

nào). Lớp Section cũng có danh sách sinh viên đăng ký môn học (enrolledStudents) và danh sách điểm đã đạt của sinh viên (assignedGrades).

Lớp Section chứa các phương thức enroll (đăng ký sinh viên học), drop (loại bỏ sinh viên đăng ký), getTotalEnrollment (xem số lượng sinh viên đăng ký), postGrade (ghi điểm cho một sinh viên chỉ định), getGrade (lấy điểm học của một sinh viên chỉ định), isSectionOf (xem có phải khóa học đang dạy môn chỉ định).

Phương thức enroll kiểm tra xem sinh viên thuộc trạng thái đăng ký (enum EnrollmentStatus) nào sau đây:

- + prevEnroll: sinh viên đã đăng ký khóa học hoặc đã hoàn thành môn học, không được đăng ký học lần nữa.
- + prereq: sinh viên chưa hoàn tất các môn tiên quyết của môn học chỉ định nên không được đăng ký môn học.
- + secFull: khóa học đã đủ học viên nên không nhận đăng ký thêm.
- + success: sinh viên đủ điều kiện đăng ký môn học, tiến hành đăng ký cho sinh viên

- Lớp Schedule thể hiện thời khóa biểu, có thuộc tính semester (học kỳ), sectionOffered (danh sách các khóa học thuộc học kỳ) và có các tác vụ: addSection (thêm khóa học vào danh sách khóa học thuộc học kỳ), findSection (tìm một lớp trong danh sách các khóa học thuộc học kỳ), isEmpty (kiểm tra danh sách các khóa học thuộc học kỳ có rỗng không).

- Lớp TranscriptEntry mô tả một mục trong học bạ, là kết quả học một lớp cụ thể của một sinh viên cụ thể. TranscriptEntry chứa các thông tin: student (sinh viên có tên trong học bạ), grade (điểm, lấy từ khóa học), section (khóa học) và transcript (học bạ chứa mục này); nó cũng chứa các phương thức dùng kiểm tra điểm nhập hợp lệ (validateGrade) và xác định điểm lớp học đó của sinh viên đã đủ qua môn chưa (passingGrade). Điểm nhập hợp lệ là A, B, C, D với điểm thêm bớt như B+, C-; và các điểm rớt là F (Failed), I (Ignored, bỏ môn). Điểm qua môn tối thiểu phải đạt D.

- Lớp Transcript thể hiện học bạ của sinh viên, chứa transcriptEntries (các mục nhập, mỗi mục nhập tương ứng một khóa học sinh viên đã học), studentOwner (tên sinh viên có học bạ). Ngoài ra có các phương thức: addTranscriptEntry (thêm một mục nhập vào học bạ) và verifyCompletion (kiểm tra xem đã học một môn học nào đó chưa).

Một số chú ý khi xây dựng các lớp:

- Viết đầy đủ các getter/setter cần thiết, viết thêm các phương thức công cụ, các constructor nạp chồng, các phương thức public khác nếu thấy cần thiết. Bảo vệ các tham chiếu cần thận bằng const.

- Viết phương thức toString cho các lớp để dễ dàng hiển thị thông tin cho đối tượng mỗi lớp. Xem mẫu xuất bên dưới.

