BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI

PHÂN HIỆU TẠI THÀNH PHỐ HỒ CHÍ MINH

BỘ MÔN CÔNG NGHỆ THÔNG TIN

BÁO CÁO BÀI TẬP LỚN
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

ĐỀ TÀI: CHƯƠNG TRÌNH QUẢN LÝ NHÂN VIÊN

Giảng viên hướng dẫn : **Trần Thị Dung**

Sinh viên thực hiện : **Huỳnh Ngọc Tài (Nhóm trưởng)**

: **Phạm Chí Hùng**

Lớp : **Công nghệ thông tin**

Khoá : **61**

Thành phố Hồ Chí Minh, tháng 11 năm 2021

LỜI MỞ ĐẦU

Quản lý nhân viên luôn là một công việc hàng đầu của các doanh nghiệp. Với lượng công nhân hàng năm gia tăng đáng kể thì việc quản lý những thông tin cá nhân của nhân viên cũng rất quan trọng. Cùng với sự phát triển của công nghệ nói chung và công nghệ thông tin nói riêng thì việc quản nhân viên cũng ngày càng được hiện đại hoá. Thay vì phải ghi sổ sách lưu trữ trên giấy tờ truyền thống thì giờ đây đã có những phần mềm được sử dụng để giúp việc quản lý nhân viên ngày càng dễ dàng hơn. Phần mềm quản lý nhân viên là phần mềm được tạo ra với mục tiêu là hỗ trợ những doanh nghiệp thuận tiện hơn trong việc quản lý nhân viên cụ thể như xem, sửa, thêm hoặc xoá thông tin của nhân viên. Việc đó tạo ra sự thuận tiện cho công tác quản lí. Bây giờ khi cần xem hoặc sửa thông tin, lương của nhân viên người quản lí không cần phải dò sổ sách với hàng tá giấy tờ như trước. Người quản lí chỉ cần đăng nhập vào chương trình, ngay lập tức với những thao tác đơn giản giờ đây các người quản lí đã có thể truy cập vào hồ sơ của nhân viên một cách nhanh chóng. Tuy nhiên các phần mềm quản lý nhân viên ngày càng được cải thiện để phục vụ tốt hơn cho các doanh nghiệp.

Thành phố Hồ Chí Minh, tháng 11 năm 2021

LỜI CẢM ƠN

Lời đầu tiên của báo cáo đề tài này, chúng em muốn gửi những lời cám ơn và biết ơn chân thành nhất của mình tới tất cả quý thầy, cô bộ môn đã luôn hỗ trợ, giúp đỡ và cung cấp cho chúng em những kiến thức bổ ích trong suốt quá trình hoàn thành đề tài bài tập lớn môn lập trình hướng đối tượng với đế tài "Quản Lí Nhân Viên" này.

Do một phần chúng em còn chưa có nhiều kinh nghiệm để xây dựng một chương trình thực tế cũng như một phần kiến thức còn nhiều hạn chế nên đề tài bài tập lớp thực hiện chắc chắn không tránh khỏi những thiếu sót nhất định.

Nhóm chúng em rất mong nhận được ý kiến đóng góp của thầy, cô và các bạn để nhóm chúng em có thêm nhiều kinh nghiệm và tiếp tục hoàn thiện đề tài bài tập lớn của mình một cách tốt hơn.

Nhóm chúng em xin chân thành cảm ơn quý thầy, cô!

Thành phố Hồ Chí Minh, tháng 11 năm 2021

NHẬN XÉT

(Giảng Viên)

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Giảng viên

(Kí ghi rõ họ và tên)

Thành phố Hồ Chí Minh, tháng 11 năm 2021

MỤC LỤC

PHẦN 1: DỊCH TÀI LIỆU

1.1. Chương 16 Class

1.1.1. Cơ bản về class

Một lớp(class) trong C++ sẽ có những đặc điểm sau:
• Là kiểu dữ liệu do người lập trình viên định nghĩa.
• Một lớp sẽ bao gồm các thành viên dữ liệu và các hàm thành viên.
• Các hàm thành viên có thể xác định ý nghĩa của việc khởi tạo, sao chép, di chuyển và huỷ.
• Để các thành viên truy cập vào đối tượng ta sử dụng dấu chấm và mũi tên −> đối với con trỏ.
• Các toán tử, chẳng hạn như + , ! , và [] , có thể được định nghĩa cho một lớp.
• Dữ liệu và hàm bên trong một lớp được gọi là các thành viên của lớp đó.
• Một cấu trúc là một lớp mà các thành viên mặc định là công khai.
Ví dụ:

```
class X {
private: // riêng tư
        int m;
public: // công khai
        X(int i =0) :m{i} { } // một phương thức khởi tạo (khởi tạo thành viên dữ
        liệu m)
        int mf(int i) // hàm thành viên
        int old = m;
        m = i; // set giá trị mới
        return old; // trả về giá trị cũ}
};
        X var {7}; // 1 kiểu biến X được khởi tạo bằng 7
        int user(X var, X∗ ptr){
        int x = var.mf(7); // truy cập thông qua dấu “.”
        int y = ptr−>mf(9); // truy cập thông qua “−>”
}
```

1.1.2. Hàm thành viên

Các hàm được khai báo trong định nghĩa lớp (struct là một loại lớp) được gọi là các hàm thành viên và chỉ có thể được gọi cho một biến cụ thể của kiểu thích hợp bằng cách sử dụng tiêu chuẩn cú pháp để truy cập thành viên cấu trúc.
Ví dụ:

```
struct Date {
int d, m, y;
void init(int dd, int mm, int yy); //khởi tạo
void add_year(int n); // thêm n năm
void add_month(int n);
```

```
void add_day(int n);
};
Date my_birthday;
void f(){
        Date today;
        today.init(16,10,1996);
        my_bir thday.init(30,12,1950);
        Date tomorrow = today;
        tomorrow.add_day(1);
}
```

Bởi vì các cấu trúc khác nhau có thể có các hàm thành viên có cùng tên, chúng ta phải chỉ định tên cấu trúc khi xác định một hàm thành viên:

```
void Date::init(int dd, int mm, int yy){
        d = dd;
        m = mm;
        y = yy;
}
```

Đó là một tính năng quan trọng của lớp là các hàm thành viên. Mỗi kiểu dữ liệu có thể có các hàm tích hợp riêng của nó (được gọi là các phương thức) có quyền truy cập vào tất cả các thành viên (public và private) của kiểu dữ liệu.

1.1.3. Lớp và cấu trúc

Theo mặc định, các thành viên của struct là công khai: struct S {…}; chỉ là cách viết tắc của class S {public: / * ... * /};

Còn mặc định đối với class là các thành viên đều là riêng tư: class X{/*…*/};

```
class Date1 {
        int d, m, y; // mặc định là private
        public:
        Date1(int dd, int mm, int yy);
        void add_year(int n); // thêm n năm
};
```

Tuy nhiên, có thể sử dụng công cụ xác định quyền truy cập riêng tư: để nói rằng các thành viên sau đây là riêng tư, cũng như công khai:

```
class Date3 {
public:
Date3(int dd, int mm, int yy);
void add_year(int n); // add n years
private:
int d, m, y;    };
```

1.1.4. Hàm khởi tạo

Ví dụ:

```
class Date {
                int d, m, y;
        public:
```

```
        Date (int dd, int mm, int yy); // constructor
    };
```

Constructor là một loại hàm thành viên đặc biệt của class, được gọi tự động khi một đối tượng của class đó được khởi tạo. Các constructors thường được sử dụng để khởi tạo các biến thành viên của class theo các giá trị mặc định phù hợp hoặc do người dùng cung cấp, hoặc để thực hiện bất kỳ các bước thiết lập cần thiết nào cho class.

Ví dụ:
```
Date today = Date (23,6,1983);
Date today = Date (23,6,1983);
Date my_bir thday; // Lỗi: chưa được khởi tạo
Date release1_0(10,12); // Lỗi: đối số thứ ba bị thiếu
```

Vì một phương thức khởi tạo xác định quá trình khởi tạo cho một lớp, chúng ta có thể sử dụng ký hiệu {} khởi tạo thay cho ký hiệu ():
```
Date today = Date {23,6,1983};
```

Bằng cách cung cấp một số hàm tạo, chúng ta có thể cung cấp nhiều cách khác nhau để khởi tạo các đối tượng của một kiểu. Ví dụ:
```
class Date {
int d, m, y;
public:
Date (int, int, int);
Date (); // default Date: today
Date (const char*);
};
```

Lưu ý rằng bằng cách đảm bảo việc khởi tạo các đối tượng một cách thích hợp, các hàm tạo sẽ đơn giản hóa rất nhiều việc thực hiện các hàm thành viên. Với các hàm tạo, các hàm thành viên khác không còn phải đối phó với khả năng dữ liệu chưa được khởi tạo.

1.1.5. Hàm tạo rõ ràng

Theo mặc định, một hàm tạo được gọi bởi một đối số duy nhất hoạt động như một chuyển đổi ngầm định từ đối số của nó đến loại của nó. Chúng ta có thể chỉ định rằng một hàm tạo không được sử dụng như một chuyển đổi *ngầm định*. Một constructor được khai báo với từ khóa explicit chỉ có thể được sử dụng để khởi tạo và chuyển đổi rõ ràng.

Ví dụ:

Một khởi tạo với dấu = được coi là một khởi tạo sao chép.

Nếu bỏ dấu = thì được coi là khởi tạo trực tiếp.

Sự phân biệt giữa khởi tạo trực tiếp và sao chép được duy trì cho việc khởi tạo danh sách.

1.1.6. Bộ khởi tạo trong lớp

Khi chúng ta sử dụng một số hàm tạo, việc khởi tạo thành viên có thể trở nên lặp đi lặp lại.

Ví dụ:
```
class Date {
int d, m, y;
public:
Date(int, int, int);
Date(int, int);
```

```
Date(int);
Date();
Date(const char*);
};
```

Chúng ta có thể giải quyết vấn đề đó bằng cách giới thiệu các đối số mặc định để giảm số lượng hàm tạo. Ngoài ra, chúng ta có thể thêm trình khởi tạo vào các thành viên dữ liệu:

```
class Date {
    int d {today.d};    int m {today.m};    int y {today.y};
public:
    Date(int, int, int);
    Date(int, int);
    Date(int);
    Date();
    Date(const char*);
```

1.1.7. Định nghĩa hàm trong lớp

Một hàm thành viên của một lớp là một hàm được định nghĩa bên trong định nghĩa lớp giống như bất kỳ biến nào khác. Nó hoạt động trên bất kỳ đối tượng nào của lớp mà nó là một thành viên, và có sự truy cập tới tất cả thành viên của một lớp cho đối tượng đó.Ví dụ:

```
class Date {
public:
    void add_month(int n) { m+=n; }
    private:
    int d, m, y;
};
```

1.1.8. Chức năng hàm thành viên const

Một đối tượng được tuyên bố là const không thể được sửa đổi và do đó, chỉ có thể gọi các hàm thành viên const vì các chức năng này đảm bảo không sửa đổi đối tượng.
Khi một hàm được tuyên bố là const, nó có thể được gọi trên bất kỳ loại đối tượng, đối tượng const cũng như các đối tượng không phải là const, trong khi một hàm thành viên không phải const chỉ có thể được gọi cho các đối tượng không phải const.
Ví dụ:

```
class Date {
    int d, m, y;
    public:
        int year() const;
        void add_year(int n); // add n years
        };
    void f (Date& d, const Date& cd) {
    int i = d.year(); // OK
    d.add_year(1); // OK
    int j = cd.year(); // OK
    cd.add_year(1); // error : cannot change value of a const Date
```

```
}
```

1.1.9. Quyền truy cập thành viên

Một thành viên của lớp X có thể được truy cập bằng cách sử dụng dấu "." hoặc bởi " ->" đối với con trỏ.Ví dụ:

```
struct X {
void f ();
        int m;  };
void user (X x, X ∗ px){
        x.m = 1;// OK
        px−> m = 1;// OK
}
```

Từ bên trong một lớp không cần toán tử. Ví dụ:

```
void X::f(){
        m = 1; // OK: ‘‘this->m = 1;”
}
```

Một hàm thành viên có thể tham chiếu đến tên của một thành viên trước khi nó được khai báo:

```
struct X {
int f() { return m; }
int m;
};
```

1.1.10. Thành viên tĩnh

Static dùng để khai báo thành viên dữ liệu dùng chung cho mọi thể hiện của lớp.
Khi chúng ta khai báo một thành viên của một lớp là static, nghĩa là, dù cho có bao nhiêu đối tượng của lớp được tạo, thì sẽ chỉ có một bản sao của thành viên static.
Một thành viên static được chia sẻ bởi tất cả đối tượng của lớp. Tất cả dữ liệu static được khởi tạo về 0 khi đối tượng đầu tiên được tạo, nếu không có mặt sự khởi tạo khác.

1.1.11. Các loại thành viên

Một lớp thành viên (thường được gọi là lớp lồng nhau) có thể tham chiếu đến các kiểu và các thành viên tĩnh bao bọc trong một lớp. Nó chỉ có thể tham chiếu đến các thành viên không tĩnh khi nó được cung cấp cho một đối tượng của lớp bao quanh tham khảo.
Một lớp lồng nhau có quyền truy cập vào các thành viên của lớp bao quanh nó, ngay cả với các thành viên riêng (giống như một hàm thành viên có), nhưng không có khái niệm về một đối tượng hiện tại của lớp bao quanh.
Một lớp lồng nhau có quyền truy cập vào các thành viên của lớp bao quanh nó, thậm chí đến các thành viên riêng (giống như một hàm thành viên có), nhưng không có khái niệm về đối tượng hiện tại của lớp bao quanh.

1.1.12. Lớp cụ thể

Một lớp cụ thể là một lớp có một triển khai cho tất cả các phương thức của nó được kế thừa từ trừu tượng hoặc được thực hiện thông qua các giao diện. Nó cũng không định nghĩa bất kỳ phương thức trừu tượng nào của riêng nó. Điều này có nghĩa là một thể hiện

của lớp có thể được tạo/cấp phát với từ khóa new mà không phải thực hiện bất kỳ phương thức nào trước.
Một lớp trừu tượng có nghĩa là được sử dụng như một lớp cơ sở nơi một số hoặc tất cả các hàm được khai báo hoàn toàn là ảo và do đó không thể được khởi tạo. Một lớp cụ thể là một lớp bình thường không có chức năng hoàn toàn ảo và do đó có thể được khởi tạo

1.1.13. Hàm thành viên lớp

Một hàm thành viên của một lớp là một hàm có định nghĩa hoặc nguyên mẫu của nó trong định nghĩa lớp như bất kỳ biến nào khác. Nó hoạt động trên bất kỳ đối tượng nào của lớp mà nó là thành viên và có quyền truy cập vào tất cả các thành viên của một lớp cho đối tượng đó.
Bạn có thể xác định cùng một chức năng bên ngoài lớp bằng cách sử dụng toán tử độ phân giải phạm vi (::)

1.1.14. Hàm hỗ trợ

Thông thường, một lớp có một số hàm được liên kết với nó mà không cần phải được xác định trong chính lớp đó vì chúng không cần quyền truy cập trực tiếp vào lớp đó. Ví dụ:

```
int diff(Date a, Date b); // số ngày trong phạm vi [a,b) hoặc [b,a)
    bool is_leapyear(int y);
    bool is_date(int d, Month m, int y);
    const Date& default_date();
    Date next_weekday(Date d);
    Date next_saturday(Date d);
```

Các chức năng của người trợ giúp (điều mà tôi tin rằng mọi người có ý nghĩa nhất khi họ nói nó) thường là các chức năng bao gồm một số chức năng hữu ích mà bạn sẽ sử dụng lại, rất có thể là lặp đi lặp lại. Bạn có thể tạo các hàm trợ giúp được sử dụng cho nhiều loại mục đích khác nhau.

1.1.15. Nạp chồng toán tử

Nạp chồng toán tử (Operator Overloading) được dùng để định nghĩa toán tử cho có sẵn trong c++ phục vụ cho dữ liệu riêng do bạn tạo ra. Nếu gặp một biểu thức phức tạp, số lượng phép tính nhiều thì việc sử dụng các phương thức trên khá khó khăn và có thể gây rối cho người lập trình. Vì thế ta sẽ nạp chồng lại các toán tử để có thể tạo một cái nhìn trực quan vào code, giảm thiểu các lỗi sai không đáng có. Ví dụ:

```
    bool operator>(Date, Date);
    bool operator!=(Date, Date);
    bool operator<(Date, Date);
```

1.2. Chương 17 Hàm khởi tạo, xoá, sao chép và di chuyển

1.2.1. Hàm khởi tạo

Hàm khởi tạo là một hàm thành viên đặc biệt của một lớp. Nó sẽ được tự động gọi đến khi một đối tượng của lớp đó được khởi tạo.
Ví dụ:

```
    class Vector {
```

```
public:
        Vector(int s);
};
```

Công việc của một phương thức khởi tạo là khởi tạo một đối tượng của lớp nó. Thông thường, việc khởi tạo đó phải thiết lập một lớp bất biến, khi nào một hàm thành viên được gọi (từ bên ngoài lớp).

1.2.2. Hàm huỷ

Hàm huỷ cũng là một hàm thành viên đặc biệt giống như hàm tạo, nó được dùng để phá huỷ hoặc xoá một đối tượng trong lớp. Ví dụ:

```
Class Vector {
public:
        Vector(int s) :elem{new double[s]}, sz{s} { }; // Hàm tạo
~       Vector() { delete[] elem; } // Hàm huỷ: giải phóng bộ nhớ
private:
double∗ elem;
int sz;
};
```

1.2.3. Cơ sở và thành viên hàm huỷ

Bộ tạo và bộ hủy tương tác trực tiếp với cấu trúc phân cấp lớp. Một hàm tạo xây dựng một đối tượng lớp ‘‘từ dưới lên ’’:
[1] đầu tiên, hàm tạo gọi các hàm tạo lớp cơ sở của nó
[2] sau đó, nó gọi các hàm tạo thành viên
[3] cuối cùng, nó tự thực thi cơ thể của mình.
Một hàm hủy " xé toạc " một đối tượng theo thứ tự ngược lại:
[1] đầu tiên, trình hủy thực thi phần thân của chính nó,
[2] sau đó, nó gọi các hàm hủy thành viên của nó
[3] cuối cùng, nó gọi các hàm hủy lớp cơ sở của nó.
Các hàm tạo thực thi các hàm khởi tạo thành viên và cơ sở theo thứ tự khai báo (không phải thứ tự của các trình khởi tạo): nếu hai hàm tạo sử dụng một thứ tự khác nhau, bộ hủy không thể đảm bảo sẽ hủy theo thứ tự ngược lại của cấu trúc (không nghiêm trọng).
Nếu một lớp được sử dụng để cần một hàm tạo mặc định và nếu lớp đó không có các hàm tạo khác, thì trình biên dịch sẽ cố gắng tạo một hàm tạo mặc định. Ví dụ:

```
struct S1 {
        string s;
};
S1 x; // OK: x.s is initialized to ""
```

1.2.4. Hàm huỷ ảo

Một hàm hủy có thể được khai báo là ảo và thường phải dành cho một lớp có hàm ảo.
Ví dụ:

```
class Shape {
```

```
public:
virtual void draw() = 0;
virtual ~Shape();
};
class Circle {
public:
void draw();
~Circle(); // Ghi dè ~Shape()
};
```

Lý do chúng ta cần một trình hủy ảo là một đối tượng thường được thao tác thông qua giao diện được cung cấp bởi một lớp cơ sở thường cũng bị xóa thông qua giao diện đó:

```
void user(Shape* p){
    p->draw();
    delete p;
};
```

1.2.5. Khởi tạo đối tượng của lớp

1.2.5.1. Hàm khởi tạo không tham số (mặc định)

Một phương thức khởi tạo có thể được gọi mà không cần đối số nào được gọi là một phương thức khởi tạo mặc định. Các hàm tạo mặc định rất phổ biến. Ví dụ:

```
class Vector {
public:
Vector(); // Hàm khởi tạo mặc định};
```

Một hàm tạo mặc định được sử dụng nếu không có đối số nào được chỉ định hoặc nếu danh sách bộ khởi tạo trống:

```
Vector v1; // OK        Vector v2 {}; // OK
```

Khi sử dụng một constructor, nếu không có giá trị khởi tạo nào do người dùng cung cấp được truyền cho constructor này, thì constructor mặc định sẽ được gọi.

