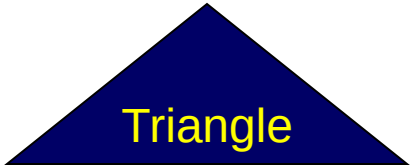
KẾ THỪA - **Inheritance**

VD dẫn nhập khái niệm kế thừa

Polygon

Rectangle

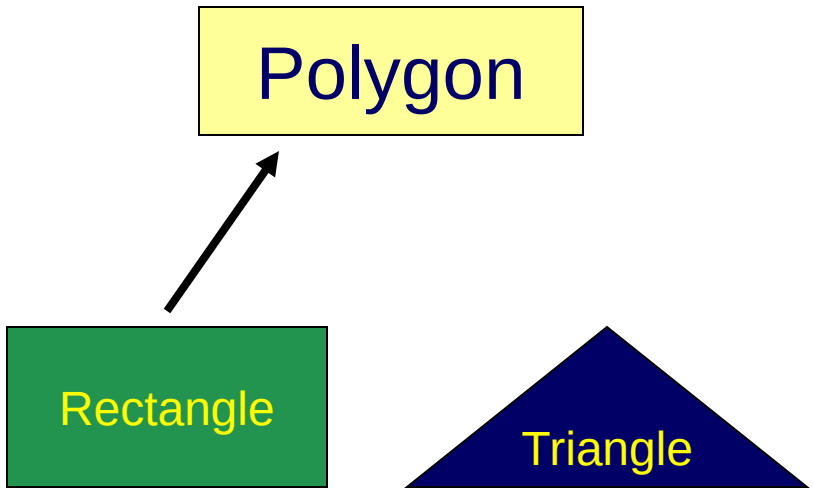



```
class Polygon
{
        private:
           int width, length;
        public:
           void set(int w, int l);
}
```

```
class Rectangle {
        private:
           int width, length;
        public:
           void set(int w, int l);
           int area();
}
```

```
class Triangle {
        private:
           int width, length;
        public:
           void set(int w, int l);
           int area();
}
```

VD dẫn nhập khái niệm kế thừa *(t.t)*



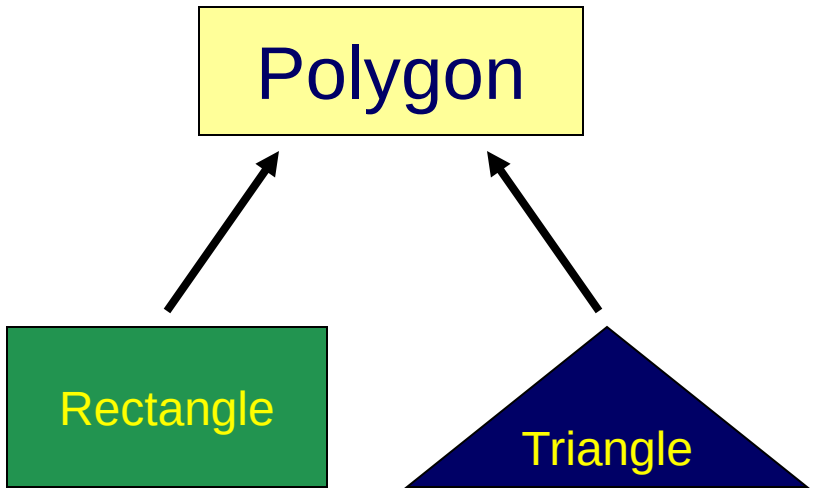

```
class Polygon
{
    protected:
      int width, length;
    public:
      void set(int w, int l);
}
```

```
class Rectangle : public Polygon
{
    public: int area();
}
```

⟺

```
class Rectangle{
    protected:
      int width, length;
    public:
      void set(int w, int l);
      int area();
}
```

VD dẫn nhập khái niệm kế thừa *(t.t)*



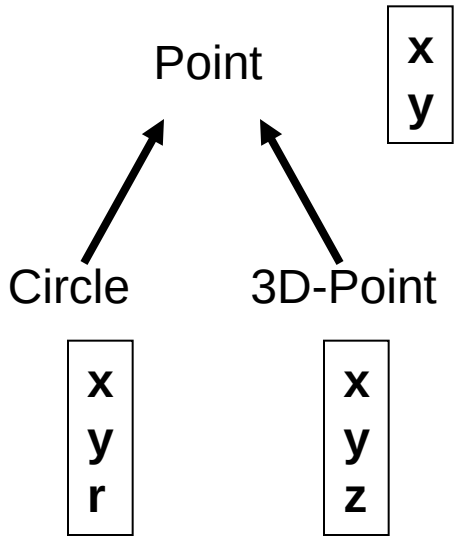

```
class Polygon
{
    protected:
      int width, length;
    public:
      void set(int w, int l);
}
```

```
class Triangle : public Polygon
{
    public: int area();
}
```