Test driver:

```
#include "section.h"
#include <iostream>
using namespace std;

int main() {
    // Tạo Schedule (thời khóa biểu) cho học kỳ
    Schedule* sc = new Schedule("Summer 2012");
    // mở Course và thêm các Course tiên quyết để đăng ký được Course đó
    Course* c1 = new Course("I2C101", "Introduction to Computing", 1.5);
    Course* c2 = new Course("00P102", "Object-Oriented Programming", 3.5);
    c2->addPrerequisite(c1);
    Course* c3 = new Course("CJV103", "Core Java Programming", 2.0);
    c3->addPrerequisite(c1);
    c3->addPrerequisite(c2);
    // thêm các Professor tham gia giảng dạy các Section
    Professor* p1 = new Professor("Greg Anderson", "123-45-7890", "Adjunct Professor", "Information Technology");
    Professor* p2 = new Professor("Bjarne Stroustrup", "135-24-7908", "Full Professor", "Computer Science");
    Professor* p3 = new Professor("James Gosling", "490-53-6281", "Full Professor", "Software Engineering");
    // thêm các Student đăng ký học các Section
    Student* s1 = new Student("Barack Obama", "123-12-4321", "IT", "Ph.D.");
    Student* s2 = new Student("Vladimir Putin", "246-13-7985", "Math", "M.S.");
    Student* s3 = new Student("Francois Hollande", "135-78-2460", "Physics", "M.S.");
    // từ Course tạo Section dạy Course đó
    Section *se1 = c1->scheduleSection('M', "07:00-10:15 AM", "302", 24);
    Section *se2 = c2->scheduleSection('T', "02:15-05:30 PM", "405", 30);
    Section *se3 = c3->scheduleSection('F', "08:45-12:00 AM", "507", 22);
    // mời các Professor dạy các Section
    p1->agreeToTeach(se1);
    p2->agreeToTeach(se2);
    p3->agreeToTeach(se3);
    // đưa các Section đã thiết lập vào Schedule
    sc->addSection(se1);
    sc->addSection(se2);
    sc->addSection(se3);
    // đăng ký các Student vào các Section trong Schedule
    se1->enroll(s1);
    se1->enroll(s2);
    se1->enroll(s3);
    se1->postGrade(s2, "A"); // s1 và s2 đã học se1, s1 và s2 có thể đăng ký se2
    se1->postGrade(s3, "B");
    se2->enroll(s2);
    se2->enroll(s3);
    se2->postGrade(s3, "A+"); // s2 đã học se2, s2 có thể học se3
```

```

se3->enroll(s3);
// hiển thị thông tin
cout << c1->toString() << endl;           // các Course
cout << c2->toString() << endl;
cout << c3->toString() << endl;
cout << p1->toString() << endl;           // các Professor
cout << p2->toString() << endl;
cout << p3->toString() << endl;
cout << se1->toString() << endl;         // các Section
cout << se2->toString() << endl;
cout << se3->toString() << endl;
cout << sc->toString() << endl;         // thời khóa biểu
cout << s1->toString() << endl;         // các Student
cout << s2->toString() << endl;
cout << s3->toString() << endl;
return 0;
}

```

Kết quả:

```

[Course Information]
Course No. : I2C101
Course Name: Introduction to Computing
Credits    : 1.5
Prerequisite Courses: [none]
Offered As Section(s): I2C101-1

[Course Information]
Course No. : 00P102
Course Name: Object-Oriented Programming
Credits    : 3.5
Prerequisite Courses:
  Introduction to Computing
Offered As Section(s): 00P102-1

[Course Information]
Course No. : CJV103
Course Name: Core Java Programming
Credits    : 2
Prerequisite Courses:
  Introduction to Computing
  Object-Oriented Programming
Offered As Section(s): CJV103-1

[Person Information]
Name       : Greg Anderson
SS Number  : 123-45-7890
Professor-Specific Information:
  Title     : Adjunct Professor
  Department : Information Technology
Teaching Assignments for Greg Anderson:
  Introduction to Computing (M, 07:00-10:15 AM)

[Person Information]
Name       : Bjarne Stroustrup
SS Number  : 135-24-7908
Professor-Specific Information:
  Title     : Full Professor
  Department : Computer Science
Teaching Assignments for Bjarne Stroustrup:
  Object-Oriented Programming (T, 02:15-05:30 PM)

[Person Information]
Name       : James Gosling
SS Number  : 490-53-6281
Professor-Specific Information:
  Title     : Full Professor
  Department : Software Engineering
Teaching Assignments for James Gosling:
  Core Java Programming (F, 08:45-12:00 AM)

[Section Information]
Semester   : Summer 2012