1.2.5.2. Hàm khởi tạo trực tiếp và sao chép

Hàm khởi tạo sao chép là một hàm tạo mà tạo một đối tượng bằng việc khởi tạo nó với một đối tượng của cùng lớp đó, mà đã được tạo trước đó.

Một hàm khởi tạo sao chép sẽ có nguyên mẫu chung như sau:

```
ClassName(const ClassName &old_obj){ //Code }
```

Trong đó Classname là tên của lớp, old_obj là đối tượng cũ sẽ lấy làm gốc để sao chép sang đối tượng mới.

1.2.6. Khởi tạo thành viên và cơ sở

1.2.6.1. Bộ khởi tạo cơ sở

Các cơ sở của một lớp dẫn xuất được khởi tạo giống như cách các thành viên không phải là dữ liệu. Nếu một cơ sở yêu cầu một bộ khởi tạo, thì nó phải được cung cấp như một bộ

khởi tạo cơ sở trong một phương thức khởi tạo. Nếu chúng ta muốn, chúng ta có thể chỉ định rõ ràng cấu trúc mặc định.
Đối với các thành viên, thứ tự khởi tạo là thứ tự khai báo, và nên chỉ định các trình khởi tạo cơ sở theo thứ tự đó. Cơ sở được khởi tạo trước thành viên và bị phá hủy sau thành viên.

1.2.6.2. Bộ khởi tạo trong lớp

ký hiệu {} và = khởi tạo có thể được sử dụng cho các bộ khởi tạo thành viên trong lớp, nhưng ký hiệu () thì không.
Theo mặc định, một phương thức khởi tạo sẽ sử dụng một trình khởi tạo trong lớp như vậy:

```
class A {
public:
        int a;
        int b;
        A() : a{7}, b{77} {}
};
```

1.2.6.3. Khởi tạo thành viên tĩnh

Một thành viên lớp tĩnh được cấp phát tĩnh chứ không phải là một phần của mỗi đối tượng của lớp. Nói chung, khai báo thành viên tĩnh hoạt động như một khai báo cho một định nghĩa bên ngoài lớp.
Ví dụ:

```
class Node {
        static int node_count; // định nghĩa
};
int Node::node_count = 0; // định nghĩa
```

Tuy nhiên, đối với một số trường hợp đặc biệt đơn giản, có thể khởi tạo một thành viên tĩnh trong khai báo lớp. Phần tử tĩnh phải là một hằng số của kiểu tích phân hoặc kiểu liệt kê, hoặc một mã liên kết của kiểu chữ và bộ khởi tạo phải là một biểu thức hằng.

1.2.7. Sao chép và di chuyển

1.2.7.1. Sao chép

Sao chép một lớp X bằng 2 cách:
+Sao chép hàm tạo: X(const X&)
+Sao chép phép gán: X& operator=(const X&)

1.2.7.2. Sao chép cơ sở

Đối với mục đích sao chép, một cơ sở là một thành viên: để sao chép một đối tượng của một lớp bạn có thể sao chép cơ sở của nó. Ví dụ:

```
struct B1 {
        B1();
        B1(const B1&);      };
```

```
struct B2 {
        B2(int);
        B2(const B2&);      };
struct D : B1, B2 {
        D(int i) :B1{}, B2{i}, m1{}, m2{2∗i} {}
        D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
        B1 m1;
        B2 m2;};
D d {1}; // construct with int argument
D dd {d}; // copy construct
```

Thứ tự khởi tạo là thông thường (cơ sở trước thành viên).

1.2.7.3. Cắt đối tượng

Cắt đối tượng xảy ra khi một đối tượng lớp có nguồn gốc được gán cho một đối tượng lớp cơ sở, các thuộc tính bổ sung của một đối tượng lớp có nguồn gốc được cắt ra để tạo thành đối tượng lớp cơ sở.

1.2.7.4. Hàm tạo di chuyển

Hàm tạo di chuyển lấy giá trị tham chiếu tới một đối tượng của lớp, và được dùng để hiện thực chuyển quyền sở hữu tài nguyên của đối tượng tham số.

1.2.8. Tạo hoạt động mặc định

Việc viết các hoạt động thông thường, chẳng hạn như một bản sao và một trình hủy, có thể dễ xảy ra lỗi, vì vậy trình biên dịch có thể tạo chúng khi cần thiết. Theo mặc định, một lớp cung cấp:

• Một hàm tạo mặc định: X ()
• Hàm tạo bản sao: X (const X &)
• Phép gán bản sao: X & operator = (const X &)
• Một hàm tạo di chuyển: X (X &&)
• Phép chuyển nhượng: X & operator = (X &&)
• Một hàm hủy: ~X ()

1.2.8.1. Mặc định rõ ràng

Vì việc tạo ra các hoạt động mặc định khác có thể bị chặn, nên phải có một cách để lấy lại mặc định. Ngoài ra, một số người muốn xem danh sách đầy đủ các thao tác trong chương trình văn bản ngay cả khi danh sách đầy đủ đó không cần thiết.

Ví dụ:

```
class gslice {
        valarray<siz e_t> siz e;
        valarray<siz e_t> stride;
        valarray<siz e_t> d1;
public:
        gslice() = default;
```

```
        ~gslice() = default;
        gslice(const gslice&) = default;
        gslice(gslice&&) = default;
        gslice& operator=(const gslice&) = default;
        gslice& operator=(gslice&&) = default;   };
```

1.2.8.2. Hoạt động mặc định

Ý nghĩa mặc định của mỗi hoạt động được tạo, như được triển khai khi trình biên dịch tạo ra nó, là áp dụng hoạt động cho từng thành viên dữ liệu cơ sở và không tĩnh của lớp. Đó là, chúng tôi nhận được bản sao của thành viên, bản dựng mặc định của thành viên, v.v.
Ví dụ:

```
struct S {
        string a;
        int b;
        };
S f(S arg){
        S s0 {};
        S s1 {s0};
        s1 = arg;
        return s1;
        }
```

Lưu ý rằng giá trị của một đối tượng được chuyển đến của một kiểu dựng sẵn là không thay đổi. Đó là điều đơn giản và nhanh nhất để trình biên dịch thực hiện. Nếu chúng ta muốn làm điều gì đó khác cho một thành viên của lớp, chúng ta phải viết các hoạt động di chuyển của chúng ta cho lớp đó.

1.2.8.3. Sử dụng thao tác mặc định

Phần này trình bày một số ví dụ chứng minh cách sao chép, di chuyển và hủy được liên kết một cách hợp lý. Nếu chúng không được liên kết, các lỗi hiển nhiên khi bạn nghĩ về chúng sẽ không được trình biên dịch bắt gặp.

1.2.8.3.1. Trình xây dựng mặc định

Trình tạo mặc định là người xây dựng không lấy bất kỳ đối số nào. Nó không có thông số.
Ví dụ:

```
struct X {
        X(int);
};
X a {1};
```

Nếu chúng ta muốn có hàm tạo mặc định, chúng ta có thể định nghĩa một hoặc khai báo rằng chúng ta muốn hàm tạo mặc định do trình biên dịch tạo ra như sau:

```
struct Y {
         string s; int n;
         Y(const string& s);
```

```
        Y() = default; };
    }
```

1.2.8.3.2. Bất biến

[1] Thiết lập một bất biến trong một phương thức khởi tạo (bao gồm cả khả năng thu nhận tài nguyên).
[2] Duy trì tính bất biến với các thao tác sao chép và di chuyển (với các tên và kiểu thông thường).
[3] Thực hiện mọi thao tác dọn dẹp cần thiết trong trình hủy (bao gồm cả giải phóng tài nguyên).

1.2.8.3.3. Tài nguyên bất biến

Nhiều ứng dụng quan trọng và rõ ràng nhất của bất biến liên quan đến quản lý tài nguyên. Ví dụ:

```
template<class T> class Handle {
T∗ p;
public:
Handle(T∗ pp) :p{pp} { }
T& operator∗() { return ∗p; }
˜Handle() { delete p; }
};
```

1.2.8.3.4. Bất biến được chỉ định một phần

Các ví dụ rắc rối dựa trên các bất biến nhưng chỉ thể hiện một phần chúng thông qua các hàm tạo hoặc hàm hủy là hiếm hơn nhưng không phải là chưa từng thấy. Ví dụ:

```
class Tic_tac_toe {
public:
Tic_tac_toe(): pos(9) {} // always 9 positions
Tic_tac_toe& operator=(const Tic_tac_toe& arg){
        for(int i = 0; i<9; ++i)
        pos.at(i) = arg.pos.at(i);
        return ∗this;
}
enum State { empty, nought, cross };
private:
vector<State> pos;
};
```

Điều này đã được báo cáo là một phần của một chương trình thực tế. Nó sử dụng ‘‘magic number ’’ 9 để thực hiện một phép gán sao chép truy cập vào đối số của nó mà không cần kiểm tra xem đối số có thực sự có chín phần tử hay không. Ngoài ra, nó thực hiện việc gán bản sao một cách rõ ràng, nhưng không thực hiện phương thức tạo bản sao.

1.2.8.3.5. Hàm đã xoá

Chúng ta có thể ‘‘xóa’’ một hàm; nghĩa là, chúng ta có thể nói rằng một hàm không tồn tại để cố gắng sử dụng nó (một cách ngầm hiểu hoặc rõ ràng) là một lỗi. Cách sử dụng rõ ràng nhất là loại bỏ chức năng. Ví dụ:

```
class Base {
Base& operator= (const Base&) = delete;// không cho phép sao chép
Base (const Base&) = delete;
Base& operator=(Base&&) = delete; // không cho phép sao chép
Base(Base&&) = delete;
};
Base x1;
Base x2 {x1}; // lỗi: không có hàm tạo bản sao
```

Lưu ý sự khác biệt giữa một hàm đã xóa và một hàm chưa được khai báo. Trong trường hợp trước đây, trình biên dịch lưu ý rằng lập trình viên đã cố gắng sử dụng hàm đã xóa và đưa ra lỗi.

1.3. Chương 18 Nạp chồng

1.3.1. Nạp chồng hàm toán tử

Các hàm định nghĩa cho các toán tử được khai bao như sau:

```
+       −       ∗       /       %       ˆ       &
|       ˜       !       =       <       >       +=
−=      ∗=      /=      %=      ˆ=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      −−      −>∗     ,
−>      []      ()      new     new [] delete delete []
```

Người dùng không thể khai báo các toán tử sau:

:: giải quyết phạm vi lựa

. chọn thành viên

.∗ lựa chọn thành viên thông qua con trỏ đến thành viên

1.3.2. Toán tử 1 và 2 ngôi

Toán tử một ngôi, dù là tiền tố hay hậu tố, đều có thể được xác định bởi một hàm thành viên không tĩnh không lấy đối số hoặc một hàm không phải hàm thành viên và lấy một đối số. Đối với bất kỳ toán tử đơn nguyên tiền tố nào @, @aa có thể được hiểu là aa.operator @ () hoặc operator @ (aa). Đối với bất kỳ toán tử đơn vị hậu tố nào @, aa @ có thể được hiểu là aa.operator @ (int) hoặc operator @ (aa, int). Một toán tử chỉ có thể được khai báo cho cú pháp được định nghĩa cho nó trong ngữ pháp.

Ví dụ: người dùng không thể xác định% bậc 1 hoặc bậc 3 +.

Ví dụ:

```
class X {
public: // members (with implicit this pointer):
X∗ operator&();  // tiền tố 1 ngôi
X operator&(X); // 2 ngôi & (and)
X operator++(int); // tăng hậu tố
};
```

Ý nghĩa mặc định của &&, ||, và, (dấu phẩy) liên quan đến trình tự: toán hạng đầu tiên được đánh giá trước thứ hai (và cho && và || toán hạng thứ hai không phải lúc nào cũng được đánh giá). Quy tắc đặc biệt này không áp dụng cho các phiên bản do người dùng xác định của &&, || và, (dấu phẩy); thay vào đó những toán tử này được xử lýchính xác như các toán tử 2 ngôi khác.

1.3.3. Ý nghĩa được xác định trước cho các toán tử

Ý nghĩa của một số toán tử cài sẵn được định nghĩa tương đương với một số kết hợp của các toán tử khác toán tử trên các đối số giống nhau. Các toán tử = (gán), & (tham chiếu) và, (trình tự; §10.3.2) có giá trị trung bình được xác định trước khi áp dụng cho các đối tượng lớp. Những ý nghĩa được xác định trước này có thể bị loại bỏ (‘‘xóa ’’):

```
class X {
public:
        void operator=(const X&) = delete;
        void operator&() = delete;
        void operator,(const X&) = delete;
};
void f(X a, X b)
{
        a = b; // error : no operator=()
        &a; // error : no operator&()
        a,b; // error : no operator,()
}
```

Ngoài ra, chúng có thể được đưa ra các nghĩa mới bằng các định nghĩa phù hợp.

1.3.4. Toán tử và các loại do người dùng xác định

Một hàm toán tử phải là một thành viên hoặc có ít nhất một đối số của kiểu do người dùng xác định (các hàm xác định lại toán tử new và delete không cần).

Một hàm toán tử nhằm chấp nhận một kiểu dựng sẵn vì toán hạng đầu tiên của nó không thể là một toán hạng.

Ví dụ: hãy xem xét việc thêm một biến phức aa vào số nguyên 2: aa + 2 có thể, với một hàm thành viên được khai báo phù hợp, được hiểu là aa.operator + (2), nhưng 2 + aa không thể vì không có lớp int nào để định nghĩa + có nghĩa là 2.operator + (aa). Bởi vì trình biên dịch không biết ý nghĩa của một + do người dùng xác định, nó không thể giả định rằng toán tử là giao hoán.

Ví dụ:

```
 enum Day { sun, mon, tue, wed, thu, fri, sat };
Day& operator++(Day& d)
{
return d = (sat==d) ? sun : static_cast<Day>(d+1);
}
```

1.3.5. Truyền đối tượng

Đối với các đối số, chúng ta có hai lựa chọn chính

- Truyền theo tham trị
- Truyền theo tham chiếu (&)

Đối với đối tượng nhỏ mang giá trị từ 1-4 thì dùng tham trị sẽ đạt hiệu quả tối ưu. Tuy nhiên, hiệu suất của việc truyền và sử dụng đối số phụ thuộc vào về kiến trúc máy, quy ước giao diện trình biên dịch (Giao diện nhị phân ứng dụng; ABI) và số lần một đối số được truy cập (hầu như luôn luôn nhanh hơn để truy cập một đối số được truyền bởi tham trị hơn một giá trị được truyền bởi tham chiếu).

Thông thường, một toán tử trả về một kết quả, trả về các đối tượng theo giá trị hoặc không trả về giá trị (void).

1.3.6. Toán tử trong Namespaces

```
namespace std {
class string {
};
class ostream {
        ostream& operator<<(const char*); // output C-style string
};
extern ostream cout;
ostream& operator<<(ostream&, const string&); // output std::string
} // namespace std
int main()
{
        const char* p = "Hello";
        std::string s = "world";
        std::cout << p << ", " << s << "!\n";
}
```

Lưu ý rằng tôi không làm cho mọi thứ từ std có thể truy cập được bằng cách viết:

```
using namespace std;
```

Thay vào đó, tôi đã sử dụng tiền tố std :: cho chuỗi và cout.

Toán tử đầu ra cho chuỗi kiểu C là một thành viên của std :: ostream, vì vậy theo định nghĩa:

```
std::cout << p
std::cout.operator<<(p)
```

Tuy nhiên, std :: ostream không có hàm thành viên để xuất ra một chuỗi std ::, vì vậy:

```
std::cout << s
operator<<(std::cout,s)
std::operator<<(std::ostream&, const std::string&)
```

Các hàm khởi tạo:

```
X operator!(X);
struct Z {
Z operator!(); // does not hide ::operator!()
X f(X x) { /* ... */ return !x; } // invoke ::operator!(X)
int f(int x) { /* ... */ return !x; } // invoke the built-in ! for ints
};
```

Đặc biệt, thư viện iostream tiêu chuẩn định nghĩa << hàm thành viên để xuất ra các kiểu tích hợp sẵn, và người dùng có thể định nghĩa << để xuất ra các kiểu do người dùng định nghĩa mà không cần sửa đổi lớp ostream.

1.3.7. Một loại số phức

```
void f()
{
        complex a {1,2};
        complex b {3};
        complex c {a+2.3};
        complex d {2+b};
        b=c*2*c;
}
```

Ngoài ra, chúng tôi mong đợi sẽ được cung cấp thêm một số toán tử, chẳng hạn như == để so sánh và << cho đầu ra, và một tập hợp các hàm toán học phù hợp, chẳng hạn như sin () và sqrt ().

1.3.8. Thành viên và không phải thành viên toán tử

Tôi muốn giảm thiểu số lượng hàm thao tác trực tiếp với biểu diễn của một đối tượng. Điều này có thể đạt được bằng cách chỉ định nghĩa các toán tử vốn đã sửa đổi giá trị của đối số đầu tiên của chúng, chẳng hạn như + =, trong chính lớp đó. Các toán tử chỉ đơn giản tạo ra một giá trị mới dựa trên các giá trị của các đối số của chúng, chẳng hạn như +, sau đó được định nghĩa bên ngoài lớp và sử dụng các toán tử thiết yếu trong việc triển khai chúng.

