Chương 9. Thừa kế

Trong thực tế hầu hết các lớp có thể kế thừa từ các lớp có trước mà không cần định nghĩa lại mới hoàn toàn. Ví dụ xem xét một lớp được đặt tên là RecFile đại diện cho một tập tin gồm nhiều mẫu tin và một lớp khác được đặt tên là SortedRecFile đại diện cho một tập tin gồm nhiều mẫu tin được sắp xếp. Hai lớp này có thể có nhiều điểm chung. Ví dụ, chúng có thể có các thành viên hàm giống nhau như là Insert, Delete, và Find, cũng như là thành viên dữ liệu giống nhau. SortedRecFile là một phiên bản đặc biệt của RecFile với thuộc tính các mẫu tin của nó được tổ chức theo thứ tự được thêm vào. Vì thế hầu hết các hàm thành viên trong cả hai lớp là giống nhau trong khi một vài hàm mà phụ thuộc vào yếu tố tập tin được sắp xếp thì có thể khác nhau. Ví dụ, hàm Find có thể là khác trong lớp SortedRecFile bởi vì nó có thể nhờ vào yếu tố thuận lợi là tập tin được sắp để thực hiện tìm kiếm nhị phân thay vì tìm tuyến tính như hàm Find của lớp RecFile.

Với các thuộc tính được chia sẻ của hai lớp này thì việc định nghĩa chúng một cách độc lập là rất dài dòng. Rõ ràng điều này dẫn tới việc phải sao chép lại mã đáng kể. Mã không chỉ mất thời gian lâu hơn để viết nó mà còn khó có thể được bảo trì hơn: một thay đổi tới bất kỳ thuộc tính chia sẻ nào có thể phải được sửa đổi tới cả hai lớp.

Lập trình hướng đối tượng cung cấp một kỹ thuật thuận lợi gọi là **thừa kế** để giải quyết vấn đề này. Với thừa kế thì một lớp có thể thừa kế những thuộc tính của một lớp đã có trước. Chúng ta có thể sử dụng thừa kế để định nghĩa những thay đổi của một lớp mà không cần định nghĩa lại lớp mới từ đầu. Các thuộc tính chia sẻ chỉ được định nghĩa một lần và được sử dụng lại khi cần.

Trong C++ thừa kế được hỗ trợ bởi các **lớp dẫn xuất** (derived class). Lớp dẫn xuất thì giống như lớp gốc ngoại trừ định nghĩa của nó dựa trên một hay nhiều lớp có sẵn được gọi là **lớp cơ sở** (base class). Lớp dẫn xuất có thể chia sẻ những thuộc tính đã chọn (các thành viên hàm hay các thành viên dữ liệu) của các lớp cơ sở của nó nhưng không làm chuyển đổi định nghĩa của bất kỳ lớp cơ sở nào. Lớp dẫn xuất chính nó có thể là lớp cơ sở của một lớp dẫn xuất khác. Quan hệ thừa kế giữa các lớp của một chương trình được gọi là **quan hệ cấp bậc lớp** (class hierarchy).

Lớp dẫn xuất cũng được gọi là **lớp con** (subclass) bởi vì nó trở thành cấp thấp hơn của lớp cơ sở trong quan hệ cấp bậc. Tương tự một lớp cơ sở có thể được gọi là **lớp cha** (superclass) bởi vì từ nó có nhiều lớp khác có thể được dẫn xuất.

9.1. Ví dụ minh họa

Chúng ta sẽ định nghĩa hai lớp nhằm mục đích minh họa một số khái niệm lập trình trong các phần sau của chương này. Hai lớp được định nghĩa trong Danh sách 9.1 và hỗ trợ việc tạo ra một thư mục các đối tác cá nhân.

Danh sách 9.1

```
1  #include <iostream.h>
2  #include <string.h>

3  class Contact {
4   public:
5    Contact(const char *name, const char *address, const char *tel);
6        ~Contact   (void);
7        const char*Name       (void) const        {return name;}
8        const char*Address(void) const {return address;}
9        const char*Tel(void) const    {return tel;}
10   friend ostream&      operator <<       (ostream&, Contact&);
11
     private:
12       char         *name;              // ten doi tac
13       char         *address;   // dia chi doi tac
14       char         *tel;          // so dien thoai
15 };
16
     //---------------------------------------------------
17 class ContactDir {
18  public:
19   ContactDir(const int maxSize);
20       ~ContactDir(void);
21       void  Insert(const Contact&);
22       void  Delete(const char *name);
23       Contact*    Find(const char *name);
24   friend    ostream&   operator <<(ostream&, ContactDir&);
25
     private:
26       int    Lookup(const char *name);
27       Contact     **contacts;// danh sach cac doi tac
         int    dirSize;           // kich thuoc thu muc hien tai
28       int    maxSize;      // kich thuoc thu muc toi da
29 };
```

Chú giải

3 Lớp Contact lưu giữ các chi tiết của một đối tác (nghĩa là, tên, địa chỉ, và số điện thoại).

18 Lớp ContactDir cho phép chúng ta thêm, xóa, và tìm kiếm một danh sách các đối tác.

22 Hàm Insert xen một đối tác mới vào thư mục. Điều này sẽ viết chồng lên một đối tác tồn tại (nếu có) với tên giống nhau.