```

Course No. : I2C101
Section No.: 1
Offered : M, 07:00-10:15 AM
In Room : 302
Professor : Greg Anderson
Total of 3 students enrolled, as follows:
Barack Obama
Francois Hollande
Vladymir Putin

[Section Information]

Semester : Summer 2012
Course No. : 00P102
Section No.: 1
Offered : T, 02:15-05:30 PM
In Room : 405
Professor : Bjarne Stroustrup
Total of 2 students enrolled, as follows:
Francois Hollande
Vladymir Putin

[Section Information]

Semester : Summer 2012
Course No. : CJV103
Section No.: 1
Offered : F, 08:45-12:00 AM
In Room : 507
Professor : James Gosling
Total of 1 students enrolled, as follows:
Francois Hollande

[Schedule Information]

Schedule for Summer 2012

F, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)
M, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)
T, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)

[Person Information]

Name : Barack Obama
SS Number : 123-12-4321

Student-Specific Information:

Major : IT
Degree : Ph.D.

Course Schedule for Barack Obama

M, 07:00-10:15 AM [Room: 302] Introduction to Computing (Greg Anderson)

Transcript for Barack Obama

[no entries]

[Person Information]

Name : Vladymir Putin
SS Number : 246-13-7985

Student-Specific Information:

Major : Math
Degree : M.S.

Course Schedule for Vladymir Putin

T, 02:15-05:30 PM [Room: 405] Object-Oriented Programming (Bjarne Stroustrup)

Transcript for Vladymir Putin

Introduction to Computing (A)

[Person Information]

Name : Francois Hollande
SS Number : 135-78-2460

Student-Specific Information:

Major : Physics
Degree : M.S.

Course Schedule for Francois Hollande

F, 08:45-12:00 AM [Room: 507] Core Java Programming (James Gosling)

Transcript for Francois Hollande

Introduction to Computing (B)
Object-Oriented Programming (A+)

Tài liệu tham khảo

(Theo năm xuất bản)

- [1] Ulla Kirch-Prinz, Peter Prinz – **A complete guide to programming in C++** – Jones and Bartlett Publishers, Inc., 2002. ISBN: 0-7637-1817-3
- [2] Ray Lischner – **C++ in a Nutshell** – O'Reilly Media, Inc., 2003. ISBN: 0-596-00298-X
- [3] Trevor Misfeldt, Gregor y Bumgardner, Andrew Gray – **The Elements of C++ Style** – Cambridge University Press, 2004. ISBN: 0-511-18532-4
- [4] Jesse Liberty, Bradley Jones – **Teach Yourself C++ in 21 Days, Fifth Edition** – Sams, 2005. ISBN: 0-672-32711-2
- [5] Marius Vasiliu – **Le language C++** – Pearson Education France, Inc., 2005. ISBN: 2-7440-7098-X
- [6] E Balagurusamy – **Object Oriented Programming with C++, Fourth Edition** – McGraw-Hill, 2008. ISBN 0-07-066907-9
- [7] Walter Savitch – **Problem Solving with C++, Seventh Edition** – Pearson Education, Inc., 2009. ISBN: 0-321-53134-5
- [8] D.S. Malik – **C++ Programming: From Problem Analysis to Program Design, Fifth Edition** – Cengage Learning, 2011. ISBN: 0-538-79808-4
- [9] Marc Gregoire, Nicholas A. Solter, Scott J. Kleper – **Professional C++, Second Edition** – John Wiley & Sons, Inc., 2011. ISBN 978-0-470-93244-5
- [10] Stephen Prata – **C++ Primer Plus, Sixth Edition** – Pearson Education, Inc., 2012. ISBN: 0-321-77640-2
- [11] Paul Deitel, Harvey Deitel – **C++: how to program, Eighth Edition** – Prentice Hall, 2012. ISBN: 978-0-13-266236-9
- [12] Gary Bronson – **A First Book of C++, Fourth Edition** – Cengage Learning, 2012. ISBN: 1-111-53100-5