Ngoại trừ sự khác biệt về hiệu quả có thể xảy ra, các tính toán của r1 và r2 là tương đương. Các toán tử gán tổng hợp như + = và ∗ = có xu hướng dễ xác định hơn so với các toán tử " đơn giản " của chúng + và ∗. Điều này làm cho hầu hết mọi người ngạc nhiên lúc đầu, nhưng nó xuất phát từ thực tế là ba đối tượng tham gia vào một phép toán + (hai toán hạng và kết quả), trong khi chỉ có hai đối tượng tham gia vào một phép toán + =. Trong trường hợp thứ hai, hiệu quả thời gian chạy được cải thiện bằng cách loại bỏ nhu cầu về các biến tạm thời. Ví dụ:

```
inline complex& complex::operator+=(complex a)
{
        re += a.re;
        im += a.im;
        return ∗this;
}
```

1.3.9. Chuyển đổi

```
complex b {3}; // should mean b.re=3, b.im=0
void comp(complex x)
{
        x = 4; // should mean x.re=4, x.im=0
}
class complex {
        double re, im;
public:
        complex(double r) :re{r}, im{0} { } // xây dựng một phức hợp từ một
double
};
```

1.3.9.1. Chuyển đổi toán hạng

Bạn có thể gần đúng với khái niệm rằng một toán tử yêu cầu một giá trị làm toán hạng bên trái của nó bằng biến nhà điều hành đó thành thành viên. Tuy nhiên, đó chỉ là ước tính vì có thể truy cập tạm thời với một thao tác sửa đổi, chẳng hạn như operator + = ():

```
complex x {4,5}
 complex z {sqr t(x)+={1,2}}; // như tmp=sqrt(x), tmp+={1,2}
```

Nếu chúng tôi không muốn chuyển đổi ngầm định, chúng tôi có thể sử dụng chuyển đổi rõ ràng để ngăn chặn chúng.

1.3.10. Chức năng người truy cập

Ví dụ: cho trước real () và image (), chúng ta có thể đơn giản hóa các phép toán đơn giản, phổ biến và hữu ích, chẳng hạn như ==, dưới dạng hàm không phải là thành viên (mà không ảnh hưởng đến hiệu suất):

```
inline bool operator==(complex a, complex b){
        return a.real()==b.real() && a.imag()==b.imag();
}
```

1.3.11. Chuyển đổi kiểu

Việc chuyển đổi kiểu có thể được thực hiện bằng
• Một hàm tạo nhận một đối số duy nhất
• Toán tử chuyển đổi
Trong cả hai trường hợp, chuyển đổi có thể

• rõ ràng; nghĩa là, việc chuyển đổi chỉ được thực hiện trong lần khởi tạo trực tiếp, tức là trình khởi tạo không sử dụng dấu =.
• Ngụ ý; nghĩa là, nó sẽ được áp dụng ở bất cứ nơi nào nó có thể được sử dụng một cách rõ ràng, ví dụ: như một đối số hàm.

1.3.12. Toán tử chuyển đổi

Sử dụng một hàm tạo lấy một đối số duy nhất để chỉ định chuyển đổi kiểu là thuận tiện nhưng có các cation hàm ý có thể không mong muốn. Ngoài ra, một hàm tạo không thể chỉ định
[1] chuyển đổi ngầm định từ loại do người dùng xác định sang loại tích hợp (vì loại không phải là lớp), hoặc
[2] một chuyển đổi từ một lớp mới sang một lớp đã xác định trước đó (mà không sửa đổi khẩu phần decla cho lớp cũ).
Do đó, tốt nhất là nên dựa vào chuyển đổi do người dùng xác định hoặc toán tử do người dùng xác định cho một loại nhất định, nhưng không phải cả hai.

1.4. Chương 19 Toán tử đặc biệt

1.4.1. Các toán tử đặc biệt

Các toán tử [] () -> ++ -- new delete là toán tử đặc biệt trong việc sử dụng của lập trình viên.
Một chút từ việc dùng cho các toán tử đơn ngôi thông thường và nhị phân, như +, <, và ˜.Các toán tử [] (subscript) và () (call) là một trong những toán tử do người dùng định nghĩa hữu ích nhất.

1.4.2. Toán tử []

Một hàm toán tử [] có thể được sử dụng để cung cấp ý nghĩa cho các chỉ số con cho các đối tượng lớp. Đối số thứ hai (chỉ số con) của một hàm toán tử [] có thể thuộc bất kỳ kiểu nào. Điều này giúp bạn có thể xác định vectơ, mảng kết hợp, v.v.
Ví dụ, chúng ta có thể xác định một kiểu mảng kết hợp đơn giản như sau:

```
struct Assoc {
vector<pair<string,int>> vec; // vectơ của các cặp {name, value}
const int& operator[] (const string&) const;
int& operator[](const string&);
};
```

1.4.3. Gọi hàm

Lời gọi hàm, nghĩa là, biểu thức ký hiệu (danh sách biểu thức), có thể được hiểu là một phép toán nhị phân với biểu thức là toán hạng bên trái và danh sách biểu thức là toán hạng bên phải. Gọi toán tử (), có thể được nạp chồng theo cách giống như các toán tử khác có thể.

```
struct Action {
```

```
        int operator()(int);
        pair<int,int> operator()(int,int);
        double operator()(double);
};
void f(Action act){
        int x = act(2);
        auto y = act(3,4);
        double z = act(2.3);
};
```

Danh sách đối số cho operator()() việc chồng hàm khi gọi toán tử dường như hữu ích chủ yếu để xác định các kiểu chỉ có một thao tác duy nhất và cho các kiểu mà một thao tác chiếm ưu thế. Gọi toán tử còn được gọi là ứng dụng toán tử.

Một đối tượng của class Add được khởi tạo bằng một số bất kỳ và khi được gọi bằng cách sử dụng (), nó sẽ thêm số đó vào đối số của nó. Ví dụ:

```
void h(vector<complex>& vec, list<complex>& lst, complex z){
        for_each(vec.begin(),vec.end(),Add{2,3});
        for_each(lst.begin(),lst.end(),Add{z});     }
```

Các cách sử dụng phổ biến khác của operator () () là một toán tử chuỗi con và như một toán tử chỉ số con cho mảng nhiều chiều. Một toán tử () () phải là một hàm thành viên không tĩnh. Các toán tử gọi hàm thường là các mẫu.

1.4.4. Toán tử mũi tên

Được định nghĩa là toán tử hậu tố. Ví dụ:

```
class Ptr {
        X∗ operator−>();
};
```

Các đối tượng của class Ptr có thể được sử dụng để truy cập các thành viên của lớp X theo cách giống với cách con trỏ được sử dụng. Ví dụ:

```
void f(Ptr p)
{
p−>m = 7; // (p.operator->())->m = 7
}
```

Việc biến đổi đối tượng p thành con trỏ p.operator -> () không phụ thuộc vào thành viên m được trỏ tới. Đó là nghĩa mà toán tử -> () là một toán tử hậu tố một ngôi. Tuy nhiên, không có cú pháp mới nào được giới thiệu, vì vậy tên thành viên vẫn được yêu cầu sau dấu ->. Ví dụ:

```
void g(Ptr p)
{
X∗ q1 = p−>; // syntax error
X∗ q2 = p.operator−>(); // OK
}
```

Toán tử -> phải là một hàm thành viên không tĩnh. Nếu được sử dụng, kiểu trả về của nó phải là một con trỏ hoặc một đối tượng của một lớp mà bạn có thể áp dụng ->. Phần thân của hàm thành viên lớp mẫu chỉ được kiểm tra nếu hàm được sử dụng (§26.2.1), vì vậy chúng ta có thể định nghĩa toán tử -> () mà không cần lo lắng về các kiểu, chẳng hạn như Ptr <int>, mà -> không có lý.

1.4.4. Tăng giảm

Khi mọi người phát minh ra " con trỏ thông minh ", họ thường quyết định cung cấp toán tử tăng ++ và toán tử giảm −− để phản ánh việc sử dụng các toán tử này cho các kiểu tích hợp. Điều này đặc biệt rõ ràng và cần thiết khi mục đích là thay thế một loại con trỏ thông thường bằng một loại " con trỏ thông minh " có cùng ngữ nghĩa, ngoại trừ việc nó thêm một chút kiểm tra lỗi thời gian chạy.

Các toán tử tăng và giảm là đặc biệt nhất trong số các toán tử C ++ ở chỗ chúng có thể được sử dụng như cả toán tử tiền tố và hậu tố. Do đó, chúng ta phải xác định gia số tiền tố và hậu tố và giảm cho Ptr <T>.

Ví dụ:

```
template<typename T>
class Ptr {
        T* ptr;
        T* array;
        int sz;
        public:
        template<int N>
        Ptr(T* p, T(&a)[N]); // liên kết với mảng a, sz == N, giá trị ban đầu p
        Ptr(T* p, T* a, int s); // liên kết với mảng a có kích thước s, giá trị ban đầu p
        Ptr(T* p); // liên kết với một đối tượng, sz == 0, giá trị ban đầu p
        Ptr& operator++(); // prefix
        Ptr operator++(int); // postfix
        Ptr& operator−−(); // prefix
        Ptr operator−−(int); // postfix
        T& operator*(); // prefix
};
```

Toán tử tăng trước có thể trả về một tham chiếu đến đối tượng của nó. Toán tử tăng sau phải tạo một đối tượng mới để trả về.

1.4.5. Lớp chuỗi

1.4.5.1. Quyền truy cập vào các kí tự

Biểu diễn cho Chuỗi được chọn để đáp ứng ba mục tiêu:

• Để dễ dàng chuyển đổi chuỗi kiểu C (ví dụ: chuỗi ký tự) thành Chuỗi và cho phép dễ dàng truy cập vào các ký tự của một chuỗi dưới dạng một chuỗi kiểu C

• Để giảm thiểu việc sử dụng cửa hàng miễn phí

• Để thêm các ký tự vào cuối một Chuỗi hiệu quả

1.4.5.2. Hàm thành viên

Hàm tạo mặc định xác định một string trống:

```
String::String() // hàm tạo mặc định: x {""}
: sz{0}, ptr{ch} // ptr trỏ đến các phần tử, ch là vị trí ban đầu (§19.3.3)
{
        ch[0] = 0; // kết thúc 0
}
```

Với copy_from () và move_from (), các hàm tạo, di chuyển và phép gán khá đơn giản để thực hiện. Hàm tạo nhận đối số C kiểu string phải xác định số và lưu trữ chúng một cách thích hợp:

```
String::String(const char∗ p)
:sz{strlen(p)},
ptr{(sz<=short_max) ? ch : new char[sz+1]},
space{0}
{
strcpy(ptr,p); // sao chép các ký tự vào ptr từ p
}
```

Hàm tạo sao chép chỉ cần sao chép biểu diễn của các đối số của nó:

```
String::String(const String& x) // sao chép cấu trúc uctor
{
copy_from(x); // sao chép biểu diễn từ x
}
```

Tương tự, hàm tạo di chuyển di chuyển biểu diễn từ nguồn của nó (và có thể đặt đối số của nó là chuỗi trống):

```
String::String(String&& x) // move constr uctor
{
move_from(x);
}
```

Giống như hàm tạo bản sao, phép gán bản sao sử dụng copy_from () để sao chép biểu diễn của đối số của nó. Ngoài ra, nó phải xóa bất kỳ cửa hàng miễn phí nào thuộc sở hữu của mục tiêu và đảm bảo rằng nó không gặp rắc rối với việc tự chuyển nhượng (ví dụ: s = s):

```
String& String::operator=(const String& x)
{
if (this==&x) return ∗this; // deal with self-assignment
char∗ p = (shor t_max<sz) ? ptr : 0;
copy_from(x);
delete[] p;
return ∗this;
}
```

Nhiệm vụ di chuyển Chuỗi sẽ xóa cửa hàng miễn phí của mục tiêu (nếu có) và sau đó di chuyển:

```
String& String::operator=(String&& x){
if (this==&x) return ∗this; // deal with self-assignment (x = move(x) is insanity)
if (short_max<sz) delete[] ptr; // delete target
move_from(x); // does not throw
return ∗this;
}
```

Phép toán Chuỗi phức tạp nhất về mặt logic là + =, thêm một ký tự vào cuối chuỗi, tăng kích thước của nó lên một.

1.4.5.3. Hàm hỗ trợ

Để hoàn thành Chuỗi lớp, tôi cung cấp một tập hợp các hàm hữu ích, luồng I / O, hỗ trợ các vòng lặp phạm vi cho, so sánh và nối. Tất cả những điều này phản ánh các lựa chọn thiết kế được sử dụng cho std :: string. Cụ thể, << chỉ in các ký tự mà không cần thêm định dạng và >> bỏ qua khoảng trắng ban đầu trước khi đọc cho đến khi tìm thấy khoảng trắng kết thúc (hoặc cuối luồng):

```
ostream& operator<<(ostream& os, const String& s){
        return os << s.c_str(); // §36.3.3
}
istream& operator>>(istream& is, String& s){
        s = ""; // clear the target string
        is>>ws; // skip whitespace (§38.4.5.1)
        char ch = ' ';
        while(is.get(ch) && !isspace(ch))
        s += ch;
        return is;
}
```

Cung cấp == và! = Để so sánh:

```
bool operator==(const String& a, const String& b){
if (a.size()!=b.siz e())
return false;
for (int i = 0; i!=a.size(); ++i)
if (a[i]!=b[i])
return false;
return true;
}
bool operator!=(const String& a, const String& b){
return !(a==b);
}
```

Với hàm thành viên + = có thêm một ký tự ở cuối, các toán tử nối dễ dàng được cung cấp dưới dạng các hàm không phải là bộ nhớ:

```
String& operator+=(String& a, const String& b) // concatenation{
        for (auto x : b)
        a+=x;
        return a;
}
String operator+(const String& a, const String& b)// concatenation{
        String res {a};
        res += b;
        return res;
}
```

1.4.5.4. Chuỗi tự định nghĩa

```
int main(){
String s ("abcdefghij");
cout << s << '\n';
s += 'k';
s += 'l';
s += 'm';
s += 'n';
cout << s << '\n';
String s2 = "Hell";
s2 += " and high water";
cout << s2 << '\n';
String s3 = "qwerty";
s3 = s3;
String s4 ="the quick bro wn fox jumped over the lazy dog";
s4 = s4;
cout << s3 << " " << s4 << "\n";
cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";
String buf;
while (cin>>buf && buf!="quit")
cout << buf << " " << buf.siz e() << " " << buf.capacity() << '\n';
}
```

Chuỗi này thiếu nhiều tính năng mà bạn có thể coi là quan trọng hoặc thậm chí là cần thiết. Tuy nhiên, về những gì nó thực hiện thì nó gần giống với std :: string và minh họa các kỹ thuật được sử dụng để triển khai chuỗi thư viện chuẩn.

1.4.5.5. Hàm bạn

Một khai báo hàm thành viên thông thường chỉ định ba điều khác biệt về mặt logic:
[1] Hàm có thể truy cập phần riêng của khai báo lớp.
[2] Hàm thuộc phạm vi của lớp.
[3] Hàm phải được gọi trên một đối tượng (có con trỏ this).

Một hàm được khai báo friend được cấp quyền truy cập vào việc triển khai một lớp giống như một hàm thành viên nhưng độc lập với lớp đó.

1.4.5.6. Tìm hàm bạn

Một người bạn phải được khai báo trước đó trong một phạm vi bao quanh hoặc được xác định trong phạm vi không phải lớp ngay lập tức bao quanh lớp đang khai báo nó là bạn. Phạm vi bên ngoài vỏ bọc trong cùng phạm vi không gian tên không được coi là tên được khai báo đầu tiên là bạn.

Một hàm friend có thể được tìm thấy thông qua các đối số của nó ngay cả khi nó không được khai báo trong phạm vi kèm theo ngay lập tức. Ví dụ:

```
void f(Matrix& m){
        inver t(m); // Bạn của ma trận invert ()
}
```

Do đó, một hàm friend nên được khai báo rõ ràng trong một phạm vi bao quanh hoặc lấy một đối số của lớp của nó hoặc một lớp có nguồn gốc từ đó. Nếu không, bạn không thể được gọi. Ví dụ:

```
// không có f () trong phạm vi này
class X {
        friend void f(); // useless
        friend void h(const X&); // có thể được tìm thấy thông qua đối số của nó
};
void g(const X& x){
        f(); // không có f () trong phạm vi
        h(x); // Bạn của X là h ()
}
```

1.4.5.7. Lời khuyên

[1] Sử dụng toán tử [] () để lập chỉ số và để lựa chọn dựa trên một giá trị duy nhất.

[2] Sử dụng toán tử () () cho ngữ nghĩa cuộc gọi, để lập chỉ mục và để lựa chọn dựa trên nhiều giá trị.

[3] Sử dụng toán tử -> () để truy cập ‘‘con trỏ thông minh’’.

[4] Ưu tiên tiền tố ++ hơn hậu tố ++;

[5] Xác định toán tử thành viên new () và toán tử thành viên delete () để kiểm soát việc cấp phát và hủy bỏ phân bổ các đối tượng của một lớp cụ thể hoặc hệ thống phân cấp của các lớp;

[6] Sử dụng các ký tự do người dùng xác định để bắt chước ký hiệu thông thường; [8] Đặt các toán tử chữ trong các không gian tên riêng biệt để cho phép sử dụng có chọn lọc;

[7] Đối với các mục đích sử dụng không chuyên biệt, hãy ưu tiên chuỗi tiêu chuẩn hơn là kết quả;

[8] Ưu tiên các chức năng thành viên hơn các chức năng bạn bè để cấp quyền truy cập vào việc triển khai lớp;

1.5. Chương 20 Lớp dẫn xuất

1.5.1. Lớp dẫn xuất

Một lớp được xây dựng thừa kế một lớp khác được gọi là lớp dẫn xuất; lớp dùng để xây dựng lớp dẫn xuất được gọi là lớp cơ sở.

Một lớp dẫn xuất ngoài các thành phần riêng của lớp đó, còn được thừa kế các thành phần của các lớp cơ sở có liên quan.Ví dụ:

Manager có nguồn gốc từ Employee và ngược lại, Employee là một lớp cơ sở của Manager.Ngoài các thành viên của chính nó (nhóm, trình độ, v.v.), lớp Manager còn có các thành viên của lớp Employee (tên, phòng ban, v.v.).

Một lớp dẫn xuất thường được cho là kế thừa các thuộc tính từ cơ sở của nó, vì vậy mối quan hệ còn được gọi là kế thừa. Một lớp cơ sở đôi khi được gọi là lớp cha và lớp dẫn xuất là lớp con. Một lớp dẫn xuất thường lớn hơn (và không bao giờ nhỏ hơn) so với lớp cơ sở của nó theo nghĩa là nó chứa nhiều dữ liệu hơn và cung cấp nhiều chức năng hơn.

