CHƯƠNG 3

SỰ THỪA KẾ

Một trong các đặc trưng quan trọng nhất của C + + và các ngôn ngữ lập trình định hướng đối tượng khác là cho phép chúng ta có thể sử dụng lại các thành phần mềm. Trong mục 2.4 chúng ta đã trình bày một phương pháp thực hiện sử dụng lại các thành phần mềm bằng cách xây dựng các lớp khuôn. Chương này sẽ trình bày một phương pháp khác: sử dụng lại các thành phần mềm thông qua tính **thừa kế (inheritance**). Sử dụng tính thừa kế, chúng ta có thể xây dựng nên các lớp mới từ các lớp đã có, tránh phải viết lại các thành phần mềm đã có.

3.1 CÁC LỚP DẪN XUẤT

Khi xây dựng một lớp mới, trong nhiều trường hợp lớp mới cần xây dựng có nhiều điểm giống một lớp đã có. Khi đó trên cơ sở lớp đã có, bằng cách sử dụng tính thừa kế, chúng ta có thể xây dựng nên lớp mới. Lớp đã có được gọi là **lớp cơ sở (base class**), lớp mới được xây dựng nên từ lớp cơ sở được gọi là **lớp dẫn xuất (derived class**). Một lớp dẫn xuất có thể được thừa kế từ nhiều lớp cơ sở, điều này được gọi là **tính đa thừa kế** (**multiple inheritance**). Song để đơn giản cho trình bày, sau đây chúng ta chỉ đề cập tới sự thiết kế lớp dẫn xuất thừa kế từ một lớp cơ sở.

Tính thừa kế cho phép ta sử dụng lại các thành phần mềm khi chúng ta xây dựng một lớp mới. Lớp dẫn xuất có thể thừa kế các thành phần dữ liệu và các hàm thành phần từ lớp cơ sở, trừ các hàm kiến tạo và hàm huỷ. Lớp dẫn xuất có thể thêm vào các thành phần dữ liệu mới và các hàm thành phần mới cần thiết cho các phép toán của nó. Ngoài ra, lớp dẫn xuất còn có

thể xác định lại bất kỳ hàm thành phần nào của lớp cơ sở cho phù hợp với các đặc điểm của lớp dẫn xuất.

Cú pháp xác định một lớp dẫn xuất như sau: Đầu lớp bắt đầu bởi từ khoá class, sau đó là tên lớp dẫn xuất, rồi đến dấu hai chấm, theo sau là từ khoá chỉ định dạng thừa kế (public, private, protected), và cuối cùng là tên lớp cơ sở. Chẳng hạn, nếu ta muốn xác định một lớp dẫn xuất D từ lớp cơ sở B thì có thể sử dụng một trong ba khai báo sau:

```
class D : public B { … } ;
class D : private B { … } ;
class D : protected B { … } ;
```

Chúng ta sẽ nói tới đặc điểm của các dạng thừa kế ở cuối mục này, còn bây giờ chúng ta sẽ xét một ví dụ minh hoạ. Giả sử chúng ta muốn xây dựng lớp Ball (lớp quả bóng) từ lớp Sphere (lớp hình cầu). Giả sử lớp hình cầu được xác định như sau:

```
class    Sphere
{
 public :
    Sphere (double R = 1) ;
    double   Radius ( ) const ;
    double   Area ( ) const ;
    double   Volume ( ) const ;
    void   WhatIsIt ( ) const ;
 private :
    double  radius ;
}
```

Lớp Sphere chỉ có một thành phần dữ liệu radius là bán kính của hình cầu, và các hàm thành phần: hàm kiến tạo ra hình cầu có bán kính R, hàm

cho biết bán kính hình cầu Radius ( ), hàm tính diện tích hình cầu Area ( ) và hàm tính thể tích hình cầu Volume ( ), cuối cùng là hàm WhatIsIt ( ) cho ta câu trả lời rằng đối tượng được hỏi là hình cầu có bán kính là bao nhiêu. Hàm WhatIsIt ( ) được cài đặt như sau:

```
void   Sphere :: WhatIsIt ( )   const
{
  cout << "It is the sphere with the radius"
       << radius ;
}
```