⟺

```
class Triangle{
    protected:
      int width, length;
    public:
      void set(int w, int l);
      int area();
}
```

VD dẫn nhập khái niệm kế thừa *(t.t)*




```
class Point
{
    protected:
       int x, y;
    public:
       void set(int a, int b);
}
```

```
class Circle : public Point
{
    private:
          double r;
}
```

```
class 3D-Point: public Point
{
    private:
          int z;
}
```

Kế thừa

- Sự kế thừa là một đặc điểm của ngôn ngữ dùng để biểu diễn mối quan hệ đặc biệt giữa các lớp.
- Sự kế thừa là một mức cao hơn của trừu tượng hóa:

  ➔ *cung cấp một cơ chế gom chung các lớp có liên quan với nhau thành một mức khái quát hóa đặc trưng cho toàn bộ các lớp nói trên.*

Kế thừa *(t.t)*

- Các lớp với các đặc điểm tương tự nhau có thể được tổ chức thành một sơ đồ phân cấp kế thừa.
- Lớp ở trên cùng là trừu tượng hóa của toàn bộ các lớp ở bên dưới nó.
- Quan hệ ***"là 1"***: Kế thừa được sử dụng thông dụng nhất để biểu diễn quan hệ "là 1".
  - Một sinh viên là một người
  - Một hình tròn là một hình ellipse
  - Một tam giác là một đa giác

Lợi ích kế thừa

- Tạo khả năng xây dựng lớp mới từ lớp đã có.
- Kế thừa có khả năng tạo cơ chế khái quát hoá và chuyên biệt hoá từ một lớp đã có.
- Kế thừa cho phép tổ chức các lớp chia sẻ mã chương trình chung nhờ vậy có thể dễ dàng sửa chữa, nâng cấp hệ thống.

Cách dùng kế thừa

- Cách 1: Để phản ánh mối quan hệ giữa các lớp
  - Là công cụ để tổ chức và phân cấp lớp dựa vào sự chuyên biệt hóa
  - Một vài hàm thành phần của lớp con là phiên bản hoàn thiện hoặc đặc biệt hoá của phiên bản ở lớp cha
  - Trong C++ mối quan hệ này thường được cài đặt sử dụng:
    - *Kế thừa public.*
    - *Hàm thành phần là phương thức ảo*

Cách dùng kế thừa (tt)

- Cách 2: Để phản ánh sự chia sẻ mã chương trình giữa các lớp
  - Không có quan hệ về mặt ngữ nghĩa nhưng có thể có tổ chức dữ liệu và mã chương trình tương tự nhau.
  - Trong C++, cơ chế chia sẻ mã này thường được cài đặt dùng:
    - *Kế thừa private.*
    - *Hàm thành phần không là phương thức ảo.*

Đặc tính Kế thừa

- Thừa kế cho phép ta định nghĩa 1 lớp mới, gọi là lớp con (subclass) hay lớp dẫn xuất (derived class) từ một lớp đã có, gọi là lớp cha (superclass) hay lớp cơ sở (base class).
- Lớp dẫn xuất:
  - Thừa kế các thành phần *(dữ liệu, hàm)* của lớp cơ sở.
  - Thêm vào các thành phần mới, bao hàm cả việc làm "tốt hơn" hoặc làm lại những công việc mà trong lớp cơ sở chưa làm tốt hoặc không còn phù hợp với lớp dẫn xuất.

Đặc tính Kế thừa *(t.t)*

- Thừa kế cho phép nhiều lớp có thể dẫn xuất từ một lớp cơ sở.
- Thừa kế cũng cho phép một lớp có thể là dẫn xuất của nhiều lớp cơ sở.
- Thừa kế không chỉ giới hạn ở một mức: Một lớp dẫn xuất có thể là lớp cơ sở cho các lớp dẫn xuất khác.