1.5.1.1. Hàm thành viên

Một thành viên của lớp dẫn xuất có thể sử dụng các thành viên công khai và được bảo vệ (xem) của một lớp cơ sở như thể chúng được khai báo trong chính lớp dẫn xuất.Ví dụ:

```
void Manager :: print () const{
        cout << "name is" << full_name () << '\ n';
}
```

Tuy nhiên, một lớp dẫn xuất không thể truy cập vào các thành viên riêng tư của một lớp cơ sở:

```
void Manager :: print () const{
        cout << "name is" << family_name << '\ n'; // lỗi!
}
```

Thông thường, giải pháp rõ ràng nhất là cho lớp dẫn xuất chỉ sử dụng các thành viên công khai của lớp cơ sở của nó. Ví dụ:

```
void Manager::print() const{
        Employee::print();
        cout << level;
}
```

1.5.1.2. Hàm tạo và hàm huỷ

Như thường lệ, hàm tạo và hàm hủy là thiết yếu:

• Các đối tượng được tạo từ dưới lên (cơ sở trước thành viên và thành viên trước dẫn xuất) và hủy từ trên xuống (dẫn xuất trước thành viên và thành viên trước cơ sở);

• Mỗi lớp có thể khởi tạo các thành viên và cơ sở của nó (nhưng không trực tiếp là thành viên hoặc cơ sở của các cơ sở của nó)

• Thông thường, các hàm hủy trong một hệ thống phân cấp cần phải là ảo

• Các hàm tạo sao chép của các lớp trong một hệ thống phân cấp nên được sử dụng cẩn thận (nếu có) để tránh bị cắt;

• Độ phân giải của một lệnh gọi hàm ảo, một ép kiểu động, hoặc một kiểu trong một phương thức khởi tạo hoặc huỷ phản ánh giai đoạn khởi tạo và hủy (chứ không phải là kiểu của đối tượng chưa được hoàn thành);

1.5.2. Phân cấp lớp

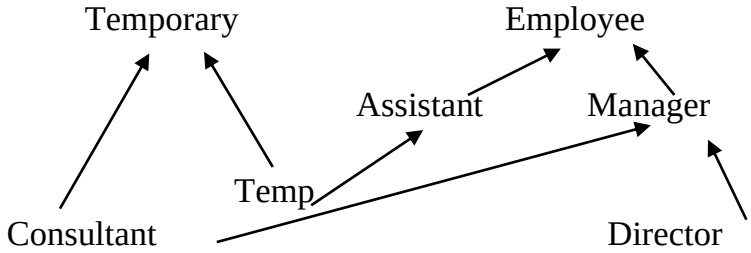
Bản thân một lớp dẫn xuất có thể là một lớp cơ sở. Ví dụ:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

Một tập hợp các lớp liên quan như vậy theo truyền thống được gọi là hệ thống phân cấp lớp. Hệ thống phân cấp như vậy thường là một dạng cây, nhưng nó cũng có thể là một cấu trúc đồ thị tổng quát hơn. Ví dụ:

```
class Temporary { /* ... */ };
class Assistant : public Employee { /* ... */ };
class Temp : public Temporary, public Assistant { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

hoặc bằng đồ thị:



1.5.2.1. Hàm ảo

Các hàm ảo khắc phục các vấn đề với giải pháp trường kiểu bằng cách cho phép lập trình khai báo các hàm trong một lớp cơ sở có thể được định nghĩa lại trong mỗi lớp dẫn xuất. Trình biên dịch và trình liên kết sẽ đảm bảo sự tương ứng chính xác giữa các đối tượng và các chức năng được áp dụng cho chúng.Ví dụ:

```
class Employee {
public:
        Employee(const string& name, int dept);
        virtual void print() const;
private:
        string first_name , family_name;
        short depar tment;
};
```

Một hàm thành viên ảo đôi khi được gọi là một phương thức.

Một hàm ảo phải được định nghĩa cho lớp mà nó được khai báo lần đầu (trừ khi nó được khai báo là một hàm ảo thuần túy.

Một kiểu có các chức năng ảo được gọi là kiểu đa hình hay. Để có được hành vi đa hình trong C ++, các hàm thành viên được gọi phải là ảo và các đối tượng phải được thao tác

thông qua con trỏ hoặc tham chiếu. Khi thao tác trực tiếp một đối tượng (thay vì thông qua con trỏ hoặc tham chiếu), kiểu chính xác của nó được trình biên dịch biết nên không cần đến tính đa hình thời gian chạy.

1.5.2.2. Khởi tạo kế thừa

Giả sử tôi muốn một vectơ giống như std::vector, tôi có thể thử điều này:

```
template<class T>
struct Vector : std::vector<T> {
        using vector<T>::vector; // kế thừa các hàm  tạo
        T& operator=[](size_type i) { check(i); return this−>elem(i); }
        const T& operator=(size_type i) const { check(i); return this−>elem(i); }
        void check(siz e_type i) { if (this−>size()<i) throw Bad_index(i); }
};
Vector<int> v { 1, 2, 3, 5, 8 }; // OK
```

Thông thường, tốt nhất là nên tránh khôn khéo và hạn chế việc sử dụng các hàm tạo kế thừa trong các trường hợp đơn giản mà không có thành viên dữ liệu nào được thêm vào.

1.5.3. Các lớp trừu tượng

Nhiều lớp giống với lớp Employee ở chỗ chúng hữu ích như chính chúng và làm khuôn mẫu cho các lớp dẫn xuất và là một phần của việc triển khai các lớp dẫn xuất.Tuy nhiên, không phải tất cả các lớp đều tuân theo khuôn mẫu đó. Một số lớp, chẳng hạn như một lớp Shape, đại diện cho các khái niệm trừu tượng mà đối tượng không thể tồn tại. Hình dạng chỉ có ý nghĩa như là cơ sở của một số lớp bắt nguồn từ nó. Điều này có thể được thấy từ thực tế là không thể cung cấp các định nghĩa hợp lý cho các chức năng ảo của nó:

```
class Shape {
public:
        virtual void rotate(int) { throw runtime_error{"Shape::rotate"}; }
        virtual void draw() const { throw runtime_error{"Shape::draw"}; }
};
```

Một giải pháp thay thế tốt hơn là khai báo các hàm ảo của lớp Shape là các hàm ảo thuần túy.

Một hàm ảo được ‘‘tạo thuần túy’’ bởi ‘‘bộ khởi tạo giả’’ = 0:

```
class Shape { // abstract class
public:
        virtual void rotate(int) = 0; // pure virtual function
        virtual void draw() const = 0; // pure virtual function
```

```
    virtual ~Shape(); // virtual
};
```

Một lớp có một hoặc nhiều hàm ảo thuần túy là một lớp trừu tượng và không có đối tượng nào của lớp trừu tượng đó có thể được tạo.

Một hàm thuần ảo không được định nghĩa trong lớp dẫn xuất vẫn là một hàm thuần ảo, vì vậy lớp dẫn xuất cũng là một lớp trừu tượng. Điều này cho phép chúng tôi xây dựng các triển khai theo từng giai đoạn:

```
class Polygon : public Shape { // abstract class
public:
bool is_closed() const override { return true; }
};
```

1.5.4. Kiểm soát truy cập

Một thành viên của một lớp có thể là riêng tư, được bảo vệ hoặc công khai:

• Nếu nó là private, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo.

• Nếu nó được bảo vệ, tên của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của lớp mà nó được khai báo và bởi các hàm thành viên và bạn bè của các lớp dẫn xuất từ lớp này.

• Nếu nó là công khai, tên của nó có thể được sử dụng bởi bất kỳ hàm nào.

Điều này phản ánh quan điểm rằng có ba loại hàm truy cập vào một lớp: các hàm thực thi lớp (bạn bè và thành viên của nó), các hàm triển khai một lớp dẫn xuất (bạn bè và thành viên của lớp dẫn xuất) và các hàm khác.

1.5.4.1. Sử dụng thành viên được bảo vệ

Mô hình ẩn dữ liệu riêng tư / công khai đơn giản phục vụ tốt khái niệm về các kiểu cụ thể. Tuy nhiên, khi các lớp dẫn xuất được sử dụng, có hai loại người dùng của một lớp: các lớp dẫn xuất và"công khai" Các thành viên và bạn bè thực hiện các thao tác trên lớp sẽ hoạt động trên các đối tượng của lớp. Mô hình private / public cho phép lập trình viên phân biệt rõ ràng giữa thực hiện và công khai, nhưng nó không cung cấp cấp cụ thể cho các lớp dẫn xuất.

Các thành viên protected dễ bị lạm dụng hơn nhiều so với các thành viên private.

Nếu bạn không muốn sử dụng dữ liệu được bảo vệ; private là mặc định trong các lớp và thường là lựa chọn tốt hơn.

1.5.4.2. Quyền truy cập vào các lớp cơ sở

Một lớp cơ sở có thể được khai báo là riêng tư, được bảo vệ hoặc công khai. Ví dụ:

```
    class X : public B { /* ... */ };
    class Y : protected B { /* ... */ };
    class Z : private B { /* ... */
```

Có thể bỏ qua thông số truy cập cho một lớp cơ sở. Trong trường hợp đó, cơ sở mặc định là cơ sở riêng cho một lớp và cơ sở công khai cho một cấu trúc. Ví dụ:

```
class XX : B { /* ... */ }; // B is a private base
struct YY : B { /* ... */ }; // B is a public base
```

Bộ định nghĩa truy cập cho một lớp cơ sở kiểm soát quyền truy cập vào các thành viên của lớp cơ sở và việc chuyển đổi con trỏ và tham chiếu từ kiểu lớp dẫn xuất sang kiểu lớp cơ sở. Ví dụ lớp dẫn xuất D bắt nguồn từ một lớp cơ sở B:

• Nếu B là cơ sở, các thành viên public và protected của nó chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của D. Chỉ bạn bè và thành viên của D mới có thể chuyển đổi D thành B.

• Nếu B là một cơ sở protected, các thành viên public và protected chỉ có thể được sử dụng bởi các hàm thành viên và bạn bè của D và bởi các hàm thành viên và bạn bè của các lớp bắt nguồn từ D. Chỉ bạn bè và thành viên của D và bạn bè và thành viên của các lớp dẫn xuất từ D có thể chuyển D∗ thành B∗.

• Nếu B là một cơ sở public, các thành viên public có thể được sử dụng cho bất kỳ hàm nào. Ngoài ra, các thành viên protected có thể được sử dụng bởi các thành viên và bạn bè của D và các thành viên và bạn bè của các lớp có nguồn gốc từ D. Bất kỳ hàm nào cũng có thể chuyển đổi từ D ∗ sang B ∗.

1.5.5. Con trỏ đến thành viên

Con trỏ đến thành viên là một cấu trúc giống như bù đắp cho phép tham chiếu gián tiếp đến thành viên của một lớp. Các toán tử -> ∗ và .∗ được cho là các toán tử C ++ chuyên dụng nhất và ít được sử dụng nhất các toán tử. Sử dụng ->, chúng ta có thể truy cập một thành viên của lớp m, bằng cách: p−> m. Sử dụng -> ∗, chúng ta có thể truy cập một thành viên có tên được lưu trữ trong một con trỏ tới thành viên, ptom: p -> ∗ ptom. Điều này cho phép truy cập các thành viên với tên của họ được chuyển làm đối số.

Một con trỏ tới thành viên không thể được gán cho void ∗ hoặc bất kỳ con trỏ thông thường nào khác. Một con trỏ null (ví dụ: nullptr) có thể được gán cho một con trỏ tới thành viên và sau đó đại diện cho "" no member ".

1.5.6. Con trỏ đến các thành viên hàm

Chúng ta có thể sử dụng một con trỏ tới thành viên để gián tiếp tham chiếu đến thành viên của một lớp và con trỏ hoặc tham chiếu đến đối tượng mà tôi muốn tạm dừng.

Con trỏ tới một thành viên ảo là một loại bù đắp, nó không phụ thuộc vào vị trí của đối tượng trong bộ nhớ. Do đó, một con trỏ đến một thành viên ảo có thể được chuyển giữa các không gian địa chỉ khác nhau miễn là sử dụng cùng một bố cục đối tượng trong cả hai. Giống như con trỏ tới các hàm thông thường, con trỏ tới các hàm thành viên không phải ảo không thể được trao đổi giữa các không gian địa chỉ. Lưu ý rằng hàm được gọi thông qua con trỏ tới hàm có thể là ảo.

Thành viên tĩnh không được liên kết với một đối tượng cụ thể, vì vậy con trỏ đến thành viên tĩnh chỉ đơn giản là một con trỏ thông thường.

1.5.7. Con trỏ đến các thành viên dữ liệu

Đương nhiên, khái niệm con trỏ tới thành viên áp dụng cho các thành viên dữ liệu và cho các hàm thành viên với các đối số và kiểu trả về. Ví dụ:

```
struct C {
        const char ∗ val;
        int i;
        void print (int x) {cout << val << x << '\ n'; }
        int f1 (int);
        void f2 ();
        C (const char ∗ v) {val = v; }
};
using Pmfi = void (C :: ∗) (int); // con trỏ tới hàm thành viên của C lấy int
using Pm = const char ∗ C :: ∗; // con trỏ tới thành viên dữ liệu char * của C
```

1.5.8. Thành viên cơ sở và dẫn xuất

Một lớp dẫn xuất có ít nhất các thành viên mà nó kế thừa từ các lớp cơ sở của nó. Chứng ta có thể gán một cách an toàn một con trỏ cho một thành viên của lớp cơ sở cho một con trỏ tới một thành viên của lớp dẫn xuất, nhưng không phải ngược lại. Tính chất này thường được gọi là độ tương phản. Ví dụ:

```
class Text : public Std_interface {
public:
        void start();
        void suspend();
        virtual void print();
        private:
        vector s;
};
void (Text::∗pmt)() = &Std_interface::start; // OK
```

Quy tắc tương phản này dường như ngược lại với quy tắc nói rằng có thể gán một con trỏ cho lớp dẫn xuất, một con trỏ đến lớp cơ sở của nó. Trên thực tế, cả hai quy tắc đều tồn tại để duy trì sự đảm bảo rằng một con trỏ có thể không bao giờ trỏ đến một đối tượng ít nhất không có các thuộc tính mà con trỏ hứa hẹn. Trong trường hợp này, Std_interface :: ∗ có thể được áp dụng cho bất kỳ Std_interface nào, và hầu hết các đối tượng như vậy có lẽ không thuộc loại Text. Do đó, chúng không có thành viên Text :: print mà chúng tôi đã cố gắng khởi tạo pmi. Bằng cách từ chối khởi tạo, trình biên dịch giúp chúng ta tránh khỏi lỗi thời gian chạy.

1.6. Chương 21 Phân cấp lớp

1.6.1. Giới thiệu

Trọng tâm chính của chương này là các kỹ thuật thiết kế, hơn là các tính năng ngôn ngữ. Các ví dụ được lấy từ thiết kế giao diện người dùng, nhưng tôi tránh chủ đề về lập trình theo hướng sự kiện thường được sử dụng cho các hệ thống giao diện người dùng đồ họa (GUI).

1.6.2. Kế thừa triển khai

Một cấu trúc phân cấp lớp sử dụng kế thừa triển khai (như thường thấy trong các chương trình cũ hơn). Lớp Ival_box xác định giao diện cơ bản cho tất cả các Ival_box và chỉ định triển khai mặc định mà các loại Ival_box cụ thể hơn có thể ghi đè bằng các phiên bản của riêng chúng. Ngoài ra, chúng tôi khai báo dữ liệu cần thiết để triển khai khái niệm cơ bản:

```
class Ival_box {
protected:
        int val;
        int low, high;
        bool chang ed {false}; // được thay đổi bởi người dùng bằng set_value ()
public:
        Ival_box(int ll, int hh) :val{ll}, low{ll}, high{hh} { }
        virtual int get_value() { chang ed = false; return val; } // cho ứng dụng
        virtual void set_value(int i) { chang ed = true; val = i; } //cho người dùng
        virtual void reset_value(int i) { chang ed = false; val = i; } // cho ứng dụng
};
```

1.6.3. Critique

Kết luận hợp lý của dòng suy nghĩ này là một hệ thống được đại diện cho người dùng như một hệ thống phân cấp của các lớp trừu tượng và được thực hiện bởi một hệ thống phân cấp cổ điển. Nói cách khác:

• Sử dụng các lớp trừu tượng để hỗ trợ kế thừa giao diện (§3.2.3, §20.1)
• Sử dụng các lớp cơ sở với việc triển khai các hàm ảo để hỗ trợ kế thừa thực thi.

1.6.4. Đa kế thừa

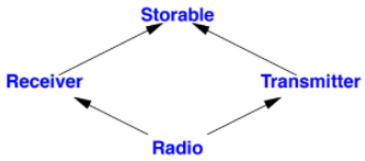
Đa kế thừa là một tính năng của ngôn ngữ C++. Trong đó một lớp có thể kế thừa từ nhiều hơn một lớp khác. Nghĩa là một lớp con được kế thừa từ nhiều hơn một lớp cơ sở.
Lưu ý: Khi đa kế thừa cần tránh trường hợp có nhiều lớp cơ sở có tên phương thức giống nhau. Vì khi gọi từ lớp con thì chương trình không biết nên gọi phương thức đó từ lớp cơ sở nào.

1.6.4.1. Lớp cơ sở ảo

Kế thừa ảo là một kỹ thuật C ++ đảm bảo rằng chỉ một bản sao của các biến thành viên của lớp cơ sở được kế thừa bởi các dẫn xuất cấp hai.