Bởi vì mỗi quả bóng là một hình cầu, nên chúng ta có thể xây dựng lớp Ball như là lớp dẫn xuất từ lớp Sphere. Lớp Ball thừa kế tất cả các thành phần của lớp Sphere, trừ ra hàm kiến tạo và xác định lại hàm WhatIsIt ( ). Chúng ta sẽ thêm vào lớp Ball một thành phần dữ liệu mới madeof để chỉ quả bóng được làm bằng chất liệu gì: cao su, nhựa hay gỗ. Một hàm thành phần mới cũng được thêm vào lớp Ball, đó là hàm Madeof ( ) trả về chất liệu tạo thành quả bóng. Lớp Ball được khai báo như sau:

```
class  Ball : public  Sphere
{
 public :
    enum Materials {RUBBER, PLASTIC, WOOD};
    Ball (double R = 1, Materials M = RUBBER) ;
    Materials   MadeOf ( ) const ;
    void   WhatIsIt ( )   const ;
 private :
    Materials   madeOf ;
} ;
```

Lớp Ball được định nghĩa như trên sẽ có hai thành phần dữ liệu: radius được thừa kế từ lớp Sphere và madeof mới được đưa vào. Ngoài hàm kiến tạo, lớp Ball có ba hàm thành phần được thừa kế từ lớp Sphere, đó là các hàm Radius ( ), Area ( ) và Volume( ), một hàm thành phần mới là hàm MadeOf( ), và hàm thành phầnWhatIsIt( ) mới, nó định nghĩa lại một hàm cùng tên đã có trong lớp cơ sở Sphere. Hàm WhatIsIt( ) trong lớp Ball được định nghĩa như sau:

```
void   Ball :: WhatIsIt ( )   const
{
  Sphere :: WhatIsIt( ) ;
  cout << " and made of " << madeof ;
}
```

Hàm kiến tạo của lớp dẫn xuất. Nếu chúng ta không cung cấp cho lớp dẫn xuất hàm kiến tạo, thì chương trình dịch sẽ tự động cung cấp hàm kiến tạo mặc định tự động. Nhưng cũng như đối với một lớp bất kỳ, khi thiết kế một lớp dẫn xuất, nói chung chúng ta cần phải cung cấp cho lớp dẫn xuất hàm kiến tạo. Bây giờ chúng ta xét xem hàm kiến tạo của lớp dẫn xuất được cài đặt như thế nào? Chú ý rằng, lớp dẫn xuất chứa các thành phần dữ liệu được thừa kế từ lớp cơ sở, ngoài ra nó còn chứa các thành phần dữ liệu mới, trong các thành phần dữ liệu mới này có thể có thành phần dữ liệu là đối tượng của một lớp khác. Nhưng trong lớp dẫn xuất, chúng ta không được quyền truy cập trực tiếp đến các thành phần dữ liệu của lớp cơ sở (và các thành phần dữ liệu của lớp khác). Vậy làm thế nào để khởi tạo các thành phần dữ liệu của lớp cơ sở (và các thành phần dữ liệu của lớp khác). Vấn đề này được giải quyết bằng cách cung cấp một **danh sách khởi tạo (initiazation list)**. Danh sách khởi tạo là danh sách các lời gọi hàm kiến tạo của lớp cơ sở (và các hàm kiến tạo của các lớp khác). Danh sách này được đặt ngay sau đầu hàm kiến tạo của lớp dẫn xuất. Ví dụ, hàm kiến tạo của lớp dẫn xuất Ball được cài đặt như sau:

```
Ball :: Ball (double R, Materials M)
: Sphere (R)
{ madeof = M; }
```

Lưu ý rằng, ngay trước danh sách khởi tạo phải có dấu hai chấm :, trong ví dụ trên danh sách khởi tạo chỉ có một lời gọi hàm kiến tạo lớp cơ sở Sphere (R), nếu có nhiều lời gọi hàm thì cần có dấu phẩy giữa các lời gọi hàm.

