BAN HỌC TẬP
CÔNG NGHỆ PHẦN MỀM

SỔ TAY KIẾN THỨC

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

BHTCNPM.COM
FACEBOOK.COM/BHTCNPM

LỜI GIỚI THIỆU

Xin chào các bạn sinh viên thân mến,

Sau những tháng ngày hoạt động và đồng hành cùng mọi người qua rất nhiều mùa thi, chúng mình nhận thấy mọi người cần một nguồn tư liệu ngắn gọn, dễ hiểu nhưng phải đầy đủ. Chính vì vậy Ban học tập Đoàn khoa Công nghệ Phần mềm đã bắt tay biên soạn cuốn sách này, sách sẽ gồm những phần như: khái quát nội dung, trọng tâm chương trình học và đề thi mẫu kèm lời giải chi tiết nhất.
Đây là dự án mà Ban học tập Đoàn khoa Công nghệ Phần mềm đã ấp ủ từ rất lâu. Và với phương châm: "Dễ hiểu nhất và tường tận nhất", chúng mình hy vọng rằng cuốn sách này sẽ là trợ thủ đắc lực nhất cho các bạn sinh viên UIT trong việc học tập và giúp các bạn đạt thành tích cao nhất trong các kì thi.

Sau những nỗ lực chúng mình đã hoàn thành sơ bộ môn Lập trình hướng đối tượng (Object Oriented Programming - OOP). Đây là một trong những kỹ thuật lập trình rất quan trọng và sử dụng nhiều hiện nay. Hầu hết các ngôn ngữ lập trình hiện này như Java, Ruby, Python… đều có hỗ trợ OOP. OOP giúp lập trình viên đặt ra mục tiêu quản lý source code giúp gia tăng khả năng tái sử dụng và quan trọng hơn hết là có thể tóm gọn được các thủ tục đã biết trước tính chất thông qua quá trình sử dụng các đối tượng.

Nếu sách có những điểm gì thắc mắc hãy liên hệ lại với chúng mình nhé!

Thông tin liên hệ của BHTCNPM:
Website: https://www.bhtcnpm.com/
Gmail: bht.cnpm.uit@gmail.com
Fanpage: https://www.facebook.com/bhtcnpm
Group BHT NNSC: https://www.facebook.com/groups/bht.cnpm.uit

Trân trọng cảm ơn các bạn đã quan tâm.

NGƯỜI BIÊN SOẠN

- 22520971 Lê Duy Nguyên
- 22520072 Phan Nguyễn Tuấn Anh
- 22521084 Hoàng Gia Phong
- 22521531 Nguyễn Lâm Thanh Triết
- 22520616 Ngô Hoàng Khang
- Các thành viên và CTV của Ban học tập Đoàn khoa Công nghệ Phần mềm

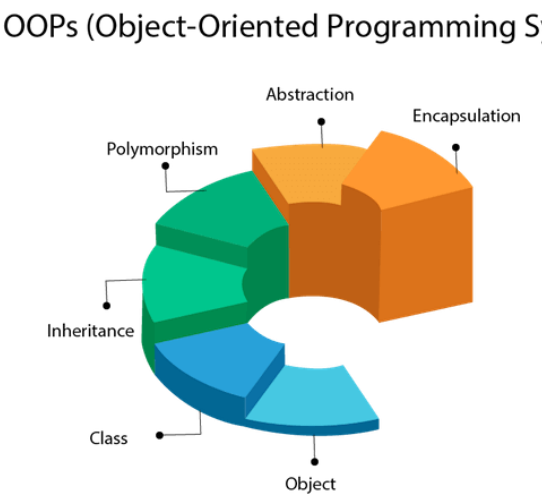
Mục lục

CHƯƠNG I: GIỚI THIỆU VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Lập trình hướng đối tượng (Object Oriented Programming - OOP) là một mô hình lập trình dựa trên khái niệm **Lớp (Class)** và **Đối tượng (Object)**. Phương pháp này được sử dụng để cấu trúc một chương trình thành các bản thiết kế đơn giản, tái sử dụng được (thường được gọi là Lớp), và qua đó dựa vào các lớp này để tạo lập các đối tượng.

Một số đặc tính, khái niệm cơ bản trong OOP:

- Lớp (Class) và Đối tượng (Object)
- Trừu tượng hóa dữ liệu (Data abstraction)
- Đóng gói (Encapsulation)
- Thừa kế (Inheritance)
- Đa hình (Polymorphism)



Hình 1: Các khái niệm, đặc tính trong OOP

Phương pháp lập trình hướng đối tượng giúp lập trình viên dễ dàng vận hành và thay đổi chương trình thông qua việc module hóa các đoạn code (bằng cách tạo ra các Lớp đối tượng), giúp bảo vệ dữ liệu và mô phỏng các khái niệm bên ngoài thế giới thực thông qua tính đóng gói và trừu tượng hóa, cũng như giúp tái sử dụng các đoạn code thông qua tính chất thừa kế.

Chúng ta sẽ đi sâu vào phần nội dung của từng khái niệm trong các chương ngay sau đây.

CHƯƠNG II: LỚP (CLASS), ĐỐI TƯỢNG (OBJECT) VÀ ĐẶC TÍNH ĐÓNG GÓI (ENCAPSULATION)

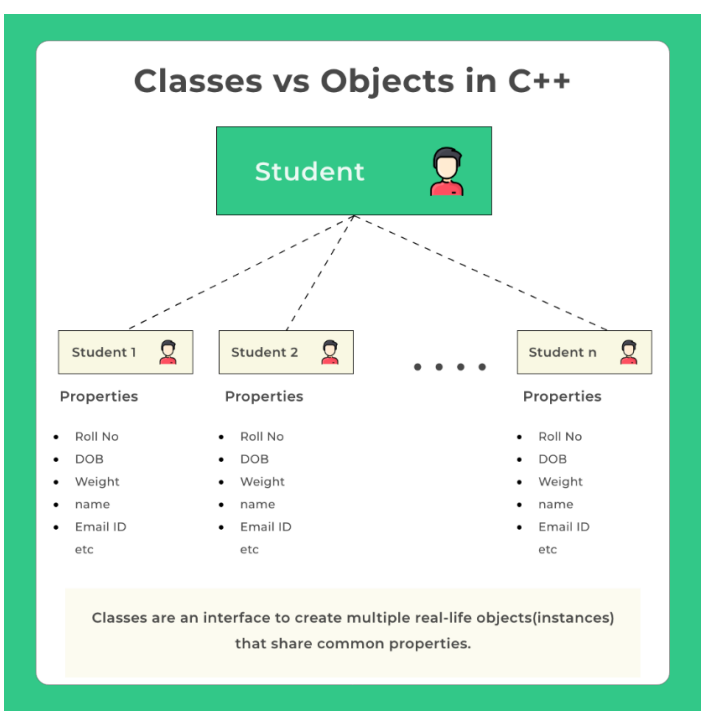
2.1. Khái niệm về Lớp (Class) và Đối tượng (Object)

Lớp đối tượng là đơn vị đóng gói cơ bản của C++, cung cấp cơ chế tạo ra một đối tượng. Có thể xem rằng **lớp là một khuôn mẫu** và đối tượng là một thực thể được thể hiện dựa trên khuôn mẫu đó. Hay nói cách khác, lớp là một mô tả trừu tượng của một nhóm các đối tượng có cùng bản chất, ngược lại, mỗi đối tượng là một thể hiện cụ thể cho những mô tả trừu tượng đó.

Một lớp đối tượng bao gồm 2 thành phần chính:

- **Thành phần dữ liệu (data member)**, hay còn được gọi là **thuộc tính (attribute)**.
- **Hàm thành phần (member function)**, còn có tên gọi khác là **phương thức (method),** là các hành động mà đối tượng của lớp có thể thực hiện.

Ví dụ, trong C++ ta có thể tự định nghĩa một lớp đối tượng có tên là `HocSinh` với các thuộc tính như mã số sinh viên, họ tên, điểm trung bình và các phương thức như đi, đứng, ngồi, học tập. Sau đó, ta có thể tạo ra các đối tượng khác nhau của lớp `HocSinh`, chẳng hạn như `hs1`, `hs2`, `hs3`, … với các giá trị khác nhau cho các thuộc tính và các hành vi khác nhau cho các phương thức.



Hình 2: Minh họa cho lớp và đối tượng

2.2. Khai báo và định nghĩa một Lớp đối tượng mới

Trong C++, để định nghĩa một lớp ta bắt đầu bằng từ khóa **class**, tiếp theo đó là tên của lớp và phần thân lớp được bao bởi cặp dấu **{}** tạo thành một phạm vi (scope). Việc định nghĩa được kết thúc bằng dấu chấm phẩy.

```
class TenLop {
    // Thành phần dữ liệu (thuộc tính).
    // Hàm thành phần (phương thức).
};
```

Cụ thể hơn, ta sẽ định nghĩa một lớp đối tượng có tên là **HocSinh** như sau:

```
class HocSinh {
private:
    int mssv;
    string hoTen;
    double diemToan;
    double diemVan;
    double diemTB;
    void XuLy();
public:
    void Nhap();
    void Xuat();
};
```

Ở ví dụ này, lớp đối tượng **HocSinh** có 5 thuộc tính bao gồm **msss**, **hoTen**, **diemToan**, **diemVan**, **diemTB** cùng với 3 phương thức là **XuLy**, **Nhap**, **Xuat**. Các thuộc tính và phương thức của một lớp đối tượng được khai báo giống như khi ta khai báo biến và hàm trong một chương trình.

Ngoài ra trong ví dụ trên ta còn sử dụng các từ khóa chỉ định phạm vi truy cập là **private** và **public**, phần nội dung này sẽ được nói ở các chương sau.

Khai báo một đối tượng:

Việc khai báo đối tượng của một lớp được thực hiện tương tự khi ta khai báo một biến bình thường.

Ví dụ 1: `HocSinh hs1;`

Trong ví dụ này, ta nói **hs1** là một đối tượng thuộc về lớp đối tượng HocSinh. Lớp HocSinh trong trường hợp này giống như một **kiểu dữ liệu** (do người lập trình tự định nghĩa) nên cũng có thể nói **hs1** là một biến có kiểu dữ liệu HocSinh.

Ví dụ 2: `HocSinh hs1,hs2,hs3;`

Ở ví dụ này, ta gọi **hs1**, **hs2**, **hs3** là ba đối tượng thuộc về lớp 1 đối tượng HocSinh. Hay nói cách khác, lớp đối tượng HocSinh có ba **sự thể hiện *khác nhau***. Lúc này ba biến **hs1**, **hs2**, **hs3** được cấp phát cho ba **vùng nhớ riêng biệt** và mỗi biến có thể giữ các giá trị khác nhau tương ứng với từng thuộc tính.

2.3. Hàm thành phần - Phương thức (Member function - Method)

2.3.1. Khái niệm

- Phương thức là các khả năng, thao tác mà một đối tượng thuộc về lớp có thể thực hiện.
- Về cơ bản, phương thức cũng không khác so với một hàm bình thường. Nó có thể có hoặc không có tham số và giá trị trả về.

2.3.2. Cách gọi phương thức

Để gọi một phương thức, ta sử dụng **toán tử chấm** (dot operator) trên một đối tượng của lớp hoặc **toán tử mũi tên** (arrow operator) lên một con trỏ giữ địa chỉ của đối tượng thuộc lớp đó, ví dụ:

```
HocSinh hs;
hs.Nhap();    // đối tượng hs gọi thực hiện phương thức Nhap
HocSinh* pHs = &hs;
pHs->Nhap(); // đối tượng mà pHs giữ địa chỉ
              // gọi thực hiện phương thức Nhap
```

Nói chung, toán tử chấm và toán tử mũi tên có thể được dùng để truy cập đến một thành viên bất kì của đối tượng khi đang trong phạm vi lớp, hoặc truy cập đến các thành viên `public` nếu ở ngoài phạm vi lớp (sẽ được nói kĩ hơn ở chương sau).

2.3.3. Định nghĩa phương thức

Trong ví dụ về lớp HocSinh ở trên ta chỉ mới khai báo các phương thức mà chưa định nghĩa chúng. Các phương thức của một lớp đối tượng phải được khai báo bên trong thân lớp, tuy nhiên việc định nghĩa có thể được thực hiện ở cả bên trong hoặc bên ngoài thân lớp.

Ta định nghĩa một phương thức bên trong thân lớp tương tự như khi định nghĩa một hàm bình thường, còn khi muốn định nghĩa một phương thức bên ngoài lớp, ta phải sử dụng **toán tử phạm vi** (scope operator – dấu `::`) để chương trình biết ta đang định nghĩa phương thức của lớp nào. Lúc này phần thân phương thức được xem như đang nằm trong phạm vi của lớp đó. Cú pháp:

```
KDL_trả_về Tên_lớp::Tên_phương_thức(<Tham_số>) {
// Thân phương thức
}
```

Ví dụ, phương thức `Xuly` và `Nhap` sẽ được định nghĩa bên ngoài lớp HocSinh như sau:

```
void HocSinh::XuLy() {
    diemTB = (diemToan + diemVan) / 2;
}

void HocSinh::Nhap() {
    cout << "Nhap MSSV: ";
    cin >> mssv;
    cout << "Nhap ho ten: ";
    cin.ignore();
    getline(cin, hoTen);
    cout << "Nhap diem toan: ";
    cin >> diemToan;
    cout << "Nhap diem van: ";
    cin >> diemVan;
    XuLy();
}
```

Tới đây có lẽ nhiều bạn đọc sẽ thắc mắc rằng các biến **`mssv`**, **`hoTen`**, **`diemToan`**, **`diemVan`**, **`diemTB`** ở đâu ra trong khi các phương thức ở trên không có tham số đầu vào. Để giải thích điều này ta sẽ tìm hiểu về khái niệm con trỏ `this`.

2.3.4. Giới thiệu về con trỏ "this"

Trước hết, hãy cùng ôn lại một số kiến thức về con trỏ trong C++:

- **Ghi nhớ: Miền giá trị của một biến con trỏ là địa chỉ ô nhớ**
- Trong câu lệnh `HocSinh* p = &hs;` ta nói **p** là một biến con trỏ kiểu `HocSinh`, địa chỉ của đối tượng **hs** thuộc lớp `HocSinh` được gán cho biến con trỏ **p**.
- Con trỏ hằng (constant pointer) là một con trỏ mà địa chỉ nó đang giữ không thể bị thay đổi, ví dụ:

  ```
  HocSinh* const p = &hs
  ```

  Trong ví dụ này, **p** là một con trỏ hằng, **p** sẽ giữ địa chỉ của đối tượng **hs** trong suốt quá trình tồn tại của mình, và ta không thể thay đổi giá trị của **p** (là địa chỉ ô nhớ).

Trở lại vấn đề, con trỏ `this` là một **con trỏ hằng** được chương trình **tự định nghĩa** bên trong một phương thức, nó sẽ giữ và chỉ có thể **giữ địa chỉ của đối tượng đang gọi thực hiện phương thức** đó. Vì thế, con trỏ `this` luôn có kiểu trùng với kiểu của lớp đối tượng mà nó thuộc về.

Nhìn lại một ví dụ về việc gọi phương thức:

```
hs.Nhap(); // đối tượng hs gọi thực hiện phương thức Nhap
```

Khi dòng lệnh trên được thực hiện, chương trình sẽ tự động gán địa chỉ của đối tượng **hs** vào biến con trỏ `this` (được định nghĩa ngầm bên trong phương thức **Nhap**). Sau đó, ta có thể dùng con trỏ này để truy cập đến các thuộc tính của đối tượng **hs**, cũng như dùng nó để gọi các phương thức khác, ví dụ:

```
void HocSinh::Nhap() {
    // ...
    cin >> this->mssv; // nhập MSSV của hs
    // ...
    this->XuLy(); // gọi Xuly() trên đối tượng hs
}
```

Tuy nhiên để cho gọn, chương trình cho phép ta truy cập trực tiếp đến các thành viên của đối tượng đang gọi thực hiện phương thức. Bất kì tên của thành viên nào được ghi ra mà không nói gì thêm thì thành viên đó sẽ xem như là được truy cập thông qua con trỏ `this`.

2.4. Trừu tượng hóa dữ liệu (Data abstraction) và Đóng gói (Encapsulation)

Một trong những ý tưởng cơ bản đằng sau việc xây dựng và thiết kế một Lớp đối tượng chính là **Trừu tượng hóa dữ liệu (Data abstraction)** và **Đóng gói (Encapsulation).**

Trừu tượng hóa dữ liệu là một kỹ thuật lập trình và thiết kế dựa trên sự tách biệt của **Giao diện (Interface)** và **Thực thi (Implementation)**. Giao diện của một Lớp đối tượng là các hoạt động mà người dùng [1]của một Lớp có thể thao tác trên các đối tượng của Lớp đó. Phần Thực thi bao gồm các dữ liệu thành viên (thuộc tính), phần định nghĩa của các phương thức.

Đóng gói chính là quá trình ẩn đi phần Thực thi khỏi người dùng bên ngoài và giới hạn quyền truy cập vào nó. Người dùng của một Lớp chỉ có thể sử dụng Giao diện mà không có quyền truy cập vào phần Thực thi.

Một Lớp đối tượng được áp dụng đặc tính Trừu tượng hóa dữ liệu và Đóng gói sẽ tạo thành một kiểu dữ liệu trừu tượng mô phỏng lại một khái niệm bên ngoài thế giới thực. Những lập trình viên sử dụng Lớp chỉ cần biết một cách *trừu tượng* rằng đối tượng của Lớp có thể thực hiện các hoạt động gì mà không cần hiểu cách thức thực hiện bên trong.

2.5. Phạm vi truy xuất

Trong C++, chúng ta thực hiện việc đóng gói bằng cách chỉ định phạm vi truy xuất (access specifiers):

- Những thành phần được khai báo sau từ khóa `public` có thể được truy cập ở **tất cả các phần của chương trình**. Các thành phần `public` tạo nên giao diện của một Lớp.
- Những thành phần được khai báo sau từ khóa `private` chỉ có thể được truy cập **bên trong phạm vi của lớp**, bởi các hàm thành viên (phương thức) của một lớp và không thể được truy cập từ bên ngoài lớp. Phần `private` ẩn đi (đóng gói) phần thực thi.

Ngoài ra còn một loại phạm vi truy xuất nữa là `protected` sẽ được nói ở phần kế thừa.

Nhìn lại ví dụ về lớp `HocSinh` ở phần trước:

```
class HocSinh {
private:
    int mssv;
    string hoTen;
    double diemToan;
```

[1] Người dùng ở đây là các lập trình viên sử dụng Lớp đối tượng mà chúng ta đang thiết kế.

```
        double diemVan;
        double diemTB;
        void XuLy();
    public:
        void Nhap();
        void Xuat();
    };
```

Các phương thức **Nhap** và **Xuat** được khai báo sau từ khóa **public** tạo nên phần giao diện. Các thuộc tính như **mssv,hoTen,diemToan,diemVan,diemTB** và phương thức **XuLy** được khai báo sau từ khóa **private**, cùng với phần định nghĩa của các phương thức tạo nên phần thực thi của lớp **HocSinh**.

Ngoài ra, các thành viên đã được đóng gói (gán nhãn **private**) thì sẽ không thể truy cập được từ bên ngoài lớp. Trong ví dụ trên, ta chỉ có thể gọi phương thức **Nhap** và **Xuat** mà không thể gọi phương thức **Xuly** hay truy cập các thuộc tính của lớp **HocSinh** khi ở ngoài phạm vi lớp:

```
    // ... khai báo lớp, thư viện, không gian tên
    int main() {
        HocSinh hs;
        hs.Nhap(); // Đúng
        hs.Xuat(); // Đúng
        hs.hoTen; // Sai
        hs.XuLy(); // Sai
        return 0;
    }
```

Lưu ý: Sự khác biệt giữa từ khóa `struct` và `class`:

Trong một lớp có thể không có hoặc có nhiều nhãn **private** và **public**, mỗi nhãn này có phạm vi ảnh hưởng cho đến khi gặp một nhãn kế tiếp hoặc hết khai báo lớp.

Nếu khai báo một Lớp sử dụng từ khóa **struct**, những thành phần được khai báo trước nhãn truy cập đầu tiên sẽ được mặc định là **public**, nếu sử dụng từ khóa **class**, những thành phần đó sẽ mặc định là **private**:

```
class HocSinh {
    int mssv;
    string hoTen; // MSSV và HoTen được xem như private
private:
    double diemToan;
    double diemVan;
    // ...
    // những thành phần còn lại như ví dụ ở trên
};

struct HocSinh {
    int mssv;
    string hoTen; // MSSV và HoTen được xem như public
private:
    double diemToan;
    double diemVan;
    // ...
    // những thành phần còn lại như ví dụ ở trên
};
```

Vì vậy khi khai báo một Lớp mà không chỉ định bất kì phạm vi truy xuất nào, thì tất cả các thuộc tính và phương thức sẽ mặc định là `public` nếu sử dụng `struct`, và là `private` nếu sử dụng `class`. Lưu ý rằng đây cũng là điểm khác biệt *duy nhất* giữa `struct` và `class`.

2.6. Phương thức truy vấn và cập nhật

Vì các thuộc tính của một đối tượng được đóng gói thì không thể được truy cập từ bên ngoài, ta phải định nghĩa các phương thức dùng để truy cập và thay đổi dữ liệu của đối tượng đó:

- Phương thức truy vấn được sử dụng để lấy giá trị của một thuộc tính `private`
- Phương thức cập nhật dùng để thay đổi giá trị của một thuộc tính `private`

Trong ví dụ về lớp `HocSinh`, ta sẽ định nghĩa một phương thức truy vấn có tên là **`getDiemToan`** và một phương thức cập nhật có tên là **`setDiemToan`** như sau (ở đây xem như đã có khai báo cho hai hàm này bên trong thân lớp):

```
double HocSinh::getDiemToan() {
    return diemToan;
}
void HocSinh::setDiemToan(double toan) {
    if (toan > 10 || toan < 0) {
        cout << "Diem toan khong duoc lon hon 10 hoac be hon 0"
             << endl;
        return;
    }
    this->diemToan = toan;
    this->XuLy(); // tính lại diemTB
}
```

Phương thức cập nhật giúp ta thay đổi dữ liệu bên trong của của một đối tượng mà vẫn đảm bảo được tính đóng gói. Ở ví dụ trên, trong hàm **`setDiemToan`**, trước khi thực hiện việc cập nhật điểm toán, ta kiểm tra xem đối số được đưa vào có thỏa mãn điều kiện hay không, nếu không thỏa mãn, chương trình sẽ báo lỗi và không thực hiện thay đổi nào.

Lưu ý: Các lợi ích khi áp dụng đặc tính Đóng gói:

- Giúp **bảo vệ dữ liệu** bên trong tránh khỏi các sai sót không đáng có từ người dùng (như ví dụ về phương thức cập nhật ở trên).
- Giúp **thay đổi phần thực thi** của lớp một cách **linh hoạt** (tức là thay đổi cách tổ chức dữ liệu, chi tiết thực hiện các phương thức). Miễn là phần giao diện không bị thay đổi thì những đoạn code sử dụng lớp sẽ không bị ảnh hưởng, do đó không làm đổ vỡ kiến trúc hệ thống.

2.7. Phương thức thiết lập (Constructor)

a) Mục tiêu

- Các phương thức thiết lập của một lớp đối tượng có nhiệm vụ thiết lập thông tin ban đầu cho các đối tượng thuộc về lớp sau khi đối tượng được khai báo.

b) Các đặc điểm

- Phương thức thiết lập là một hàm thành viên đặc biệt có tên trùng với tên lớp.
- Không có giá trị trả về.
- Được tự động gọi thực hiện ngay khi đối tượng được khai báo.