```
class Transmitter : public virtual Storable {
public:
        void write() override;
};
class Receiver : public virtual Storable {
public:
        void write() override;
};
class Radio : public Transmitter, public Receiver {
public:
        void write() override;
};
```

Bằng đồ thị:



1.6.4.2. Xây dựng cơ sở ảo

Một phương thức khởi tạo của một cơ sở ảo được gọi chính xác một lần. Hơn nữa, hàm tạo của một cơ sở (dù là ảo hay không) được gọi trước các lớp dẫn xuất của nó. Bất cứ điều gì khác sẽ gây ra hỗn loạn (nghĩa là, một đối tượng có thể được sử dụng trước khi nó được khởi tạo). Để tránh sự hỗn loạn như vậy, phương thức khởi tạo của mọi cơ sở ảo được gọi (ngầm định hoặc rõ ràng) từ phương thức khởi tạo cho đối tượng hoàn chỉnh (phương thức khởi tạo cho lớp dẫn xuất nhất). Đặc biệt, điều này đảm bảo rằng một cơ sở ảo được xây dựng chính xác một lần ngay cả khi nó được đề cập ở nhiều nơi trong hệ thống phân cấp lớp. Ví dụ:

```
struct V {
        V(int i);
};
struct A {
A(); //hàm khởi tạo mặc định A
};
struct B : virtual V, vir tual A {
        B() :V{1} { /* ... */ }; //hàm khởi tạo mặc định B; phải khởi tạo cơ sở V      };
class C : virtual V {
public:
        C(int i) : V{i} { /* ... */ }; phải khởi tạo cơ sở V
};
class D : virtual public B, virtual public C {
```

```
// ngầm định lấy cơ sở ảo V từ B và C
// ngầm nhận cơ sở ảo A từ B
public:
        D() { /* ... */ } // error: không có hàm tạo mặc định cho C hoặc V
        D(int i) :C{i} { /* ... */ }; // error: không có hàm tạo mặc định cho V
        D(int i, int j) :V{i}, C{j} { /* ... */ } // OK          };
```

1.6.5. Cơ sở sao chép so với cơ sở ảo

Việc sử dụng đa kế thừa để cung cấp các triển khai cho các lớp trừu tượng đại diện cho các giao diện thuần túy ảnh hưởng đến cách một chương trình được thiết kế.
Lớp BB_ival_slider là một ví dụ:

```
class BB_ival_slider
        : public Ival_slider, // interface
        protected BBslider // implementation
{
        // triển khai các chức năng theo yêu cầu của Ival_slider và BBslider, sử dụng các
        tiện ích từ BBslider
};
```

1.6.6. Ghi đè các hàm cơ sở

Một lớp dẫn xuất có thể ghi đè một hàm ảo của lớp cơ sở ảo trực tiếp hoặc gián tiếp của nó. Đặc biệt, hai lớp khác nhau có thể ghi đè các chức năng ảo khác nhau từ cơ sở ảo. Theo cách đó, một số lớp dẫn xuất có thể đóng góp các triển khai cho giao diện được trình bày bởi một lớp cơ sở ảo. Nếu hai lớp ghi đè một hàm của lớp cơ sở, nhưng không ghi đè lên lớp kia, thì cấu trúc phân cấp lớp là một lỗi. Lý do là không một hàm đơn lẻ nào có thể được sử dụng để cung cấp một ý nghĩa nhất quán cho tất cả các lệnh gọi một cách độc lập với lớp mà chúng sử dụng làm giao diện.

1.6.7. Lời khuyên

[1] Sử dụng unique_ptr hoặc shared_ptr để tránh quên xóa các đối tượng được tạo bằng new;
[2] Tránh các thành viên ngày tháng trong các lớp cơ sở dùng làm giao diện;
[3] Sử dụng các lớp trừu tượng để thể hiện các giao diện;
[4] Cung cấp cho một lớp trừu tượng một hàm hủy ảo để đảm bảo việc dọn dẹp thích hợp;
[5] Sử dụng ghi đè để làm cho việc ghi đè trở nên rõ ràng trong các cấu trúc phân cấp lớp lớn;
[6] Sử dụng các lớp trừu tượng để hỗ trợ kế thừa giao diện;
[7] Sử dụng các lớp cơ sở với các thành viên dữ liệu để hỗ trợ kế thừa triển khai;
[8] Sử dụng đa kế thừa thông thường để thể hiện sự kết hợp của các tính năng;
[9] Sử dụng đa kế thừa để tách việc triển khai khỏi giao diện;
[10] Sử dụng cơ sở ảo để biểu diễn một cái gì đó chung cho một số, nhưng không phải tất cả, các lớp trong phân cấp

PHẦN 2: CHƯƠNG TRÌNH ỨNG DỤNG

2.1. Giới thiệu chương trình quản lí nhân viên

2.1.1. Lý do chọn đề tài

Quản lý nhân viên luôn là một công việc hàng đầu của các doanh nghiệp. Với lượng công nhân hàng năm gia tăng đáng kể thì việc quản lý những hồ sơ thông tin của nhân viên cũng rất quan trọng. Cùng với sự phát triển của công nghệ nói chung và công nghệ thông tin nói riêng thì việc quản lý nhân viên cũng ngày càng được cải thiện và hiện đại hơn. Thay vì phải ghi sổ sách lưu trữ trên giấy tờ truyền thống thì giờ đây đã có những phần mềm được sử dụng để giúp việc quản lý nhân viên ngày càng dễ dàng hơn. Phần mềm quản lý nhân viên là phần mềm được tạo ra với mục tiêu là hỗ trợ những doanh nghiệp thuận tiện hơn trong việc quản lý nhân viên cụ thể như xem, sửa, thêm hoặc xoá thông tin của nhân viên. Việc đó tạo ra sự thuận tiện cho công tác quản lí. Bây giờ khi cần xem hoặc sửa thông tin, lương của nhân viên người quản lí không cần phải dò sổ sách với hàng tá giấy tờ như trước. Người quản lí chỉ cần đăng nhập vào chương trình, ngay lập tức với những thao tác đơn giản giờ đây các người quản lí đã có thể truy cập vào hồ sơ của nhân viên một cách nhanh chóng. Cũng vì vậy, nhóm chúng em quyết định thực hiện xây dựng chương trình "quản lí nhân viên".

2.1.2. Mô tả bài toán

Đầu tiên để xây dựng được chương trình quản lí nhân viên thì trước hết chúng ta cần phải xác định được những thuộc tính cơ bản của một nhân viên cần khai báo trong class gồm những gì? Ví dụ: Họ và tên, mã số nhân viên, tuổi… Cùng với đó là những phương thức như nhập, xuất, tính lương … cho nhân viên. Sau đó sử dụng các cấu trúc cơ bản đã học như for, if else, while… kết hợp với các cấu trúc dữ liệu giải thuật và những hàm có sẵn trong các thư thiện để xây dựng các chức năng và tạo menu cho chương trình để dễ dàng lựa chọn và sử dụng các chức năng mà bạn muốn.

Sau khi chạy chương trình sẽ có những chức năng như: nhập, xuất, sắp xếp, tìm kiếm, tính lương, xoá nhân viên cho người quản lí tự lựa chọn chức năng mà họ muốn thực hiện.

Chức năng nhập – xuất: dùng để nhập – xuất thông tin của nhân viên thâm niên và thời vụ.

Chức năng sắp xếp: dùng để sắp xếp thông tin nhân viên theo chiều tăng dần với những tiêu chí: mã số, lương, tuổi.

Chức năng tìm kiếm: dùng để tìm kiếm thông tin nhân viên với những lựa chọn như tìm theo họ và tên, tìm theo tuổi, tìm theo lương, quê quán.

Chức năng tính tổng lương: dùng để xuất tổng lương của tất cả nhân viên hoặc tổng lương của tường loại nhân viên (thâm niên, thời vụ).

Chức năng xoá: cho người quản lí xoá thông tin nhân viên theo các lựa chọn như xoá theo tuổi, lương, họ và tên.

2.1.3. Giao diện chương trình

- Đăng ký tài khoản

```
            HE THONG QUAN LY NHAN VIEN

                *****DANG KY******

              Tai Khoan: 1

               Mat khau: 1
            --------------------------------
Tai Khoan Cua Ban Dang Duoc Tao Xin Vui Long Doi Trong Giay Lat!!......
        Xin Chuc Mung Ban Da Tao Tai Khoan Thanh Cong!!
```

- Đăng nhập chương trình

```
          HE THONG QUAN LY NHAN VIEN

            ******DANG NHAP******

            Tai Khoan: 1

             Mat khau: 1

    Chuc mung ban da dang nhap thanh cong!!
```

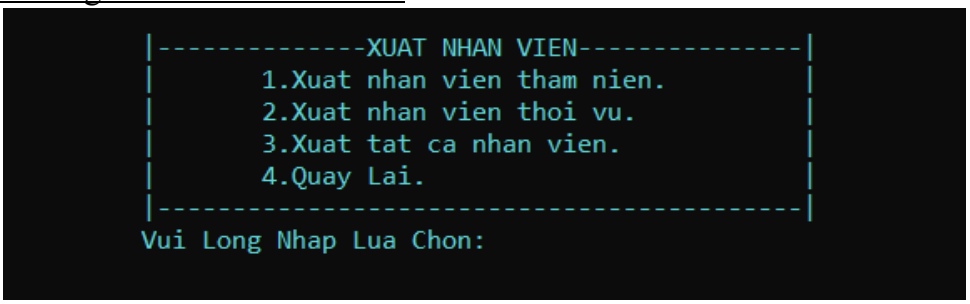
- Menu chương trình

```
 |-------------------------------------------------------------|
 | C-H-U-O-N-G---T-R-I-N-H---Q-U-A-N---L-Y---N-H-A-N---V-I-E-N |
 | <------------* Nhap so tuong ung trong menu   *------------> |
 | 1.-----------*         Nhap nhan vien         *------------- |
 | 2.-----------*         Xuat nhan vien         *------------- |
 | 3.-----------*         Sap xep nhan vien      *------------- |
 | 4.-----------*         Tim kiem nhan vien     *------------- |
 | 5.-----------*         Tong luong nhan vien   *------------- |
 | 6.-----------*         Xoa nhan vien          *------------- |
 | 7.---------*T-H-O-A-T----C-H-U-O-N-G---T-R-I-N-H*------------|
 |-------------------------------------------------------------|
Vui Long Nhap Lua Chon: 1
```

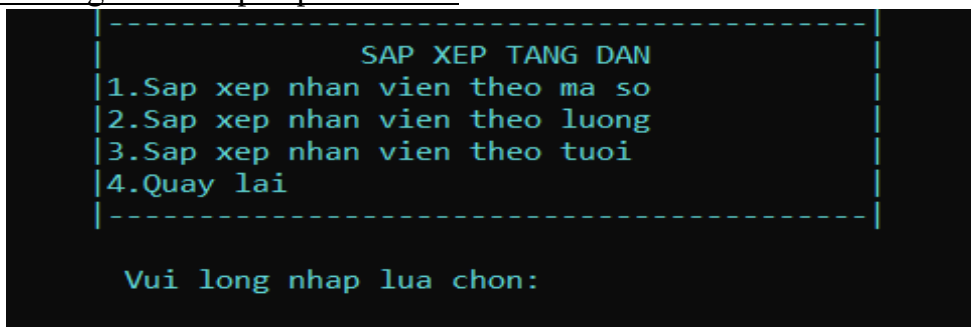
- Chức năng thứ 1: nhập nhân viên

```
    |--------------NHAP NHAN VIEN---------------|
    |       1.Nhap nhan vien tham nien.         |
    |       2.Nhap nhan vien thoi vu.           |
    |       3.Quay Lai.                         |
    |-------------------------------------------|
   Vui Long Nhap Lua Chon:
```

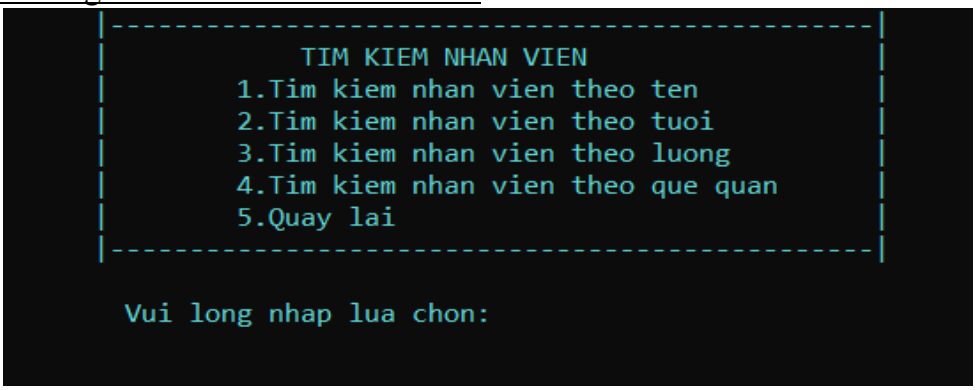
- Chức năng thứ 2: xuất nhân viên



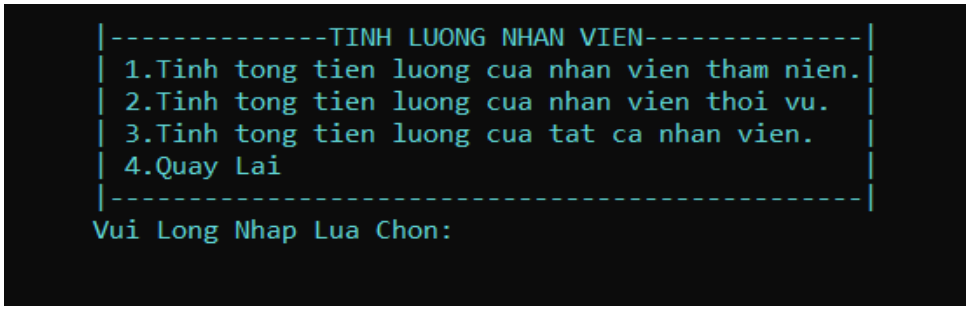
- Chức năng thứ 3: Sắp xếp nhân viên



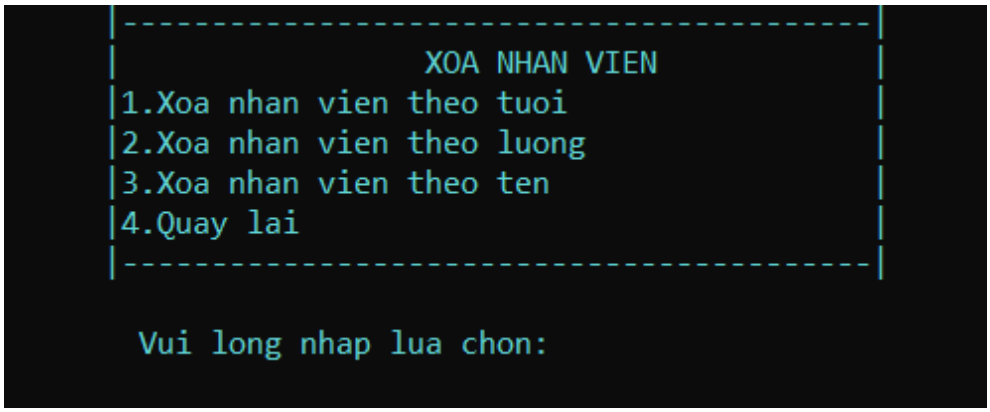
- Chức năng thứ 4: Tìm kiếm nhân viên



- Chức năng thứ 5: Tính tổng lương nhân viên



- Chức năng thứ 6: Xoá nhân viên



- Chức năng thứ 7: thoát chương trình

2.2. Nội dung lý thuyết

2.2.1. Tính chất hướng đối tượng

2.2.1.1. Tính trừu tượng

Trừu tượng hóa dữ liệu (Data abstraction) liên quan tới việc chỉ cung cấp thông tin cần thiết tới bên ngoài và ẩn chi tiết cơ sở của chúng, ví dụ: để biểu diễn thông tin cần thiết trong chương trình mà không hiển thị chi tiết về chúng.

Trừu tượng hóa dữ liệu (Data abstraction) là một kỹ thuật lập trình mà dựa trên sự phân biệt của Interface và Implementation (trình triển khai).

Trong ngôn ngữ lập trình C++, thì các lớp C++ cung cấp Trừu tượng hóa dữ liệu (Data abstraction) ở mức thật tuyệt vời. Chúng cung cấp đủ các phương thức public tới bên ngoài để thao tác với tính năng của đối tượng và để thao tác dữ liệu đối tượng

Trong bất kỳ chương trình C++ nào, nơi bạn triển khai một lớp với các thành viên là public và private, thì đó là một ví dụ của trừu tượng hóa dữ liệu. Bạn xem xét ví dụ sau:

Các thành viên public là Nhập và Xuất là các Interface (có thể nhìn thấy) tới bên ngoài và một người sử dụng cần biết chúng để sử dụng lớp đó. Thành viên private là MaSo,HoTen,Tuoi,Luong,... là cái gì đó mà người sử dụng không cần biết đến, nhưng là cần thiết cho lớp đó hoạt động một cách chính xác.

Lợi ích của tính trừu tượng

Trừu tượng hóa dữ liệu trong C++ mang lại hai lợi thế quan trọng:

- Phần nội vi hay bên trong lớp được bảo vệ tránh khỏi các lỗi do người dùng vô ý, mà có thể gây hư hỏng trạng thái của dữ liệu.
- Triển khai lớp có thể tiến hành qua thời gian để đáp ứng yêu cầu thay đổi hoặc bug các báo cáo mà không yêu cầu thay đổi trong code của người dùng.

Bằng việc định nghĩa các thành viên dữ liệu chỉ trong khu vực private của lớp, tác giả của lớp có thể tự do tạo các thay đổi trong dữ liệu. Nếu trình triển khai thay đổi, thì chỉ mã hóa lớp là cần kiểm tra để biết khía cạnh nào đem lại thay đổi. Nếu dữ liệu là public, thì khi đó bất kỳ hàm nào mà truy cập một cách trực tiếp tới các thành viên dữ liệu của phép biểu diễn cũ có thể bị phá vỡ.

2.2.1.2. Tính đóng gói

Tính đóng gói được thể hiện khi mỗi đối tượng mang trạng thái là private ở bên trong một class và những đối tượng khác không thể truy cập trực tiếp vào phạm vi này. Thay vào đó họ chỉ có thể gọi các hàm mang phạm vi public được gọi là phương thức.
Cụ thể, đối tượng sẽ mang trạng thái riêng thông qua các phương thức và không một class nào khác có thể truy cập vào được trừ khi cho phép. Nói chung trạng thái đối tượng không hợp lệ thường do chưa được kiểm tra tính hợp lệ, các bước thực hiện không đúng trình tự hoặc bị bỏ qua nên trong OOP có một quy tắc quan trọng cần nhớ đó là phải luôn khai báo các trạng thái bên trong của đối tượng là private và chỉ cho truy cập qua các public/protected method.

Hiểu một cách đơn giản, "đóng gói" là việc đưa tất cả thông tin, dữ liệu quan trọng vào bên trong một đối tượng (object). Sau đó, khi một đối tượng được khởi tạo từ lớp (class), thì dữ liệu và phương thức (method) đã được đóng gói trong đối tượng đó. Khi sử dụng, ta chỉ cần gọi tên phương thức chứ không cần truy cập đến dữ liệu bên trong.

Có thể nói tính đóng gói (Encapsulation) là cơ chế của che giấu dữ liệu (Data Hiding) bởi chúng được lớp (class) che giấu đi (ở dạng private) một số dữ liệu, hàm và phương thức để đảm bảo rằng các dữ liệu đó sẽ được truy cập và sử dụng đúng mục đích, đúng cách thông qua các hàm và phương thức ở dạng public mà class cung cấp. Đó là lý do bạn không thể truy cập đến các thuộc tính private hoặc gọi đến phương thức private của class từ bên ngoài class đó.
Ví dụ:

```
92  class NhanVienThamNien:public NhanVien{
93      private:
94          float HeSoLuong,ThamNien;
95      public:
96          NhanVienThamNien(){
97              HeSoLuong = 0;
98              ThamNien = 0;
99          }
100         ~NhanVienThamNien(){}
101
102         void Nhap();
103         void Xuat();
104         float TinhLuong();
105 };
```

Trong đoạn code trên tính đóng gói được thể hiện qua các thuộc tính MaSo, HoTen, Tuoi, Luong... và phương thức Nhap(), Xuat(), TinhLuong() vào trong class NhanVien. Bạn không thể truy cập đến các private data hoặc gọi đến private methods của class từ bên ngoài class đó.

Lợi ích của tính đóng gói

Nhìn chung, Tính đóng gói có một số ưu điểm như sau:

- Tính linh hoạt: Mã được đóng gói sẽ linh hoạt, dễ sửa đổi hơn là những đoạn mã độc lập.
- Khả năng tái sử dụng: Mã đã đóng gói có thể được tái sử dụng trong một ứng dụng hoặc nhiều ứng dụng. Từ một đối tượng, người dùng có thể chuyển sang dùng một đối tượng khác mà không phải đổi mã. Bởi vì cả hai đối tượng đều có giao diện như nhau.
- Khả năng bảo trì: Mã được đóng gói trong những phần riêng biệt, như là lớp, phương thức, giao diện,… Do đó, việc thay đổi, cập nhật một phần của ứng dụng không ảnh hưởng đến những phần còn lại. Điều này giúp giảm công sức và tiết kiệm thời gian cho các nhà phát triển.
- Khả năng kiểm thử: Đối với một lớp được đóng gói, Tester sẽ diễn viết những bài kiểm thử hơn. Các biến thành viên sẽ tập trung ở một nơi chứ không nằm rải rác. Do đó, kiểm thử viên cũng tiết kiệm được thời gian và công sức hơn.
- Che giấu dữ liệu: Khi sử dụng phương thức, người dùng chỉ cần biết nó tạo ra kết quả gì. Họ không cần quan tâm đến những chi tiết bên trong của đối tượng để sử dụng nó.

2.2.1.3. Tính kế thừa

Tính kế thừa là một trong những đặc tính quan trọng nhất của lập trình hướng đối tượng, lấy một thuộc tính, đặc tính của một lớp cha để áp dụng lên lớp con.Lớp kế thừa các thuộc tính từ một lớp khác được gọi là lớp con hoặc lớp dẫn xuất.Lớp có các thuộc tính được kế thừa bởi lớp con được gọi là lớp cha hoặc lớp cơ sở.

Trong lập trình hướng đối tượng, kế thừa là việc tái sử dụng lại một số thuộc tính, phương thức đã có sẵn từ lớp cơ sở. Là một đặc điểm của ngôn ngữ dùng để biểu diễn mối quan hệ đặc biệt hóa – tổng quát hóa giữa các lớp. Khái niệm kế thừa được phát minh năm 1967 cho ngôn ngữ Simula.

Cú pháp

```
0
1 class subclass_name : access_mode base_class_name
2 {
3   //body of subclass
4 };
5
```

Trong đó:

- subclass_name là tên lớp con sẽ áp dụng kế thừa
- base_class_name là tên lớp cha
- access_mode có thể là public, private hoặc protected

Nếu các bạn không chỉ rõ access_mode thì mặc định sẽ là private.

Ví dụ:

```cpp
#include<iostream>
#include<iomanip>
#include<vector>
#include<windows.h>
#include<string>
#include<fstream>
using namespace std;

class NhanVien{
    protected:
        string MaSo, HoTen;
        int Tuoi;
        float Luong;
        string GioiTinh, QueQuan;
        bool Check;
    public:
        void textcolor(int x);
        NhanVien(){
            MaSo = " ";
            HoTen = " ";
            Tuoi = 0;
            GioiTinh = " ";
            QueQuan = " ";
            Luong = 0;
        }
        ~NhanVien(){}

        virtual void Nhap();
        virtual void Xuat();
        virtual float TinhLuong()=0;
```

```cpp
class NhanVienThamNien:public NhanVien{
    private:
        float HeSoLuong,ThamNien;
    public:
        NhanVienThamNien(){
            HeSoLuong = 0;
            ThamNien = 0;
        }
        ~NhanVienThamNien(){}

        void Nhap();
        void Xuat();
        float TinhLuong();
};
```

Trong chương trình trên, class NhanVienThamNien là lớp con, nó sẽ được kế thừa các thành viên dữ liệu dạng public từ class NhanVien.

Phạm vi kế thừa: có 3 loại chính

1. public: Nếu kế thừa ở dạng này, sau khi kế thừa, tất cả các thành viên dạng public lớp cha sẽ public ở lớp con, dạng protected ở lớp cha vẫn sẽ là protected ở lớp con.
2. protected: Nếu dùng protected thì sau khi kế thừa, tất cả các thành viên dạng public lớp cha sẽ trở thành protected tại lớp con.
3. private: Trường hợp ta sử dụng private, thì sau khi kế thừa, tất cả các thành viên dạng public và protected ở lớp cha sẽ thành private tại lớp con.

Các loại kế thừa

Đơn kế thừa (Single Inheritance)

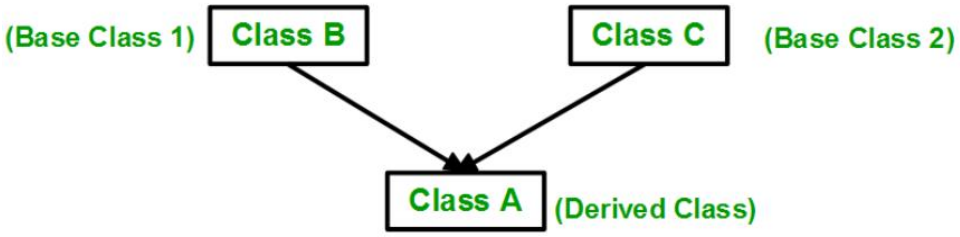
Đơn kế thừa: nghĩa là một lớp chỉ được kế thừa từ đúng một lớp khác. Hay nói cách khác, lớp con chỉ có duy nhất một lớp cha.

```
92  class NhanVienThamNien:public NhanVien{
93      private:
94          float HeSoLuong,ThamNien;
95      public:
96          NhanVienThamNien(){
97              HeSoLuong = 0;
98              ThamNien = 0;
99          }
100         ~NhanVienThamNien(){}
101
102         void Nhap();
103         void Xuat();
104         float TinhLuong();
105 };
```

Class NhanVienThamNien là lớp con kế thừa lớp cha duy nhất là Class NhanVien

Đa kế thừa (Multiple Inheritance)

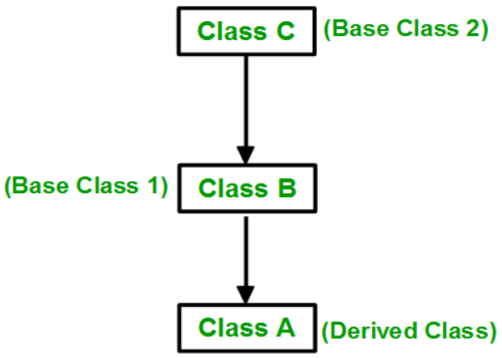
Đa kế thừa là một tính năng của ngôn ngữ C++. Trong đó một lớp có thể kế thừa từ nhiều hơn một lớp khác. Nghĩa là một lớp con được kế thừa từ nhiều hơn một lớp cơ sở.



Lưu ý: Khi đa kế thừa cần tránh trường hợp có nhiều lớp cơ sở có tên phương thức giống nhau. Vì khi gọi từ lớp con thì chương trình không biết nên gọi phương thức đó từ lớp cơ sở nào.

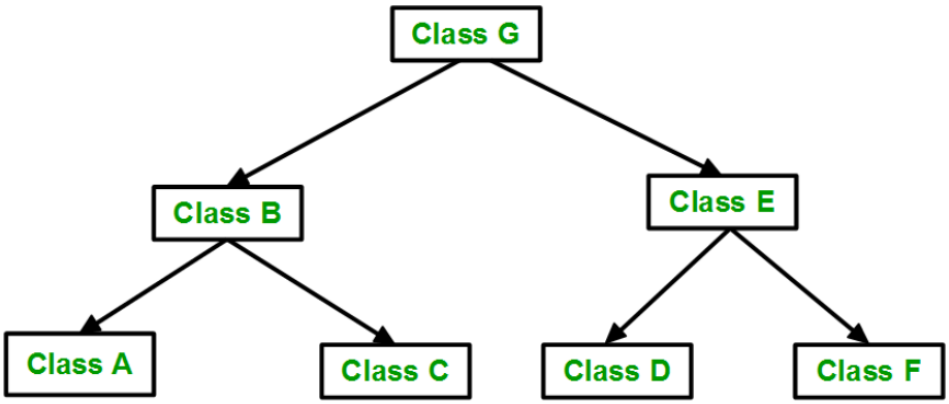
Kế thừa đa cấp (Multilevel Inheritance)

Kế thừa đa cấp: Trong kiểu thừa kế này, một lớp dẫn xuất được tạo từ một lớp dẫn xuất khác.



Kế thừa phân cấp (Hierarchical Inheritance)

Kế thừa phân cấp: Trong kiểu thừa kế này, sẽ có nhiều hơn một lớp con được kế thừa từ một lớp cha duy nhất.



Ví dụ:

```
#include<iostream>
#include<iomanip>
#include<vector>
#include<windows.h>
#include<string>
#include<fstream>
using namespace std;
class NhanVien{
    protected:
        string MaSo, HoTen;
        int Tuoi;
        float Luong;
        string GioiTinh, QueQuan;
        bool Check;
    public:
        void textcolor(int x);
        NhanVien(){
            MaSo = " ";
            HoTen = " ";
            Tuoi = 0;
            GioiTinh = " ";
            QueQuan = " ";
            Luong = 0;
        }
        ~NhanVien(){}

        virtual void Nhap();
        virtual void Xuat();
        virtual float TinhLuong()=0;
```

```
class NhanVienThamNien:public NhanVien{
    private:
        float HeSoLuong,ThamNien;
    public:
        NhanVienThamNien(){
            HeSoLuong = 0;
            ThamNien = 0;
        }
        ~NhanVienThamNien(){}

        void Nhap();
        void Xuat();
        float TinhLuong();
};
```

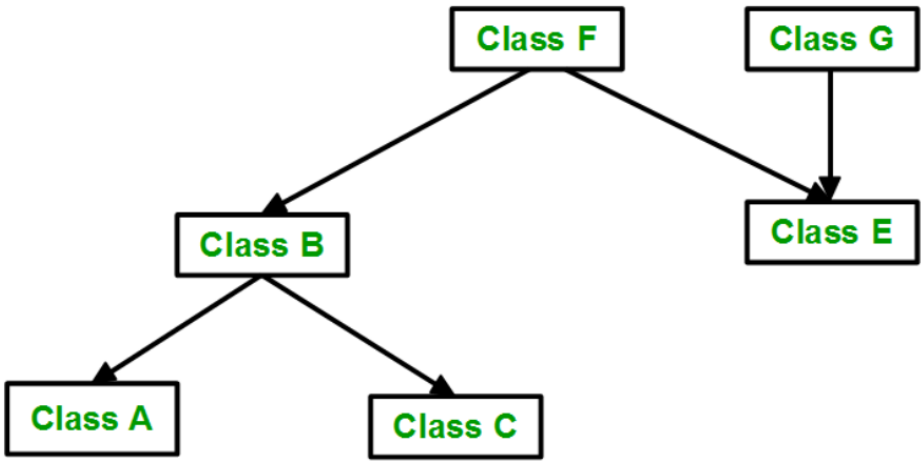
```
133 class NhanVienThoiVu:public NhanVien{
134     private:
135         int SoGioLam;
136         float DonGia;
137     public:
138         NhanVienThoiVu(){
139             SoGioLam = 0;
140             DonGia = 0;
141         }
142         ~NhanVienThoiVu(){}
143         void Nhap();
144         void Xuat();
145         float TinhLuong();
146 };
```

Class NhanVien là class cha được hai class con là NhanVienThamNien và NhanVienThoiVu kế thừa.

Kế thừa ảo (Hybrid (Virtual) Inheritance)

Kế thừa lai (Kế thừa ảo): được thực hiện bằng cách kết hợp nhiều hơn một loại thừa kế.

Ví dụ: Kết hợp kế thừa phân cấp và đa kế thừa.



2.2.1.4. Tính* đa *hình

Đa hình (polymorphism) nghĩa là có nhiều hình thái khác nhau. Tiêu biểu là, đa hình xuất hiện khi có một cấu trúc cấp bậc của các lớp và chúng là liên quan với nhau bởi tính kế thừa.

Đa hình trong C++ nghĩa là một lời gọi tới một hàm thành viên sẽ làm cho một hàm khác để được thực thi phụ thuộc vào kiểu của đối tượng mà triệu hồi hàm đó.

Trong đa hình khi định nghĩa một phương thức của base class (lớp cha) là hàm ảo (virtural fuction) thì các derived class (lớp dẫn xuất) có thể định nghĩa lại phương thức đó hoặc không.

Ví dụ:

```cpp
#include<iostream>
#include<iomanip>
#include<vector>
#include<windows.h>
#include<string>
#include<fstream>
using namespace std;

class NhanVien{
    protected:
        string MaSo, HoTen;
        int Tuoi;
        float Luong;
        string GioiTinh, QueQuan;
        bool Check;
    public:
        void textcolor(int x);
        NhanVien(){
            MaSo = " ";
            HoTen = " ";
            Tuoi = 0;
            GioiTinh = " ";
            QueQuan = " ";
            Luong = 0;
        }
        ~NhanVien(){}

        virtual void Nhap();
        virtual void Xuat();
        virtual float TinhLuong()=0;
```

```cpp
class NhanVienThamNien:public NhanVien{
    private:
        float HeSoLuong,ThamNien;
    public:
        NhanVienThamNien(){
            HeSoLuong = 0;
            ThamNien = 0;
        }
        ~NhanVienThamNien(){}

        void Nhap();
        void Xuat();
        float TinhLuong();
};
```

```
111 void NhanVienThamNien::Nhap(){
112     NhanVien::Nhap();
113     do{
114         cout<<"Nhap He So Luong: ";
115         cin>>HeSoLuong;
116         if(HeSoLuong<=0)
117             cout<<"He So Luong Khong Hop Le.Xin Kiem Tra Lai!!"<<endl;
118     }while(HeSoLuong<=0);
119 
120     do{
121         cout<<"Nhap Tham Nien: ";
122         cin>>ThamNien;
123         if(ThamNien<=0)
124             cout<<"Tham Nien Khong Hop Le.Xin Kiem Tra Lai!!"<<endl;
125     }while(ThamNien<=0);
126 }
```

Phương thức Nhap () của class NhanVienThamNien tận dụng những thông tin đã có trong phương thức Nhap () của class NhanVien và thêm vào những thuộc tính khác như HeSoLuong, ThamNien.

Trong đa hình khi định nghĩa một phương thức của base class (lớp cha) là hàm thuần ảo (pure virtural fuction) thì các derived class (lớp dẫn xuất) bắt buộc phải định nghĩa lại phương thức đó .

```
1  #include<iostream>
2  #include<iomanip>
3  #include<vector>
4  #include<windows.h>
5  #include<string>
6  #include<fstream>
7  using namespace std;
8  
9  class NhanVien{
10     protected:
11         string MaSo, HoTen;
12         int Tuoi;
13         float Luong;
14         string GioiTinh, QueQuan;
15         bool Check;
16     public:
17         void textcolor(int x);
18         NhanVien(){
19             MaSo = " ";
20             HoTen = " ";
21             Tuoi = 0;
22             GioiTinh = " ";
23             QueQuan = " ";
24             Luong = 0;
25         }
26         ~NhanVien(){}
27 
28         virtual void Nhap();
29         virtual void Xuat();
30         virtual float TinhLuong()=0;
```

```
92  class NhanVienThamNien:public NhanVien{
93      private:
94          float HeSoLuong,ThamNien;
95      public:
96          NhanVienThamNien(){
97              HeSoLuong = 0;
98              ThamNien = 0;
99          }
100         ~NhanVienThamNien(){}
101
102         void Nhap();
103         void Xuat();
104         float TinhLuong();
105 };
106
107 float NhanVienThamNien::TinhLuong(){
108     return HeSoLuong*1600 + ThamNien*500;
109 }
```

Phương thức TinhLuong trong class NhanVienThamNien đã được định nghĩa lại.

2.2.2. Cấu trúc dữ liệu giải thuật

2.2.2.1. Thuật toán sắp xếp đổi chỗ trực tiếp (Interchange Sort)

Như đã biết, để sắp xếp một dãy số, ta có thể xét các nghịch thế có trong dãy và triệt tiêu dần chúng đi. Ý tưởng chính của giải thuật là xuất phát từ đầu dãy, tìm tất cả nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế. Lặp lại xử lý trên với các phần tử kế tiếp theo trong dãy.

Giải thuật:

Bước 1:

i=0;/ Bắt đầu từ đầu dãy

Bước 2:

j = i+1; // Tìm tất cả các phần tử a[j]<a[i], với j>i

Bước 3:

Trong khi j < N thực hiện

Nếu a[j]<a[i] *//xét cặp a[i], a[j]*

Swap(a[i],a[j]);

j=j+1;

Bước 4:

i = i+1;

Nếu i<n-1: Lặp lại Bước 2

Ngược lại: Dừng.

Ví dụ:

```
445             case 1:
446                 {
447                     for(int i=0; i<NV.size()-1; i++) {
448                         for(int j=i+1; j< NV.size(); j++) {
449                             if(NV[i]->GetMaSo() > NV[j]->GetMaSo()){
450                                 swap(NV[i],NV[j]);
451                             }
452                         }
453                     }
454                     XuatDSS();
455                     system("pause");
456                 }
457                 break;
```

2.2.2.2. Vecto trong C++

Không giống như array (mảng), chỉ một số giá trị nhất định có thể được lưu trữ dưới một tên biến duy nhất. Vector trong C++ giống dynamic array (mảng động) nhưng có khả năng tự động thay đổi kích thước khi một phần tử được chèn hoặc xóa tùy thuộc vào nhu cầu của tác vụ được thực thi, với việc lưu trữ của chúng sẽ được vùng chứa tự động xử lý. Các phần tử vector được đặt trong contiguous storage (bộ nhớ liền kề) để chúng có thể được truy cập và duyệt qua bằng cách sử dụng iterator.

Lý do nên dùng vecto

Nếu bạn đã phát chán việc quản lý mảng động qua con trỏ trong C++ hay chán phải tạo mảng mới, copy các phần tử qua mảng mới, rồi lại xóa mảng cũ mỗi khi bạn muốn resize kích thước mảng động trong C++. Thật thừa thải, tốn thời gian và đó là thời điểm ta nhận ra C++ còn có vector.

Một số điểm nổi trội của Vector:

– Bạn không cần phải khai báo kích thước của mảng ví dụ int A[100]…, vì vector có thể tự động nâng kích thước lên.

– Nếu bạn thêm 1 phần tử vào vector đã đầy rồi, thì nó sẽ tự động tăng kích thước của nó lên để dành chỗ cho giá trị mới này.

– Vector còn giúp cho bạn biết số lượng các phần tử mà bạn đang lưu trong đó.

– Dùng số phần tử âm vẫn được trong vector ví dụ A[-6], A[-9], rất tiện trong việc cài đặt các giải thuật.

Cú pháp

```
#include <vector>

//...

vector<object_type> variable_name;
```

Ví dụ

```cpp
#include <vector>

int main()

{

    std::vector<int> my_vector;

}
```

Áp dụng

```cpp
class QuanLyNhanVien{
    private:
        vector<NhanVien *> NV;
```

Cơ chế ngăn chặn rò rỉ bộ nhớ của vecto

Khi một biến vector rời khỏi phạm vi đoạn code mà chương trình đang chạy, nó sẽ tự động giải phóng những phần bộ nhớ mà nó kiểm soát (nếu cần). Điều này không chỉ tiện dụng (vì bạn không cần tự tay giải phóng bộ nhớ), mà nó còn giúp ngăn ngừa lỗi rò rỉ bộ nhớ (memory leaks).