Các mục public, private và protected của một lớp

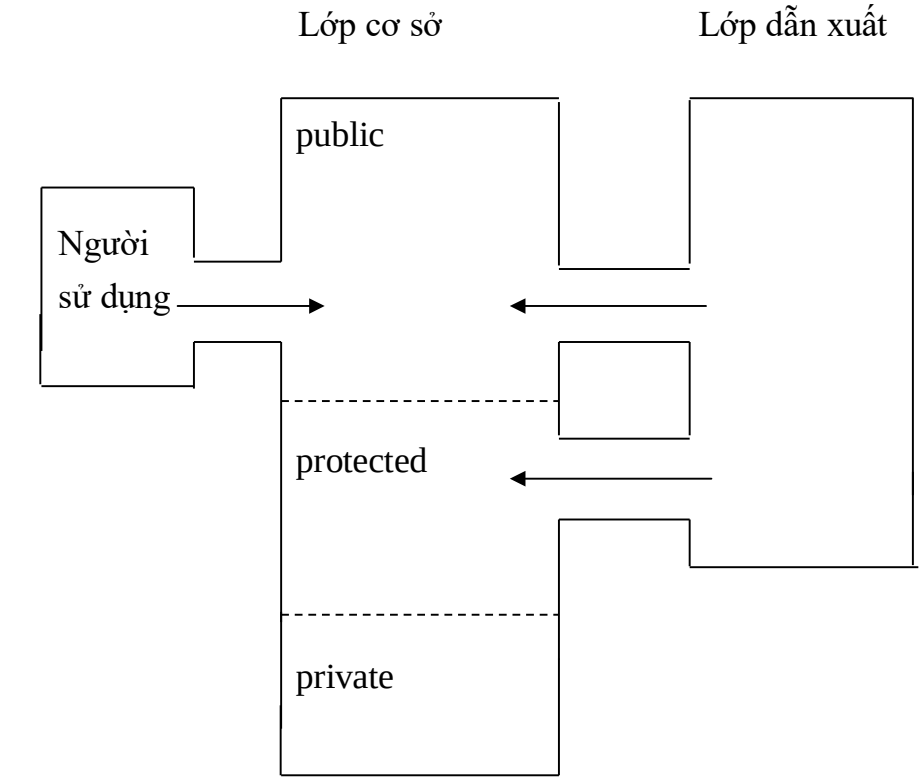
Trong các ví dụ mà chúng ta đưa ra từ trước tới nay, các thành phần của lớp được đưa vào hai mục: public và private. Các thành phần nằm trong mục public là các thành phần công khai, khách hàng của lớp có thể sử dụng trực tiếp các thành phần này. Các thành phần nằm trong mục private là các thành phần cá nhân của lớp, chỉ được phép sử dụng trong phạm vi lớp. Song khi chúng ta thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta mong muốn rằng một số thành phần của lớp, khách hàng không được quyền sử dụng, nhưng cho phép các lớp dẫn xuất được quyền sử dụng. Muốn vậy chúng ta đưa các thành phần đó vào mục protected. Như vậy các thành phần nằm trong mục protected là các thành phần được bảo vệ đối với khách hàng, nhưng các lớp dẫn xuất được quyền truy cập. Hình 3.1 minh hoạ quyền truy cập đến các mục public, protected và private của một lớp. Đến đây chúng ta có thể đưa ra cấu trúc tổng quát của một định nghĩa lớp:

```
class  tên_lớp
{
 public:
        danh sách các thành phần công khai
 protected :
        danh sách các thành phần được bảo vệ
```

private :

danh sách các thành phần cá nhân

};



Hình 3.1. Quyền truy cập đến các thành phần của lớp

Các dạng thừa kế public, private và protected

Khi xây dựng một lớp dẫn xuất từ một lớp cơ sở, chúng ta có thể sử dụng một trong ba dạng thừa kế: public, private hay protected. Tức là, định nghĩa một lớp dẫn xuất được bắt đầu bởi:

class tên_lớp_dẫn_xuất : dạng_thừa_kế tên_lớp_cơ_sở

Trong đó, dạng_thừa_kế là một trong ba từ khoá chỉ định dạng thừa kế: public, private, protected. Dù dạng thừa kế là gì (là public hoặc private hoặc protected) thì lớp dẫn xuất đều có quyền truy cập đến các thành phần trong mục public và protected của lớp cơ sở. Như chúng ta đã nhấn mạnh, lớp dẫn xuất được thừa kế các thành phần của lớp cơ sở, trừ ra các hàm kiến tạo, hàm huỷ và các hàm được định nghĩa lại. Vấn đề được đặt ra là, quyền truy cập đến các thành phần được thừa kế của lớp dẫn xuất như thế nào? Câu trả lời phụ thuộc vào dạng thừa kế. Sau đây là các quy tắc quy định quyền truy cập đến các thành phần được thừa kế của lớp dẫn xuất.