Cú pháp khai báo kế thừa

❖ Cú pháp:

```
class DerivedClassName : access-level BaseClassName
```

- access-level - kiểu truy cập
  - protected
  - public
  - private

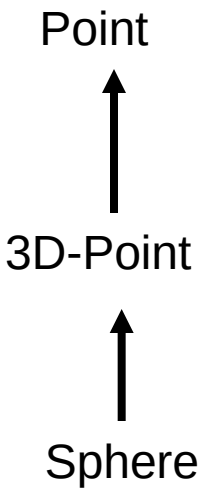
Cú pháp khai báo kế thừa (tt)

❖ Cú pháp:

class *DerivedClassName* : access-level *BaseClassName1*,
access-level *BaseClassName2*,
……

- access-level - kiểu truy cập
  - protected
  - public
  - private

Ví dụ: Khai báo lớp cơ sở và lớp dẫn xuất




```
class Point{
    protected:
      int x, y;
    public:
      void set(int a, int b);
}
```

```
class 3D-Point : public
    Point{
    private:  double z;
    … …
}
```

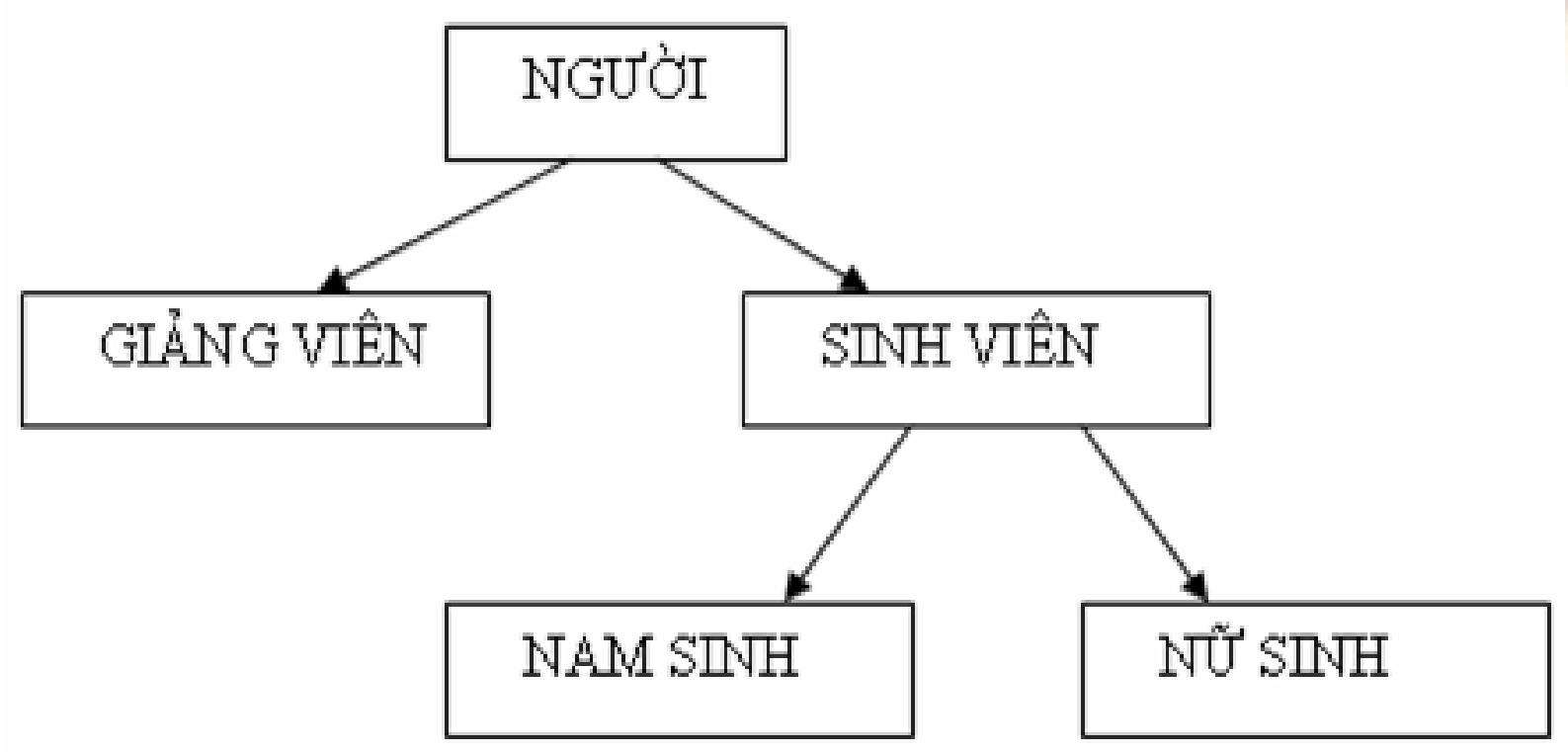
```
class Sphere : public 3D-
    Point{
    private:  double r;
    … …
}
```