- Có thể có nhiều phương thức thiết lập trong 1 lớp.
- Trong một quá trình sống của đối tượng thì chỉ có 1 lần duy nhất một phương thức thiết lập được gọi thực hiện mà thôi đó là khi đối tượng ra đời.

c) Phân loại phương thức thiết lập.

- Phương thức thiết lập mặc định (default constructor).
- Phương thức thiết lập nhận tham số đầu vào (parameterized constructor).
- Phương thức thiết lập sao chép (copy constructor).

2.7.1. Phương thức thiết lập mặc định (default constructor)

Trong các ví dụ về lớp `HocSinh` ở trên, mặc dù chúng ta chưa định nghĩa bất kì phương thức thiết lập nào cho lớp, chương trình sử dụng lớp `HocSinh` vẫn có thể chạy một cách bình thường. Ví dụ, trong hàm main, khi chúng ta khai báo:

```
HocSinh hs;
```

Biến `hs` lúc này sẽ được khởi tạo mặc định bằng **phương thức thiết lập mặc định**. Phương thức thiết lập mặc định ***không có tham số đầu vào*** và được chương trình tự định nghĩa khi người thiết kế lớp không định nghĩa bất kì phương thức thiết lập nào cho lớp.

Khi thực hiện lệnh `hs.Xuat()`, ta sẽ nhận được kết quả như sau:

```
-858993460

-9.25596e+61
-9.25596e+61
-9.25596e+61
```

Lúc này phương thức thiết lập mặc định do chương trình tự định nghĩa cho các thuộc tính `mssv,diemToan,diemVan,diemTB` nhận giá trị ngẫu nhiên, còn `hoTen` nhận giá trị là một chuỗi rỗng.

Ngoài ra, chúng ta có thể tự định nghĩa một phương thức thiết lập mặc định của riêng mình bên trong thân của lớp `HocSinh` như sau:

```
HocSinh() {
    cout << "Default constructor of HocSinh has been called"
         << endl;
    mssv = 0;
```

```
        hoTen = "Unknown";
        diemToan = 0.0;
        diemVan = 0.0;
        diemTB = 0.0;
    }
```

Để ý ở đây, tên hàm là **HocSinh** trùng với tên lớp, không có giá trị trả về, và vì là phương thức thiết lập mặc định nên sẽ không có tham số đầu vào.

Khi đó, nếu chúng ta khai báo

```
    HocSinh hs;
```

thì chương trình sẽ dùng phương thức thiết lập mặc định do chúng ta vừa định nghĩa để khởi tạo cho đối tượng **hs**, và đây là kết quả khi xuất **hs** ra màn hình:

```
Default constructor of HocSinh has been called
0
Unknown
0
0
0
```

Lưu ý:

- Nếu chúng ta có định nghĩa các phương thức thiết lập khác thì chương trình sẽ không tự định nghĩa phương thức thiết lập mặc định cho chúng ta, do đó ta phải tự định nghĩa một phiên bản của riêng mình như ví dụ ở trên.
- Phương thức thiết lập mặc định còn được chương trình tự gọi khi ta khai báo một mảng các đối tượng của một lớp như các cách sau:

  ```
  HocSinh arr[5];
  HocSinh* arr = new HocSinh[5];
  ```

  Khi chạy chương trình, ta thấy câu thông báo được xuất ra 5 lần, chứng tỏ phương thức thiết lập mặc định đã được gọi trên 5 phần tử của mảng **arr**:

```
Default constructor of HocSinh has been called
Default constructor of HocSinh has been called
Default constructor of HocSinh has been called
Default constructor of HocSinh has been called
Default constructor of HocSinh has been called
```

2.7.2. Phương thức thiết lập nhận tham số đầu vào (parameterized constructors)

Là các phương thức thiết lập sử dụng các đối số được truyền vào nó để khởi tạo dữ liệu cho các thuộc tính của đối tượng.

Tiếp tục với ví dụ về lớp HocSinh, ta sẽ định nghĩa bên trong thân lớp một phương thức thiết lập nhận 4 tham số đầu vào lần lượt là mã số sinh viên, họ tên, điểm toán, điểm văn như sau:

```
    HocSinh(int id, string name, int toan, int van) {
        mssv = id;
        hoTen = name;
        diemToan = toan;
        diemVan = van;
        XuLy(); // tính diemTB
    }
```

Để gọi phương thức thiết lập vừa được định nghĩa ở trên, ta sẽ khai báo đối tượng **hs** như dưới đây:

```
    HocSinh hs(22520971, "Le Duy Nguyen", 8, 7);
```

Trong cùng một chương trình đó, ta cũng có thể định nghĩa thêm nhiều phương thức thiết lập khác miễn là chúng có danh sách tham số đầu vào khác nhau, ví dụ ở đây chúng sẽ định nghĩa thêm một phương thức thiết lập nhận 2 tham số đầu vào là mã số sinh viên và họ tên:

```
    HocSinh(int id, string name) {
        mssv = id;
        hoTen = name;
        diemToan = 0.0;
        diemVan = 0.0;
        XuLy(); // tính diemTB
    }
```

2.7.3. Phương thức thiết lập sao chép (copy constructor)

Trước khi đến với khái niệm phương thức thiết lập sao chép, chúng ta sẽ nhắc sơ lại về khái niệm **tham chiếu (reference)** trong C++:

- Tham chiếu (hay tham biến) là một ***cái tên khác*** cho một đối tượng.
- Tham chiếu được khai báo bằng cách thêm kí tự **'&'** vào trước tên biến, ví dụ:

```
HocSinh &r = hs;
```

Khi khai báo một tham chiếu `r` như trên, chương trình sẽ ***không sao chép*** giá trị của `hs` vào `r` mà chỉ xem `r` như là một **cái tên khác** của đối tượng `hs`.

- Tham chiếu hằng: một tham chiếu mà không thể dùng để thay đổi giá trị của đối tượng nó được gắn vào, ví dụ:

```
const HocSinh &r = hs;
```

- Một tham chiếu bình thường không thể được gắn với một biến hằng, một tham chiếu hằng có thể được gắn với một biến hằng lẫn biến bình thường.

Trở lại với vấn đề chính, **phương thức thiết lập sao chép** của một lớp đối tượng là phương thức thiết lập có 1 **tham số đầu vào là tham chiếu** tới một đối tượng của lớp đó. Mục đích của phương thức này là để sao chép dữ liệu của một đối tượng vào một đối tượng khác vừa được khai báo.

Vấn đề tại sao tham số đầu vào phải là tham chiếu sẽ được giải thích sau một lát nữa, trước hết chúng ta sẽ xem qua ví dụ về việc định nghĩa một phương thức thiết lập sao chép trong lớp `HocSinh`:

```
HocSinh(const HocSinh& temp) {
    cout << "Copy constructor of HocSinh has been called"<<endl;
    mssv = temp.mssv;
    hoTen = temp.hoTen;
    diemToan = temp.diemToan;
    diemVan = temp.diemVan;
    diemTB = temp.diemTB;
}
```

Trong chương trình, ta gọi thực hiện phương thức sao chép bằng cách khai báo:

```
HocSinh hs2(hs);
```

Hoặc:

```
HocSinh hs2 = hs;
```

Lúc này chương trình sẽ tự động thực hiện dòng lệnh sau:

```
const HocSinh &temp = hs;
```

Chương trình tới đây sẽ **không sao chép dữ liệu** của `hs` vào `temp` mà chỉ xem `temp` như là một **cái tên khác** của `hs`, những thao tác lúc này được thực hiện bên trong thân phương thức ở trên chính là gán các giá trị của `hs` cho `hs2`.

Tới đây bạn đọc có thể nhận thấy rằng nếu `temp` không được khai báo là tham chiếu mà chỉ là một biến bình thường thì chương trình sẽ ngầm thực hiện dòng lệnh:

```
    const HocSinh temp = hs;
```

Lúc này, trong quá trình thực hiện phương thức thiết lập sao chép để sao chép `hs` vào `hs2`, chương trình lại phải gọi thêm một phương thức thiết lập sao chép khác để sao chép `hs` vào `temp`, và cứ như vậy tạo thành một **vòng lặp vô hạn**.

Trong phương thức trên, ta khai báo tham chiếu `temp` là hằng để đảm bảo đối số truyền vào **không thể bị sửa đổi** một cách vô ý, cũng như đảm bảo rằng có thể truyền vào phương thức một **đối số hằng**.

***Một số trường hợp khác mà phương thức thiết lập sao chép được gọi thực hiện:**

- Khi truyền một đối số vào lời gọi hàm của một hàm có tham số tương ứng không phải là tham chiếu (như ví dụ ở trên).
- Khi kết thúc lời gọi hàm, trả về một đối tượng mà kiểu dữ liệu trả về của hàm không phải tham chiếu.
- Khi khởi tạo các phần tử của một mảng sử dụng dấu ngoặc nhọn, ví dụ:

  ```
  HocSinh arr[2] = {hs};
  ```

  Lúc này chương trình gọi phương thức thiết lập sao chép để sao chép `hs` vào phần tử đầu tiên của mảng, và gọi thực hiện phương thức thiết lập mặc định để khởi tạo giá trị cho phần tử thứ 2.

  Khi chạy chương trình, sẽ thấy có dòng thông báo sau xuất hiện

  ```
  Copy constructor of HocSinh has been called
  Default constructor of HocSinh has been called
  ```

Một điều cần lưu ý ở đây là nếu chúng ta không tự tạo một phương thức thiết lập sao chép của riêng mình, chương trình sẽ cũng sẽ tự định nghĩa cho ta một phương thức thiết lập sao chép gần giống với ví dụ ở trên. Tuy nhiên, cần phải nhận thức được rằng phương thức thiết lập sao chép do chương trình tự định nghĩa không phải lúc nào cũng sẽ hoạt động như ý chúng ta muốn. Ta sẽ bắt gặp một vài ví dụ về vấn đề này ở các chương sau.

2.8. Phương thức phá hủy (Destructor)

a) Mục đích.

- Thông thường, phương thức phá hủy có nhiệm vụ thu hồi lại tất cả các tài nguyên đã cấp phát cho đối tượng khi đối tượng hết phạm vi hoạt động (scope).

b) Đặc điểm.

- Tên phương thức trùng với tên lớp nhưng có dấu ngã ở đằng trước.
- Không có giá trị trả về.
- Không có tham số đầu vào.
- Được tự động gọi thực hiện *trước khi* đối tượng bị hủy.
- Có và chỉ có duy nhất một phương thức phá huỷ trong 1 lớp.
- Trong một quá trình sống của đối tượng có và chỉ có một lần phương thức phá hủy được gọi thực hiện mà thôi.

c) Ý nghĩa

- Một cách dùng của phương thức phá hủy là để giải phóng bộ nhớ của các thuộc tính được cấp phát động trong một đối tượng. Nếu chúng ta không giải phóng các vùng nhớ này, nó sẽ bị tồn đọng lại và chiếm không gian không cần thiết.

Trước khi đến với các ví dụ về destructor, ta sẽ ôn lại một chút về cấp phát động:

Nhắc lại về cấp phát động:

- **Ghi nhớ: Miền giá trị của một biến con trỏ là địa chỉ ô nhớ**
- Trong câu lệnh `HocSinh* arr;` ta nói `arr` là một biến con trỏ kiểu `HocSinh`.
- Về bản chất, **tên của một mảng là một con trỏ** giữ địa chỉ của phần tử đầu tiên trong mảng.
- `HocSinh* arr = new HocSinh[n];` có nghĩa là xin cấp phát một vùng nhớ có kích thước bằng kích thước của **n** kiểu `HocSinh`, nếu cấp phát thành công, giá trị của biến con trỏ `arr` sẽ là **địa chỉ ô nhớ** đầu tiên của vùng nhớ được cấp phát.
- Lúc này `arr` được xem như là một mảng `HocSinh` có **n** phần tử.

- Với cách khai báo bằng cấp phát động như vậy, các phần tử của mảng **arr** sẽ được lưu trong **vùng nhớ heap.**[2]
- Đối với các đối tượng được cấp phát động, sau khi sử dụng xong phải thu hồi bộ nhớ đã cấp phát bằng toán tử `delete`.

2.8.1. Phương thức phá hủy và cấp phát động

Ta sẽ cho ví dụ về một lớp đối tượng có tên là `LopHoc` để minh họa cho một trường hợp cần sử dụng phương thức phá hủy:

```
class LopHoc {
private:
    HocSinh* arr;
    int size;
public:
    LopHoc(int size) {
        this->size = size;
        arr = new HocSinh[size];
    }
};
```

Lớp đối tượng `LopHoc` có hai thành phần dữ liệu là con trỏ **arr** kiểu `HocSinh` tượng trưng cho mảng các `HocSinh` và biến số nguyên **size** tượng trưng cho sỉ số lớp. `LopHoc` có một phương thức thiết lập nhận 1 tham số đầu vào là sỉ số của lớp, ví dụ cho chương trình sau đây:

```
// ... khai báo thư viện, lớp, không gian tên
int main() {
    LopHoc lop(70);
    return 0;
}
```

Thành phần **arr** của đối tượng **lop** lúc này được cấp phát cho một vùng nhớ có kích thước gấp 70 lần kích thước kiểu `HocSinh`. Ở đây, khi chương trình kết thúc, đối tượng **lop** sẽ bị

[2] Bạn đọc có thể tìm hiểu thêm về bộ nhớ heap trong link sau đây: https://codelearn.io/sharing/heap-va-stack-khac-biet-den-nhu-the-nao

phá hủy, kéo theo các dữ liệu thành phần là **arr** và **size** sẽ bị phá hủy theo, tuy nhiên vùng nhớ được cấp phát cho biến con trỏ **arr** vẫn còn đó mà chưa được thu hồi.

Để giải quyết vấn đề này, ta sẽ định nghĩa cho lớp LopHoc một phương thức phá hủy như sau:

```
~LopHoc() {
    cout << "Destructor has been called" << endl;
    delete[] arr;
}
```

Lúc này, trước khi đối tượng **lop** bị hủy, chương trình sẽ tự động gọi thực hiện phương thức phá hủy vừa được định nghĩa ở trên để thu hồi bộ nhớ được cấp phát cho thành phần dữ liệu **arr**, rồi sau đó mới thực hiện việc hủy đối tượng.

Nếu chúng ta không tự định nghĩa một phương thức phá hủy, chương trình cũng sẽ tự định cho ta một phiên bản như đây:

```
~LopHoc() { }
```

Phương thức phá hủy này không thực hiện bất cứ thao tác gì nên có thân hàm rỗng. Đối với những lớp đối tượng có thành phần được khai báo tĩnh thì có thể sử dụng phiên bản do chương trình tự định nghĩa mà không cần phải tự định nghĩa một phiên bản riêng.

Một số trường hợp khác mà phương thức phá hủy được gọi:

- Đối tượng bị hủy khi ra khỏi phạm vi hoạt động.
- Đối tượng được cấp phát động bị hủy khi sử dụng toán tử delete lên biến con trỏ trỏ vào nó, ví dụ:

```
HocSinh* hsPtr = new HocSinh();
delete hsPtr; // Phương thức phá hủy của lớp HocSinh được gọi
```

- Các dữ liệu thành viên bị hủy khi đối tượng chúng thuộc về bị hủy.

2.8.2. Phương thức phá hủy và phương thức thiết lập sao chép

Khi thiết kế các lớp đối tượng, có một quy luật là nếu một lớp cần phải tự định nghĩa phương thức phá hủy, thì lớp đó cũng cần tự định nghĩa một phương thức thiết lập sao chép của riêng mình.

Trong ví dụ về lớp LopHoc ở trên, nếu không làm gì thêm, chương trình sẽ tự định nghĩa cho ta một phương thức thiết lập sao chép như sau:

```
1.   LopHoc(const LopHoc& temp) {
2.       this->arr = temp.arr;//sau bước này this->arr và temp.arr
3.                            //cùng nắm giữ một vùng nhớ
4.       this->size = temp.size;
5.   }
```

Tại dòng số 2, địa chỉ mà biến-con-trỏ-**arr**-thuộc-đối-tượng-temp đang nắm giữ được sao chép vào biến con trỏ **arr** của đối tượng đang thực hiện lời gọi phương thức. Lúc này hai con trỏ có giá trị bằng nhau, tức là chúng đang **cùng nắm giữ một vùng nhớ**.

Khi thực hiện đoạn chương trình sau đây:

```
1.   LopHoc item1(5);
2.   {
3.       LopHoc item2 = item1; // Copy constructor do chương trình
4.                             // tự định nghĩa được gọi.
5.   }                          // Destructor được gọi trên item2.
6.   item1.Xuat();              // lỗi! item1 bây giờ bị mất vùng nhớ.
```

Trong phạm vi từ dòng số 2 đến dòng số 5, **item2** được khởi tạo bằng copy constructor do chương trình tự định nghĩa, sao chép dữ liệu từ **item1**. Khi **item2** ra khỏi phạm vi, destructor của lớp LopHoc được gọi trên **item2** thu hồi vùng nhớ được cấp phát cho biến này nhưng cũng "vô tình" thu hồi vùng nhớ được cấp phát cho **item1** vì thành phần **arr** trong hai đối tượng này đang cùng sở hữu một vùng nhớ.

Để tránh lỗi trên, ta cần phải định nghĩa cho lớp LopHoc một phương thức thiết lập sao chép như dưới đây:

```
LopHoc(const LopHoc& temp) {
    size = temp.size;
    arr = new HocSinh[size];
    for (int i = 0; i < size; ++i) {
        arr[i] = temp.arr[i];
    }
}
```

Lúc này, thành phần **arr** trong 2 biến **item1** và **item2** sẽ giữ địa chỉ của 2 vùng nhớ khác nhau, các thao tác trên đối tượng này sẽ không ảnh hưởng tới đối tượng kia.

Ví dụ trên cho thấy không phải lúc nào các phương thức do chương trình tự định nghĩa cũng hoạt động như ý chúng ta mong muốn, nên ta cần phải chú ý khi nào thì nên tự định nghĩa các phương thức thiết lập của riêng mình.

Trong thực tế, việc quản lí bộ nhớ cho các đối tượng được cấp phát động thường gây khó khăn và dễ phát sinh ra lỗi, vì thế ngôn ngữ C++ có hỗ trợ cho ta các thư viện và lớp đối tượng được cài đặt sẵn như `vector` (có công dụng như một mảng động), `shared_ptr` (con trỏ tự thu hồi bộ nhớ), … để việc lập trình trở nên dễ dàng hơn.

2.9. Thành phần tĩnh (Static member)

Các thành phần tĩnh là các thành phần thuộc về cả một lớp chứ **không thuộc về một đối tượng** cụ thể. Điều này có nghĩa là tất cả các đối tượng của một lớp đều chia sẻ chung một thành phần tĩnh, và nó có thể được truy cập mà không cần thông qua một đối tượng.

Có hai loại thành phần tĩnh:

- Các thuộc tính (dữ liệu thành viên) tĩnh (Static data member)
- Hàm thành viên (phương thức) tĩnh (Static member function)

Một ví dụ về trường hợp cần sử dụng thành phần tĩnh là khi ta cần biết có bao nhiêu đối tượng của một lớp đã được tạo, lúc này lớp đó cần có một thuộc tính tĩnh gọi là biến đếm. Để ý rằng biến đếm ở đây thuộc về cả lớp đối tượng đó chứ không phải là thuộc tính của một đối tượng cụ thể, vì nó đại diện cho số đối tượng đã được tạo.

2.9.1. Khởi tạo thành viên tĩnh

Muốn khởi tạo một thành viên tĩnh chúng ta sẽ thêm từ khóa `static` vào trước dòng khai báo của nó. Như các thành viên khác thì thành viên tĩnh cũng có thể được khai báo như là một thành phần `private`, `public` hay là `protected`.

Ví dụ , chúng ta sẽ thêm vào lớp `HocSinh` một thuộc tính tĩnh là **`demHS`** để đếm số lượng học sinh đã được tạo, và một hàm thành viên tĩnh tên là **`getDemHS`** để lấy giá trị của **`demHS`**:

```
class HocSinh {
private:
    static int demHS;       // Biến static
public:
    static int getDemHS(); // Hàm static
    // ... phần còn lại như các ví dụ trên
};
```

Thuộc tính tĩnh **không phải là thuộc tính của một đối tượng** nào cả, vì vậy mỗi đối tượng thuộc lớp `HocSinh` được tạo ra sẽ chỉ có 5 thuộc tính là: **`mssv,hoTen,diemToan, diemVan,diemTB`** mà không có thuộc tính **`demHS`**. Thêm vào đó, biến **`demHS`** được tạo ra ngay cả khi lớp chưa có một đối tượng nào, chỉ có **một phiên bản** của nó tồn tại từ đầu cho tới cuối chương trình và được các đối tượng của lớp `HocSinh` sử dụng.

Tương tự như vậy, một hàm thành viên tĩnh **không gắn với bất kì đối tượng nào** (không tồn tại con trỏ `this` bên trong một hàm thành viên tĩnh) nên ta không thể truy cập đến các thuộc tính non-static của lớp bên trong phương thức tĩnh.

2.9.2. Cách gọi các thành viên tĩnh

Chúng ta có thể truy cập trực tiếp một thành viên tĩnh thông qua toán tử phạm vi, ví dụ:

```
int dem = HocSinh::getDemHS();
// gọi hàm thành viên tĩnh thông qua toán tử phạm vi
```

Mặc dù các thành viên tĩnh không phải là một phần của đối tượng, chúng ta vẫn có thể dùng các đối tượng của một lớp để truy cập đến các thành phần tĩnh của lớp đó:

```
HocSinh h1;
int dem = h1.getDemHS();
```

Các hàm thành viên có thể truy cập trực tiếp đến các thành phần tĩnh mà không cần toán tử phạm vi. Trong ví dụ về lớp `HocSinh`, khi một đối tượng `HocSinh` được tạo ra thì các phương thức thiết lập sẽ làm thêm một việc đó là tăng giá trị của **`demHS`** lên một:

```
// Phương thức thiết lập mặc định:
HocSinh() {
    // ... khởi tạo các giá trị mặc định
    demHS++;
    // truy cập đến demHS mà không cần toán tử phạm vi
}
```

2.9.3. Định nghĩa thành viên tĩnh

Đối với các hàm thành viên tĩnh, chúng có thể được định nghĩa bên trong hoặc ngoài thân của lớp như các phương thức khác. Khi chúng ta định nghĩa ở bên ngoài thì không dùng từ khóa `static` (Lưu ý: Phương thức tĩnh chỉ có thể truy cập đến các thuộc tính tĩnh)

```
// định nghĩa hàm getdemHS
int HocSinh::getDemHS() {
    return demHS;
}
```

Bởi vì các thuộc tính tĩnh không phải là một phần của một đối tượng, chúng **không được khởi tạo** khi gọi phương thức thiết lập trên các đối tượng của lớp. Vì thế nên trong đa số các trường hợp, ta sẽ khởi tạo giá trị cho các thuộc tính tĩnh **bên ngoài thân lớp**. Mỗi dữ liệu thành viên tĩnh chỉ được định nghĩa một lần. Như biến toàn cục , dữ liệu thành viên tĩnh được định nghĩa bên ngoài tất cả các hàm, do đó một khi được định nghĩa nó sẽ tồn tại cho đến khi chương trình kết thúc:

```
class HocSinh {
private:
    // khai báo thuộc tính tĩnh demHS
    static int demHS;
    // ... phần còn lại như các ví dụ trên
};
// định nghĩa và khởi tạo demHS
int HocSinh::demHS = 0;
```

2.10. Hàm bạn, lớp bạn (Friends)

2.10.1. Hàm bạn

Hàm bạn là hàm có thể truy cập thành phần `private` hoặc `protected` của lớp xem nó là bạn.