1. Thừa kế public: các thành phần public và protected của lớp cơ sở trở thành các thành phần public và protected tương ứng của lớp dẫn xuất.
2. Thừa kế protected: các thành phần public và protected của lớp cơ sở trở thành các thành phần protected của lớp xuất.
3. Thừa kế private: các thành phần public và protected của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất.
4. Trong bất kỳ trường hợp nào, các thành phần private của lớp cơ sở, lớp dẫn xuất đều không có quyền truy cập tới mặc dầu nó được thừa kế.

Trong ba dạng thừa kế đã nêu, thì thừa kế public là quan trọng nhất. Nó được sử dụng để mở rộng một định nghĩa lớp, tức là để cài đặt một lớp mới (lớp dẫn xuất) có đầy đủ các tính chất của lớp cơ sở và được bổ sung thêm các tính chất mới. Thừa kế private được sử dụng để cài đặt một lớp mới bằng các phương tiện của lớp cơ sở. Thừa kế protected ít được sử dụng.

Sự tương thích kiểu.

Khi lớp cơ sở là lớp cơ sở public, thì một lớp dẫn xuất có thể xem như lớp con của lớp cơ sở theo nghĩa thuyết tập, tức là mỗi đối tượng của lớp dẫn xuất là một đối tượng của lớp cơ sở, song điều ngược lại thì không đúng. Do đó, khi ta khai báo một con trỏ trỏ tới đối tượng của lớp cơ sở, thì trong thời

gian chạy con trỏ này có thể trỏ tới đối tượng của lớp dẫn xuất. Ví dụ, giả sử chúng ta có các khai báo sau:

```
class   Polygon { … };
class   Rectangle : public Polygon { … };
```

Chú ý rằng, mỗi lớp là một kiểu dữ liệu. Do đó, ta có thể khai báo các biến sau:

```
Polygon   *P ;
Polygon    Pobj;
Rectangle  *R;
Rectangle   Robj;
```

Khi đó:

```
P = new Polygon( ); // hợp lệ
P = new Rectangle ( ); // hợp lệ
R = new Polygon( ); // không hợp lệ
R = new Rectangle( ); // hợp lệ
P = R; // hợp lệ
R = P; // không hợp lệ
```

Chúng ta cũng có thể gán đối tượng lớp dẫn xuất cho đối tượng lớp cơ sở, song ngược lại thì không được phép, chẳng hạn:

```
Pobj = Robj; // hợp lệ
Robj = Pobj; // không hợp lệ
```

3.2 HÀM ẢO VÀ TÍNH ĐA HÌNH

Tính đa hình (polymorphism) để chỉ một hàm khai báo trong một lớp cơ sở có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, tức là hàm có nội dung khác nhau trong các lớp dẫn xuất. Để hiểu được tính đa hình, chúng ta hãy xét một ví dụ đơn giản. Giả sử chúng ta có một lớp cơ sở:

```
class    Alpha
{
   public :
          ……
      void   Hello( )   const
      { cout << “I am Alpha” << endl ; }
          ……
} ;
```

Lớp Alpha chứa hàm Hello( ), hàm này in ra thông báo “I am Alpha”. Chúng ta xây dựng lớp Beta dẫn xuất từ lớp Alpha, lớp Beta cũng chứa hàm Hello( ) nhưng với nội dung khác: nó in ra thông báo “I am Beta”.

```
class   Beta : public   Alpha
{
  public :
          …….
     void   Hello( )   const
     { cout << “I am Beta” << endl ; }
          …….
} ;
```

Bây giờ, ta khai báo

Alpha Obj;

Khi đó, lời gọi hàm Obj.Hello( ) sẽ cho in ra “I am Alpha”, tức là bản hàm Hello( ) trong lớp Alpha được thực hiện. Còn nếu ta khai báo

Beta   Obj ;

thì lời gọi hàm obj.Hello( ) sẽ sử dụng bản hàm trong lớp Beta và sẽ cho in ra “I am Beta”. Như vậy, bản hàm nào của hàm Hello( ) được sử dụng khi thực hiện một lời gọi hàm được quyết định bởi kiểu đã khai báo của đối tượng, tức là được quyết định trong thời gian dịch. Hoàn cảnh này được gọi

là **sự ràng buộc tĩnh (static binding)**. Song sự ràng buộc tĩnh không đáp ứng được mong muốn của chúng ta trong tình huống sau đây:

Giả sử, Aptr là con trỏ trỏ tới đối tượng lớp Alpha:

Alpha *Aptr ;

Kiểu của con trỏ Aptr khi khai báo là kiểu tĩnh. Trong thời gian chạy, con trỏ Aptr có thể trỏ tới một đối tượng của lớp dẫn xuất. Kiểu của con trỏ Aptr lúc đó là kiểu động. Chẳng hạn, khi gặp các dòng lệnh:

Aptr = new Beta ;

Aptr →Hello( );

Aptr trỏ tới đối tượng của lớp Beta. Do đó, chúng ta mong muốn rằng, lời gọi hàm Aptr →Hello( ) sẽ cho in ra "I am Beta". Song đáng tiếc là không phải như vậy, kiểu tĩnh của con trỏ Aptr đã quyết định bản hàm Hello( ) trong lớp Alpha được thực hiện và cho in ra "I am Alpha".

Chúng ta có thể khắc phục được sự bất cập trên bằng cách khai báo hàm Hello( ) trong lớp cơ sở Alpha là **hàm ảo (virtual function)**. Để chỉ một hàm là ảo, chúng ta chỉ cần viết từ khoá virtual trước khai báo hàm trong định nghĩa lớp cơ sở. Chẳng hạn, lớp Alpha được khai báo lại như sau:

```
class   Alpha
{
  public :
        ……
        virtual  void  Hello( )  const
        { cout << "I am Alpha << endl; }
        ……
};
```

Khi một hàm được khai báo là ảo trong lớp cơ sở, như hàm Hello( ) trong lớp Alpha, thì nó có thể được định nghĩa lại với các nội dung mới trong các lớp dẫn xuất. Chẳng hạn, trong lớp Beta:

```
class   Beta : public   Alpha
```

```
{
  public :
        ……..
        virtual   void   Hello( )   const
        { cout << “I am Beta” << endl; }
        ……..
};
```

Chúng ta xây dựng một lớp dẫn xuất khác Gama từ lớp cơ sở Alpha. Trong lớp Gama, hàm Hello( ) cũng được định nghĩa lại:

```
class   Gama : public   Alpha
{
  public :
        ……..
        virtual   void   Hello( )   const
        { cout << “I am Gama” << endl; }
        ……..
};
```

Như vậy, hàm ảo Hello( ) có ba dạng khác nhau. Chúng ta thử xem bản hàm nào được sử dụng trong chương trình sau:

```
int   main( )
{
  Alpha   Aobj;
  Beta   Bobj;
  Alpha   *Aptr;
  Aptr = &Aobj; // con trỏ Aptr trỏ tới đối tượng lớp Alpha.
  Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Alpha được sử dụng.
  Aptr = &Bobj; // con trỏ Aptr trỏ tới đối tượng lớp Beta.
```

```
    Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Beta được sử dụng.
    Aptr = new Gama( ); // con trỏ Aptr trỏ tới đối tượng lớp Gama.
    Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Gama được sử dụng.
    return   0;
}
```

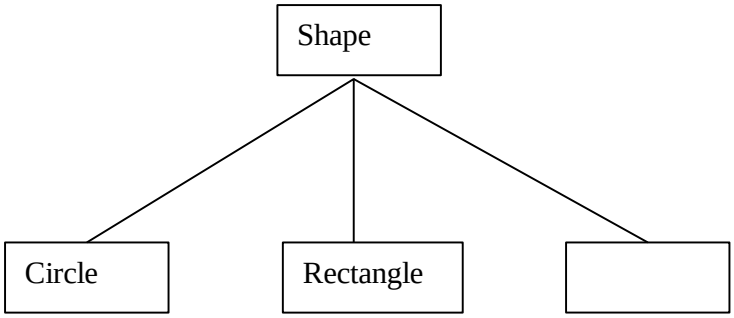
Tóm lại, một hàm được khai báo là ảo trong một lớp cơ sở, nó có thể được định nghĩa lại trong các lớp dẫn xuất và do đó nó có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, chẳng hạn hàm Hello( ) có ba bản hàm khác nhau. Bản hàm nào được sử dụng trong lời gọi hàm (chẳng hạn, Aptr → Hello( )) được quyết định bởi kiểu của đối tượng mà con trỏ lớp cơ sở trỏ tới, tức là được xác định trong thời gian chạy. Điều này được gọi là **sự ràng buộc động (dynamic binding)**. Như vậy, một hàm được khai báo là ảo trong lớp cơ sở là hàm có tính đa hình, tức là hàm có nhiều dạng khác nhau. Dạng hàm thích hợp được lựa chọn để thực hiện phụ thuộc vào kiểu động của đối tượng kích hoạt hàm.