Point là lớp cơ sở của lớp 3D-Point; và 3D-Point là lớp cơ sở của lớp Sphere

Ví dụ: Kế thừa đơn

- Xét hai khái niệm ***người*** và ***sinh viên*** với mối quan hệ tự nhiên: *một 'sinh viên' là một 'người'*.
- Trong C++, ta có thể biểu diễn khái niệm trên, một sinh viên là một người có thêm một số thông tin và một số thao tác riêng biệt của sinh viên.
- Ta tổ chức lớp sinh viên kế thừa từ lớp người:
  - Lớp người được gọi là lớp cha *(superclass)* hay lớp cơ sở *(base class)*
  - Lớp sinh viên được gọi là lớp con *(subclass)* hay lớp dẫn xuất *(derived class)*.

Ví dụ: Kế thừa đơn *(t.t)*



Ví dụ: Kế thừa đơn *(t.t)*

```
class Nguoi {
   char *HoTen;
   int NamSinh;
public:
   Nguoi();
   Nguoi(char *ht, int ns):NamSinh(ns)
   {HoTen=strdup(ht);}
   ~Nguoi() {delete [] HoTen;}
   void An() const { cout<<HoTen<<" an 3 chen com";}
   void Ngu() const { cout<<HoTen<< " ngu ngay 8
   tieng";}
   void Xuat() const;
   friend ostream& operator << (ostream &os, Nguoi&
   p);
};
```

Ví dụ: Kế thừa đơn *(t.t)*

```
class SinhVien : public Nguoi {
   char *MaSo;
public:
   Sinhvien();
   SinhVien(char *ht, char *ms, int ns) :
   Nguoi(ht,ns) {  MaSo = strdup(ms);
   }
   ~SinhVien() {delete [] MaSo;}
   void Xuat() const;
};
```

Kế thừa đơn – ví dụ *(t.t)*

```
class SinhVien : public Nguoi {
//...
};
```

- Lớp con sẽ tự động kế thừa các đặc tính của lớp cha. Khi đó sinh viên được thừa hưởng các đặc tính của lớp người như sau:
  - *Về mặt dữ liệu: Mỗi đối tượng sinh viên tự động có thành phần dữ liệu họ tên và năm sinh của người.*
  - *Về mặt thao tác: Lớp sinh viên được tự động kế thừa các thao tác của lớp cha là phương thức An(), Ngu()*
  - *Riêng phương thức thiết lập không được kế thừa*

Kế thừa đơn – ví dụ *(t.t)*

```
Nguoi p1;
SinhVien s1;
p1.An(); cout << "\n";
s1.An(); cout << "\n";     // Tu lop Nguoi
p1.Xuat(); cout << "\n";
```

- Kế thừa public như trên hàm ý rằng ***một đối tượng sinh viên là một đối tượng người.***

Định nghĩa lại thao tác ở lớp con

- Ta có thể định nghĩa lại các đặc tính ở lớp con đã có ở lớp cha. Việc định nghĩa chủ yếu là thao tác, bằng cách khai báo giống hệt như ở lớp cha.

```
class SinhVien : public Nguoi {
    char *MaSo;
public:
    //...
    void Xuat();
};
void SinhVien::Xuat() {
   cout << "Sinh vien, ma so: " << MaSo << ",
                        ho ten: " << HoTen;
}
```

Định nghĩa lại thao tác ở lớp con *(t.t)*

- Việc định nghĩa lại thao tác ở lớp con được thực hiện khi thao tác ở lớp con khác thao tác ở lớp cha. Thông thường là các thao tác xuất, nhập.
- Ta cũng có thể định nghĩa lại thao tác ở lớp con trong trường hợp giải thuật ở lớp con đơn giản hơn (tô màu đa giác, tính diện tích...).