```cpp
void doSomething(bool earlyExit)
{
        int *array = new int[3]{ 1, 3, 2 };

        if (earlyExit) // thoát khỏi hàm
                return;

        delete[] array; // trường hợp hàm thoát sớm, array sẽ không bị xóa
}
```

Vecto tự ghi độ dài của mình

Không giống như mảng động được tích hợp sẵn của C++, cái mà không biết được độ dài của mảng mà nó đang trỏ tới là bao nhiêu, std::vectors tự theo dõi độ dài của chính nó. Chúng ta có thể lấy được độ dài của vector thông qua hàm size():

```cpp
#include <iostream>
#include <vector>

void printLength(const std::vector<int>& array)
{
    std::cout << "The length is: " << array.size() << '\n';
}

int main()
{
    std::vector array { 9, 7, 5, 3, 1 };
    printLength(array);

    return 0;
}
```

Các hàm trong vecto

1. Nhóm Modifiers

1. push_back(): Hàm đẩy một phần tử vào vị trí sau cùng của vector. Nếu kiểu của đối tượng được truyền dưới dạng tham số trong push_back() không giống với kiểu của vector thì sẽ bị ném ra.

ten-vector.push_back(ten-cua-phan-tu);

Ví dụ:

```
219                     NV.push_back(nv);
```

2. erase(): Hàm được sử dụng để xóa các phần tử tùy theo vị trí vùng chứa

ten-vector.erase(position);

ten-vector.erase(start-position, end-position);

Ví dụ:

```
640                     NV.erase(NV.begin() + i);
```

3. swap(): Hàm được sử dụng để hoán đổi nội dung của một vector này với một vector khác cùng kiểu. Kích thước có thể khác nhau.

ten-vector-1.swap(ten-vector-2);

Ví dụ:

```
450                         swap(NV[i],NV[j]);
```

2. Nhóm Iterators

begin(): đặt iterator đến phần tử đầu tiên trong vector

ten-vector.begin();

Ví dụ:

```
640                     NV.erase(NV.begin() + i);
```

3. Nhóm Capacity

size(): hàm sẽ trả về số lượng phần tử đang được sử dụng trong vector

ten-vector.size();

Ví dụ:

```
637                 for(int i=0; i<NV.size(); i++)
```

4. Nhóm Element access

at(g): Trả về một tham chiếu đến phần tử ở vị trí ‘g’ trong vectorten-vector.at(position);

Ví dụ:

```
352                 this->NV.at(i)->Xuat();
```

2.2.3. Đồ hoạ windows.h

2.2.3.1. Windows.h là gì?

Windows.h là một header của Windows dành riêng cho ngôn ngữ lập trình C và C++. Trong đó chứa các khai báo cho tất cả các hàm (function) trong Windows API, tất cả các macro thường dùng bởi các lập trình viên Windows, và tất cả các kiểu dữ liệu (data type) sử dụng cho nhiều hàm và hệ thống con (subsystem).

2.2.3.2. Áp dụng

Để chương trình có thêm chút gì đó là màu sắc chúng em đã sử dụng hàm textcolor cho màn hình console.

```
void textcolor(int x)
{
    HANDLE mau;
    mau = GetStdHandle

        (STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(mau, x);
}
```

```
|-------------------------------------------------------------|
| C-H-U-O-N-G---T-R-I-N-H---Q-U-A-N---L-Y---N-H-A-N---V-I-E-N  |
| <------------* Nhap so tuong ung trong menu   *------------> |
| 1.-----------*         Nhap nhan vien         *------------- |
| 2.-----------*         Xuat nhan vien         *------------- |
| 3.-----------*         Sap xep nhan vien      *------------- |
| 4.-----------*         Tim kiem nhan vien     *------------- |
| 5.-----------*         Tong luong nhan vien   *------------- |
| 6.-----------*         Xoa nhan vien          *------------- |
| 7.----------*T-H-O-A-T----C-H-U-O-N-G---T-R-I-N-H*-----------|
|-------------------------------------------------------------|
Vui Long Nhap Lua Chon:
```

2.3. Kết luận

Kết quả đạt được	Sau thời gian dài thực hiện đề tài, chương trình đã hoàn thành và đạt được một số kết quả sau: - Hiểu rõ được tầm quan trọng của việc quản lí hồ sơ nhân viên. - Xây dựng chương trình tương đối thành công đáp ứng nhu cầu đặt ra của doanh nghiệp trong việc quản lí nhân viên. - Chương trình hỗ trợ tiết kiệm được thời gian và công sức trong việc quản lý thông tin chi tiết từng nhân viên.

Hạn chế của đề tài	- Chưa thể có thêm nhiều chức năng mới mẻ trong đề tài quản lí nhân viên. - Với nỗ lực của bản thân, chúng em đã cố gắng hoàn thành đề tài. Do hạn chế về một phần nào đó về kiến thức và kinh nghiệm thực tiễn nên chúng em chưa thể mở rộng được thêm đề tài quản lí nhân viên này.
Hướng phát triển của đề tài	- Nhóm chúng em mong muốn đề tài phát triển trở thành một phần mềm quản lí nhân viên chuyên nghiệp, xây dựng giao diện hoàn thiện hơn. Cung cấp đầy đủ những chức năng mới mẻ hơn cho việc quản lí nhân viên. - Và mong rằng chương trình sẽ được cải thiện một cách tốt nhất để một doanh nghiệp nào đó có thể sử dụng chương trình này cho việc quản lí nhân viên.
Đề nghị ý kiến	- Trong thời gian thực hiện đề tài không thể tránh khỏi những thiếu sót nhất định,chúng em rất mong nhận được ý kiến đóng góp từ phía quý thầy cô và các bạn để đề tài ngày càng hoàn thiện và hay hơn.
Tài liệu tham khảo	- https://codelearn.io/ - https://nguyenvanhieu.vn/ - https://khiemle.dev/ - https://stackoverflow.com/ -Sách: . The C++ Programming Language, 4th Ed . C++ và lập trình hướng đối tượng - Phạm Văn Ất

PHẦN 3: PHỤ LỤC

3.1. Link gihub

HuynhTai81/Project (github.com)

3.2. Cách cài đặt chương trình

Để có thể chạy được chương trình quản lí nhân viên thì trước hết chúng ta cần phải cài đặt một trong những phần mềm sau:

	Link hướng dẫn tải	Link video hướng dẫn tải

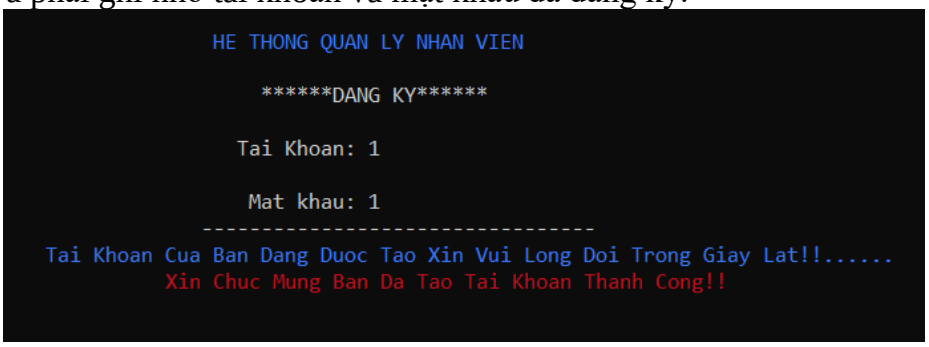
Dev-C++	https://codecute.com/c/huong-dan-cai-dat-ide-dev-c-lap-trinh-c-c.html	https://www.youtube.com/watch?v=LKom13wmYtk
Visual studio code	https://codelearn.io/sharing/huong-dan-cai-dat-visual-studio-code-lap-trinh-cpp	https://www.youtube.com/watch?v=a4NIBn44FSE

3.3. Hướng dẫn sử dụng chương trình

- Lưu ý: Sau khi thực hiện một chức năng con bất kỳ của một chức năng chính nào đó cần phải nhấn một phím bất kỳ để quay lại giao diện của chương trình chức năng đó.

3.3.1. Đăng ký tài khoản

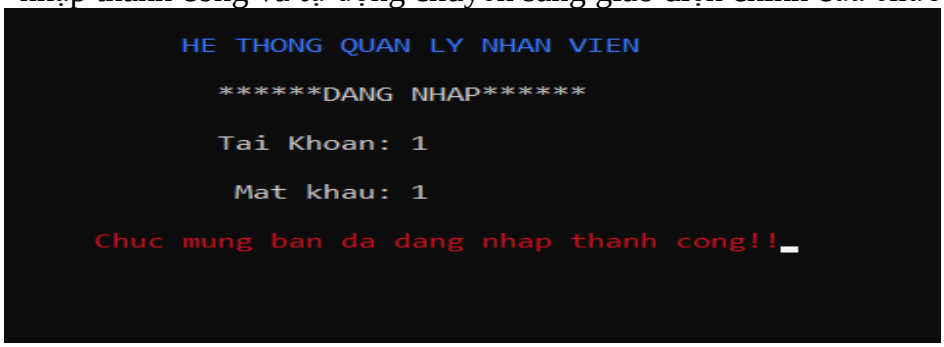
Để có thể vào giao diện của chương trình chúng ta cần phải tạo tài khoản để có thể đăng nhập và phải ghi nhớ tài khoản và mật khẩu đã đăng ký.



3.3.2. Đăng nhập

Sau khi đăng ký tài khoản thành công, chương trình sẽ tự động chuyển sang giao diện đăng nhập và chúng ta cần nhập tài khoản và mật khẩu vừa đăng ký vào để có thể đăng nhập vào giao diện chương trình quản lí nhân viên.

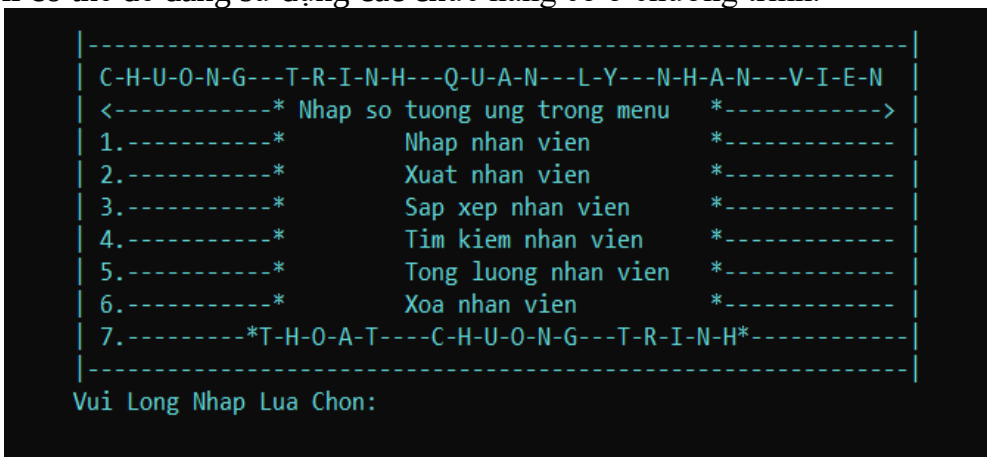
- Nếu chúng ta điền tài khoản & mật khẩu đúng thì chương trình sẽ báo đăng nhập thành công và tự động chuyển sang giao diện chính của chương trình.



- Ngược lại nếu tài khoản hoặc mật khẩu sai chương trình sẽ báo lỗi tài khoản hoặc mật khẩu sai và không thể đăng nhập vào giao diện chính chương trình được và chúng ta cần phải đăng nhập lại.

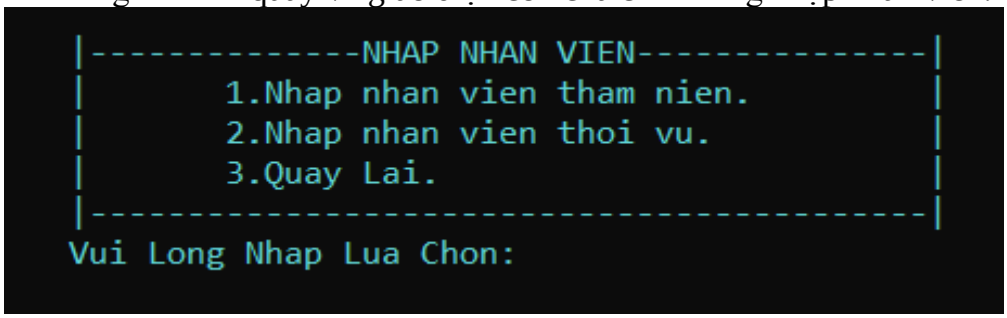
3.3.3. Giao diện chính của chương trình

Sau khi đăng nhập thành công, chương trình sẽ chuyển sang giao diện chính để người quản lí có thể dễ dàng sử dụng các chức năng có ở chương trình.



3.3.4. Chức năng thứ 1: Nhập nhân viên

Nếu chúng ta chọn chức năng thứ 1 thì chương trình sẽ hiện ra giao diện con với 2 lựa chọn là nhập nhân viên thâm niên và nhân viên thời vụ và sau khi nhập xong thông tin nhân viên thì chương trình sẽ quay về giao diện con của chức năng nhập nhân viên.



- Nếu chúng ta cần thêm nhân viên thâm niên thì chỉ cần nhập số 1 chương trình sẽ hiện ra như sau:

```
        NHAP NHAN VIEN THAM NIEN
Nhap so luong nhan vien tham nien: 2

Nhan Vien Thu 1
Nhap Ma So: 1
Nhap Ho Ten: Nguyen Van A
Nhap Gioi Tinh: Nam
Nhap Tuoi: 19
Nhap Que Quan: TPHCM
Nhap He So Luong: 10
Nhap Tham Nien: 10

Nhan Vien Thu 2
Nhap Ma So: 2
Nhap Ho Ten: Nguyen Thi B
Nhap Gioi Tinh: Nu
Nhap Tuoi: 20
Nhap Que Quan: Ca Mau
Nhap He So Luong: 20
Nhap Tham Nien: 20
```

- Lựa chọn số 2 là thêm nhân viên thời vụ:

```
          NHAP NHAN VIEN THOI VU
Nhap so luong nhan vien thoi vu: 2

Nhan Vien Thu 1
Nhap Ma So: 3
Nhap Ho Ten: Nguyen Van C
Nhap Gioi Tinh: Nam
Nhap Tuoi: 40
Nhap Que Quan: Nghe An
Nhap So Gio Lam: 20
Nhap Don Gia: 10

Nhan Vien Thu 2
Nhap Ma So: 4
Nhap Ho Ten: Nguyen Thi D
Nhap Gioi Tinh: Nu
Nhap Tuoi: 25
Nhap Que Quan: TPHCM
Nhap So Gio Lam: 30
Nhap Don Gia: 10
```

- Bấm lựa chọn thứ 3 để quay lại giao diện chính của chương trình để sử dụng các chức năng khác.

3.3.5. Chức năng thứ 2: Xuất nhân viên

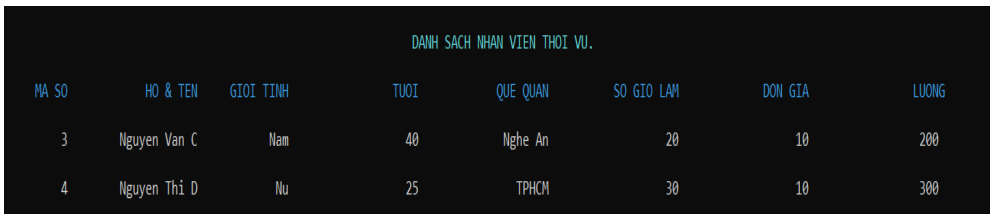
Đến với lựa chọn thứ 2, thì chương trình sẽ có 3 lựa chọn con là xuất nhân viên thâm niên, xuất nhân viên thời vụ và cuối cùng là xuất tất cả nhân viên cả thâm niên và thời vụ.

```
|--------------XUAT NHAN VIEN---------------|
|       1.Xuat nhan vien tham nien.         |
|       2.Xuat nhan vien thoi vu.           |
|       3.Xuat tat ca nhan vien.            |
|       4.Quay Lai.                         |
|-------------------------------------------|
Vui Long Nhap Lua Chon:
```

- Lựa chọn thứ 1: xuất nhân viên thâm niên:

```
                    DANH SACH NHAN VIEN THAM NIEN.
MA SO     HO & TEN    GIOI TINH     TUOI     QUE QUAN     HE SO LUONG     THAM NIEN     LUONG
  1       Nguyen Van A      Nam       19        TPHCM          10             10         21000
  2       Nguyen Thi B      Nu        20        Ca Mau         20             20         42000
```

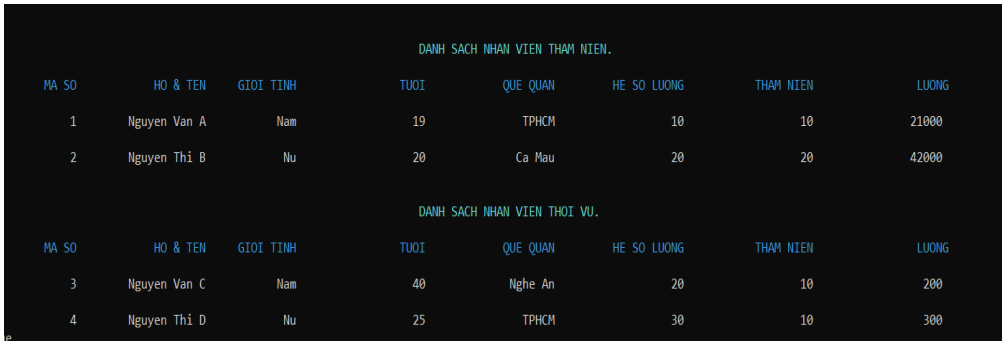
- Lựa chọn thứ 2: xuất nhân viên thời vụ:



DANH SACH NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
3	Nguyen Van C	Nam	40	Nghe An	20	10	200
4	Nguyen Thi D	Nu	25	TPHCM	30	10	300

- Lựa chọn thứ 3: xuất tất cả nhân viên:



DANH SACH NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
1	Nguyen Van A	Nam	19	TPHCM	10	10	21000
2	Nguyen Thi B	Nu	20	Ca Mau	20	20	42000

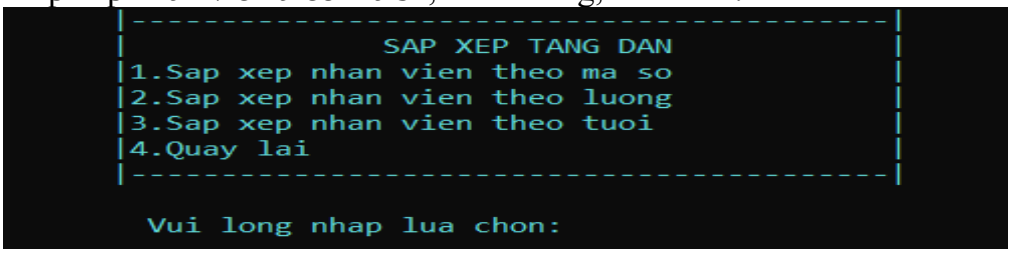
DANH SACH NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
3	Nguyen Van C	Nam	40	Nghe An	20	10	200
4	Nguyen Thi D	Nu	25	TPHCM	30	10	300

- Bấm lựa chọn 4 để quay về giao diện chính của chương trình.

3.3.6. Chức năng thứ 3: Sắp xếp nhân viên

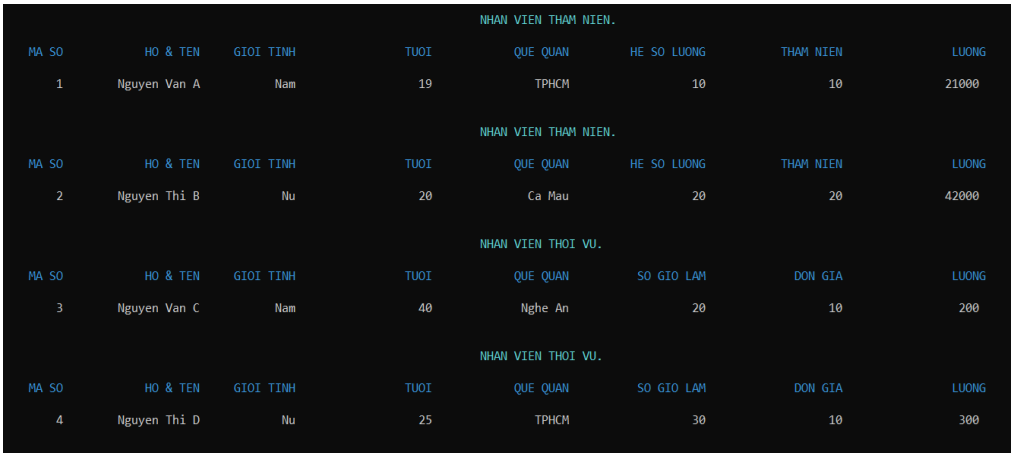
Đến với lựa chọn thứ 3 là chức năng sắp xếp nhân viên theo chiều tăng dần với các lựa chọn như sắp xếp nhân viên theo mã số, theo lương, theo tuổi.