Khi thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta cần chú ý đến các luật tổng quát sau:

- Các hàm cần định nghĩa lại trong các lớp dẫn xuất cần phải là ảo.
- Hàm kiến tạo không thể là ảo, song hàm huỷ cần phải là ảo.

3.3 LỚP CƠ SỞ TRỪU TƯỢNG

Giả sử chúng ta cần thiết kế các lớp sau: lớp các hình tròn (Circle), lớp các hình chữ nhật (Rectangle), và nhiều lớp các hình phẳng có dạng đặc biệt khác. Trong các lớp đó chúng ta cần phải đưa vào các hàm thành phần thực hiện các hành động có đặc điểm chung cho tất cả các loại hình, chẳng hạn tính chu vi, tính diện tích, vẽ hình, … Trong tình huống này, chúng ta cần thiết kế một lớp, lớp các hình (Shape), làm cơ sở để dẫn xuất ra các lớp Circle, Rectangle, …, như được minh hoạ trong hình sau:



Lớp Shape sẽ chứa các khai báo các hàm thực hiện các xử lý có đặc điểm chung cho các lớp dẫn xuất, chẳng hạn các hàm tính chu vi Perimeter( ), hàm tính diện tích Area( ), … Song chúng ta không thể cài đặt được các hàm này trong lớp Shape (bởi vì chúng ta không thể tính được chu vi và diện tích của một hình trừu tượng), các hàm này sẽ được cài đặt trong các lớp dẫn xuất từ lớp Shape, chẳng hạn được cài đặt trong các lớp Circle, Rectangle, …Và do đó, trong lớp Shape chúng được khai báo là **hàm ảo thuần tuý (pure virtual function**) hay còn gọi là **hàm trừu tượng (abstract function**). Một hàm thành phần trong một lớp được gọi là ảo thuần tuý (hay trừu tượng) nếu nó chỉ được khai báo, nhưng không được định nghĩa trong lớp đó. Một hàm ảo thuần tuý được khai báo bằng cách khai báo nó là ảo và đặt = 0 ở sau mẫu hàm. Ví dụ, để khai báo các hàm Permeter( ) và Area( ) là ảo thuần tuý, ta viết:

```
virtual double Perimeter( ) const = 0;
virtual double Area( ) const = 0;
```

Một lớp chứa ít nhất một hàm ảo thuần tuý được gọi là **lớp cơ sở trừu tượng (abstract base class**). Lớp cơ sở trừu tượng không có đối tượng nào

cả, nó chỉ được sử dụng làm cơ sở để xây dựng các lớp khác. Lớp cơ sở trừu tượng chứa các hàm ảo thuần tuý biểu diễn các xử lý có đặc điểm chung cho các lớp đối tượng khác nhau. Các hàm đó sẽ được định nghĩa trong các lớp dẫn xuất. Và như vậy, các hàm ảo thuần tuý trong một lớp cơ sở trừu tượng là các hàm có tính đa hình. Sử dụng tính đa hình, người lập trình có thể viết các phần mềm dễ dàng hơn khi mở rộng, có tính khái quát cao, dễ đọc, dễ hiểu,… Mô hình thiết kế các lớp dẫn xuất từ lớp cơ sở trừu tượng là mô hình thiết kế mà chúng ta cần sử dụng trong các hoàn cảnh tương tự như khi thiết kế các lớp đối tượng hình học phẳng.