23 Hàm Delete xóa một đối tác (nếu có) mà tên của đối tác trùng với tên đã cho.

24 Hàm Find trả về một con trỏ tới một đối tác (nếu có) mà tên của đối tác khớp với tên đã cho.

27 Hàm Lookup trả về chỉ số vị trí của một đối tác mà tên của đối tác khớp với tên đã cho. Nếu không tồn tại thì sau đó hàm Lookup trả về chỉ số của vị trí mà tại đó mà một đầu vào như thế sẽ được thêm vào. Hàm Lookup được định nghĩa như là riêng (private) bởi vì nó là một hàm phụ được sử dụng bởi các hàm Insert, Delete, và Find.

Cài đặt của hàm thành viên và hàm bạn như sau:

```
Contact::Contact (const char *name,
                  const char *address, const char *tel)
{
    Contact::name = new char[strlen(name) + 1];
    Contact::address = new char[strlen(address) + 1];
    Contact::tel = new char[strlen(tel) + 1];
    strcpy(Contact::name, name);
    strcpy(Contact::address, address);
    strcpy(Contact::tel, tel);
}

Contact::~Contact (void)
{
    delete name;
    delete address;
    delete tel;
}

ostream&    operator << (ostream &os, Contact &c)
{
    os << "(" << c.name << " , "
       << c.address << " , " << c.tel << ")";
  return os;
}

ContactDir::ContactDir (const int max)
{
    typedef Contact *ContactPtr;
    dirSize = 0;
    maxSize = max;
    contacts = new ContactPtr[maxSize];
};

ContactDir::~ContactDir (void)
{
    for (register i = 0; i < dirSize; ++i)
        delete contacts[i];
    delete [] contacts;
}

void ContactDir::Insert (const Contact& c)
{
    if (dirSize < maxSize) {
        int idx = Lookup(c.Name());
        if (idx > 0 &&
            strcmp(c.Name(), contacts[idx]->Name()) == 0) {
    delete contacts[idx];
```

```
            } else {
                for (register i = dirSize; i > idx; --i) // dich phai
                    contacts[i] = contacts[i-1];
                ++dirSize;
            }
            contacts[idx] = new Contact(c.Name(), c.Address(), c.Tel());
        }
    }

    void ContactDir::Delete (const char *name)
    {
        int idx = Lookup(name);
        if (idx < dirSize) {
            delete contacts[idx];
            --dirSize;
            for (register i = idx; i < dirSize; ++i)          // dich trai
                contacts[i] = contacts[i+1];
        }
    }

    Contact *ContactDir::Find (const char *name)
    {
        int idx = Lookup(name);
        return (idx < dirSize &&
                strcmp(contacts[idx]->Name(), name) == 0)
                    ? contacts[idx]
                    : 0;
    }

    int ContactDir::Lookup (const char *name)
    {
        for (register i = 0; i < dirSize; ++i)
            if (strcmp(contacts[i]->Name(), name) == 0)
                return i;
        return dirSize;
    }

    ostream  &operator << (ostream &os, ContactDir &c)
    {
        for (register i = 0; i < c.dirSize; ++i)
        os << *(c.contacts[i]) << '\n';
      return os;
    }
```

Hàm main sau thực thi lớp ContactDir bằng cách tạo ra một thư mục nhỏ và gọi các hàm thành viên:

```
    int main (void)
    {
        ContactDir dir(10);
        dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
        dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
        dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
        dir.Insert(Contact("Jack", "42 Wayne St", "663 2989"));
        dir.Insert(Contact("Fred", "2 High St", "458 2324"));

        cout << dir;
        cout << "Find Jane: " << *dir.Find("Jane") << '\n';
        dir.Delete("Jack");
```

```
        cout << "Deleted Jack\n";
        cout << dir;
        return 0;
    };
```

Khi chạy nó sẽ cho kết quả sau:

```
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
(Jane , 321 Yara Ln , 982 6252)
(Jack , 42 Wayne St , 663 2989)
(Fred , 2 High St , 458 2324)
Find Jane: (Jane , 321 Yara Ln , 982 6252)
Deleted Jack
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
(Jane , 321 Yara Ln , 982 6252)
(Fred , 2 High St , 458 2324)
```

9.2. Lớp dẫn xuất đơn giản

Chúng ta muốn định nghĩa một lớp gọi là SmartDir ứng xử giống như là lớp ContactDir và theo dõi tên của đối tác mới vừa được tìm kiếm gần nhất. Lớp SmartDir được định nghĩa tốt nhất như là một dẫn xuất của lớp ContactDir như được minh họa bởi Danh sách 9.2.