Ràng buộc ngữ nghĩa ở lớp con

- Kế thừa có thể được áp dụng cho quan hệ kế thừa mang ý nghĩa ràng buộc, đối tượng ở lớp con là đối tượng ở lớp cha nhưng có **dữ liệu bị ràng buộc**
  - Hình tròn là Ellipse ràng buộc bán kính ngang dọc bằng nhau.
  - Số ảo là số phức ràng buộc phần thực bằng 0.
  - Hình vuông là hình chữ nhật ràng buộc hai cạnh ngang và dọc bằng nhau…
- Trong trường hợp này, các hàm thành phần phải bảo đảm sự ràng buộc dữ liệu được tôn trọng

Phạm vi truy xuất

- ***Truy xuất theo chiều dọc***: Hàm thành phần của lớp con có thể có quyền truy xuất đến các thành phần riêng tư của lớp cha. Vì chiều truy xuất là từ lớp con, cháu lên lớp cha nên ta gọi là truy xuất theo chiều dọc
- ***Truy xuất theo chiều ngang***: Các thành phần của lớp cha, sau khi kế thừa xuống lớp con, thì thế giới bên ngoài có quyền truy xuất thông qua đối tượng của lớp con hay không? Trong trường hợp này, ta gọi là truy xuất theo chiều ngang.

Phạm vi truy xuất (tt)

- **Thuộc tính public:** Thành phần nào có thuộc tính public thì có thể được truy xuất từ bất cứ nơi nào (từ sau khai báo lớp).
- **Thuộc tính private:** Thành phần nào có thuộc tính private thì nó là riêng tư của lớp đó. Chỉ có các hàm thành phần của lớp và ngoại lệ là các hàm bạn được phép truy xuất, ngay cả các lớp con cũng không có quyền truy xuất.
- **Thuộc tính protected:** cho phép qui định một vài thành phần nào đó của lớp là bảo mật, theo nghĩa thế giới bên ngoài không được phép truy xuất, nhưng tất cả các lớp con, cháu… đều được phép truy xuất

Ví dụ

```
class Nguoi {
protected:
   char *HoTen;
   int NamSinh;
public:
//...
};
```

Ví dụ *(t.t)*

```
class SinhVien : public Nguoi {
protected:
   char *MaSo;
public:
   SinhVien(char *ht, char *ms, int ns) : Nguoi(ht,ns) {
        MaSo = strdup(ms);
   }
   ~SinhVien() {delete [] MaSo;}
   void Xuat() const;          // Co the truy xuat
                        // Nguoi::HoTen va Nguoi::NamSinh
};

class NuSinh : public SinhVien {
public:
   NuSinh(char *ht, char *ms, int ns) : SinhVien(ht,ms,ns) {}
   void An() const {
        cout << HoTen << " ma so " << MaSo << " an 2 to pho";
   }
};      // Co the truy xuat Nguoi::HoTen va
        // Nguoi::NamSinh va SinhVien::MaSo
```

Ví dụ *(t.t)*

```
void Nguoi::Xuat() const {
   cout << "Nguoi, ho ten: " << HoTen << " sinh " <<
   NamSinh;
}

void SinhVien::Xuat() const {
   cout << "Sinh vien, ma so: " << MaSo << ", ho ten:
   " << HoTen;      // Ok: co quyen truy xuat
                    // Nguoi::HoTen, Nguoi::NamSinh
}
```

Phạm vi truy xuất *(t.t)*

- *Kế thừa public:* Lớp con kế thừa public từ lớp cha thì:
  - Các thành phần **protected** của lớp cha trở thành **protected** của lớp con
  - Các thành phần **public** của lớp cha trở thành **public** của lớp con.
  - Nói cách khác mọi thao tác của lớp cha được kế thừa xuống lớp con.
  - Vì vậy ta có thể sử dụng thao tác của lớp cha cho đối tượng thuộc lớp con.

Phạm vi truy xuất *(t.t)*

- Kế thừa public *(t.t)*
  - Do được thừa hưởng các đặc tính của lớp cha nên ta dùng kế thừa public khi và chỉ khi có quan hệ là một từ lớp con đến lớp cha.
  - Hầu hết các trường hợp kế thừa là kế thừa public, nó cho phép tận dụng lại mã chương trình, đồng thời tạo khả năng thu gom các đặc điểm chung của các lớp vào một lớp cơ sở, nhờ đó dễ dàng nâng cấp và sửa chữa bảo trì.