Để làm ví dụ minh hoạ cho khái niệm lớp cơ sở trừu tượng, chúng ta xây dựng lớp Shape. Một lớp cơ sở trừu tượng có thể chứa các thành phần dữ liệu là chung cho tất cả các lớp dẫn xuất. Chẳng hạn, lớp Shape chứa một biến thành phần private có tên là name để lưu tên các loại hình. Chúng ta đưa vào lớp Shape một hàm kiến tạo, nó không được gọi trực tiếp để khởi tạo ra đối tượng của lớp Shape (vì Shape là lớp trừu tượng, nên không có đối tượng nào cả), song nó sẽ được gọi để khởi tạo các đối tượng của các lớp dẫn xuất. Lớp Shape chứa hai hàm ảo thuần tuý: hàm Perimeter( ) và hàm Area( ). Ngoài các hàm ảo thuần tuý, lớp cơ sở trừu tượng còn có thể chứa các hàm ảo và các hàm thành phần khác. Chẳng hạn, lớp Shape chứa hàm operator <= để so sánh các hình theo diện tích và hàm ảo print để in ra một số thông tin về các hình. Định nghĩa lớp trừu tượng Shape được cho trong hình 3.2.

```
class   Shape
{
  public:
      Shape (const string & s = “  ”)
          : name (s) {  }
      ~ Shape( ) {  }
      virtual  double  Perimeter( )  const = 0;
```

```
        virtual  double  Area( )  const = 0;
        bool  operator <=(const Shape & sh)  const
        { return   Area( ) <=  sh. Area( ); }
        virtual  void  Print (ostream & out = cout)  const
        { out << name << "of area" << Area( ) ; }
    private:
        string name;
};
```

Hình 3.2. Lớp cơ sở trừu tượng Shape.

Chú ý rằng, hàm Print đã được cài đặt trong lớp Shape, nhưng nó được khai báo là ảo, mục đích là để bạn có thể cài đặt lại hàm này trong các lớp dẫn xuất để in ra các thông tin khác nếu bạn muốn. Sử dụng hàm Print, bạn có thể định nghĩa lại toán tử << để nó in ra các thông tin về các hình như sau:

```
ostream & operator << (ostream & out, const  Shape & sh)
{
    sh.Print (out);
    return   out;
}
```

Bây giờ dựa trên lớp cơ sở trừu tượng Shape, chúng ta sẽ xây dựng một loạt lớp dẫn xuất: lớp các hình có dạng đặc biệt, chẳng hạn các lớp Circle, Rectangle,… Lớp Rectangle được thiết kế như sau: ngoài thành phần dữ liệu name được thừa kế từ lớp Shape, nó chứa hai thành phần dữ liệu khác là length (chỉ chiều dài) và width (chỉ chiều rộng của hình chữ nhật). Lớp chứa hàm kiến tạo để khởi tạo nên hình chữ nhật có chiều dài là l, chiều rộng là w.

```
Rectangle ::  Rectangle (double l = 0.0, double w = 0.0)
: Shape (“rectangle”)
{
  length = l;
  width = r ;
}
```

Chú ý rằng, trong hàm kiến tạo trên, chúng ta đã gọi hàm kiến tạo của lớp Shape để đặt tên cho hình. Trong lớp Rectangle, chúng ta cần cung cấp định nghĩa cho các hàm Perimeter( ) và Area( ) được khai báo là trừu tượng trong lớp Shape. Chúng ta cũng định nghĩa lại hàm ảo Print để nó cho biết thêm một số thông tin khác về hình chữ nhật. Hàm Print được cài đặt như sau:

```
void   Rectangle :: Print (ostream & out = cout)  const
{
  Shape :: Print(out);
  out << “and of length” << length
      << “and of width” << width;
}
```

Lớp Circle được thiết kế một cách tương tự. Định nghĩa lớp Rectangle và lớp Circle được cho trong hình 3.3.

```
class   Rectangle : public  Shape
{
  public:
     Rectangle (double l = 0.0, double w = 0.0)
     : Shape (“rectangle”), length(l), width(w) { }
     double   GetLength( )   const
     { return   length; }
```

```
    double  GetWidth( )  const
    { return   width; }
    double   Perimeter( )  const
    {return   2*(length + width); }
    double   Area( )   const
    { return   length * width; }
    void   Print(ostream & out = cout)
    {
      // như đã trình bày ở trên.
    }
  private :
        double   length;
        double   width;
};

class   Circle : public  Shape
{
   public :
     const  double  PI = 3.14159;
     Circle(double r = 0.0)
     : Shape("circle"), radius (r) { }
     double  GetRadius( ) const
     { return   radius; }
     double   Perimeter( ) const
     { return   2 * radius * PI; }
     double   Area( ) const
     { return   radius * radius * PI; }
     void   Print (ostream & out = cout) const
     {
       Shape :: Print(out);
       out << "of radius" << radius;
```

```
    }
  private :
    double   radius ;
} ;
```

Hình 3.3. Các lớp Rectangle và Circle.