```
|-------------------------------------------|
|              SAP XEP TANG DAN             |
|1.Sap xep nhan vien theo ma so             |
|2.Sap xep nhan vien theo luong             |
|3.Sap xep nhan vien theo tuoi              |
|4.Quay lai                                 |
|-------------------------------------------|

  Vui long nhap lua chon:
```

- Lựa chọn thứ 1: sắp xếp nhân viên theo mã số



NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
1	Nguyen Van A	Nam	19	TPHCM	10	10	21000

NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
2	Nguyen Thi B	Nu	20	Ca Mau	20	20	42000

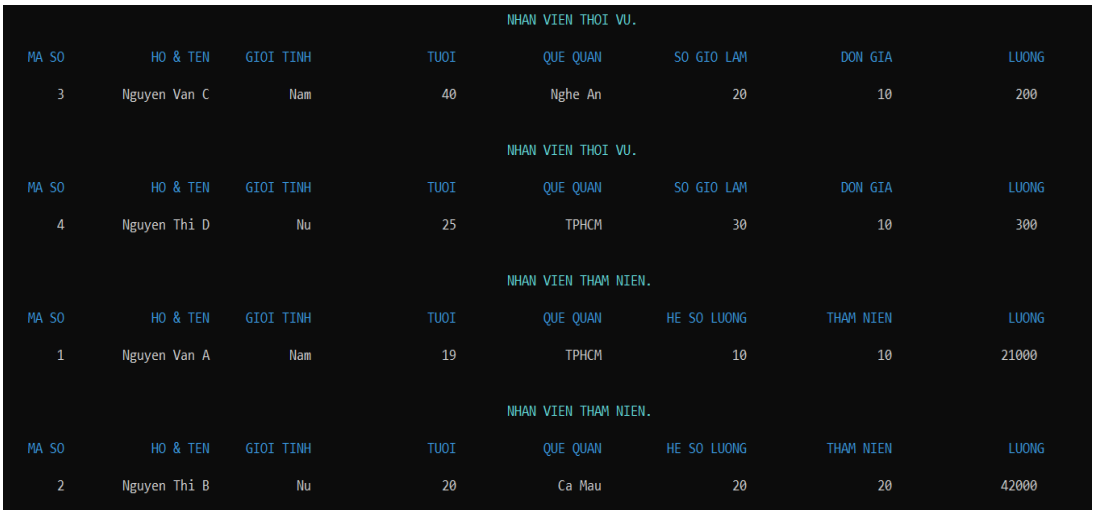
NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
3	Nguyen Van C	Nam	40	Nghe An	20	10	200

NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
4	Nguyen Thi D	Nu	25	TPHCM	30	10	300

- Lựa chọn thứ 2: sắp xếp nhân viên theo lương



NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
3	Nguyen Van C	Nam	40	Nghe An	20	10	200

NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
4	Nguyen Thi D	Nu	25	TPHCM	30	10	300

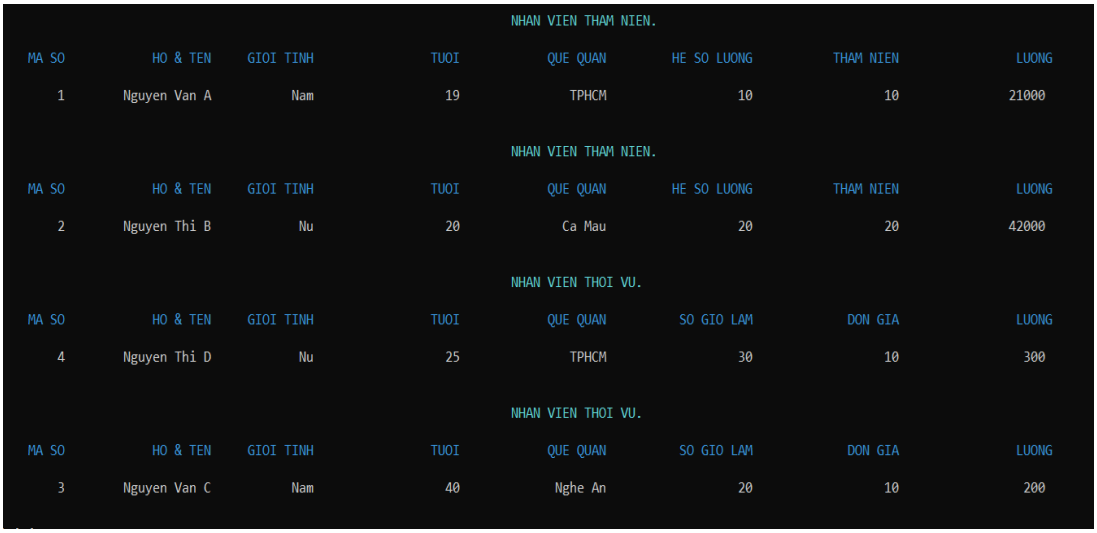
NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
1	Nguyen Van A	Nam	19	TPHCM	10	10	21000

NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
2	Nguyen Thi B	Nu	20	Ca Mau	20	20	42000

- Lựa chọn thứ 3: sắp xếp nhân viên theo tuổi



NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
1	Nguyen Van A	Nam	19	TPHCM	10	10	21000

NHAN VIEN THAM NIEN.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	HE SO LUONG	THAM NIEN	LUONG
2	Nguyen Thi B	Nu	20	Ca Mau	20	20	42000

NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
4	Nguyen Thi D	Nu	25	TPHCM	30	10	300

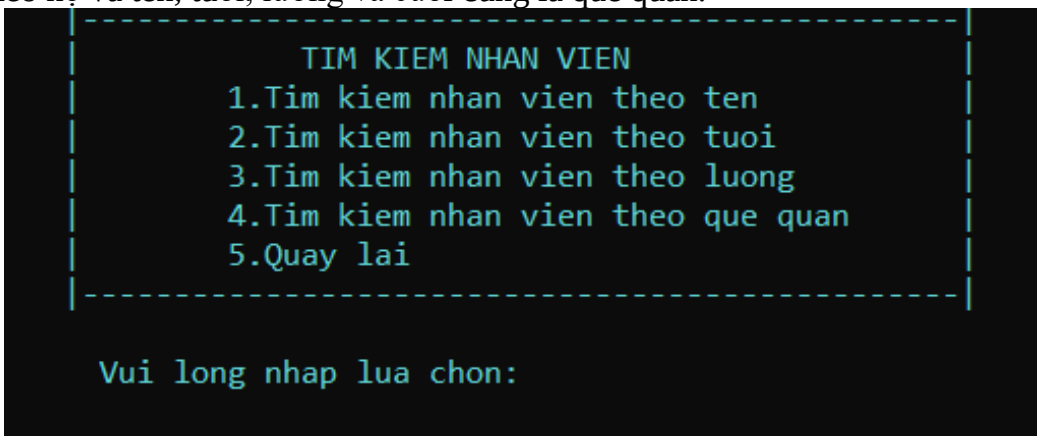
NHAN VIEN THOI VU.

MA SO	HO & TEN	GIOI TINH	TUOI	QUE QUAN	SO GIO LAM	DON GIA	LUONG
3	Nguyen Van C	Nam	40	Nghe An	20	10	200

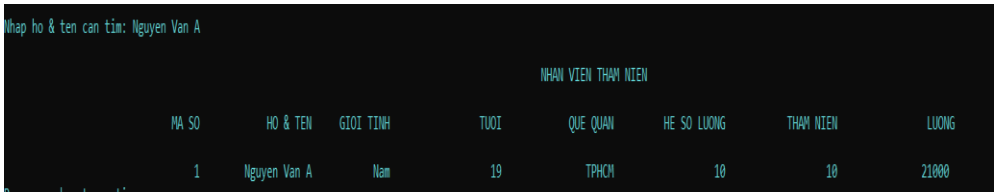
- Bấm lựa chọn 4 để quay lại giao diện chính của chương trình để chọn các chức năng khác.

3.3.7. Chức năng thứ 4: Tìm kiếm nhân viên

Tiếp theo đến với lựa chọn thứ 4 là chức năng tìm kiếm nhân viên chúng ta sẽ dễ dàng tìm kiếm thông tin của một nhân viên bất kỳ thông qua các lựa chọn như tìm kiếm nhân viên theo họ và tên, tuổi, lương và cuối cùng là quê quán.

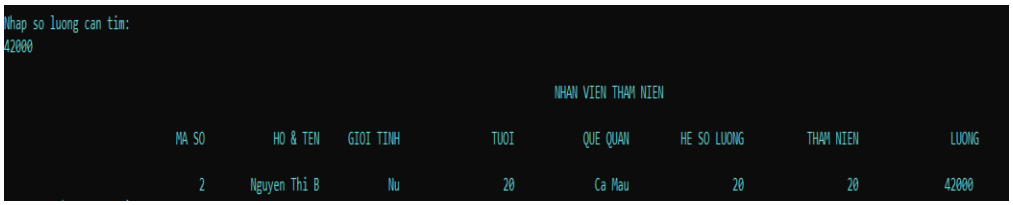


- Lựa chọn thứ 1: tìm kiếm nhân viên theo họ và tên

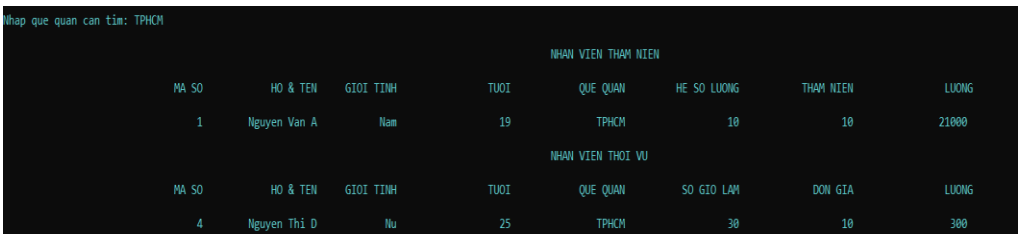


- Lựa chọn thứ 2: tìm kiếm nhân viên theo tuổi

- Lựa chọn thứ 3: tìm kiếm nhân viên theo lương



- Lựa chọn thứ 4: tìm kiếm nhân viên theo quê quán

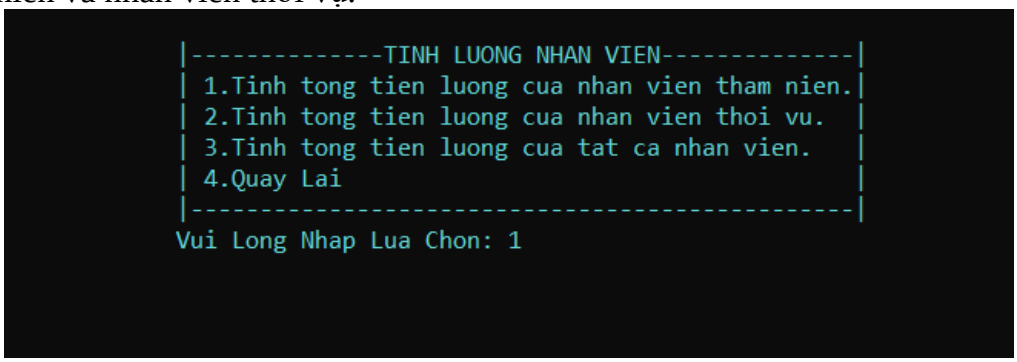


- Bấm lựa chọn 5 để quay lại giao diện chính của chương trình.

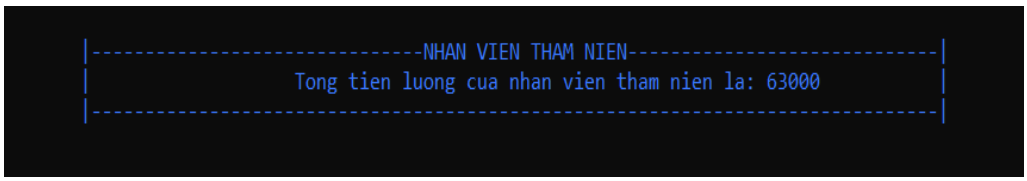
Lưu ý: Khi nhập sai thông tin cần tìm chương trình sẽ không hiển thị ra màn hình bất kỳ thông tin nào.

3.3.8. Chức năng thứ 5: Tính tổng lương nhân viên

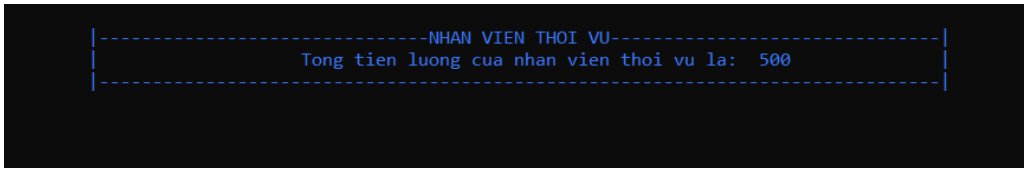
Tiếp đến với lựa chọn thứ 5 là chức năng tính tổng lương nhân viên bao gồm tổng lương nhân viên thâm niên, tổng lương nhân viên thời vụ và tính tổng lương của cả nhân viên thâm niên và nhân viên thời vụ.



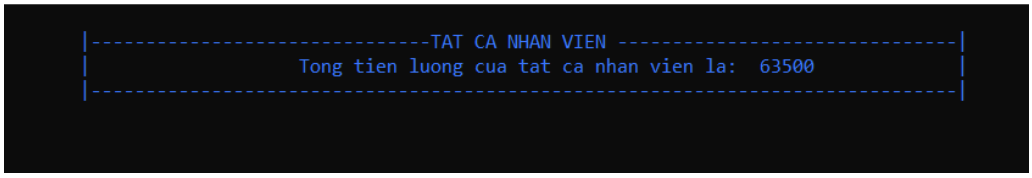
- Lựa chọn thứ 1: tính tổng tiền lương của nhân viên thâm niên



- Lựa chọn thứ 2: tính tổng tiền lương của nhân viên thời vụ



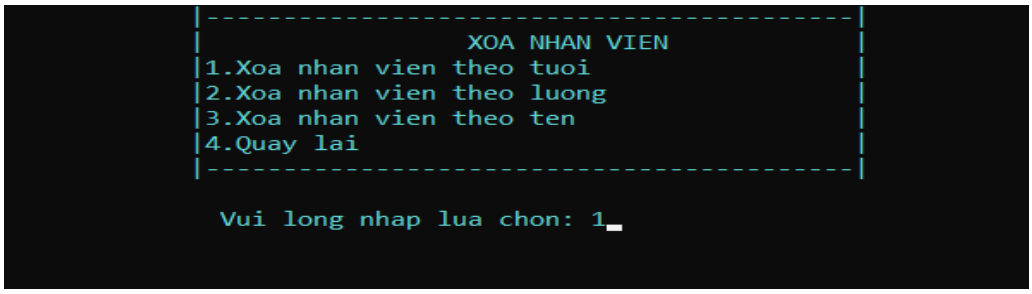
- Lựa chọn thứ 3: tính tổng tiền lương của nhân viên



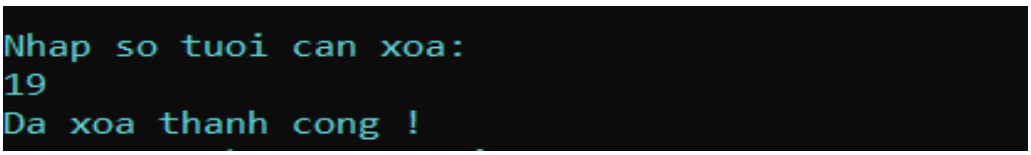
- Bấm lựa chọn 4 để quay lại giao diện chính của chương trình để sử dụng những chức năng khác.

3.3.9. Chức năng thứ 6: Xoá nhân viên

Đến với lựa chọn thứ 6 là chức năng xoá nhân viên, chúng ta chỉ cần nhập tên hoặc lương hoặc tuổi của nhân viên cần xoá vào thì chương trình sẽ tự động xoá đi những nhân viên đó.



- Lựa chọn thứ 1: xoá nhân viên theo tuổi



- Lựa chọn thứ 2: xoá nhân viên theo lương

- Lựa chọn thứ 3: xoá nhân viên theo tên

- Bấm lựa chọn 4 để quay lại giao diện chính của chương trình.

Lưu ý: Khi nhập sai thông tin nhân viên cần xoá chương trình sẽ báo không có thông tin nhân viên nào như trên để xoá.

3.3.10. Thoát chương trình

Bấm lựa chọn 7 thì chương trình sẽ tự động thoát.