Danh sách 9.2

```
1  class SmartDir : public ContactDir {
2    public:
3      SmartDir(const int max) : ContactDir(max) {recent = 0;}
4          Contact* Recent (void);
5          Contact* Find  (const char *name);

6    private:
7          char      *   recent;  // ten duoc tim gan nhat
8  };
```

Chú giải

1 Phần đầu của lớp dẫn xuất chèn vào các lớp cơ sở mà nó thừa kế. Một dấu hai chấm (:) phân biệt giữa hai phần. Ở đây, lớp ContactDir được đặc tả là lớp cơ sở mà lớp SmartDir được dẫn xuất. Từ khóa public phía trước lớp ContactDir chỉ định rằng lớp ContactDir được sử dụng như một lớp cơ sở chung.

3 Lớp SmartDir có hàm xây dựng của nó, hàm xây dựng này triệu gọi hàm xây dựng của lớp cơ sở trong danh sách khởi tạo thành viên của nó.

4 Hàm Recent trả về một con trỏ tới đối tác được tìm kiếm sau cùng (hoặc 0 nếu không có).

5 Hàm Find được định nghĩa lại sao cho nó có thể ghi nhận đầu vào được tìm kiếm sau cùng.

7 Con trỏ recent được đặt tới tên của đầu vào đã được tìm sau cùng.

Các hàm thành viên được định nghĩa như sau:

```
Contact* SmartDir::Recent (void)
{
      return recent == 0 ? 0 : ContactDir::Find(recent);
}

Contact* SmartDir::Find (const char *name)
{
      Contact *c = ContactDir::Find(name);
      if (c != 0)
            recent = (char*) c->Name();
  return c;
}
```

Bởi vì lớp ContactDir là một lớp cơ sở chung của lớp SmartDir nên tất cả thành viên chung của lớp ContactDir trở thành các thành viên chung của lớp martDir. Điều này nghĩa là chúng ta có thể triệu gọi một hàm thành viên như là Insert trên một đối tượng SmartDir và đây là một lời gọi tới ContactDir::Insert. Tương tự, tất cả các thành viên riêng của lớp ContactDir trở thành các thành viên riêng của lớp SmartDir.

Phù hợp với các nguyên lý ẩn thông tin, các thành viên riêng của lớp ContactDir sẽ không thể được truy xuất bởi SmartDir. Vì thế, lớp SmartDir sẽ không thể truy xuất tới bất kỳ thành viên dữ liệu nào của lớp ContactDir cũng như là hàm thành viên riêng Lookup.

Lớp SmartDir định nghĩa lại hàm thành viên Find. Điều này không nên nhầm lẫn với tái định nghĩa. Có hai định nghĩa phân biệt của hàm này: ContactDir::Find và SmartDir::Find (cả hai định nghĩa có cùng dấu hiệu dẫu cho chúng có thể có các dấu hiệu khác nhau nếu được yêu cầu). Triệu gọi hàm Find trên đối tượng SmartDir thứ hai sẽ được gọi. Như được minh họa bởi định nghĩa của hàm Find trong lớp SmartDir,hàm thứ nhất có thể vẫn còn được triệu gọi bằng cách sử dụng tên đầy đủ của nó.

Đoạn mã sau minh họa lớp SmartDir cư xử như là lớp ContactDir nhưng cũng theo dõi đầu vào được tìm kiếm được gần nhất:

```
SmartDir          dir(10);
dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
dir.Insert(Contact("Fred", "2 High St", "458 2324"));
dir.Find("Jane");
dir.Find("Peter");
cout << "Recent: " << *dir.Recent() << '\n';
```

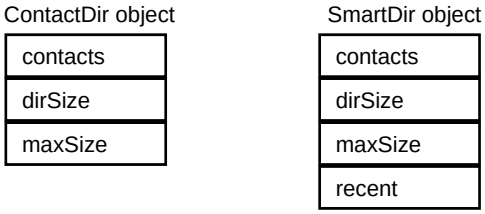
Điều này sẽ cho ra kết quả sau:

```
Recent: (Peter , 9 Port Rd , 678 9862)
```

Một đối tượng kiểu SmartDir chứa đựng tất cả dữ liệu thành viên của ContactDir cũng như là bất kỳ dữ liệu thành viên thêm vào được giới thiệu bởi

SmartDir. Hình 9.1 minh họa việc tạo ra một đối tượng ContactDir và một đối tượng SmartDir.