Phạm vi truy xuất *(t.t)*

- *Kế thừa private:* Lớp con kế thừa private từ lớp cha thì:
  - Các thành phần ***protected*** và ***public*** của lớp cha trở thành **private** của lớp con.
  - Nói cách khác mọi thao tác của lớp cha đều bị lớp con che dấu.
  - Sử dụng kế thừa private ta có thể chia sẻ mã chương trình giữa các lớp có cấu trúc dữ liệu tương tự nhau nhưng vẫn giữ được tinh thần của từng lớp.

Phạm vi truy xuất *(t.t)*

- *Kế thừa protected*
  - Lớp con và friend có thể truy cập các thành viên protected của lớp cơ sở.

Phạm vi truy nhập đến các thành phần của lớp cơ sở

- Các thành phần của lớp cơ sở trở nên như thế nào (private, protected, public) trong lớp dẫn xuất?
  - **kế thừa public**: Các thành phần public và protected của lớp cơ sở là các thành phần public và protected của lớp dẫn xuất.
  - **kế thừa protected**: Các thành phần public và protected của lớp cơ sở là các thành phần protected của lớp dẫn xuất.
  - **kế thừa private**: Các thành phần public và protected của lớp cơ sở là các thành phần private của lớp dẫn xuất.

Bảng tầm vực trong kế thừa của lớp dẫn xuất:

Tầm vực (khai báo trong lớp cơ sở)	Kế thừa public	Kế thừa protected	Kế thừa private
public	public	protected	private
protected	protected	protected	private
private	***Không thể truy xuất***	***Không thể truy xuất***	***Không thể truy xuất***

Phương thức thiết lập và huỷ bỏ

- Phương thức thiết lập và huỷ bỏ là các hàm thành phần đặc biệt dùng để tự động khởi động đối tượng khi nó được tạo ra và tự động dọn dẹp đối tượng khi nó bị hủy đi.
- Một đối tượng thuộc lớp con có chứa các thành phần dữ liệu của các lớp cơ sở.
  - Có thể xem lớp con có các thành phần ngầm định ứng với các lớp cơ sở.
  - Khi một đối tượng thuộc lớp con được tạo ra, các thành phần cơ sở cũng được tạo ra, nghĩa là phương thức thiết lập của các lớp cơ sở phải được gọi.

Phương thức thiết lập và huỷ bỏ

- Trình biên dịch tự động gọi phương thức thiết lập của các lớp cơ sở cho các đối tượng cơ sở nhúng vào đối tượng đang được tạo ra.
- Đối với phương thức thiết lập của một lớp con, công việc đầu tiên là gọi phương thức thiết lập của các lớp cơ sở.
- Nếu mọi phương thức thiết lập của lớp cơ sở đều đòi hỏi phải cung cấp tham số thì lớp con bắt buộc phải có phương thức thiết lập để cung cấp các tham số đó.

Ví dụ

```
class Diem {
    double x,y;
public:
    Diem(double x, double y):x(xx),y(yy){}
    //...
};

class HinhTron:public Diem
{
    double r;
public:
    void Ve(int color) const;
};

HinhTron t;      // SAI
```

Ví dụ *(t.t)*

```
class HinhTron:public Diem {
    double r;
public:
    HinhTron(double tx, double ty, double rr)
                    :Diem(tx,ty),r(rr){}
    void Ve(int color) const;
    void TinhTien(double dx, double dy) const;
};
HinhTron t(200,200,50);      // Dung
```

Ví dụ *(t.t)*

```
class Nguoi {
protected:
   char *HoTen;
   int NamSinh;
public:
   Nguoi(char *ht, int ns):NamSinh(ns) {
       HoTen = strdup(ht);
   }
//...
};

class SinhVien : public Nguoi {
   char *MaSo;
public:
   SinhVien(char *ht, char *ms, int ns) :
   Nguoi(ht,ns) {  MaSo = strdup(ms);
   }
   void Xuat() const;
};
```

Hủy đối tượng

- Khi một đối tượng bị huỷ đi, phương thức huỷ bỏ của nó sẽ được gọi, sau đó phương thức huỷ bỏ của các lớp cơ sở sẽ được gọi một cách tự động. Vì vậy lớp con không cần và cũng không được thực hiện các thao tác dọn dẹp cho các thành phần thuộc lớp cha.