Tóm lại, trong một lớp các hàm ảo thuần tuý (hay các hàm trừu tượng) là các hàm chỉ được khai báo, nhưng không được định nghĩa. Một lớp chứa ít nhất một hàm ảo thuần tuý được gọi là lớp trừu tượng, nó không có đối tượng nào cả, nhưng được sử dụng làm cơ sở để xây dựng các lớp dẫn xuất. Nó cung cấp cho người sử dụng một dao diện chung cho tất cả các lớp dẫn xuất. Trong một lớp dẫn xuất từ lớp cơ sở trừu tượng, chúng ta có thể cung cấp định nghĩa cho các hàm trừu tượng của lớp cơ sở. Một hàm ảo thuần tuý cũng giống như hàm ảo thông thường ở đặc điểm là hàm có tính đa hình, bản hàm nào (trong số các bản hàm được cài đặt ở các lớp dẫn xuất) được thực hiện phụ thuộc vào đối tượng kích hoạt hàm thuộc lớp dẫn xuất nào, đối tượng đó được xác định trong thời gian chạy (sự ràng buộc động).

Mặc dù lớp cơ sở trừu tượng không có đối tượng nào cả, song chúng ta có thể khai báo một biến tham chiếu đến lớp cơ sở trừu tượng, chẳng hạn trong khai báo hàm sau:

```
ostream & operator << (ostream & out, const Shape & sh);
```

Chúng ta cũng có thể khai báo các con trỏ trỏ tới lớp cơ sở trừu tượng, chẳng hạn

```
Shape * aptr, * bptr;
aptr = new Rectangle (2.3, 5.4); // hợp lệ   bptr
= new Shape( “circle” ) ; // không hợp lệ
```

BÀI TẬP

1. Giả sử chúng ta xây dựng lớp Beta dẫn xuất từ lớp cơ sở Alpha; ngoài các thành phần dữ liệu được thừa kế từ lớp Alpha, lớp Beta còn chứa thành phần dữ liệu là đối tượng của một lớp khác: lớp Gama và các thành phần dữ liệu khác. Nếu ta không cung cấp cho lớp Beta hàm kiến tạo, hàm huỷ, toán tử gán thì C++ sẽ cung cấp hàm kiến tạo mặc định tự động, hàm kiến tạo copy tự động, hàm huỷ tự động và toán tử gán tự động cho lớp dẫn xuất Beta. Hãy cho biết hiệu quả các hàm tự động đó.

2. Giả sử ta xây dựng lớp dẫn xuất Beta như trong bài tập 1. Nói chung, chúng ta cần đưa vào lớp Beta hàm kiến tạo. Hàm kiến tạo của lớp dẫn xuất Beta thực hiện các nhiệm vụ gì? Nó cần được cài đặt như thế nào? Hãy đưa ra ví dụ minh hoạ.

3. Giả sử trong giao diện của lớp Alpha có chứa hàm foo( ) thực hiện một nhiệm vụ nào đó.

```
class Alpha
{
    public :
        void foo( ) ;
        …
 } ;
```

Giả sử ta xây dựng lớp dẫn xuất Beta từ lớp cơ sở Alpha với dạng thừa kế private :

```
class Beta : private Alpha
```

Chúng ta muốn hàm foo( ) là hàm public của lớp Beta. Để có điều đó, ta cần làm gì?

4. Cho các khai báo lớp như sau:

```
class      Alpha
{
  private :
     int   w ;
  protected :
     int   x ;
  public :
     Alpha(  ) { w = 1; x = 2; }
     void   foo( ) {cout << "w =" << w << endl ; }
     virtual   void   bar( )
     {cout << "x =" << x << endl; }
} ;

class    Beta : public   Alpha
{
   private :
     int   y ;
   protected :
     int   z ;
   public:
     Beta(  ) {y = 3 ; z = 4; }
     void   foo( ) {cout << "y =" << y << endl; }
     void   bar( ) {cout << "z =" << z << endl; }
} ;
```

Hãy cho biết chương trình sau in ra cái gì? Giải thích?

```
int main( )
{
   Alpha   A;
   Beta   B;
```

```
    Alpha* Bptr = &A ;
    A.foo( ) ;
    A.bar( ) ;
    B.foo( ) ;
    B.bar( ) ;
    Bptr → foo( )
    ; Bptr → bar(
    ) ; Bptr = &B
    ; Bptr → foo(
    ) ; Bptr →
    bar( ) ;
}
```