Giả sử mình có lớp `MyClass1` và hàm `myFunc` như sau:

```
class MyClass1 {
private:
    int tpRiengTu;
};

void myFunc(MyClass1 mClass) {
    cout << mClass.tpRiengTu; // Không hợp lệ
}
```

Như bạn thấy trong đoạn code trên, dòng in ra giá trị thuộc tính `tpRiengTu` là không hợp lệ do nó là thuộc tính `private` và không được phép truy cập từ bên ngoài. Tuy nhiên mọi chuyện sẽ khác nếu hàm `myFunc` là hàm bạn của lớp `MyClass1`.

Cú pháp khai báo hàm bạn đơn giản là thêm dòng khai báo của hàm đó vào bên trong lớp, kèm theo từ khóa `friend` ở trước . Vậy thì đối với ví dụ trên ta sẽ thêm như sau:

```
class MyClass1 {
    friend void myFunc(MyClass1 mClass);
    // ...
};
```

Lúc này hàm **myFunc** có thể truy cập vào **TpRiengTu** của MyClass.

Lưu ý: nhiều bạn thường nhầm hàm **myFunc** là thành phần của MyClass1. Hàm **myFunc** **không phải** là thành phần của lớp MyClass1, nó chỉ là một hàm bình thường (không cần có toán tử phạm vi (::) trước tên hàm khi định nghĩa).

2.10.2. Lớp bạn

Tương tự, lớp bạn là lớp có thể truy cập thành phần **private** hoặc **protected** của lớp xem nó là bạn.

Giả sử mình có lớp MyClass2 như sau:

```
class MyClass2 {
public:
    void myMethod(MyClass1 mClass) {
        cout << mClass.tpRiengTu; // Không hợp lệ
    }
};
```

Trong đoạn code trên, dòng in giá trị thuộc tính **tpRiengTu** trong phương thức **myMethod** của MyClass2 không hợp lệ do ta không thể truy cập thành phần **private** của MyClass1 từ bên ngoài.

Để lớp MyClass2 có thể truy cập thành phần **private**, **protected** của lớp MyClass1, ta thêm dòng khai báo của MyClass2 bên trong MyClass1, theo sau bởi từ khóa **friend**:

```
class MyClass1 {
    friend class MyClass2;
    // ...
}
```

Như vậy, lớp MyClass2 đã là bạn của lớp MyClass1, các phương thức của lớp MyClass2 sẽ có quyền truy cập các thành phần **private**, **protected** của lớp MyClass1 một cách hợp lệ.

Lưu ý: lớp bạn là mối quan hệ một chiều, có nghĩa là lớp này có thể xem lớp kia là bạn, nhưng không có nghĩa lớp kia xem lớp này là bạn.

CHƯƠNG III: ĐA NĂNG HÓA TOÁN TỬ (OVERLOAD OPERATOR)

3.1. Giới thiệu tính năng và cú pháp khai báo

Nếu bạn đã học qua lập trình C++ cơ bản, chắc chắn rằng trong hầu hết các bài tập về C++, bạn đều sử dụng các toán tử số học như cộng, trừ, nhân, chia. Hầu hết các toán tử đó đều được thực hiện trên toán hạng có kiểu dữ liệu cơ bản như int, float, double…

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

Vậy nếu như bạn muốn thực hiện các toán tử đó đối với toán hạng có kiểu dữ liệu bạn tự định nghĩa thì làm sao?

```
PhanSo ps1(1, 2);
PhanSo ps2(2, 3);
// Làm sao để có thể cộng hai phân số?
PhanSo ketQua = ps1 + ps2;
```

Đây chính là lúc chúng ta sử dụng nạp chồng toán tử.

3.1.1. Nạp chồng toán tử là gì?

Cũng tương tự như nạp chồng hàm (overload function), bạn có thể định nghĩa nhiều hàm có cùng tên, nhưng khác tham số truyền vào. Nạp chồng toán tử (overload operator) cũng tương tự, bạn **định nghĩa lại toán tử đã có** trên kiểu dữ liệu người dùng tự định nghĩa để dễ dàng thể hiện các câu lệnh trong chương trình.

Ví dụ như bạn định nghĩa phép cộng cho kiểu dữ liệu phân số thì sẽ thực hiện cộng hai phân số rồi trả về một phân số mới. So với việc thực hiện gọi hàm, việc overload toán tử sẽ làm cho câu lệnh ngắn gọn, dễ hiểu hơn:

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

3.1.2. Cơ chế hoạt động

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

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

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

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

Các toán tử có thể overload:

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

3.1.3. Cú pháp overload:

Như đã giới thiệu, bản chất việc dùng toán tử là lời gọi hàm, do đó chúng ta overload toán tử cũng giống overload hàm, vậy chúng ta sẽ overload hàm nào? Chúng ta sẽ overload hàm có tên là "**operator@**", với @ là toán tử cần overload (+, -, *, /, ...). Có hai loại là **hàm cục bộ (dùng phương thức của lớp)** và **hàm toàn cục (dùng hàm bạn)**. Chúng ta sẽ lần lượt tìm hiểu cách overload toán tử bằng cả hai cách.

Cài đặt với hàm cục bộ

Đối với hàm cục bộ hay còn gọi là phương thức của lớp, số tham số sẽ ít hơn hàm toàn cục một tham số vì tham số đầu tiên mặc định chính là đối tượng gọi phương thức (toán hạng

đầu tiên). Vậy, đối với toán tử hai ngôi, ta chỉ cần truyền vào một tham số cho hàm, chính là toán hạng thứ hai. Ví dụ:

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

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

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

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

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

Do toán tử overload theo cách này là phương thức, được gọi từ một đối tượng, do đó mặc định toán hạng đầu tiên phải là toán hạng có kiểu dữ liệu của lớp đó, điều này cũng có nghĩa

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

Cài đặt hàm toàn cục

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

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

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

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

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

3.1.4. Chuyển kiểu

Có hai loại chuyển kiểu là chuyển kiểu bằng constructor và bằng toán tử chuyển kiểu.

Chuyển kiểu bằng constructor

Dùng để chuyển các kiểu dữ liệu cơ bản thành kiểu dữ liệu do ta định nghĩa:

```
// Thêm constructor bên trong lớp
PhanSo(int a) {
    tu = a;
    mau = 1;
}
```

Với constructor được định nghĩa như trên, khi ta thực hiện cộng một số nguyên với một kiểu phân số, hay một kiểu phân số với số nguyên thì số nguyên sẽ được trình biên dịch chuyển thành kiểu phân số thông qua việc gọi constructor bên trên, với mẫu số là 1 và tử chính là số nguyên ta đang cộng. Vì thế, lúc này chúng ta chỉ cần một hàm bạn:

```
class PhanSo{
     friend PhanSo operator+(const PhanSo& ps1,const PhanSo& ps2)
    //...
};
PhanSo operator+(const PhanSo& ps1, const PhanSo& ps2){
    PhanSo kq;
    kq.tu = ps1.tu * ps2.mau + ps1.mau * ps2.tu;
    kq.mau = ps1.mau * ps2.mau;
    return kq;
}
```

Và chỉ cần như thế, lúc này ta có thể thực hiện như sau:

```
PhanSo a(2,3), b(4,1), c;
c = a + b; // c = operator+(a,b)
c = a + 5; // c = operator+(a,PhanSo(5))
c = 3 + a; // c = operator+(PhanSo(3),a)
```

Chuyển kiểu bằng toán tử chuyển kiểu

Dùng để chuyển kiểu dữ liệu ta định nghĩa sang các kiểu dữ liệu cơ bản.

*Cú pháp:

```
operator KieuDL() {
    return x; // x có kiểu dữ liệu là Kieu_DL
}
```

Ví dụ như mình muốn chuyển phân số về số thực, mình sẽ overload toán tử chuyển kiểu `float`:

```
class PhanSo {
    //...
    operator float();
};
// Định nghĩa
PhanSo::operator float() {
    return (float)this->tu / this->mau;
}
```

Lúc này ta có thể chuyển kiểu dữ liệu như sau:

```
PhanSo ps1(1, 2);
PhanSo ps2(2, 3);
float a = ps1 + ps2;
cout << a << endl;  // ~ 1.67
cout << (float)ps1; // = 0.5
```

3.1.5. Sự nhập nhằng

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

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

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

Cách xử lý duy nhất cho việc này là thực hiển chuyển kiểu tường minh, việc này làm mất đi sự tiện lợi của cơ chế chuyển kiểu tự động. Do đó khi thực hiện chuyển kiểu, ta chỉ được chọn một trong hai, hoặc là chuyển kiểu bằng constructor, hoặc là overload toán tử chuyển kiểu.

3.2. Toán tử nhập, xuất (Input, output operator)

Để nạp chồng toán tử nhập xuất, chúng ta sử dụng **hàm toàn cục** (và là **hàm bạn**), có **hai tham số**, tham số đầu tiên là một **tham chiếu**[3] đến đối tượng kiểu `istream` hoặc `ostream`, tham số thứ hai là một **tham chiếu** tới đối tượng cần nhập/xuất, kiểu trả về của hàm chính là **tham chiếu** đến tham số đầu tiên của hàm (`istream` hoặc `ostream`).

Toán tử nhập

Chúng ta sẽ thực hiện overload toán tử nhập cho lớp phân số như sau:

```
class PhanSo {
    //...
    friend istream& operator>>(istream&, PhanSo&);
};
istream& operator>>(istream& is, PhanSo& ps) {
    cout << "Nhap tu: ";
    is >> ps.tu;
    cout << "Nhap mau: ";
    is >> ps.mau;
    return is;
}
```

[3] Một đối tượng thuộc lớp istream hoặc ostream thì không sao chép được nên phải sử dụng tham chiếu (xem lại về tham chiếu ở chương 2.7.3 phần phương thức thiết lập sao chép).

Như vậy toán tử nhập đã được nạp chồng cho lớp PhanSo, bây giờ sử dụng toán tử nhập trên một đối tượng của lớp PhanSo như sau:

```
PhanSo ps;
cin >> ps;
```

Hàm nạp chồng toán tử nhập trả về tham chiếu tới một đối tượng thuộc lớp istream để ta sử dụng các toán tử nhập một cách liên tiếp. Ví dụ:

```
PhanSo ps1, ps2;
cin >> ps1 >> ps2; // operator>>(operator>>(cin,ps1),ps2)
```

Trong ví dụ trên hàm operator>>(cin,ps1) được thực hiện trước, hàm này trả về một tham chiếu tới đối tượng cin, đối tượng này được sử dụng để làm đối số cho lời gọi hàm operator>> tiếp theo.

Toán tử xuất

Đối với toán tử xuất chúng ta cũng thực hiện tương tự như sau:

```
class PhanSo {
    //...
    friend ostream& operator<<(ostream&, const PhanSo&);
};
ostream& operator<<(ostream& os, const PhanSo& ps) {
    os << ps.tu << "/" << ps.mau;
    return os;
}
```

Khác biệt duy nhất nằm ở chỗ tham số thứ 2 của hàm này nên là một **tham chiếu hằng** [4] nhằm tránh việc phải thực hiện sao chép quá nhiều (tốn tài nguyên) đồng thời vẫn đảm bảo đối số truyền vào sẽ không bị thay đổi.

Bây giờ bạn có thể sử dụng toán tử xuất bình thường như các kiểu dữ liệu cơ bản khác:

```
PhanSo ps(1,2);
cout << ps; // operator<<(cout,ps)
```

[4] Xem lại về tham chiếu hằng ở chương 2.7.3 phần phương thức thiết lập sao chép

3.3. Toán tử so sánh (Relational operator)

Một số toán tử so sánh thường gặp: ==, !=, >, <, >=, <= .

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

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

```
// Khai báo lớp, các thuộc tính, phương thức cần thiết
bool PhanSo::operator>(const PhanSo& x) {
    float gt1 = (float)tu / mau;
    float gt2 = (float)x.tu / x.mau;
    if (gt1 > gt2)
        return true;
    return false;
}
```

Trong đoạn chương trình sau:

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

Ta có thể hiểu rằng đối tượng **a** đang gọi thực hiện phương thức operator> với đối số là đối tượng **b**. Các hàm nạp chồng những toán tử so sánh khác cũng được định nghĩa tương tự.

3.4. Toán tử gán (Assignment operator)

Trong quá trình thực hiện phép gán giữa hai đối tượng, toán tử gán (=) chỉ đơn giản là sao chép dữ liệu đối tượng nguồn (đối tượng bên phải toán tử gán) sang đối tượng đích (đối tượng bên trái toán tử gán).

Đặc biệt, hàm đa năng hóa toán tử gán chỉ có thể được định nghĩa dưới dạng **phương thức** của lớp.

Hàm đa năng hóa toán tử gán của lớp PhanSo được định nghĩa như sau:

```
//Khai báo lớp, các thuộc tính, phương thức cần thiết
PhanSo& PhanSo::operator=(const PhanSo& x) {
    tu = x.tu;
    mau = x.mau;
    return *this;
}
```

Thông thường, giá trị trả về của toán tử gán sau khi được nạp chồng là **tham chiếu** tới đối tượng bên trái toán tử =. Điều này phù hợp với chức năng truyền thống của toán tử gán mặc định. Dòng lệnh `return *this` giúp trả về đối tượng đang gọi thực hiện phương thức (được trỏ đến bởi con trỏ `this`). Vì vậy, ta có thể thực hiện chuỗi phép gán sau:

```
PhanSo a, b, c(1,2);
a = b = c; // a.operator=(b.operator=(c))
```

Tương tự như **phương thức thiết lập sao chép**[5], hàm đa năng hóa toán tử gán cũng nhận vào một **tham chiếu hằng** để đảm bảo đối số truyền vào không thể bị sửa đổi một cách vô ý, cũng như đảm bảo rằng có thể truyền vào phương thức một đối số hằng.

Nếu không làm gì, chương trình sẽ tự nạp chồng cho lớp một toán tử gán, tuy nhiên ta cần phải tự nạp chồng toán tử gán với những lớp có thuộc tính đặc biệt như con trỏ.

3.5. Toán tử số học, gán kết hợp (Compound-assignment operator)

Là sự kết hợp giữa toán tử số học và toán tử gán.

Ví dụ:

```
//Khai báo lớp, các thuộc tính, phương thức cần thiết
PhanSo& PhanSo::operator+=(const PhanSo& x) {
    tu = tu *x.mau + mau * x.Tu;
    mau = mau * x.mau;
    return *this;
}
```

3.6. Toán tử tăng một, giảm một (Increment, decrement operator)

Loại toán tử này có hai phiên bản là tiền tố và hậu tố. Cả hai đều tăng giá trị của đối tượng lên 1, tuy nhiên phiên bản **tiền tố** sẽ trả về giá trị của đối tượng **sau khi đã tăng thêm 1**, còn

[5] Xem lại về phương thức thiết lập sao chép ở chương 2.7.3

phiên bản **hậu tố** thì trả về giá trị của đối tượng **trước khi tăng thêm 1**. Cả hai đều được nạp chồng bằng **phương thức** của lớp.

Phiên bản tiền tố (++a)

Vì là toán tử 1 ngôi và toán hạng duy nhất của nó là đối tượng đang gọi thực hiện phương thức nên hàm nạp chồng sẽ không có tham số đầu vào:

```
PhanSo& PhanSo::operator++() {
    *this += PhanSo(1);
    return *this;
}
```

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

Phiên bản hậu tố (a++)

Để chương trình có thể phân biệt được 2 kiểu `operator++` khác nhau thì phiên bản hậu tố sẽ có thêm 1 tham số đầu vào giả:

```
PhanSo PhanSo::operator++(int) {
    PhanSo ret = *this;
    ++ *this; // sử dụng lại phiên bản tiền tố đã định nghĩa
    return ret;
}
```

Lúc lập trình ta chỉ cần ghi các câu lệnh như bình thường, chương trình sẽ tự biết là nên gọi phương thức nào:

```
PhanSo a(2, 3);
++a; // gọi phiên bản tiền tố
a++; // gọi phiên bản hậu tố
```

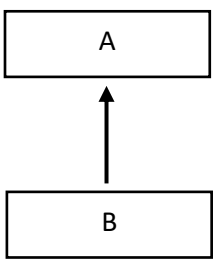
Kiểu dữ liệu trả về của phiên bản hậu tố là kiểu `PhanSo` bình thường bởi vì không nên trả về một tham chiếu hoặc một con trỏ tới một biến cục bộ của hàm[6] (ở đây là `ret`).

[6] Các bạn tự tìm hiểu thêm ha chứ ghi vào thì bị rối và loãng nội dung<3: https://www.educative.io/answers/resolving-the-function-returns-address-of-local-variable-error

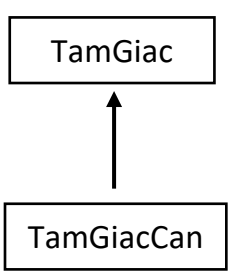
CHƯƠNG IV: KẾ THỪA (INHERITANCE) VÀ ĐA HÌNH (POLYMORPHISM)

4.1. Mối quan hệ đặc biệt hóa, tổng quát hóa

- Hai lớp đối tượng được gọi là quan hệ đặc biệt hóa – tổng quát hóa với nhau khi, lớp đối tượng này là trường hợp đặc biệt của lớp đối tượng kia và lớp đối tượng kia là trường hợp tổng quát của lớp đối tượng này.
- Ký hiệu:



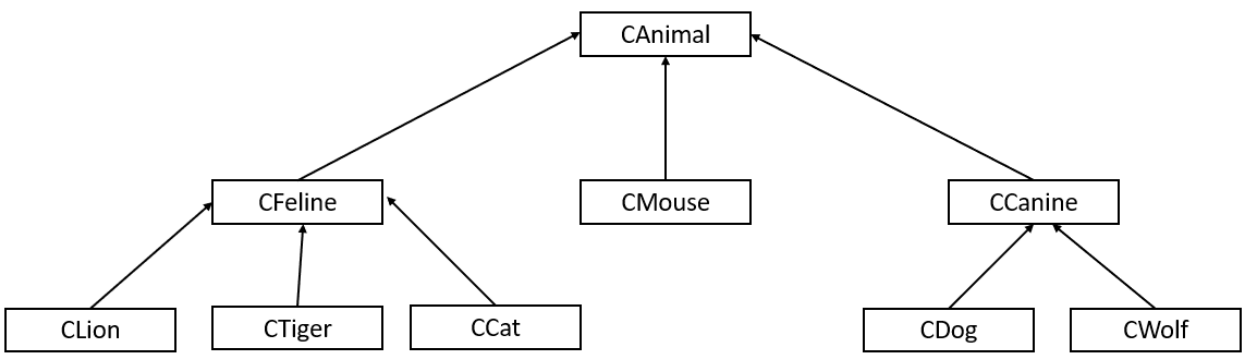
- Trong hình vẽ trên: lớp đối tượng B là trường hợp đặc biệt của lớp đối tượng A và lớp đối tượng A là trường hợp tổng quát của lớp đối tượng B.
- Ví dụ:



- Trong hình vẽ trên: lớp đối tượng TamGiacCan là trường hợp đặc biệt của lớp đối tượng TamGiac và lớp đối tượng TamGiac là trường hợp tổng quát của lớp đối tượng TamGiacCan

Cây kế thừa

- Cây kế thừa là một cây đa nhánh thể hiện mối quan hệ đặc biệt hóa-tổng quát hóa giữa các lớp trong hệ thống, trong chương trình.
- Ví dụ:



4.2. Kế thừa

- Kế thừa là một đặc điểm của ngôn ngữ lập trình dùng để biểu diễn mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp. Nó cho phép một lớp có thể được **thừa hưởng các thuộc tính, phương thức** từ một lớp khác.
- Lớp kế thừa từ một lớp khác thì được gọi là **lớp con (child class, subclass)** hay **lớp dẫn xuất (derived class)**. Lớp được các lớp khác kế thừa được gọi là **lớp cha (parent class, superclass)** hay **lớp cơ sở (base class)**.

Ví dụ thực tế

- Bạn có một lớp đối tượng con người, có các thuộc tính như họ tên, ngày sinh, quê quán, ta khai báo thêm một lớp sinh viên kế thừa từ lớp con người.
- Khi đó lớp sinh viên sẽ có các thuộc tính họ tên, ngày sinh, quê quán từ lớp con người mà không cần phải khai báo lại. Lớp con người được gọi là lớp cha và lớp sinh viên là lớp con
- Ngoài các thuộc tính của lớp cha, lớp con còn có thể có thêm các thuộc tính, phương thức của riêng nó. Ở ví dụ trên thì lớp sinh viên có thể có thêm các thuộc tính như MSSV, tên trường, chuyên ngành …

Lợi ích của kế thừa

- Kế thừa cho phép xây dựng lớp mới từ lớp đã có.
- Kế thừa cho phép tổ chức các lớp chia sẻ mã chương trình chung, nhờ vậy có thể dễ dàng sửa chữa, nâng cấp hệ thống.
- Định nghĩa sự tương thích giữa các lớp, nhờ đó ta có thể chuyển kiểu tự động.

4.3. Định nghĩa lớp cơ sở và lớp dẫn xuất trong C++

4.3.1. Bài toán quản lí cửa hàng sách

Lấy ví dụ về bài toán quản lí hóa đơn trong một hiệu sách, chủ cửa hàng muốn áp dụng các phương thức tính tiền khác nhau cho các loại sách khác nhau. Một vài quyển sách chỉ được bán ở một mức giá cố định, trong khi một vài quyển khác sẽ được giảm giá khi mua với số lượng lớn.

Để giải quyết vấn đề, ta sẽ tạo một lớp cơ sở có tên là `Quote` đại diện cho những quyển sách không được giảm giá và lớp `BulkQuote` được kế thừa từ lớp `Quote`, tượng trưng cho những loại sách sẽ được áp mã giảm giá khi mua trên một số lượng nhất định. Hai lớp sẽ có chung các thuộc tính là mã số sách và giá tiền, cùng với một phương thức đặc biệt dùng để tính tiền. Trước tiên hãy cùng xem sơ qua định nghĩa của hai lớp này.

4.3.2. Định nghĩa lớp cơ sở

```
class Quote {
private:
    string bookNo; // mã số sách
protected:
    double price; // giá của một quyển sách
public:
    // phương thức thiết lập:
    Quote();
    Quote(const string& book, double salesPrice);
    // phương thức truy vấn:
    string getBookNo() { return bookNo; }
    // phương thức tính tiền khi biết số lượng sách được mua:
    virtual double NetPrice(int n) { return price * n; }
};
```

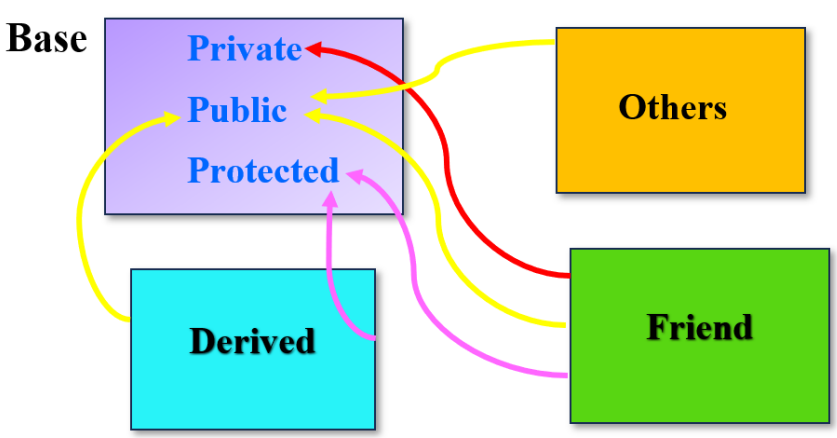
Điểm mới trong phần định nghĩa của lớp cơ sở `Quote` là sự xuất hiện của từ khóa `virtual` trước dòng khai báo của phương thức `NetPrice` và phạm vi truy xuất `protected`. Chúng ta sẽ tìm hiểu kĩ hơn về `virtual` ở các chương sau, bây giờ bạn đọc chỉ cần biết rằng từ khóa `virtual` dùng để chỉ những hàm mà lớp cha muốn lớp con của nó tự định nghĩa một phiên bản của riêng mình.

4.3.3. Phạm vi truy xuất `protected` trong kế thừa

Tương tự với cách mà người dùng sử dụng một lớp đối tượng, các lớp dẫn xuất có thể truy cập đến các thành viên `public` và không thể truy cập đến các thành viên `private` trong lớp cơ sở của nó. Tuy nhiên có những trường hợp mà ta muốn một vài thành viên của lớp cơ sở **có thể được truy cập ở lớp dẫn xuất** trong khi vẫn **giới hạn quyền truy cập từ người dùng**. Những thành viên như vậy sẽ được chỉ định là các thành viên `protected`.

Phạm vi truy xuất `protected` có thể được xem như là một sự kết hợp giữa `public` và `private`:

- Giống với `private`, các thành viên `protected` không thể được truy cập bởi người dùng bên ngoài.
- Giống với `public`, các thành viên `protected` của lớp cơ sở có thể được truy cập bởi bạn và các lớp dẫn xuất của nó.



Hình 3: Minh họa cho tính chất của các phạm vi truy xuất.

Trong ví dụ về lớp cơ sở `Quote`, thuộc tính `price` được chỉ định là `protected` nên các hàm thành viên của các lớp kế thừa từ `Quote` có thể truy cập trực tiếp vào nó trong khi người dùng thì không. Thuộc tính `bookNo` được chỉ định là `private` nên không thể được truy cập trực tiếp từ bên ngoài lớp, kể cả là các lớp con hay người dùng. Còn lại các thành viên `public` thì có thể được truy cập ở bất cứ đâu.

4.3.4. Định nghĩa lớp dẫn xuất

Trong C++, lớp dẫn xuất được khai báo theo cú pháp:

```
class B : <từ khóa dẫn xuất> A {
   ...
};
```

Ở đây ta hiểu rằng lớp B kế thừa từ lớp A, lớp A là lớp cơ sở và lớp B là lớp dẫn xuất từ A. Từ khóa dẫn xuất có ba loại đó là **private**, **public** và **protected** sẽ được giải thích ở phần sau, trước hết ta hãy cùng xem qua định nghĩa của lớp BulkQuote:

```
// lớp BulkQuote kế thừa từ lớp Quote
class BulkQuote : public Quote {
private:
    double discount; // tỉ lệ phần trăm được giảm
    int minQty; // số lượng mua tối thiểu để được áp mã
public:
    BulkQuote();
    BulkQuote(const string& book, double salesPrice,
              double disc, int cnt);
    // phương thức NetPrice sẽ có một phiên bản khác
    double NetPrice(int n);
};
```

Lớp dẫn xuất sẽ được **kế thừa** các thành viên nằm trong lớp cơ sở (*trừ phương thức thiết lập*), thêm vào đó nó có thể **định nghĩa thêm** các thành viên mới của riêng mình hoặc **định nghĩa một phiên bản khác** cho vài phương thức ở lớp cơ sở. Lớp BulkQuote trong ví dụ này đã được thừa hưởng từ lớp Quote 2 thuộc tính là **bookNo**, **price** cùng với phương thức **getBookNo**. Lớp BulkQuote còn có thêm 2 thuộc tính mới là **discount**, **minQty** và 2 phương thức thiết lập của riêng mình, ngoài ra, nó sẽ tự định nghĩa lại hàm **NetPrice** ở lớp Quote. Những thành viên đã được thừa hưởng thì không cần phải ghi lại trong phần thân của lớp dẫn xuất, ta chỉ cần viết khai báo và định nghĩa cho các thành viên mới cùng với các phương thức cần định nghĩa lại.

Lưu ý: Mặc dù lớp dẫn xuất không có quyền truy cập trực tiếp vào các thuộc tính **private** ở lớp cơ sở nhưng nó vẫn được thừa hưởng các thuộc tính đó. Trong ví dụ trên, lớp BulkQuote được kế thừa thuộc tính **bookNo** từ Quote, tuy nhiên thuộc tính này chỉ có thể được truy cập thông qua phương thức **getBookNo**.

4.4. Các kiểu kế thừa

Trong các phần trước, ta đã tìm hiểu cách một lớp đối tượng quản lí quyền truy cập vào các thành viên của mình thông qua phạm vi truy xuất được chỉ định bên trong lớp. Đến với kế thừa, các từ khóa dẫn xuất còn được sử dụng trong khai báo lớp con để có thể **thay đổi mức độ truy cập** của các thành viên ở lớp cha khi chúng được kế thừa xuống lớp con. Tức là kiểm soát quyền truy cập mà người dùng của lớp dẫn xuất có đối với các thành viên được kế thừa từ lớp cơ sở.

Để cụ thể hơn, ta có bảng quy tắc kế thừa như sau:

Phạm vi truy xuất được chỉ định ở lớp cơ sở:	**Từ khóa dẫn xuất khi khai báo lớp con:**	**Phạm vi truy xuất ở lớp con được chuyển thành:**
`Public`	`Public`	`public`
`Protected`		`protected`
`Private`		**Không truy cập được**
`Public`	`protected`	`protected`
`Protected`		`protected`
`Private`		**Không truy cập được**
`Public`	`private`	`private`
`Protected`		`private`
`Private`		**Không truy cập được**

Từ bảng có thể rút ra được các thông tin:

- Thành phần `private` ở lớp cha thì không truy xuất được ở lớp con
- Lớp con kế thừa kiểu `public` từ lớp cha thì các thành phần `protected` của lớp cha trở thành `protected` của lớp con, các thành phần `public` của lớp cha trở thành `public` của lớp con.
- Lớp con kế thừa kiểu `private` từ lớp cha thì các thành phần `protected` và `public` của lớp cha trở thành `private` của lớp con.

- Lớp con kế thừa kiểu `protected` từ lớp cha thì các thành phần `protected` và `public` của lớp cha trở thành `protected` của lớp con.

Lưu ý: Bảng quy tắc kế thừa ở trên không ảnh hưởng đến cách mà lớp con truy cập đến các thành phần ở lớp cha (phần này được đảm nhiệm bởi các phạm vi truy xuất do lớp cha chỉ định) mà chỉ điều khiển quyền truy cập của *người dùng* của lớp con với các thành viên kế thừa từ lớp cha.

4.5. Phương thức thiết lập trong kế thừa

Mặc dù một đối tượng của lớp con được thừa hưởng các thuộc tính từ lớp cha, nó **không nên trực tiếp khởi tạo dữ liệu** cho các thuộc tính đó (trong vài trường hợp thì là không thể luôn). Ở các ví dụ trên, lớp `BulkQuote` kế thừa 2 thuộc tính là **`bookNo`** và **`price`** từ `Quote`, tuy nhiên **`bookNo`** là thành phần `private` của `Quote`, kể cả lớp con cũng không truy cập được thuộc tính này, dẫn đến việc `BulkQuote` không thể trực tiếp gán dữ liệu cho **`bookNo`**.

Cho dù các thuộc tính này đều là `protected` và lớp con có thể truy cập được đi chăng nữa, lớp con cũng không nên khởi tạo dữ liệu trực tiếp cho chúng (bởi vì lớp cha có cung cấp các giao diện để tương tác với các thuộc tính của nó, lớp con nên tôn trọng và sử dụng giao diện này).

Tóm lại, lớp con nên **sử dụng phương thức thiết lập của lớp cha** để khởi tạo dữ liệu cho các thuộc tính chung mà nó được thừa kế. Cách sử dụng như sau (lấy ví dụ là phương thức thiết lập cho lớp `BulkQuote`):

```
//Phương thức thiết lập của Quote
Quote::Quote(string id, double price) : bookNo(id), price(price) {}
//Phương thức thiết lập của Bulk_quote
BulkQuote::BulkQuote(string id, double price, double disc, int n)
                     : Quote(id,price), discount(disc), minQty(n) {}
```

Một điểm mới trong ví dụ trên chính là việc sử dụng **danh sách khởi tạo** trong phương thức thiết lập **(constructor initializer list)**. Có một vài điểm khác biệt giữa việc sử dụng danh sách khởi tạo và việc thực hiện phép gán (`operator=`) trong thân phương thức thiết lập ở ví dụ của các chương trước, tuy nhiên để tránh loãng nội dung thì kiến thức này sẽ không nói ở đây☹.

Trở lại vấn đề chính, với cách định nghĩa như trên, khi một phương thức thiết lập của `BulkQuote` được gọi với 4 đối số đầu vào:

```
    BulkQuote derived("893-523-52-2211-3", 150000, 0.2, 3);
```

2 đối số đầu tiên sẽ được dùng để gọi phương thức thiết lập của `Quote`, sau khi phương thức này thực hiện xong nhiệm vụ và các thuộc tính chung đã được khởi tạo, hai đối số còn lại sẽ lần lượt được dùng để khởi tạo cho giá trị cho 2 thuộc tính riêng của `BulkQuote`.

Constructor mặc định

Khi một đối tượng của lớp con được khởi tạo mặc định, thứ tự như trên cũng được thực hiện, tức là **constructor mặc định của lớp cha được gọi trước** để khởi tạo dữ liệu mặc định cho các thuộc tính chung, sau đó constructor của lớp con mới bắt đầu thực hiện việc gán dữ liệu cho các thuộc tính riêng.

```
Quote::Quote() {
    cout << “Ham khoi tao cua lop co so!” << endl;
}
BulkQuote::BulkQuote() {
    cout << “Ham khoi tao cua lop dan xuat!” << endl;
}

int main() {
    BulkQuote obj;
    return 0;
}
```

★ **Kết quả:** Hàm khởi tạo ở lớp cơ sở được **gọi đến trước** rồi mới tới hàm khởi tạo ở lớp dẫn xuất.

```
Ham khoi tao cua lop co so!
Ham khoi tao cua lop dan xuat!
```

4.6. Phép gán và con trỏ trong kế thừa

Thông thường, một biến con trỏ chỉ có thể giữ địa chỉ của các đối tượng có cùng kiểu với nó, tuy nhiên trong kế thừa có một ngoại lệ: **một con trỏ đối tượng thuộc lớp cơ sở có thể giữ địa chỉ của một đối tượng thuộc lớp dẫn xuất**. Dẫu vậy, một con trỏ đối tượng thuộc lớp dẫn xuất lại *không thể* giữ địa chỉ của một đối tượng thuộc lớp cơ sở, như ví dụ dưới đây:

```
Quote base;
BulkQuote derived;
Quote* basePtr = &derived; // Đúng
BulkQuote* derivedPtr = &base; // Sai
```

Việc có thể gán địa chỉ của một đối tượng thuộc lớp dẫn xuất cho một biến con trỏ đối tượng thuộc lớp cơ sở dẫn đến một kết quả quan trọng: Khi sử dụng một con trỏ đối tượng thuộc lớp cơ sở, chúng ta **không biết chắc** được đối tượng mà con trỏ đó sẽ giữ có kiểu dữ liệu gì, nó có thể là một đối tượng thuộc lớp cơ sở hoặc lớp dẫn xuất. Điều này góp phần tạo nên tính đa hình trong OOP sẽ được tìm hiểu ở chương tới.

Phép gán trực tiếp giữa các đối tượng

Ta cũng có thể gán trực tiếp một đối tượng thuộc lớp con cho một đối tượng thuộc lớp cha. Ngược lại, không thể gán một đối tượng của lớp cha cho một đối tượng thuộc lớp con. Tuy nhiên có một vài điểm cần lưu ý: Khi dùng một biến có kiểu lớp con khởi tạo cho một đối tượng thuộc lớp cha, chương trình sẽ chỉ **sao chép những thuộc tính chung** giữa 2 lớp mà **không sao chép các thuộc tính riêng** của lớp con. Đơn giản là vì lớp cha thì không thể biết được các lớp con của nó có các thuộc tính mới nào, nó chỉ biết được những thuộc tính đã được định nghĩa sẵn bên trong mình mà thôi.

Ví dụ:

```
Quote item1;
BulkQuote item2("13hd", 50000, 0.2, 3);
item1 = item2;
```

Sau dòng lệnh trên, `item1` lúc này chỉ chứa 2 thuộc tính là `bookNo = 13hd` và `price = 50000`, 2 thuộc tính `discount = 0.2` và `minQty = 3` trong `item2` đã bị lược bỏ.

4.7. Phương thức ảo (Virtual function) và Đa hình (Polymorphism)

Lớp dẫn xuất được kế thừa các phương thức đã có ở lớp cơ sở, tuy nhiên hành vi của chúng có thể được tinh chỉnh để tương thích hơn với lớp dẫn xuất. Để làm vậy, lớp dẫn xuất cần phải định nghĩa lại **một phiên bản khác** cho các phương thức đó.

Trong lớp cơ sở, ta thêm từ khóa `virtual` vào trước phần khai báo của những phương thức mà các lớp dẫn xuất có thể **ghi đè** lại **(override)**, những phương thức này sẽ được gọi là **phương thức ảo**. Khi gọi các phương thức ảo thông qua một con trỏ đối tượng, lời gọi sẽ được thực hiện theo cơ chế **đa hình**, cho phép xác định đúng hành vi (phương thức) sẽ được thực thi. Tùy thuộc vào kiểu dữ liệu của đối tượng mà con trỏ giữ địa chỉ, phiên bản của phương thức ảo nằm trong lớp cơ sở hoặc lớp dẫn xuất sẽ được thực hiện (nhớ lại rằng một con trỏ thuộc lớp cơ sở có thể giữ địa chỉ của một đối tượng thuộc lớp dẫn xuất).

Ngoại trừ các *phương thức tĩnh* và *phương thức thiết lập*, các phương thức khác đều có thể được khai báo là phương thức ảo. Những phương thức đã được khai báo là `virtual` trong

lớp cơ sở thì thì những phương thức **cùng tên** và **cùng danh sách tham số** đầu vào trong lớp dẫn xuất cũng sẽ là phương thức ảo.

Trong bài toán quản lí cửa hàng sách, ta đã khai báo phương thức `NetPrice` trong `Quote` là phương thức ảo, lớp `BulkQuote` muốn định nghĩa một phiên bản khác của `NetPrice` nên sẽ ghi lại dòng khai báo của của nó (lưu ý rằng tên hàm, kiểu dữ liệu trả về, danh sách tham số đầu vào phải giống hệt nhau):

```
class Quote {
// ...
public:
    virtual double NetPrice(int n) { return price * n; }
};

class BulkQuote : public Quote {
// ...
public:
    double NetPrice(int n);
};
```

Phương thức ảo có thể được định nghĩa bên trong hoặc bên ngoài lớp, khi định nghĩa bên ngoài thì cũng không cần phải ghi lại từ khóa `virtual`. Ta sẽ định nghĩa một phiên bản của `NetPrice` cho lớp `BulkQuote` như sau:

```
double BulkQuote::NetPrice(int n) {
    // nếu số lượng mua lớn hơn minQty thì sẽ áp dụng giảm giá
    if (n > minQty) {
        return n * (1 - discount) * price;
    }
    return n * price;
}
```

Để hiểu rõ hơn cơ chế hoạt động của đa hình, hãy cùng xem qua ví dụ dưới đây:

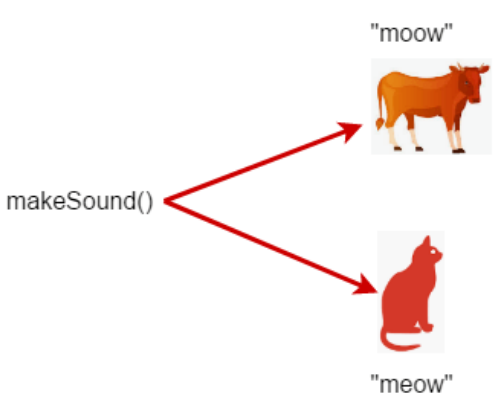
```
1.  // base là đối tượng thuộc lớp cơ sở Quote
2.  Quote base("978-179-64-4473-5", 150000);
3.  // derived là đối tượng thuộc lớp dẫn xuất BulkQuote
4.  BulkQuote derived("978-179-64-4473-5", 150000, 0.2, 3);
5.  // basePtr là con trỏ thuộc lớp cơ sở Quote
6.  Quote* basePtr;
```

```
7.  // basePtr giữ địa chỉ của base
8.  basePtr = &base;
9.  cout << "Gia cua don hang 1: ";
10. cout << basePtr->NetPrice(5) << endl;
11. // phương thức NetPrice của Quote được thực hiện

12. // basePtr giữ địa chỉ của derived
13. basePtr = &derived;
14. cout << "Gia cua don hang 2: ";
15. cout << basePtr->NetPrice(5) << endl;
16. // phương thức NetPrice của BulkQuote được thực hiện
```

Bởi vì **basePtr** là con trỏ kiểu Quote và **NetPrice** là phương thức ảo, phiên bản của **NetPrice** được gọi sẽ phụ thuộc vào kiểu dữ liệu của đối tượng mà **basePtr** giữ địa chỉ. Tại dòng số 10, **basePtr** đang giữ địa chỉ đối tượng **base** thuộc lớp Quote nên phiên bản **NetPrice** của lớp Quote được thực hiện. Còn ở dòng số 15, **basePtr** lúc này giữ địa chỉ của đối tượng **derived** thuộc lớp BulkQuote, do đó hàm **NetPrice** được định nghĩa bởi lớp BulkQuote được gọi. Khi chạy chương trình sẽ thấy hai kết quả in ra khác nhau, chứng tỏ hai phiên bản khác nhau của **NetPrice** đã được thực hiện:

```
Gia cua don hang 1: 750000
Gia cua don hang 2: 600000
```

Trong ví dụ trên, ta thấy thông qua cùng một giao diện là phương thức **NetPrice** (dòng số 10 và dòng số 15), ta vẫn có thể thực hiện các hành động khác nhau (thanh toán khi có giảm giá và không có giảm giá) tùy thuộc vào ngữ cảnh. Đây chính là sự thể hiện của tính đa hình trong OOP.



Hình 4: Một minh họa khác cho tính đa hình

4.8. Lớp cơ sở trừu tượng (Abstract base class)

a. Khái niệm

Lớp cơ sở trừu tượng là một loại lớp đặc biệt vì ta **không thể khởi tạo các đối tượng** của lớp này. Mục đính nó tồn tại là chỉ để làm lớp cơ sở cho các lớp dẫn xuất khác kế thừa lên, cung cấp một **khuôn mẫu, giao diện chung** cho các lớp con đó. Gọi là lớp cơ sở **trừu tượng** vì nó đại diện cho một khái niệm chung chung, trừu tượng và các lớp con của nó sẽ là các phiên bản cụ thể, rõ ràng hơn. Ta sẽ làm rõ vấn đề hơn qua các ví dụ ở chương sau.

a. Cách định nghĩa một lớp cơ sở trừu tượng

Để một lớp cơ sở là trừu tượng, nó phải có ít nhất một **phương thức thuần ảo**. Khác với một phương thức ảo bình thường, phương thức thuần ảo **không nhất thiết phải được định nghĩa**. Ta chỉ định một phương thức ảo là thuần ảo bằng cách viết `= 0` sau dòng khai báo của nó.

Lấy ví dụ về lớp cơ sở `Shape`, đại diện chung cho các đối tượng là hình vẽ, các lớp con của nó sẽ là `Circle`, `Square`, ... Các lớp trong cây kế thừa này sẽ có một phương thức ảo là `Draw` dùng để vẽ hình. Bởi vì `Shape` chỉ là một khái niệm trừu tượng về hình vẽ, việc gọi phương thức `Draw` trên một đối tượng `Shape` là không hợp lí (vì có biết vẽ cái gì đâu:v) nên ta sẽ cho `Shape` là một lớp cơ sở trừu tượng và `Draw` là một phương thức thuần ảo của `Shape`:

```
class Shape {
public:
    // phương thức thuần ảo:
    virtual void Draw() = 0;
    // phương thức Draw của Shape không có hành động cụ thể
};
```

Thật ra hàm `Draw` có thể được định nghĩa là không làm gì bằng cách chỉ thực hiện câu lệnh `return` bên trong thân hàm, khi đó `Shape` không phải lớp cơ sở trừu tượng và ta có thể tạo các đối tượng của `Shape` và gọi phương thức `Draw` trên chúng. Tuy nhiên, ghi nhớ rằng một trong các mục đính của việc tạo lớp cơ sở trừu tượng là để ngăn **không cho người dùng tạo ra các đối tượng** của lớp này và thực hiện các hành động vô nghĩa trên đó.

Các lớp kế thừa từ `Shape` phải định nghĩa phương thức `Draw`, nếu không các lớp đó cũng sẽ là lớp trừu tượng. Do đó việc để một phương thức là thuần ảo còn có tác dụng là bắt buộc các lớp con phải ghi đè lại một hàm nào đó:

```
class Square : public Shape {
private:
    int edgeLength;
public:
    void Draw() {
        // định nghĩa phương thức Draw cho lớp Square
    }
};
```

4.9. Phương thức phá hủy trong kế thừa

Như đã nói ở phần phương thức thiết lập trong kế thừa, khi một đối tượng được khởi tạo, các thuộc tính chung sẽ được khởi tạo trước rồi mới đến các thuộc tính riêng. Phương thức phá hủy thì ngược lại, khi một destructor của lớp con được gọi, nó sẽ **thu hồi các tài nguyên đã cấp phát cho các thuộc tính riêng trước**, rồi sau đó destructor của lớp cha mới được gọi để dọn dẹp các thuộc tính chung (rác của ai người đó dọn:v).

★ **Ví dụ:**

```
class Quote {
// ...
public:
    ~Quote() {
        cout << “Ham huy cua lop co so!” << endl;
    }
};
class BulkQuote : public Quote {
// ...
public:
    ~BulkQuote() {
        cout << “Ham huy cua lop dan xuat!” << endl;
    }
};
int main() {
    BulkQuote obj;
    return 0;
}
```

★ **Kết quả:** Hàm hủy của lớp dẫn xuất được **gọi đến trước** hàm hủy của lớp cơ sở.

```
Ham huy cua lop dan xuat!
Ham huy cua lop co so!
```

CHƯƠNG V: GIẢI CÁC DẠNG BÀI TẬP TRONG ĐỀ THI

5.1. Dạng câu 1: Lý thuyết

Câu 1.1. Trình bày các đặc điểm quan trọng của lập trình hướng đối tượng.

- **Trừu tượng hóa – Abstraction:** Trừu tượng là quá trình tạo ra các lớp trừu tượng để mô hình hóa các đối tượng trong thế giới thực. Nó giúp tập trung vào các khía cạnh quan trọng và ẩn đi các chi tiết không cần thiết.
- **Đóng gói – Encapsulation:** Đóng gói là quá trình kết hợp dữ liệu và các phương thức hoạt động trên dữ liệu thành một đơn vị độc lập. Điều này giúp bảo vệ dữ liệu và ẩn thông tin chi tiết về cách hoạt động bên trong của đối tượng.
- **Thừa kế - Inheritance:** Kế thừa cho phép lớp con kế thừa các thuộc tính và phương thức từ lớp cha. Điều này giúp tái sử dụng mã nguồn và xây dựng các mối quan hệ phân cấp giữa các lớp.
- **Đa hình – Polymorphism:** Đa hình cho phép sử dụng các phương thức cùng tên nhưng với cách thức thực hiện khác nhau trong các lớp khác nhau. Điều này cho phép gọi cùng một phương thức và có kết quả khác nhau tuỳ thuộc vào đối tượng gọi phương thức đó..

Câu 1.2. Trình bày sự hiểu biết và cho ví dụ minh họa về khái niệm class (lớp) và object (đối tượng) trong lập trình hướng đối tượng

- **Lớp là một khuôn mẫu** và đối tượng là một thực thể được thể hiện dựa trên khuôn mẫu đó. Hay nói cách khác, lớp là một mô tả trừu tượng của một nhóm các đối tượng có cùng bản chất, ngược lại, mỗi đối tượng là một thể hiện cụ thể cho những mô tả trừu tượng đó.
- Một lớp đối tượng bao gồm 2 thành phần chính:
    - **Thành phần dữ liệu (data member)**, hay còn được gọi là **thuộc tính (attribute)**.
    - **Hàm thành phần (member function)**, còn có tên gọi khác là **phương thức (method),** là các hành động mà đối tượng của lớp có thể thực hiện.

Câu 1.3. Trình bày những đặc điểm của tính đóng gói (Encapsulation) trong lập trình hướng đối tượng. Trường hợp nào có thể vi phạm tính đóng gói? Cho ví dụ minh họa.

- **Các đặc điểm:**
  - Giúp **bảo vệ dữ liệu** bên trong tránh khỏi các sai sót không đáng có từ người dùng.
  - Giúp **thay đổi phần thực thi** của lớp một cách **linh hoạt** (tức là thay đổi cách tổ chức dữ liệu, chi tiết thực hiện các phương thức) -> Dễ bảo trì, dễ kiểm thử, tăng khả năng tái sử dụng.
- **Trường hợp vi phạm tính đóng gói:** cố gắng **truy cập đến thuộc tính private/ gọi đến phương thức private** của lớp khi đang nằm ngoài phạm vi truy cập của nó.
- **Ví dụ:**

```
class Diem {
private:
   int x;
   int y;
};
```

```
int main() {
   Diem P;
   P.x = 0; //Sai
   return 1;
}
```

Câu 1.4. Cho biết ý nghĩa và mục đích của các hàm get/set trong một lớp.

- **Hàm get và set** là các hàm thành viên của một lớp được sử dụng để **truy cập** và **thay đổi giá trị của các thuộc tính của lớp** đó. Hàm **get** được sử dụng để **truy xuất giá trị** của một thuộc tính trong khi hàm **set** được sử dụng để **gán giá trị** cho thuộc tính đó.
- **Mục đích:**
  - Đảm bảo rằng dữ liệu của lớp được **bảo vệ** và **kiểm soát** tốt hơn. Chúng cho phép lập trình viên kiểm soát cách thức truy cập và thay đổi dữ liệu của lớp, giúp ngăn chặn việc truy cập trực tiếp vào các thuộc tính của lớp từ bên ngoài lớp.
  - Tăng tính linh hoạt và **tái sử dụng** của mã nguồn. Nếu bạn muốn thay đổi cách thức tính toán giá trị của một thuộc tính, bạn chỉ cần thay đổi mã nguồn trong hàm set tương ứng mà không cần thay đổi mã nguồn ở những nơi khác trong chương trình.

Câu 1.5. Nêu khái niệm constructor và destructor. Phân biệt các kiểu constructor.

- **Constructor: Phương thức thiết lập -** có nhiệm vụ **thiết lập thông tin ban đầu** cho các đối tượng thuộc về lớp ngay khi đối tượng được khai báo.
- **Destructor: Phương thức phá hủy -** có nhiệm vụ **thu hồi lại tất cả các tài nguyên** đã cấp phát cho đối tượng khi đối tượng hết phạm vi hoạt động (scope).
- **Phân biệt:** Ta có thể chia các phương thức thiết lập của một lớp thành 3 nhóm như sau:
  - **Phương thức thiết lập mặc định** (default constructor): thiết lập các thông tin ban đầu cho đối tượng thuộc về lớp bằng **những giá trị mặc định** (do người lập trình quyết định). Phương thức thiết lập mặc định **không có tham số đầu vào.**
  - **Phương thức thiết lập sao chép** (copy constructor): nhận **tham số đầu vào là một đối tượng cùng thuộc về lớp**. Các **thông tin** ban đầu của đối tượng sẽ **hoàn toàn giống** thông tin của đối tượng tham số đầu vào.
  - **Phương thức thiết lập nhận tham số đầu vào** (user define constructor – parameterized constructors): không phải là phương thức thiết lập mặc định và phương thức thiết lập sao chép.

Câu 1.6. Trình bày những ưu điểm của kế thừa trong lập trình hướng đối tượng và cho ví dụ minh họa?

- **Các ưu điểm:**
  - Kế thừa cho phép xây dựng lớp mới từ lớp đã có.
  - Kế thừa cho phép tổ chức các lớp chia sẻ mã chương trình chung, nhờ vậy có thể dễ dàng sửa chữa, nâng cấp hệ thống.
  - Định nghĩa sự tương thích giữa các lớp, nhờ đó ta có thể chuyển kiểu tự động.
- **Ví dụ:**

```
class HocSinh {
    // ...
};
// lớp HocSinhGioi kế thừa các thành viên từ lớp HocSinh mà
// không cần phải khai báo.
class HocSinhGioi : public HocSinh {
    // ...
};
```

Câu 1.7. Phân biệt các kiểu kế thừa private, public, protected.

Phạm vi truy cập / Từ khóa dẫn xuất	Private	Public	Protected
Private	X	Private	Private
Public	X	Public	Protected
Protected	X	Protected	Protected

- Thành phần **private** ở lớp cha thì **không truy xuất** được ở lớp con.
- **Kế thừa public:** Lớp con kế thừa public từ lớp cha thì các thành phần **protected** của lớp cha trở thành **protected** của lớp con, các thành phần **public** của lớp cha trở thành **public** của lớp con.
- **Kế thừa private:** Lớp con kế thừa private từ lớp cha thì các thành phần **protected** và **public** của lớp cha trở thành **private** của lớp con.
- **Kế thừa protected:** Lớp con kế thừa protected từ lớp cha thì các thành phần **protected** và **public** của lớp cha trở thành **protected** của lớp con.

Câu 1.8. Phân biệt các phạm vi truy cập private, public, protected.

- Phạm vi truy cập `public`: có thể được truy cập ở **tất cả các phần của chương trình**. Các thành phần `public` tạo nên giao diện của một Lớp.
- Phạm vi truy cập `private`: chỉ có thể được truy cập **bên trong phạm vi của lớp**, bởi các hàm thành viên (phương thức) của một lớp và không thể được truy cập từ bên ngoài lớp. Phần `private` ẩn đi (đóng gói) phần thực thi.
- Phạm vi truy cập `protected`: Là sự kết hợp giữa `public` và `private`, có thể được truy cập ở lớp dẫn xuất trong khi vẫn giới hạn quyền truy cập từ người dùng.

Câu 1.9. Phân biệt khái niệm overload (nạp chồng) và override (ghi đè).

- **Overload (nạp chồng):** Là một kĩ thuật cho phép **trong cùng một class** có thể có **nhiều phương thức cùng tên nhưng khác** nhau về **số lượng tham số** hoặc **kiểu dữ liệu tham số.**
- **Overriding (ghi đè):** Được sử dụng trong trường hợp **lớp con** kế thừa từ lớp cha và muốn **định nghĩa lại một phương thức đã có mặt ở lớp cha**. Một lớp cha thông thường có thể có nhiều lớp con kế thừa, tuy nhiên phương thức ở lớp cha có thể phù hợp với lớp con này nhưng không phù hợp với lớp con khác, do đó lớp con cần ghi đè lại phương thức đó cho phù hợp. Các phương thức được ghi đè **giống nhau hoàn toàn** về tên, danh sách tham số, kiểu dữ liệu trả về.

Câu 1.10. Hàm thuần ảo là gì? Lớp trừu tượng là gì? Cho ví dụ minh họa.

- **Hàm thuần ảo:**
  - **Khái niệm:** Một hàm được **tạo ra mà không có nội dung gì** để **phục vụ cho các lớp dẫn xuất** kế thừa từ lớp cơ sở ghi đè lại.
- **Lớp trừu tượng:**
  - **Khái niệm:** Một **lớp có ít nhất một hàm thuần ảo** được gọi là lớp trừu tượng. **Mục đích** lớp trừu tượng được tạo ra là để **các lớp dẫn xuất kế thừa lại.**
- **Ví dụ:**

```
// Shape là lớp trừu tượng
class Shape {
public:
   virtual int DienTich() = 0; // Lớp thuần ảo
   virtual int ChuVi() = 0; // Lớp thuần ảo
};
```

5.2. Dạng câu 2: Thiết kế các lớp đơn giản

Câu 2 trong đề thi các năm gần đây thường yêu cầu ta phải thiết kế các lớp đơn giản như phân số, đa thức, ngày, thời gian, mảng động, … Ở đây chủ yếu áp dụng các kiến thức về phương thức thiết lập, nạp chồng toán tử, cấp phát động, …

Link các lời giải tham khảo:

https://github.com/bht-cnpm-uit/OOP.git

Câu 2.1 (Đề 2022 – 2023) (HK1)

Xây dựng class `IntArr` để hàm main hoạt động đúng như mong đợi.

```
class IntArr {
private:
   int count; // tổng số lượng phần tử có trong values
   int* values; // mảng các số nguyên đang có trong đối tượng hiện tại
public:
   /* Sinh viên bổ sung đầy đủ các thành phần cần thiết để hàm main hoạt động
   như mong đợi */
};
int main() {
   IntArr l1; // tạo mảng không chứa bất kì phần tử nào
   IntArr l2(3, 2); /* tạo một mảng với 3 phần tử, tất cả phần tử đều có giá trị
                       là 2 */
   IntArr l3(2);//tạo một mảng với 2 phần tử, tất cả phần tử đều có giá trị là 0
   IntArr l4 = l2.concat(l3); /* tạo ra một IntArr mới có nội dung là kết quả
   của việc nối các phần tử l3 vào cuối các phần tử của l2 theo thứ tự */
   l2.push(3); // thêm số 3 vào cuối danh sách trong đối tượng l2
   cin >> l2;  /* Xoá các giá trị hiện có trong l2 và cho phép người dùng nhập
   số lượng phần tử mới và giá trị các phần tử mới vào l2 (cần xoá các vùng nhớ
   không sử dụng nếu có) */
   cout << l2; // in ra các số nguyên có trong danh sách
   /* Khi vượt quá phạm vi sử dụng cần huỷ tất cả các vùng nhớ được cấp phát cho
   các values của IntArr */
   return 0;
}
```

Source code tham khảo:

Lưu ý: Dựa vào các câu lệnh trong hàm main để xác định đúng các phương thức cần định nghĩa. Khi xây dựng các lớp liên quan tới cấp phát động thì nên định nghĩa các phương thức thiết lập sao chép, toán tử gán cho phù hợp mặc dù đề không yêu cầu (Xem lại mục **2.8.2** phần phương thức phá hủy và phương thức thiết lập sao chép).

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

class IntArr {
   friend istream& operator>> (istream& is, IntArr& rhs);
   friend ostream& operator<< (ostream& os, const IntArr& rhs);
private:
   int count;
   int* values;
public:
   IntArr();
   IntArr(int size);
   IntArr(int size, int init);
   IntArr(const IntArr& rhs);
   IntArr& operator= (const IntArr & rhs);
   IntArr concat(const IntArr& rhs);
   void push(int val);
   ~IntArr();
};
IntArr::IntArr() {
   count = 0;
   values = NULL;
}
IntArr::IntArr(int size) {
   count = size;
   values = new int[count];
   for (int i = 0; i < size; ++i) {
      values[i] = 0;
   }
}
IntArr::IntArr(int size, int init) {
   count = size;
   values = new int[count];
   for (int i = 0; i < count; ++i) {
      values[i] = init;
```

```
    }
}
IntArr::IntArr(const IntArr& rhs) {
    count = rhs.count;
    values = new int[count];
    for (int i = 0; i < count; ++i) {
        values[i] = rhs.values[i];
    }
}
IntArr& IntArr::operator= (const IntArr& rhs) {
    delete[] this->values;
    count = rhs.count;
    values = new int[count];
    for (int i = 0; i < count; ++i) {
        values[i] = rhs.values[i];
    }
    return *this;
}
IntArr IntArr::concat(const IntArr& rhs) {
    IntArr ret(this->count + rhs.count);
    int iRet = 0;
    for (int i = 0; i < this->count; ++i) {
        ret.values[iRet] = this->values[i];
        ++iRet;
    }
    for (int i = 0; i < rhs.count; ++i) {
        ret.values[iRet] = rhs.values[i];
        ++iRet;
    }
    return ret;
}
void IntArr::push(int val) {
    IntArr temp(1, val);
    *this = this->concat(temp);
}
IntArr::~IntArr() {
    cout << "Destructor has been called" << endl;
    delete[] values;
}
istream& operator>> (istream& is, IntArr& rhs) {
    delete[] rhs.values;
    cout << "Nhap so luong: ";
    is >> rhs.count;
```

```
   rhs.values = new int[rhs.count];
   cout << "Nhap cac phan tu :\n";
   for (int i = 0; i < rhs.count; ++i) {
      is >> rhs.values[i];
   }
   return is;
}
ostream& operator<< (ostream& os, const IntArr& rhs) {
   for (int i = 0; i < rhs.count; ++i) {
      os << rhs.values[i] << " ";
   }
   return os;
}
```

Câu 2.2 (Đề 2018 – 2019)

Xây dựng lớp Thời gian (giờ, phút, giây). Định nghĩa các phép toán:

++ để tăng thời gian thêm 1 giây.

>> và << để nhập, xuất dữ liệu thời gian.

+ để cộng 2 thời gian.

Source code tham khảo:

```
class Time {
private:
   long seconds = 0;
public:
   friend istream& operator>> (istream& is, Time& rhs);
   friend ostream& operator<< (ostream& os, const Time& rhs);
   Time& operator++();
   Time operator++(int);
   Time operator+ (const Time& rhs);
};
istream& operator>> (istream& is, Time& rhs) {
   long gio = 0, phut = 0, giay = 0;
   cout << "Nhap gio: ";
   is >> gio;
   cout << "Nhap phut: ";
   is >> phut;
   cout << "Nhap giay: ";
   is >> giay;
```

```
    rhs.seconds = gio * 3600 + phut * 60 + giay;
    return is;
}
ostream& operator<< (ostream& os, const Time& rhs) {
    os << rhs.seconds / 3600 << ":";
    os << (rhs.seconds % 3600) / 60 << ":";
    os << rhs.seconds % 60;
    return os;
}
Time& Time::operator++() {
    ++seconds;
    return *this;
}
Time Time::operator++(int) {
    Time ret = *this;
    ++seconds;
    return ret;
}
Time Time::operator+ (const Time& rhs) {
    Time ret;
    ret.seconds = (this->seconds + rhs.seconds) % (3600 * 24);
    return ret;
}
```

Câu 2.3 (Đề 2016-2017) (HK3)

Xây dựng lớp đa thức bậc hai (1 điểm) để thể hiện các đa thức bật hai có dạng:

$$f(x) = ax^2 + bx + c \ \ (a \neq 0)$$

Xây dựng các phương thức: (2 điểm)

a. Phương thức cho phép xác định giá trị của đa thức ứng với $x = \ x_0$ (tính $f(x_0)$).

b. Phép toán cộng (operator +) để cộng hai đa thức bậc hai.

Source code tham khảo:

```
class DaThucBacHai {
private:
    float a;
    float b;
    float c;
public:
```

```
    float TinhGiaTri(DaThucBacHai dt, float x0);
    DaThucBacHai operator+(const DaThucBacHai&);
};
float DaThucBacHai::TinhGiaTri(DaThucBacHai dt, float x0) {
    return a * pow(x0, 2) + b * x0 + c;
}
DaThucBacHai DaThucBacHai::operator+(const DaThucBacHai & dt) {
    DaThucBacHai tong;
    tong.a = this->a + dt.a;
    tong.b = this->b + dt.b;
    tong.c = this->c + dt.c;
    return tong;
}
```

Câu 2.4 (Đề 2017-2018) (HK1)

Hãy định nghĩa lớp Date thích hợp để chương trình dưới đây không bị lỗi biên dịch và chạy đúng. Lưu ý rằng không được chỉnh sửa hàm main và sinh viên cần viết cả các lệnh #include thích hợp.

```
int main() {
    Date ng1; // ng1 sẽ có giá trị là ngày 1 tháng 1 năm 1
    Date ng2(2017, 1); // ng2 sẽ có giá trị là ngày 1 tháng 1 năm 2017
    Date ng3(2017, 1, 7); // ng3 sẽ có giá trị là ngày 7 tháng 1 năm 2017
    cin >> ng1;
    ng1++;
    cout << ng1;
    if (ng1 < ng2)
        cout << "Ngay 1 truoc ngay 2" << endl;
    else
        cout << "Ngay 1 khong truoc ngay 2" << endl;
    return 0;
}
```

Source code tham khảo:

```
#include <iostream>
#include <string>
using namespace std;
class Date {
    friend istream& operator>> (istream& is, Date& rhs);
    friend ostream& operator<< (ostream& os, const Date& rhs);
```

```
private:
   int ngay;
   int thang;
   int nam;
public:
   Date();
   Date(int nam, int thang);
   Date(int nam, int thang, int ngay);
   int IsNhuan();
   bool operator< (const Date& rhs);
   Date operator++(int);
};

Date::Date() {
   nam = 1;
   thang = 1;
   ngay = 1;
}
Date::Date(int nam, int thang) {
   this->nam = nam;
   this->thang = thang;
   ngay = 1;
}
Date::Date(int nam, int thang, int ngay) {
   this->nam = nam;
   this->thang = thang;
   this->ngay = ngay;
}
int Date::IsNhuan() {
   if ((nam % 4 == 0 && nam % 100 != 0) || nam % 400 == 0)
      return 1;
   return 0;
}
istream& operator>> (istream& is, Date& rhs) {
   cout << "Nhap ngay: ";
   is >> rhs.ngay;
   cout << "Nhap thang: ";
   is >> rhs.thang;
   cout << "Nhap nam: ";
   is >> rhs.nam;
   return is;
}
```

```
ostream& operator<< (ostream& os, const Date& rhs) {
   os << rhs.ngay << "/";
   os << rhs.thang << "/";
   os << rhs.nam;
   return os;
}
bool Date::operator< (const Date& rhs) {
   if (nam < rhs.nam)
      return true;
   if (thang < rhs.thang)
      return true;
   if (ngay < rhs.ngay)
      return true;
   return false;
}
Date Date::operator++(int) {
   Date ret = *this;
   int ngayTrongThang[13] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 };
   if (IsNhuan()) {
      ngayTrongThang[2] = 29;
   }
   ++ngay;
   if (ngay > ngayTrongThang[thang]) {
      ++thang;
      if (thang > 12) {
         ++nam;
         thang = 1;
      }
      ngay = 1;
   }
   return ret;
}
```

Câu 2.5 (Đề 2017-2018) (HK2)

Xây dựng lớp Đa thức bậc n với các toán tử >>, <<, +. (3 điểm)

Source code tham khảo:

```
class DaThuc {
private:
   int n;
   float a[100];
public:
   DaThuc operator+(const DaThuc&);
   friend istream& operator>> (istream& is, DaThuc& rhs);
   friend ostream& operator<< (ostream& os, const DaThuc& rhs);
};
istream& operator>> (istream& is, DaThuc& rhs) {
   cout << "Nhap bac cua da thuc: ";
   is >> rhs.n;
   for (int i = rhs.n; i >= 0; i--) {
      cout << "Nhap he so cua don thuc bac " << i << ": ";
      is >> rhs.a[i];
   }
   return is;
}
ostream& operator<< (ostream& os, const DaThuc& rhs) {
   cout << "Da thuc: " << endl;
   for (int i = rhs.n; i > 0; i--) {
      cout << rhs.a[i] << "x^" << i << " + ";
   }
   cout << rhs.a[0] << endl;
   return os;
}
DaThuc DaThuc::operator+(const DaThuc& P) {
   DaThuc temp;
   if (this->n > P.n)
      temp.n = this->n;
   else
      temp.n = P.n;
    for (int i = temp.n; i >= 0; i--) {
        temp.a[i] = 0;
    }
    for (int i = this->n; i >= 0; i--) {
        temp.a[i] += a[i];
    }
```

```
    for (int i = P.n; i >= 0; i--) {
        temp.a[i] += P.a[i];
    }
    return temp;
}
```

Câu 2.6 (Đề 2019-2020)

Cho lớp Phân số (PhanSo). Hãy khai báo và định nghĩa các phương thức cần thiết để các đối tượng thuộc lớp PhanSo có thể thực hiện được các câu lệnh sau:

```
PhanSo a(5, 3);
PhanSo b, c, kq;
cin >> b >> c;
kq = a + b + 5 + c;
cout << "Ket qua la: " << kq;
if (a == b)
   cout << "Phan so a bang phan so b" << endl;
```

Source code tham khảo:

```
class PhanSo {
private:
   int ts, ms;
public:
   PhanSo();
   PhanSo(int a);
   PhanSo operator+(const PhanSo&);
   bool operator==(PhanSo&);
   friend istream& operator>>(istream& is, PhanSo& x);
   friend ostream& operator<<(ostream& os, const PhanSo& x);
};
PhanSo::PhanSo() {
   ts = 0;
   ms = 1;
}
PhanSo::PhanSo(int a) {
   ts = a;
   ms = 1;
}
PhanSo PhanSo::operator + (const PhanSo& y) {
   PhanSo temp;
   temp.ts = this->ts * y.ms + y.ts * this->ms;
```

```
    temp.ms = this->ms * y.ms;
    return temp;
}
bool PhanSo::operator ==(PhanSo& y) {
    float s1 = this->ts / this->ms;
    float s2 = y.ts / y.ms;
    if (s1 == s2)
        return true;
    return false;
}
istream& operator>>(istream& is, PhanSo& x) {
    cout << "Nhap tu so: ";
    is >> x.ts;
    cout << "Nhap mau so: ";
    is >> x.ms;
    return is;
}
ostream& operator<<(ostream& os, const PhanSo& x) {
    os << "Tu so: ";
    os << x.ts;
    os << "Mau so: ";
    os << x.ms;
    return os;
}
```

Câu 2.7 (Đề 2013-2014) (Đọc thêm)

a) Xét đoạn chương trình sau:

```
#include <iostream>
using namespace std;
class A {
public:
    A() {
        cout << "Constructing A " << endl;
    }
    ~A() {
        cout << "Destructing A " << endl;
    }
};
class B : public A {
public:
    B() {
        cout << "Constructing B " << endl;
    }
```

```
   ~B() {
      cout << "Destructing B " << endl;
   }
};
int main() {
   B b1;
   return 0;
}
```

Hãy cho biết kết quả xuất ra màn hình khi thực thi đoạn chương trình trên. Giải thích ngắn gọn tại sao có kết quả đó.

Lời giải tham khảo:

Kết quả khi xuất ra màn hình:

```
Constructing A
Constructing B
Destructing B
Destructing A
```

Giải thích:

Khi một đối tượng của lớp con được khởi tạo mặc định, **constructor mặc định của lớp cha được gọi trước** để khởi tạo dữ liệu mặc định cho các thuộc tính chung, sau đó constructor của lớp con mới bắt đầu thực hiện việc gán dữ liệu cho các thuộc tính riêng. Phương thức phá hủy thì ngược lại, khi một destructor của lớp con được gọi, nó sẽ **thu hồi các tài nguyên đã cấp phát cho các thuộc tính riêng trước**, rồi sau đó destructor của lớp cha mới được gọi để dọn dẹp các thuộc tính chung

b) Xét đoạn chương trình sau:

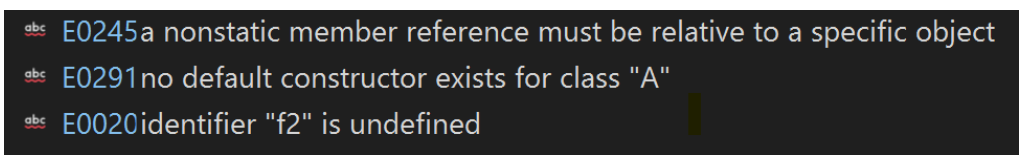
```
#include <iostream>
using namespace std;
class A {
private:
   int x;
public:
   A(int t) {
      x = t;
   }
   static void f() {
      cout << x;
   }
   int f2() {
      return x;
   }
```

```
};
void main() {
   A a;
   f2(a);
}
```

Cho biết đoạn chương trình trên khi biên dịch có lỗi xảy ra hay không? Nếu có lỗi, hãy chỉ ra các lỗi đó và sửa lỗi để chương trình có thể thực thi được.

Lời giải tham khảo:

Đoạn chương trình trên có 3 lỗi:



- Lỗi thứ 1:
  - Phương thức `f()` là thành phần `static` nên nó không được gắn với một đối tượng nào, thế nên bên trong phương thức `static` ta không thể truy cập vào thuộc tính của một đối tượng cụ thể. Trong đoạn code trên ta muốn xuất thuộc tính x nhưng mà x ở đâu ra? Thông thường ta nói x là thuộc tính của đối tượng đang gọi phương thức, tuy nhiên phương thức `static` có thể được gọi độc lập không cần thông qua một đối tượng nào.
  - Cách sửa: Bỏ từ khóa `static` đi, lúc này phương thức trở thành phương thức bình thường.
- Lỗi thứ 2:
  - Thiếu constructor mặc định: khi ta định nghĩa bất kì một constructor nào, chương trình sẽ không tự định nghĩa constructor mặc định cho ta.
  - Cách sửa: thêm constructor mặc định:

```
A() {
   x = 0;
}
```

- Lỗi thứ 3:
  - Sai dòng `f2(a)`, vì hàm `f2` là phương thức thuộc lớp `A`. Cú pháp để gọi một phương thức thuộc lớp phải là: <object>.method()
  - Cách sửa: đổi thành `a.f2()`

Câu 2.8 (Đề 2014-2015) (Đọc thêm)

a. Xét lớp phân số được khai báo như sau:

```
class PhanSo {
private:
   int ts, ms;
public:
   PhanSo(int ts = 0, int ms = 1);
   PhanSo operator+ (PhanSo);
};
```

Hãy cho biết trong các dòng lệnh sau đây, dòng nào có lỗi xảy ra, giải thích và sửa lỗi nếu có:

```
PhanSo a, b(3, 4), c(2, 5);
a = b + c;
a = b + 3;
a = 5 + c;
```

Lời giải tham khảo:

Dòng lệnh `a = 5 + c` bị sai vì toán tử cộng đang được nạp chồng dưới dạng phương thức của lớp nên toán hạng đầu tiên phải là một đối tượng thuộc lớp `PhanSo`. Dòng lệnh `a = b + 3` không sai vì chương trình trên có cơ chuyển kiểu tự động bằng constructor khi truyền vào phương thức một đối số là số nguyên, câu lệnh này có thể được hiểu là:

```
a.operator+(PhanSo(3));
```

Cách sửa: Để toán hạng đầu tiên không nhất thiết phải là một đối tượng của lớp, ta phải nạp chồng toán tử cộng bằng hàm bên ngoài như sau:

```
class PhanSo { //...
     friend PhanSo operator+(const PhanSo& ps1,const PhanSo& ps2);
};
PhanSo operator+(const PhanSo& ps1, const PhanSo& ps2) {
    PhanSo kq;
    kq.tu = ps1.tu * ps2.mau + ps1.mau * ps2.tu;
    kq.mau = ps1.mau * ps2.mau;
    return kq;
}
```

Khi đó dòng lệnh `a = 5 + c` có thể được hiểu như này: `operator+(PhanSo(5),a);`

5.3. Dạng câu 3 (Thiết kế lớp phức tạp sử dụng kế thừa và đa hình)

Để làm được dạng bài này, trước hết, phải bình tĩnh và không được hoảng loạn sau khi đọc đề:v. Đề câu 3 thường nhiều chữ khiến sinh viên lười đọc và đôi khi có các thông tin nhiễu để đánh lừa người đọc, tuy nhiên cấu trúc bài giải thường không quá phức tạp. Thế nên cần bình tĩnh đọc đề và xác định đúng nội dung của đề theo các bước như sau:

Bước 1: Xác định được các lớp đối tượng có trong đề bài:

- Mối quan hệ giữa các lớp: Lớp nào là lớp cha, lớp nào là lớp con, lớp nào là một thuộc tính của lớp khác?
- Các lớp con đa số sẽ được liệt kê sẵn trong đề bài, và lớp cha sẽ là lớp trừu tượng được đúc kết từ các điểm chung của lớp con.
- Thông thường sẽ có thêm một lớp "danh sách" gì đó (có thể gồm một mảng các con trỏ thuộc lớp cha) để biểu diễn một danh sách các đối tượng trong đề.

Bước 2: Thiết kế thuộc tính:

- Thuộc tính nào mà tất cả các lớp con đều có thì sẽ được khai báo ở lớp cha.
- Các thuộc tính riêng chỉ xuất hiện ở một số lớp con thì nên đặt ở lớp con tương ứng.
- Hạn chế để lớp con sỡ hữu các thuộc tính mà nó không cần dùng tới.

Bước 3: Thiết kế phương thức:

- Dựa vào các yêu cầu của đề bài để xác định phương thức cho từng lớp.
- Tìm ra các phương thức ở lớp cha trước, sau đó xác định phương thức nào nên là phương thức ảo.
- Tiếp theo là xác định xem các lớp con sẽ định nghĩa thêm các phương thức mới nào và cần ghi đè những phương thức ảo nào từ lớp cha.
- Lưu ý là những phương thức nào mà chỉ lớp con mới cần sử dụng thì không nên đặt ở lớp cha.

Một số tip khác:

- Khi vẽ sơ đồ lớp thì phải ghi đầy đủ kiểu dữ liệu của các thuộc tính, kiểu trả về của các phương thức, vẽ đường nối biểu diễn các mối quan hệ (kế thừa, một nhiều, ...)
- Khi code giấy thì nên khai báo hết tất cả các phương thức bên trong lớp trước rồi định nghĩa sau.
- Khi thao tác trên nhiều đối tượng, để áp dụng tính đa hình, ta sẽ thường sử dụng một mảng các con trỏ thuộc lớp cha. Trước khi nhập từng đối tượng trong mảng, ta sẽ cấp phát vùng nhớ cho các biến con trỏ tương ứng với loại lớp con mà ta muốn, ví dụ:

```
// Mảng các con trỏ thuộc lớp cha
LopCha* arr[50];
// Nhập số lượng đối tượng sẽ quản lý
int n; cin >> n;
// Khởi tạo các đối tượng lớp con
for (int i = 0; i < n; i++) {
   int loai; cin >> loai;
   if (loai == 1) arr[i] = new LopCon1();
   if (loai == 2) arr[i] = new LopCon2();
   ...
}
```

Câu 3.1 (Đề 2019-2020)

Trước hết phải khẳng định, đất đai là nguồn tài nguyên vô cùng quý giá, là tài sản quan trọng của quốc gia, là tư liệu sản xuất,... Đặc biệt, đất đai là điều kiện cần cho mọi hoạt động sản xuất và đời sống. Ở nước ta, khi còn nhiều người sống nhờ vào nông nghiệp, thì đất đai càng trở thành nguồn lực rất quan trọng.

Muốn phát huy tác dụng của nguồn lực đất đai, ngoài việc bảo vệ đất của quốc gia, còn phải quản lý đất đai hợp lý, nâng cao hiệu quả sử dụng đất sao cho vừa đảm bảo được lợi ích trước mắt, vừa tạo điều kiện sử dụng đất hiệu quả lâu dài để phát triển bền vững đất nước.

Hiện nay, ở Việt Nam đất đai được phân chia thành 2 loại chính sau:

- Đất nông nghiệp.
- Đất phi nông nghiệp (đất ở).

Quan điểm nhất quán của Đảng, Nhà nước và nhân dân ta đã được xác định từ năm 980 đến nay là đất đai thuộc sở hữu toàn dân, do Nhà nước đại diện chủ sở hữu và thống nhất quản lý. Để góp phần nâng cao hiệu quả quản lý nhà nước về đất đai, mỗi thửa đất được nhà nước quản lý và **cấp quyền sử dụng** cho một hoặc nhiều người dân (nhà nước cho phép nhiều người dân có thể đồng sở hữu quyền sử dụng đất) có nhu cầu sử dụng (**Giấy chứng nhận quyền sử dụng đất** hay còn được gọi là **Sổ hồng**).

- Với các **thửa đất nông nghiệp**, thông tin cần quản lý gồm: số giấy chứng nhận (chuỗi), người sở hữu quyền sử dụng đất (gồm họ và tên, năm sinh, CMND, địa chỉ thường trú), số thửa đất, số tờ bản đồ, địa chỉ thửa đất, diện tích (m^2), thời gian sử dụng (được sử dụng đến năm nào), ngày cấp, đơn giá thuế phải đóng cho nhà nước hàng năm/$1m^2$.
- Với các **thửa đất phi nông nghiệp (đất ở)**, thông tin cần quản lý gồm: số giấy chứng nhận (chuỗi), người sở hữu quyền sử dụng đất (gồm họ và tên, năm sinh, CMND, địa chỉ thường trú), số thửa đất, số tờ bản đồ, địa chỉ thửa đất, diện tích (m^2), ngày cấp, đơn giá thuế phải đóng cho nhà nước hàng năm/$1m^2$.

Áp dụng kiến thức lập trình hướng đối tượng (kế thừa, đa hình) thiết kế sơ đồ chi tiết các lớp đối tượng (1 điểm) và khai báo các lớp (1 điểm) để xây dựng chương trình thực hiện các yêu cầu sau:

1. Tạo danh sách các giấy chứng nhận quyền sử dụng đất mà nhà nước đã cấp cho người dân. (1 điểm)
2. Tính tiền thuế mà người sử dụng đất phải đóng cho nhà nước và cho biết thửa đất nào (thông tin thửa đất) có tiền thuế phải đóng nhiều nhất. (1 điểm)
3. Xuất ra màn hình thông tin các thửa đất nông nghiệp đã hết thời hạn sử dụng (năm sử dụng < năm hiện tại). (1 điểm)

Lưu ý: Các thông tin trong đề chỉ mô phỏng các thông tin với mục tiêu để sinh viên vận dụng kiến thức lập trình hướng đối tượng. Do vậy, các thông tin trong đề KHÔNG nhất thiết phải đúng hoặc khớp với các thông tin hiện tại trong thế giới thực. Sinh viên cần bám sát các mô tả trong đề thi để làm bài.

Phân tích đề:

Bước 1: Xác định các lớp

Vừa đọc đề ta thấy được có hai lớp đối tượng chính là **Đất nông nghiệp** và **Đất ở**. Ta sẽ tạo ra một **lớp cơ sở** có tên là **Đất đai** để 2 lớp trên kế thừa. Theo yêu cầu ở câu 1 và để tiện làm các câu 2, 3 thì ta sẽ tạo thêm lớp **Danh sách đất**.

Bước 2: Thiết kế thuộc tính

Ngoại trừ thuộc tính **thời gian sử dụng** chỉ có ở lớp **Đất nông nghiệp**, các **thuộc tính còn lại** ở 2 lớp con **đều giống nhau** nên ta sẽ cho tất cả các thuộc tính chung nằm ở lớp cơ sở Đất đai. Lớp Đất nông nghiệp khi kế thừa từ lớp Đất đai sẽ bổ sung thêm thuộc tính thời gian sử dụng, còn lớp Đất ở khi kế thừa từ lớp Đất đai thì sẽ không cần định nghĩa thêm thuộc tính nào nữa. Lớp

Cả 2 lớp con đều cần lưu thông tin của **người sở hữu**, mỗi người sỡ hữu thì lại có **nhiều thông tin nhỏ** khác, thế nên ta sẽ tạo thêm một lớp đối tượng là **Người sỡ hữu**. Tương tự thì cũng nên tạo thêm lớp **Ngày** để biểu diễn cho thuộc tính ngày cấp.

Lớp **Danh sách đất** sẽ là một **mảng các con trỏ** thuộc lớp Đất đai để có thể áp dụng tính đa hình

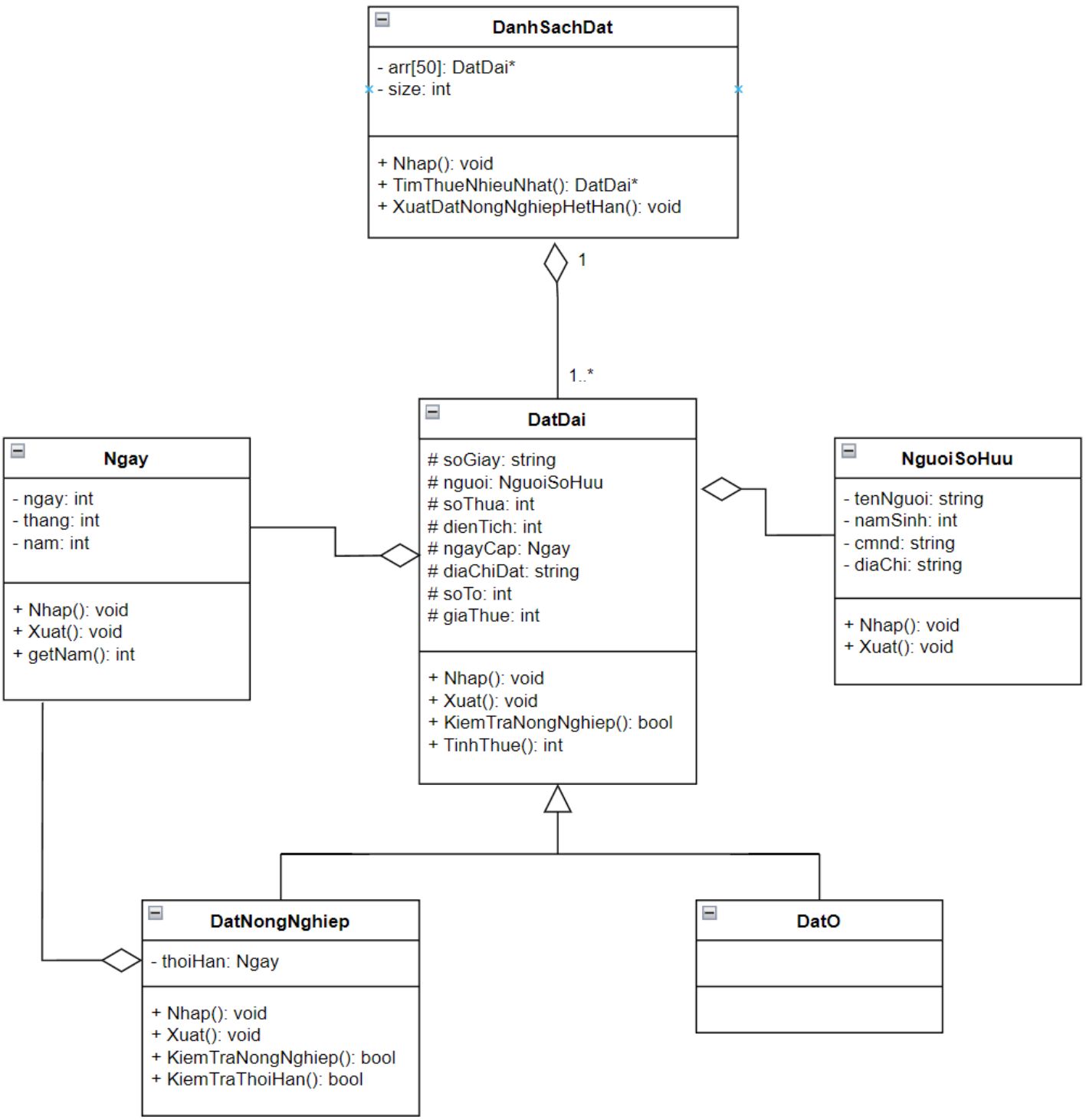
Bước 3: Thiết kế phương thức

1. Đa số các bài dạng này đều cần phải làm các **phương thức nhập, xuất** cho từng lớp đối tượng. Các lớp nhập, xuất ở lớp cơ sở sẽ là **phương thức ảo**

2. Tạo **phương thức tính tiền** ở lớp cơ sở, cả 2 loại đất có cách thức tính tiền như nhau nên không cần là phương thức ảo. Phương thức tìm đất có thuế nhiều nhất ở lớp Danh sách.

3. Tạo một **phương thức kiểm tra** ở lớp cơ sở để xem một loại đất có phải nông nghiệp không. Nó sẽ là **phương thức ảo**, phiên bản ở lớp nông nghiệp sẽ trả về giá trị true, phiên bản ở lớp đất ở trả về giá trị false. Ngoài ra **lớp nông nghiệp** sẽ có thêm phương thức **kiểm tra hết hạn.**

Sơ đồ thiết kế lớp:



Lời giải tham khảo:

https://github.com/bht-cnpm-uit/OOP.git

Source code tham khảo:

```
#include <iostream>
#include <string>
using namespace std;
class NguoiSoHuu {
private:
   string tenNguoi;
   int namSinh;
   string cmnd;
   string diaChi;
public:
   void Nhap() {
      cout << "Nhap ten nguoi so huu: ";
      cin >> tenNguoi;
      cin.ignore();
      cout << "Nhap nam sinh: ";
      cin >> namSinh;
      cout << "Nhap cmnd: ";
      cin >> cmnd;
      cin.ignore();
      cout << "Nhap dia chi nguoi so huu: ";
      cin >> diaChi;
      cin.ignore();
   }
   void Xuat() {
      cout << tenNguoi << endl << namSinh << endl
         << cmnd << endl << diaChi << endl;
   }
};
class Ngay {
private:
   int ngay;
   int thang;
   int nam;
public:
   void Nhap() {
      cout << "Nhap ngay: ";
      cin >> ngay;
      cout << "Nhap thang: ";
```

```
      cin >> thang;
      cout << "Nhap nam: ";
      cin >> nam;
   }
   void Xuat() {
      cout << ngay << "/" << thang << "/" << nam << endl;
   }
   int getNam() {
      return nam;
   }
};
class DatDai {
protected:
   string soGiay;
   NguoiSoHuu nguoi;
   int soThua;
   int dienTich;
   Ngay ngayCap;
   string diaChiDat;
   int soTo;
   int giaThue;
public:
   virtual void Nhap() {
      cout << "Nhap so giay: ";
      cin >> soGiay;
      cin.ignore();
      cout << "Nhap nguoi so huu:\n";
      nguoi.Nhap();
      cout << "Nhap so thua dat: ";
      cin >> soThua;
      cout << "Nhap dien tich dat: ";
      cin >> dienTich;
      cout << "Nhap ngay cap: ";
      ngayCap.Nhap();
      cout << "Nhap dia chi dat: ";
      cin >> diaChiDat;
      cin.ignore();
      cout << "Nhap so to ban do: ";
      cin >> soTo;
      cout << "Nhap don gia thue: ";
      cin >> giaThue;
   }
   virtual void Xuat() {
```

```
      cout << soGiay << endl;
      nguoi.Xuat();
      cout << soThua << endl << dienTich << endl;
      ngayCap.Xuat();
      cout << diaChiDat << endl << soTo << endl << giaThue << endl;
   }
   virtual bool KiemTraNongNghiep() {
      return false;
   }
   int TinhThue() {
      return dienTich * giaThue;
   }
};
class DatNongNghiep : public DatDai {
protected:
   Ngay thoiHan;
public:
   void Nhap() {
      DatDai::Nhap();
      cout << "Nhap thoi han su dung: ";
      thoiHan.Nhap();
   }
   void Xuat() {
      DatDai::Xuat();
      thoiHan.Xuat();
   }
   bool KiemTraThoiHan() {
      if (thoiHan.getNam() < 2023)
         return true;
      return false;
   }
   bool KiemTraNongNghiep() {
      return true;
   }
};

class DatO : public DatDai {};

class DanhSachDat {
protected:
   DatDai* arr[50];
   int size;
public:
```

```
    void Nhap() {
        cout << "Nhap so luong dat: ";
        cin >> size;
        for (int i = 0; i < size; ++i) {
            cout << "Nhap kieu dat: 1 = Nong nghiep, 2 = Dat o: ";
            int type;
            cin >> type;
            if (type == 1) {
                arr[i] = new DatNongNghiep();
            }
            if (type == 2) {
                arr[i] = new DatO();
            }
            arr[i]->Nhap();
        }
    }
    DatDai* TimThueNhieuNhat() {
        int maxIndex = 0;
        for (int i = 1; i < size; ++i) {
            if (arr[i]->TinhThue() > arr[maxIndex]->TinhThue()) {
                maxIndex = i;
            }
        }
        return arr[maxIndex];
    }
    void XuatDatNongNghiepHetHan() {
        for (int i = 0; i < size; ++i) {
            if (arr[i]->KiemTraNongNghiep()) {
                if (((DatNongNghiep*)arr[i])->KiemTraThoiHan()) {
                    arr[i]->Xuat();
                }
            }
        }
    }
};
int main() {
    DanhSachDat item;
    item.Nhap();
    DatDai* kq = item.TimThueNhieuNhat();
    kq->Xuat();
    item.XuatDatNongNghiepHetHan();
    return 0;
}
```

Câu 3.2 (Đề 2022-2023) (HK1)

Một tổ chức chuyên trưng bày, mua bán các sản phẩm liên quan đến nghệ thuật đang muốn xây dựng một ứng dụng để **quản lý các hoá đơn** khi bán các sản phẩm. Mỗi lần bán sản phẩm thành công, cửa hàng sẽ lưu trữ các **hoá đơn chứa thông tin sản phẩm** liên quan. Mỗi hoá đơn sẽ có thông tin: mã hoá đơn, **thông tin khách hàng**, ngày lập hoá đơn, **danh sách sản phẩm**, tổng giá (tổng giá trị các sản phẩm trong đơn hàng). Tổ chức này hiện tại chỉ kinh doanh **2 loại sản phẩm:** tranh ảnh và CD âm nhạc (tương lai có thể thay đổi sản phẩm kinh doanh khác). Mỗi sản phẩm sẽ có thông tin chung cần quản trị: mã số, tiêu đề, giá bán. Ngoài thông tin chung, các sản phẩm tranh ảnh cần thêm thông tin kích thước của bức tranh (chiều rộng, chiều cao), tên hoạ sĩ. Sản phẩm CD âm nhạc sẽ có thêm tên ca sĩ, tên đơn vị sản xuất. Mỗi khách hàng sẽ được lưu trữ các thông tin: mã khách hàng, tên khách hàng, số điện thoại.

Áp dụng tư tưởng lập trình hướng đối tượng (có sử dụng kế thừa, đa hình), anh/chị hãy:

1. (1.5 điểm) Thiết kế và vẽ sơ đồ lớp cho ứng dụng theo bài toán được mô tả

2. Cài đặt chi tiết theo sơ đồ lớp đã thiết kế và cũng như thành phần cần thiết khác để xây dựng chương trình thực hiện các tính năng sau:

a) (1.5 điểm) Nhập và xuất danh sách các hoá đơn bán hàng

b) (1 điểm) Tính tổng thu nhập của cửa hàng

c) (1 điểm) Tìm các khách hàng mua nhiều nhất ở cửa hàng (dựa vào tổng giá trị các hoá đơn khách hàng đã mua).

Lưu ý: Các thông tin trong đề thi chỉ mang tính chất giả sử, KHÔNG nhất thiết phải đúng hoặc khớp với các thông tin hiện tại trong thế giới thực. Sinh viên cần bám sát các mô tả trong đề thi và vận dụng kiến thức về lập trình hướng đối tượng để làm bài theo yêu cầu.

Phân tích đề:

Bước 1: Xác định các lớp

Vừa đọc đề ta thấy yêu cầu là phải **quản lí các hóa đơn** nên tất nhiên là phải có lớp **Hóa đơn** và **Danh sách hóa đơn**. Bên trong lớp Hóa đơn sẽ có thêm lớp **Người** để biểu diễn thông tin khách hàng, lớp **Ngày** để mô tả ngày lập hóa đơn, và lớp **Sản phẩm**. Có 2 lớp con kế thừa từ lớp Sản phẩm là **Tranh ảnh** và **CD âm nhạc**.

Bước 2: Thiết kế thuộc tính

Lớp Danh sách hóa đơn sẽ là một **mảng các Hóa đơn** (vì chỉ có 1 loại hóa đơn nên không cần dùng con trỏ). Lớp Hóa đơn có **5 thông tin** cần thể hiện, trong đó **danh sách sản phẩm** được biểu diễn bằng 1 **mảng các con trỏ thuộc lớp Sản phẩm** (vì có nhiều loại sản phẩm nên sử dụng con trỏ để áp dụng tính đa hình). Đề này đã chỉ rõ luôn cho ta biết các thuộc tính nào là chung, riêng của lớp Sản phẩm, Tranh ảnh và CD âm nhạc.

Bước 3: Thiết kế phương thức

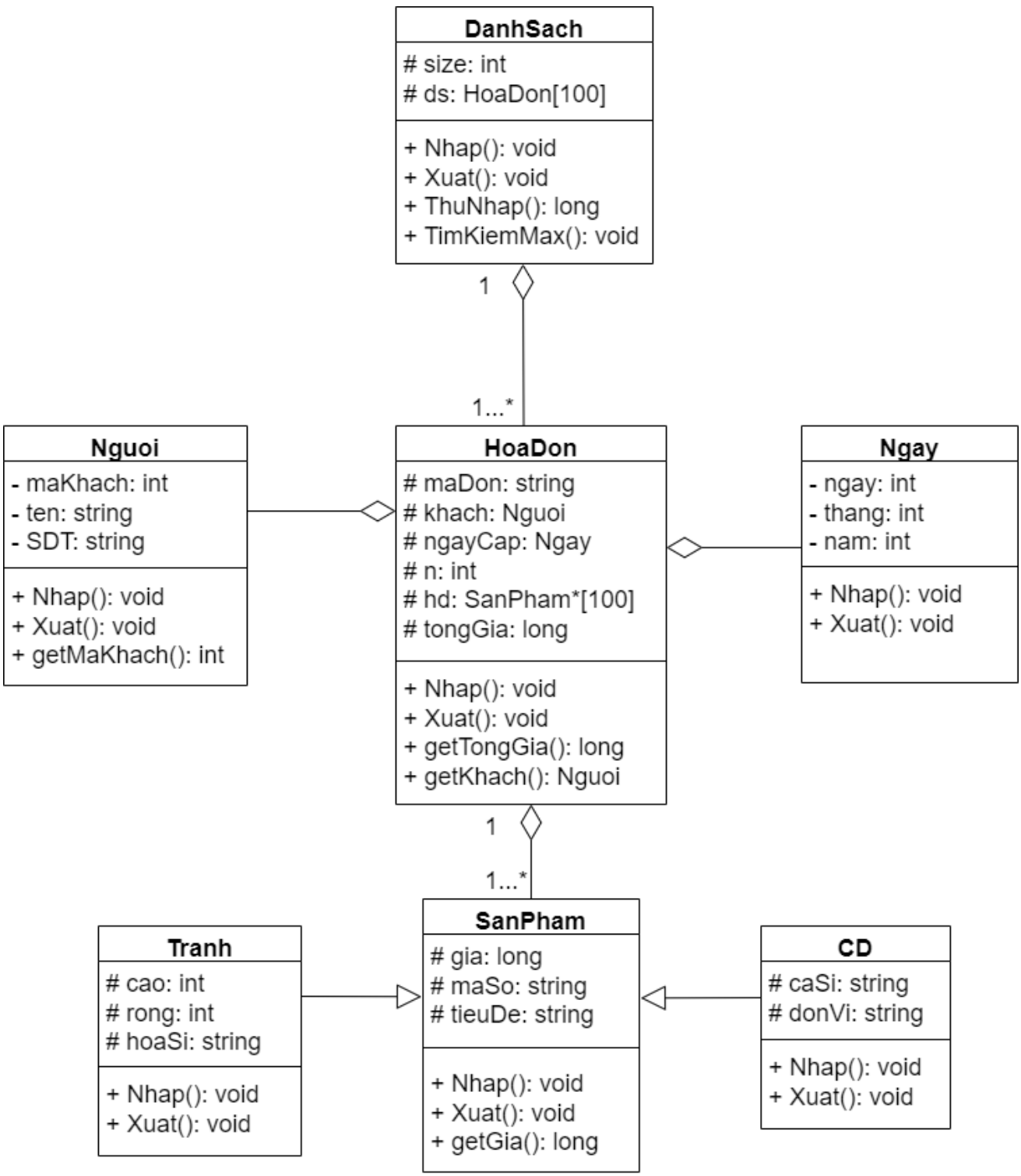
a. Các **phương thức nhập, xuất** cho các lớp. Phương thức nhập, xuất ở lớp Sản phẩm là phương thức ảo.

b. Phương thức tính **tổng giá các sản phẩm** ở lớp Hóa đơn, tính **tổng giá các hóa đơn** ở lớp Danh sách.

c. Phương thức tìm max ở lớp danh sách hóa đơn.

Ngoài ra cần định nghĩa thêm các **phương thức truy vấn** phù hợp.

Sơ đồ thiết kế lớp:



Lời giải tham khảo:

https://github.com/bht-cnpm-uit/OOP.git

Source code tham khảo:

```
#include<iostream>
#include<string>
#include<map>
using namespace std;
class Nguoi {
private:
   int maKhach;
   string ten;
   string SDT;
public:
   void Nhap();
   void Xuat();
   int getMaKhach();
};
class Ngay {
private:
   int ngay;
   int thang;
   int nam;
public:
   void Nhap();
   void Xuat();
};
class SanPham {
protected:
   string maSo;
   string tieuDe;
   long gia;
public:
   virtual void Nhap();
   virtual void Xuat();
   long getGia();
};
class Tranh :public SanPham {
protected:
   int rong;
   int cao;
   string hoaSi;
```

```
public:
   void Nhap();
   void Xuat();
};
class CD :public SanPham {
protected:
   string caSi;
   string donVi;
public:
   void Nhap();
   void Xuat();
};
class HoaDon {
protected:
   string maDon;
   Nguoi khach;
   Ngay ngayCap;
   int n;
   SanPham* hd[100];
   long tongGia;
public:
   void Nhap();
   void Xuat();
   long getTongGia();
   Nguoi getKhach();
};
class DanhSach {
protected:
   int size;
   HoaDon ds[100];
public:
   void Nhap();
   void Xuat();
   long ThuNhap();
   void TimKiemMax();
};
void Ngay::Nhap() {
   cout << "Nhap ngay: ";
   cin >> ngay;
   cout << "Nhap thang: ";
   cin >> thang;
   cout << "Nhap nam: ";
   cin >> nam;
```

```
}
void Nguoi::Nhap() {
   cout << "Nhap ma khach hang: ";
   cin >> maKhach;
   cout << "Nhap ho ten: ";
   cin >> ten;
   cout << "Nhap so dien thoai: ";
   cin >> SDT;
}
void Nguoi::Xuat() {
   cout << "Ma khach hang: " << maKhach << endl;
   cout << "Ho ten: " << ten << endl;
   cout << "So dien thoai: " << SDT << endl;
}
int Nguoi::getMaKhach() {
   return maKhach;
}
void Ngay::Xuat() {
   cout << ngay << "/" << thang << "/" << nam << endl;
}
void SanPham::Nhap() {
   cout << "Nhap ma so: ";
   cin >> maSo;
   cout << "Nhap tieu de: ";
   cin >> tieuDe;
   cout << "Nhap gia ban: ";
   cin >> gia;
}
void SanPham::Xuat() {
   cout << "Ma so: " << maSo << endl;
   cout << "Tieu de: " << tieuDe << endl;
   cout << "Gia ban: " << gia << endl;
}
long SanPham::getGia() {
   return gia;
}
void Tranh::Nhap() {
   SanPham::Nhap();
   cout << "Nhap kich thuoc: " << endl;
   cout << "Nhap chieu rong: ";
   cin >> rong;
   cout << "Nhap chieu cao: ";
   cin >> cao;
```

```
    cout << "Nhap ten hoa si: ";
    cin >> hoaSi;
}
void Tranh::Xuat() {
    SanPham::Xuat();
    cout << "Chieu rong: " << rong << endl;
    cout << "Chieu cao: " << cao << endl;
    cout << "Ten hoa si: " << hoaSi << endl;
}
void CD::Nhap() {
    SanPham::Nhap();
    cout << "Nhap ten ca si: ";
    cin >> caSi;
    cout << "Nhap don vi san xuat: ";
    cin >> donVi;
}
void CD::Xuat() {
    SanPham::Xuat();
    cout << "Ten ca si: " << caSi << endl;
    cout << "Don vi san xuat: " << donVi << endl;
}
void HoaDon::Nhap() {
    cout << "Nhap ma hoa don: ";
    cin >> maDon;
    cout << "Nhap thong tin khach hang: " << endl;
    khach.Nhap();
    cout << "Nhap ngay cap: " << endl;
    ngayCap.Nhap();
    cout << "Nhap so luong san pham: ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        cout << "Nhap san pham thu " << i + 1 << endl;
        int loai;
        cout << "Nhap loai (0. Tranh anh, 1. CD): ";
        cin >> loai;
        if (loai == 0) {
            hd[i] = new Tranh();
        }
        else {
            hd[i] = new CD();
        }
        hd[i]->Nhap();
    }
```

```
}
void HoaDon::Xuat() {
   cout << "Ma hoa don: " << maDon << endl;
   cout << "Ngay cap: ";
   ngayCap.Xuat();
   cout << "Thong tin khach hang: " << endl;
   cout << "Tong gia tien: " << getTongGia() << endl;
   khach.Xuat();
   for (int i = 0; i < n; i++) {
      cout << "San pham thu " << i + 1 << endl;
      hd[i]->Xuat();
   }
}
long HoaDon::getTongGia() {
   tongGia = 0;
   for (int i = 0; i < n; i++) {
      tongGia += hd[i]->getGia();
   }
   return tongGia;
}
Nguoi HoaDon::getKhach() {
   return khach;
}
void DanhSach::Nhap() {
   cout << "Nhap so luong hoa don: ";
   cin >> size;
   for (int i = 0; i < size; i++) {
      cout << "Nhap hoa don thu " << i + 1 << endl;
      ds[i].Nhap();
   }
}
void DanhSach::Xuat() {
   for (int i = 0; i < size; i++) {
      cout << "Hoa don thu " << i + 1 << endl;
      ds[i].Xuat();
   }
}
long DanhSach::ThuNhap() {
   int s = 0;
   for (int i = 0; i < size; i++) {
      s += ds[i].getTongGia();
   }
   return s;
```

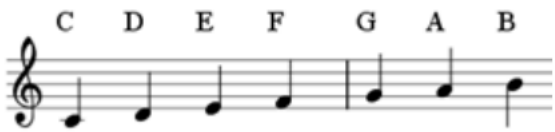
```
}
void DanhSach::TimKiemMax() {
   map<int, long> mp;
   for (int i = 0; i < size; ++i) {
      int id = ds[i].getKhach().getMaKhach();
      mp[id] += ds[i].getTongGia();
   }
   long max = 0;
   for (const auto& pair : mp) {
      if (pair.second > max) {
         max = pair.second;
      }
   }
   cout << "ID cac khach mua nhieu nhat: \n";
   for (const auto& pair : mp) {
      if (pair.second == max) {
         cout << pair.first << endl;
      }
   }
}

int main() {
   DanhSach l;
   l.Nhap();
   l.Xuat();
   cout << "Tong thu nhap: " << l.ThuNhap() << endl;
   l.TimKiemMax();
}
```

Câu 3.2 (Đề 2015-2016)

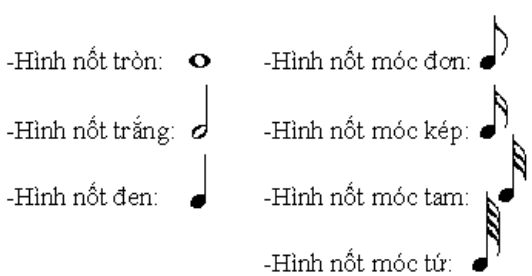
Xây dựng chương trình mô phỏng biên soạn nhạc với các mô tả ký kiệu âm nhạc như sau:

Nốt nhạc: là ký hiệu trong bản nhạc dùng để xác định *cao độ* (độ cao), *trường độ* (độ dài, độ ngân vang) của từng âm thanh được vang lên trong bản nhạc.

Có 7 ký hiệu nốt nhạc dùng để xác định *cao độ* theo thứ tự từ thấp đến cao, đó là Đô (C), Rê (D), Mi (E), Fa (F), Sol (G), La (A), và Si (B)

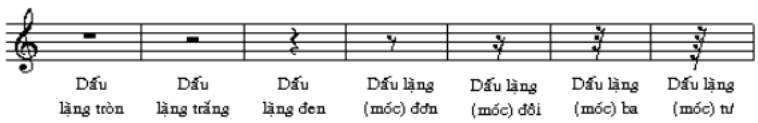


Để xác định *trường độ* của nốt nhạc có cao độ kể trên, người ta cũng dùng 7 hình nốt để thể hiện, đó là:



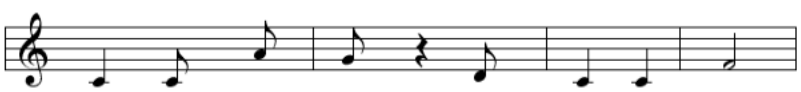
- Nốt tròn có trường độ tương đương với trường độ của 4 nốt đen
- Nốt trắng có trường độ bằng 2 nốt đen
- Nốt đen có trường độ bằng 1 phách (đơn vị thời gian trong âm nhạc- vd như 1 bước chân người đi trong không gian)
- Nốt móc đơn có trường độ bằng 1/2 nốt đen
- Nốt móc đôi có trường độ bằng 1/4 nốt đen
- Nốt móc tam có trường độ bằng 1/8 nốt đen
- Nốt móc tứ có trường độ bằng 1/16 nốt đen

Dấu lặng (Z - Zero) là ký hiệu cho biết phải ngưng, không diễn tấu âm thanh (không có cao độ) trong một thời gian nào đó. Các dấu lặng trong thời gian tương ứng (giá trị trường độ) với dạng dấu nhạc nào, thì cũng có tên gọi tương tự.



Trường độ	4	2	1	1/2	1/4	1/8	1/16

Ví dụ: Ký hiệu bản nhạc



C1 - C1/2 - A1/2 -G1/2 - Z1 - D1/2 - C1 - C1 - F2

Trường độ	1	1/2	1/2	1/2	1	1/2	1	1	2
Cao độ	C	C	A	G	Không có (Z)	D	C	C	F
Nốt	Đô đen	Đô móc đơn	La móc đơn	Sol móc đơn	Dấu lặng đen	Rê móc đơn	Đô đen	Đô đen	Fa trắng

Áp dụng kiến thức lập trình hướng đối tượng (kế thừa, đa hình) thiết kế sơ đồ chi tiết các lớp đối tượng (1.5đ) và xây dựng chương trình thực hiện các yêu cầu sau:

1. Soạn một bản nhạc (1.5đ)

2. Tìm và đếm có bao nhiêu dấu lặng đen (Q) trong bản nhạc(1đ)

3. Cho biết nốt nhạc có cao độ cao nhất trong bản nhạc (1đ)

Phân tích đề:

Bước 1 : Thiết kế lớp

Dựa vào đề hoặc nếu đề dài khó hiểu thì chúng ta có thể dựa vào các câu hỏi để mường tượng ra các đối tượng có trong lớp. Ví dụ , như ở **câu b** yêu cầu :" Tìm và đếm có bao nhiêu dấu lặng đen (Q) trong bản nhạc " thì chắc chắn trong bài chúng ta phải có đối tượng `DauLang`

Như vậy bài này khi đọc vào chúng ta có thể phát hiện được sẽ có một lớp `KyHieu` sẽ là lớp cha của lớp `NotNhac` và `DauLang`. Bên cạnh đó , để quản lí tốt hơn ta cần tạo thêm 1 lớp `BanNhac` để quản lí các `KyHieu`

Bước 2 : Xác định các thuộc tính:

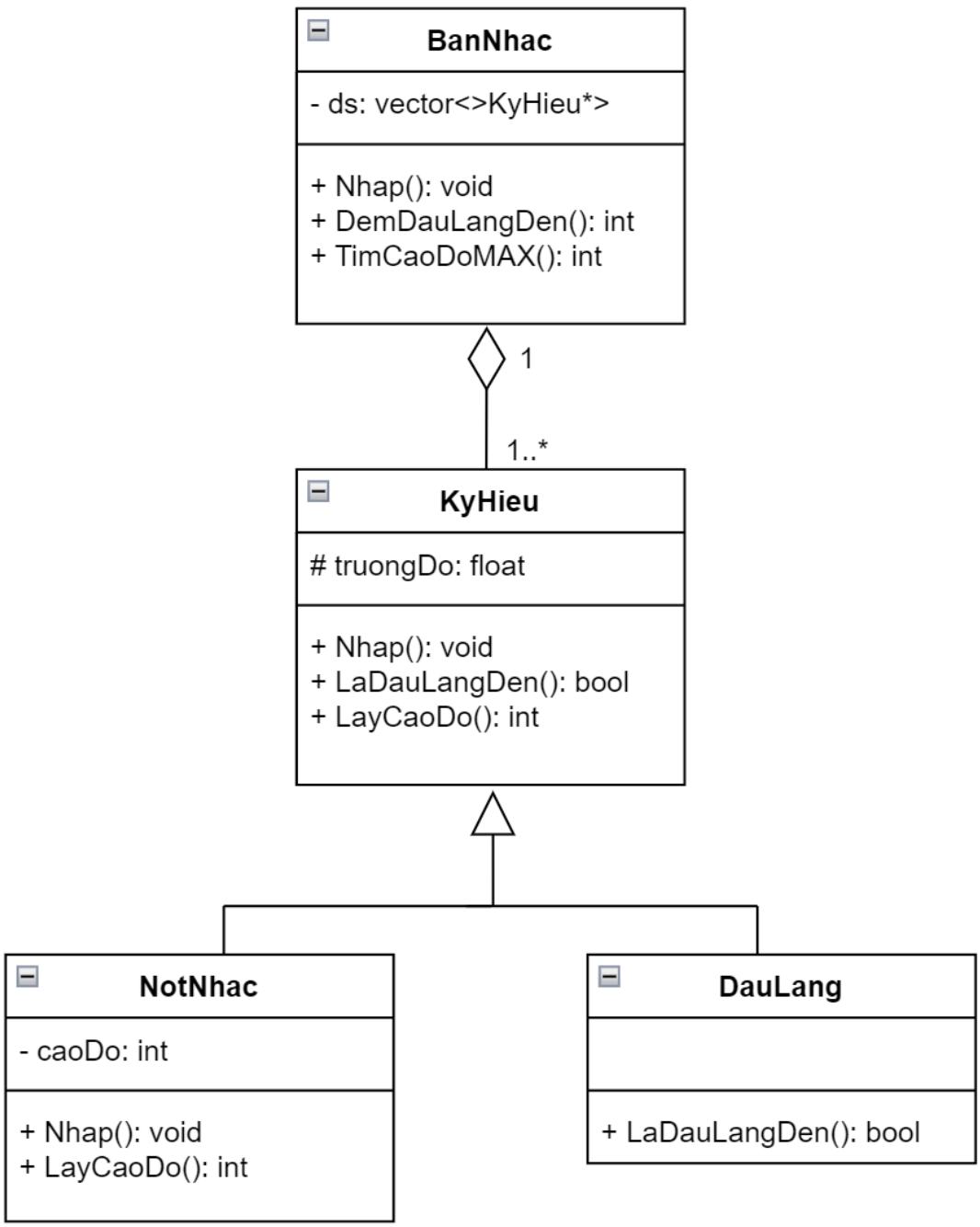
Ở đây ta dễ dàng xác định thuộc tính chung của `DauLang` và `NotNhac` là **trường độ** vì thế lớp `KyHieu` sẽ có thuộc tính là **`truongDo`** . Bên cạnh đó lớp `NotNhac` sẽ có thêm thuộc tính riêng là **`caoDo`**.

Bài này nhìn chung các thuộc tính khá dễ xác định nhưng có một điểm lưu ý là ở kiểu dữ liệu của **`caoDo`** vì đề bài không bắt buộc kiểu nào cố định nên vì thế thay vì kiểu kí tự là string ta sẽ trừu tượng nó thành kiểu dữ liệu int lưu các các cao độ tương ứng (1 – Đô ,2- Rê,…) .Từ đó giúp ta dễ dàng làm **câu c**

Bước 3: Xác định phương thức:

1. Tương tự như các bài ở trên, các đối tượng có các thuộc tính khác nhau nên việc nhập thông tin cho các đối tượng đó cũng khác nhau. Chính vì thế ở bài này, phương thức nhập lớp cơ sở `KyHieu` sẽ là phương thức ảo.
2. Tạo phương thức để kiểm tra dấu lặng đen là **`LaDauLangDen()`** . Nó sẽ là phương thức ảo và mặc định trả về false, chúng ta sẽ override nó ở lớp dẫn xuất `DauLang` khi đó nó sẽ trả về true nếu đó là dấu lặng đen.
3. Cũng tương tự phương thức **`LaDauLangDen()`**. Ta tạo một phương thức ảo để lấy cao độ là **`LayCaoDo()`** mặc định sẽ trả về **0**, nhưng ta sẽ override nó trong lớp `NotNhac` và trả về cao độ cần tìm.

Sơ đồ thiết kế lớp:



Lời giải tham khảo:

https://github.com/bht-cnpm-uit/OOP.git

Source code tham khảo:

```
#include<iostream>
#include<vector>
using namespace std;
class KyHieu {
protected:
   float truongDo;
public:
   virtual void Nhap() {
      int loai;
      cout << "Nhap vao loai hinh not(1-Tron, 2-Trang, 3-Den, 4-Moc don, 5-Moc
     kep, 6-Moc tam, 7-Moc tu): ";
      cin >> loai;
      switch (loai) {
      case 1: { truongDo = 4; break; }
      case 2: { truongDo = 2; break; }
      case 3: { truongDo = 1; break; }
      case 4: { truongDo = 0.5; break; }
      case 5: { truongDo = 0.25; break; }
      case 6: { truongDo = 0.125; break; }
      case 7: { truongDo = 0.0625; break; }
      }
   }
   virtual bool LaDauLangDen() {
      return false;
   }
   virtual int LayCaoDo() {
      return 0;
   }
};
class NotNhac : public KyHieu {
private:
   int caoDo;
public:
   void Nhap() {
      cout << "Nhap vao cao do(1-Do(C), 2-Re(D), 3-Mi(E), 4-Fa(F), 5-Sol(G), 6-
     La(A), 7-Si(B)): ";
      cin >> caoDo;
   }
```

```
    int LayCaoDo() {
        return caoDo;
    }
};
class DauLang :public KyHieu {
public:
    bool LaDauLangDen() {
        if (truongDo == 1)
            return true;
        return false;
    }
};

class BanNhac {
private:
    vector<KyHieu*> ds;
public:
    void Nhap() {
        cout << "Nhap vao so ky hieu cua ban nhac: ";
        int n;
        cin >> n;
        KyHieu* tmp;
        for (int i = 0; i < n; i++) {
            int loai;
            tmp = NULL;
            cout << "Nhap ky hieu thu " << i + 1 << " can them vao ban nhac(1 -
            NotNhac, 2 - DauLang) : ";
            cin >> loai;
            if (loai == 1) tmp = new NotNhac;
            else tmp = new DauLang;
            tmp->Nhap();
            ds.push_back(tmp);
        }
    }
    // cau b
    int DemDauLangDen() {
        int dem = 0;
        for (int i = 0; i < ds.size(); i++) {
            if (ds[i]->LaDauLangDen()) {
                dem++;
            }
        }
        return dem;
```

```
    }
    // cau c
    int TimCaoDoMAX() {
        int max = ds[0]->LayCaoDo();
        int index = 0;
        for (int i = 0; i < ds.size(); i++) {
            if (ds[i]->LayCaoDo() > max) {
                max = ds[i]->LayCaoDo();
                index = i;
            }
        }
        return index + 1;
    }
};

int main() {
    BanNhac b;
    b.Nhap();
    cout << "So dau lang den cua ban nhac la: " << b.DemDauLangDen();
    cout << "\nVi tri not nhac co cao do lon nhat la :" << b.TimCaoDoMAX();
}
```

Câu 3.3 (Đề 2017-2018)

Đầu những năm 1900, dựa trên sự hiện diện của các kháng nguyên trên màng hồng cầu, các nhà khoa học đã xác định rằng con người có 4 **nhóm máu** khác nhau: **O, A, B và AB**. Hệ thống phân loại nhóm máu này (gọi là hệ thống nhóm máu ABO) cung cấp cho bác sĩ các thông tin quan trọng để lựa chọn nhóm máu phù hợp trong việc truyền máu. Và đồng thời có thể tiên đoán được nhóm máu tương đối của người con dựa trên nhóm máu của cha mẹ theo cơ chế di truyền học.

Nhóm máu của người con khi biết được nhóm máu của cha và mẹ:

		Nhóm máu người cha				
		A	**B**	**AB**	**O**	
Nhóm máu người mẹ	**A**	A hoặc O	A, B, AB hoặc O	A, B hoặc AB	A hoặc O	**Dự đoán khả năng nhóm máu người con**
	B	A, B, AB hoặc O	B hoặc O	A, B hoặc AB	B hoặc O	
	AB	A, B hoặc AB	A, B hoặc AB	A, B hoặc AB	A hoặc B	
	O	A hoặc O	B hoặc O	A hoặc B	O	

Ngoài ra còn có thêm hệ thống phân loại **Rh (Rhesus)**

Căn cứ vào sự khác biệt khi nghiên cứu về sự vận chuyển oxy của hồng cầu thì các hồng cầu có thể mang ở mặt ngoài một protein gọi là Rhesus. Nếu có kháng nguyên D thì là nhóm Rh+ (dương tính), nếu không có là Rh- (âm tính). Các nhóm máu A, B, O, AB mà Rh- thì được gọi là âm tính A-, B-, O-, AB-. Nhóm máu Rh- chỉ chiếm 0,04% dân số thế giới. Đặc điểm của nhóm máu Rh này là chúng chỉ có thể nhận và cho người cùng nhóm máu, đặc biệt phụ nữ có nhóm máu Rh- thì con rất dễ tử vong.

Người có nhóm máu Rh+ chỉ có thể cho người cũng có nhóm máu Rh+ và nhận người có nhóm máu Rh+ hoặc Rh-

Người có nhóm máu Rh- có thể cho người có nhóm máu Rh+ hoặc Rh- nhưng chỉ nhận được người có nhóm máu Rh- mà thôi

Trường hợp người có nhóm máu Rh- được truyền máu Rh+, trong lần đầu tiên sẽ không có bất kỳ phản ứng tức thì nào xảy ra nhưng nếu tiếp tục truyền máu Rh+ lần thứ 2 sẽ gây ra những hậu quả nghiêm trọng do tai biến truyền máu. Tương tự với trường hợp mẹ Rh- sinh con (lần đầu và lần thứ hai trở đi).

- Khả năng tương thích:
  - ✓: Có thể cho - nhận.
  - ✗: Không thể cho - nhận.

Bảng khả năng tương thích hồng cầu

Người nhận	Người cho							
	O−	O+	A−	A+	B−	B+	AB−	AB+
O−	✓	✗	✗	✗	✗	✗	✗	✗
O+	✓	✓	✗	✗	✗	✗	✗	✗
A−	✓	✗	✓	✗	✗	✗	✗	✗
A+	✓	✓	✓	✓	✗	✗	✗	✗
B−	✓	✗	✗	✗	✓	✗	✗	✗
B+	✓	✓	✗	✗	✓	✓	✗	✗
AB−	✓	✗	✓	✗	✓	✗	✓	✗
AB+	✓	✓	✓	✓	✓	✓	✓	✓

Áp dụng kiến thức lập trình hướng đối tượng (kế thừa, đa hình) thiết kế sơ đồ chi tiết các lớp đối tượng (*1.5 điểm*) và xây dựng chương trình thực hiện các yêu cầu sau:

1. Nhập **danh sách các nhóm máu** của một nhóm người. (*1 điểm*)

2. Cho một bộ 3 nhóm máu của 3 người là cha, mẹ, con. Hãy kiểm tra và đưa ra kết quả nhóm máu có phù hợp với quy luật di truyền hay không? (*1 điểm*)

3. Chọn một người X trong danh sách. Hãy liệt kê tất cả các người còn lại trong danh sách có thể cho máu người X này. (*1 điểm*)

Lưu ý: Trong trường hợp sinh viên không biết về nhóm máu và di truyền học trước đây thì phải đọc kỹ thông tin trên (các thông tin trên đủ để sinh viên thực hiện các yêu cầu của đề thi) và nghiêm túc làm bài. Giám thị coi thi không giải thích gì thêm.

Phân tích đề:

Bước 1: Xác định được các lớp đối tượng có trong đề bài:

- Từ các câu hỏi, ta có thể thấy vấn đề cần giải quyết đều liên quan tới danh sách nhóm máu của từng người. Vì vậy, ta nên thiết kế một lớp `DanhSach` để phục vụ cho việc giải quyết các vấn đề ở từng câu.
- Vì đối tượng chủ yếu được đề cập trong đề thi là các nhóm máu(**tổng quát**) mà **cụ thể** là những nhóm A, B, AB , O. Cùng với mục đích áp dụng kiến thức về kế thừa, đa hình, ta sẽ xây dựng một lớp `NhomMau` là lớp cơ sở còn những lớp `A, B, AB, O` là các lớp dẫn xuất kế thừa từ lớp `NhomMau`.

Bước 2: Thiết kế thuộc tính

- Đối với lớp `DanhSach`, thuộc tính cần thiết là một mảng con trỏ đối tượng thuộc lớp `NhomMau` để gọi thực hiện các phương thức ảo theo cơ chế đa xạ (có thể lựa chọn cấp phát động, cấp phát tĩnh,...)
- Từ các thông tin của đề thi, ta có thể xác định các thuộc tính chung của các nhóm máu là **Rh** và các lớp con không có thuộc tính riêng nào.

Bước 3: Thiết kế phương thức:

- Từ các câu hỏi, ta xác định được những phương thức cần định nghĩa ở lớp `DanhSach` là phương thức **nhập thông tin**; phương thức **kiểm tra theo quy luật di truyền** để xác định nhóm máu của cha, mẹ, con có phù hợp hay không; phương thức **kiểm tra xem nhóm máu người cho có phù hợp** với người nhận. Đây là các phương thức nên được **ưu tiên định nghĩa trước.**
- Ở lớp `NhomMau`, ta cần định nghĩa các phương thức để hỗ trợ cho việc định nghĩa các hàm thành phần của lớp `DanhSach`:

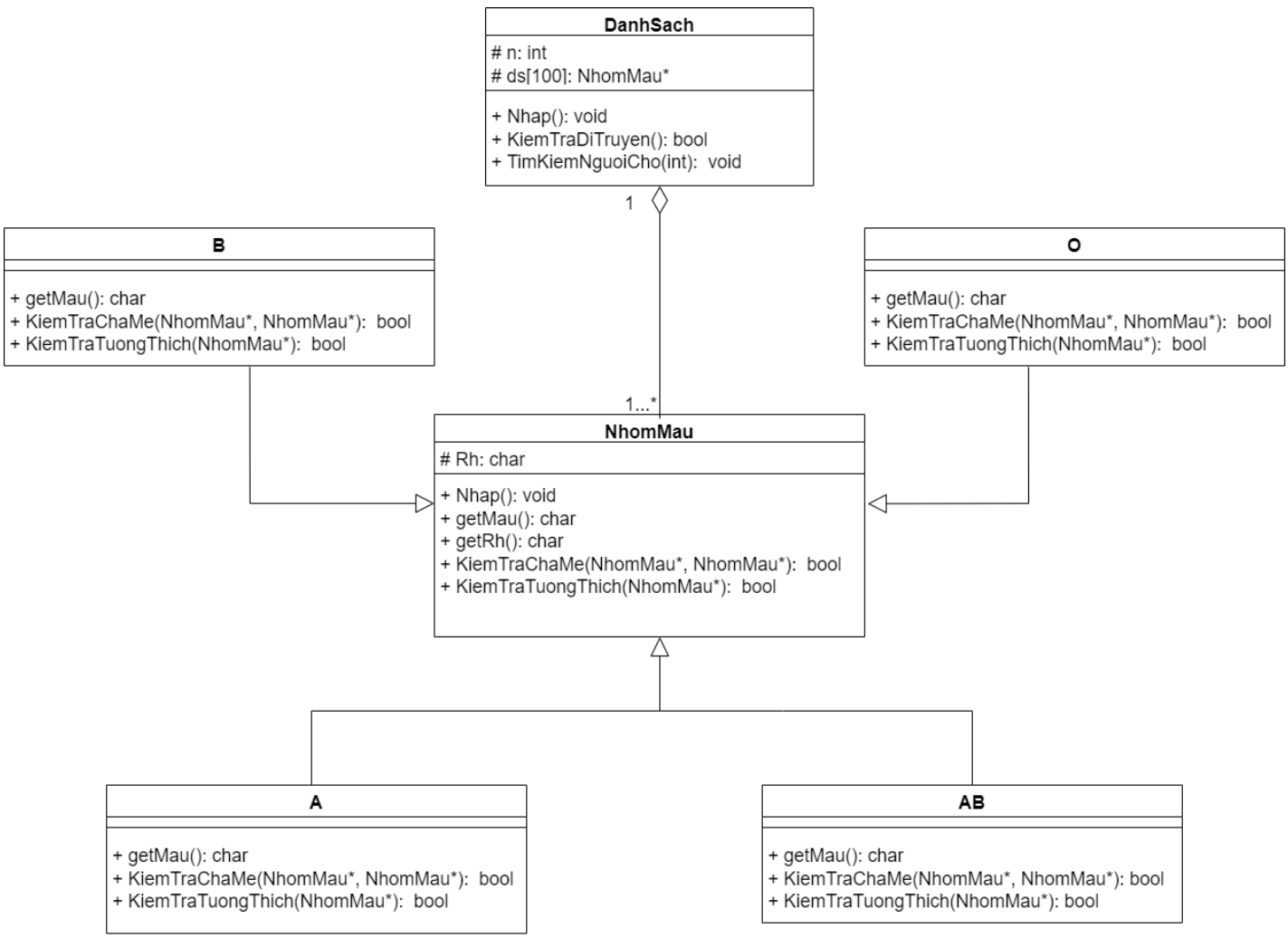
- Phương thức **nhập thông tin**: Cụ thể hơn là nhập Rh, không cần nhập tên nhóm máu A, B, AB hay O vì các thông tin này sẽ được nhập khi gọi thực hiện phương thức nhập thông tin của lớp `DanhSach`. Ở lớp cha, phương thức này **là phương thức không ảo** vì các lớp con không có thuộc tính riêng(không cần ghi đè).
- Các phương thức **kiểm tra**: Do các quy luật ứng với từng nhóm máu là khác nhau nên những phương thức có nhiệm vụ tương tự cần được định nghĩa lại ở các lớp dẫn xuất. Vì vậy, ở lớp `NhomMau`, các phương thức kiểm tra nên là **phương thức thuần ảo.**

- Các phương thức **truy vấn**: Hai thông tin cần được truy vấn để hỗ trợ cho việc định nghĩa các phương thức kiểm tra là **tên nhóm máu** và **Rh**. **Phương thức truy vấn đến Rh là phương thức không ảo** vì lí do tương tự phương thức nhập thông tin. Ngược lại, **phương thức truy vấn đến tên nhóm máu nên là phương thức thuần ảo** để giá trị trả về là loại máu của đối tượng đang gọi thực hiện phương thức.

- Ở các lớp dẫn xuất, việc định nghĩa lại các phương thức truy vấn chỉ đơn giản là trả về tên nhóm máu của lớp (có thể là kiểu kí tự hoặc kiểu dữ liệu khác theo quy ước mỗi người). Để định nghĩa các phương thức kiểm tra, ta cần đọc kĩ thông tin của đề kết hợp với kiến thức bản thân để tìm ra những cách kiểm tra tổng quát nhất theo từng quy luật. Sau đây là một số nhận định rút ra từ đề thi giúp ta định nghĩa những phương thức này nhanh hơn:

- Nếu cha hoặc mẹ có nhóm máu A hoặc AB thì con có thể có nhóm máu A.
- Nếu cha hoặc mẹ có nhóm máu B hoặc AB thì con có thể có nhóm máu B
- Nếu cha và mẹ có nhóm máu AB thì con không thể có nhóm máu O
- Nếu cha và mẹ có nhóm máu giống nhau nhưng khác nhóm máu AB hoặc một trong hai người có nhóm máu O thì con không thể có nhóm máu AB
- Người có nhóm máu A chỉ nhận được nhóm máu A và O(chưa xét Rh)
- Người có nhóm máu B chỉ nhận được nhóm máu B và O(chưa xét Rh)
- Người có nhóm máu O chỉ nhận được nhóm máu O(chưa xét Rh)
- Người có nhóm máu AB có thể nhận được tất cả nhóm máu (chưa xét Rh)
- Người có nhóm máu Rh+ có thể nhận nhóm máu Rh+ hoặc Rh-
- Người có nhóm máu Rh- có thể cho người có nhóm máu Rh+ hoặc Rh-

Sơ đồ thiết kế lớp:



Lời giải tham khảo:

https://github.com/bht-cnpm-uit/OOP.git

Source code tham khảo:

```
#include<iostream>
using namespace std;
class NhomMau {
protected:
   char Rh;
public:
   void Nhap();/*Các lớp con không có thuộc tính riêng nên hàm Nhap()
                 không cần là phương thức ảo*/
   virtual char getMau() = 0;/*Lấy nhóm máu của các lớp con(không xét Rh)
                               để thực hiện các phương thức kiểm tra*/
   char getRh();//Tương tự như phương thức Nhap()
   virtual bool KiemTraChaMe(NhomMau*, NhomMau*) = 0;
   //Kiểm tra xem nhóm  máu cha mẹ có phù hợp với nhóm máu của conn
   virtual bool KiemTraTuongThich(NhomMau*) = 0;
   //Kiểm tra nhóm máu người cho có phù hợp
};
class DanhSach { // Lớp DanhSach dùng để quản lí các nhóm mau
protected:
   int n;                       // Số lượng loại máu
   NhomMau* ds[100];    //Mảng ds dùng để quản lí nhóm máu của từng người
public:
   void Nhap();                 //Nhập danh sách nhóm máu của từng người
   bool KiemTraDiTruyen();      //Kiểm tra nhóm máu của cha, mẹ, con
   void TimKiemNguoiCho(int); //Xuất ra chỉ số của những người cho phù hợp
};
class A : public NhomMau {
public:
   char getMau();
   bool KiemTraChaMe(NhomMau*, NhomMau*);
   bool KiemTraTuongThich(NhomMau*);
};
class B : public NhomMau {
public:
   char getMau();
   bool KiemTraChaMe(NhomMau*, NhomMau*);
   bool KiemTraTuongThich(NhomMau*);
};
class O : public NhomMau {
```

```
public:
   char getMau();
   bool KiemTraChaMe(NhomMau*, NhomMau*);
   bool KiemTraTuongThich(NhomMau*);
};
class AB : public NhomMau {
public:
   char getMau();
   bool KiemTraChaMe(NhomMau*, NhomMau*);
   bool KiemTraTuongThich(NhomMau*);
};
//Định nghĩa các phương thức cho câu 1
void NhomMau::Nhap() {
   cout << "Nhap Rh: ";
   cin >> Rh;
}
void DanhSach::Nhap() {
   cout << "Nhap so luong nguoi: ";
   cin >> n;
   for (int i = 0; i < n; i++) {
      char loai;
      cout << "Nhap loai mau (nhap C thay cho AB) thu [" << i << "]: ";
      cin >> loai;
      switch (loai) {
      case 'A':
         ds[i] = new A;
         break;
      case 'B':
         ds[i] = new B;
         break;
      case 'O':
         ds[i] = new O;
         break;
      case 'C':
         ds[i] = new AB;
      }
      ds[i]->Nhap();
   }
}
//Định nghĩa các phương thức cho câu 2
char A::getMau() {
   return 'A';
}
```

```
char B::getMau() {
   return 'B';
}
char O::getMau() {
   return 'O';
}
char AB::getMau() {
   return 'C';
}
char NhomMau::getRh() {
   return Rh;
}
bool A::KiemTraChaMe(NhomMau* cha, NhomMau* me) {
   char mauCha = cha->getMau();
   char mauMe = me->getMau();
   //Nếu cha hoặc mẹ có nhóm máu A hoặc AB thì con có thể có nhóm máu A
   if (mauCha == 'A' || mauCha == 'C' || mauMe == 'A' || mauMe == 'C') {
      return true;
   }
   return false;
}
bool B::KiemTraChaMe(NhomMau* cha, NhomMau* me) {
   char mauCha = cha->getMau();
   char mauMe = me->getMau();
   //Nếu cha hoặc mẹ có nhóm máu B hoặc AB thì con có thể có nhóm máu B
   if (mauCha == 'B' || mauCha == 'C' || mauMe == 'B' || mauMe == 'C') {
      return true;
   }
   return false;
}
bool O::KiemTraChaMe(NhomMau* cha, NhomMau* me) {
   char mauCha = cha->getMau();
   char mauMe = me->getMau();
   //Nếu cha và mẹ có nhóm máu AB thì con không thể có nhóm máu O
   if (mauCha == 'C' && mauMe == 'C') {
      return false;
   }
   return true;
}
bool AB::KiemTraChaMe(NhomMau* cha, NhomMau* me) {
   char mauCha = cha->getMau();
   char mauMe = me->getMau();
   /*Các trường hợp con không thể có nhóm máu AB:
```

```
      - Cha và mẹ có nhóm máu giống nhau nhưng khác nhóm máu AB
      - Cha hoặc mẹ có nhóm máu O*/

   if ((mauCha == mauMe && mauMe != 'C') || mauCha == 'O' || mauMe == 'O') {
      return false;
   }
   return true;
}
bool DanhSach::KiemTraDiTruyen() {
   int cha, me, con;
   cout << "Nhap lan luot chi so cua cha, me, con: ";
   cin >> cha >> me >> con;
   bool check = ds[con]->KiemTraChaMe(ds[cha], ds[me]);
   if (check == 1) {
      return true;
   }
   return false;
}
//Định nghĩa các phương thức cho câu 3
void DanhSach::TimKiemNguoiCho(int x) {
   for (int i = 0; i < n; i++) {
      if (i != x && ds[x]->KiemTraTuongThich(ds[i]) == 1) {
         cout << i << endl;
      }
   }
}
bool A::KiemTraTuongThich(NhomMau* nguoiCho) {
   //Người có nhóm máu A chỉ nhận được nhóm máu A và O(chưa xét Rh)
   if (nguoiCho->getMau() == 'A' || nguoiCho->getMau() == 'O') {
      //Người có nhóm máu Rh+ có thể nhận nhóm máu Rh+ hoặc Rh-
      //Người có nhóm máu Rh- có thể cho người có nhóm máu Rh+ hoặc Rh-
      if (this->getRh() == '+' || nguoiCho->getRh() == '-') {
         return true;
      }
   }
   return false;
}
bool B::KiemTraTuongThich(NhomMau* nguoiCho){
   //Người có nhóm máu B chỉ nhận được nhóm máu B và O(chưa xét Rh)
   if (nguoiCho->getMau() == 'O' || nguoiCho->getMau() == 'B') {
      if (this->getRh() == '+' || nguoiCho->getRh() == '-') {
         return true;
      }
```

```
    }
    return false;
}

bool O::KiemTraTuongThich(NhomMau* nguoiCho) {
    //Người có nhóm máu O chỉ nhận được nhóm máu O(chưa xét Rh)
    if (nguoiCho->getMau() == 'O') {
        if (this->getRh() == '+' || nguoiCho->getRh() == '-') {
            return true;
        }
    }
    return false;
}
bool AB::KiemTraTuongThich(NhomMau* nguoiCho) {
    //Người có nhóm máu AB có thể nhận được tất cả nhóm máu (chưa xét Rh)
    if (this->getRh() == '+' || nguoiCho->getRh() == '-') {
        return true;
    }
    return false;
}
```

TÀI LIỆU THAM KHẢO

[1] Website: https://www.geeksforgeeks.org/.

[2] Website: https://cplusplus.com/.

[3] Website: https://topdev.vn/.

[4] *The C++ Programming Language Bjarne Stroustrup, Fourth Edition*, B. Stroustrup, Addison-Wesley, 2013.

[5] *C++ Primer (5th Edition)*, J. L. B. E. M. Stanley B. Lippman, Addison-Wesley, 1986.

[6] Các tài liệu giảng dạy môn Lập trình hướng đối tượng của Trường Đại học Công nghệ thông tin - ĐHQG TP.HCM