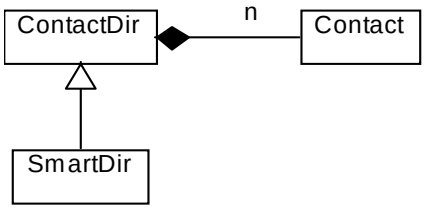
Hình 9.1 Các đối tượng lớp cơ sở và lớp dẫn xuất.



9.3. Ký hiệu thứ bậc lớp

Thứ bậc lớp thường được minh họa bằng cách sử dụng ký hiệu đồ họa đơn giản. Hình 9.2 minh họa ký hiệu của ngôn ngữ UML mà chúng ta sẽ đang sử dụng trong giáo trình này. Mỗi lớp được biểu diễn bằng một hộp được gán nhãn là tên lớp. Thừa kế giữa hai lớp được minh họa bằng một mũi tên có hướng vẽ từ lớp dẫn xuất đến lớp cơ sở. Một đường thẳng với hình kim cương ở một đầu miêu tả **composition** (tạm dịch là quan hệ bộ phận, nghĩa là một đối tượng của lớp được bao gồm một hay nhiều đối tượng của lớp khác). Số đối tượng chứa bởi đối tượng khác được miêu tả bởi một nhãn (ví dụ, *n*).

Hình 9.2 Một thứ bậc lớp đơn giản



Hình 9.2 được thông dịch như sau. Contact, ContactDir, và SmartDir là các lớp. Lớp ContactDir gồm có không hay nhiều đối tượng Contact. Lớp SmartDir được dẫn xuất từ lớp ContactDir.

9.4. Hàm xây dựng và hàm hủy

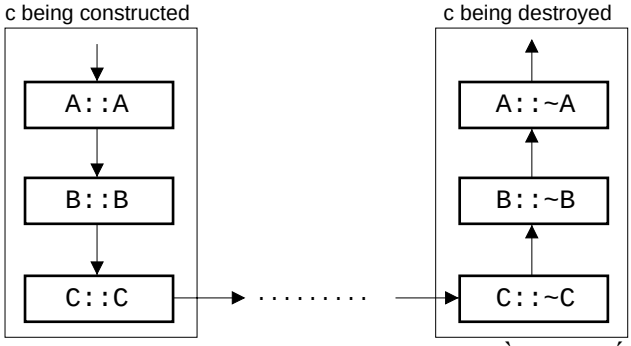
Lớp dẫn xuất có thể có các hàm xây dựng và một hàm hủy. Bởi vì một lớp dẫn xuất có thể cung cấp các dữ liệu thành viên dựa trên các dữ liệu thành viên từ lớp cơ sở của nó nên vai trò của hàm xây dựng và hàm hủy là để khởi tạo và hủy bỏ các thành viên thêm vào này.

Khi một đối tượng của một lớp dẫn xuất được tạo ra thì hàm xây dựng của lớp cơ sở được áp dụng tới nó trước tiên và theo sau là hàm xây dựng của lớp dẫn xuất. Khi một đối tượng bị thu hồi thì hàm hủy của lớp dẫn xuất được áp dụng trước tiên và sau đó là hàm hủy của lớp cơ sở. Nói cách khác thì các hàm xây dựng được ứng dụng theo thứ tự từ gốc (lớp cha) đến ngọn (lớp con)

và các hàm hủy được áp dụng theo thứ tự ngược lại. Ví dụ xem xét một lớp C được dẫn xuất từ lớp B, mà lớp B lại được dẫn xuất từ lớp A. Hình 9.3 minh họa một đối tượng c thuộc lớp C được tạo ra và hủy bỏ như thế nào.

```
class A                    { /* ... */ }
class B : public A         { /* ... */ }
class C : public B         { /* ... */ }
```

Hình 9.3 **Thứ tự xây dựng và hủy bỏ đối tượng của lớp dẫn xuất.**



Bởi vì hàm xây dựng của lớp cơ sở yêu cầu các đối số, chúng cần được chỉ định trong phần định nghĩa hàm xây dựng của lớp dẫn xuất. Để làm công việc này, hàm xây dựng của lớp dẫn xuất triệu gọi rõ ràng hàm xây dựng lớp cơ sở trong danh sách khởi tạo thành viên của nó. Ví dụ, hàm xây dựng SmartDir truyền đối số của nó tới hàm xây dựng ContactDir theo cách này:

```
SmartDir::SmartDir (const int max) : ContactDir(max)
{ /* ... */ }
```

Thông thường, tất cả những gì mà một hàm xây dựng lớp dẫn xuất yêu cầu là một đối tượng từ lớp cơ sở. Trong một vài tình huống, điều này thậm chí có thể không cần tham khảo tới hàm xây dựng lớp cơ sở:

```
extern ContactDir cd;                  // được định nghĩa ở đâu đó
SmartDir::SmartDir (const int max) : cd
{ /* ... */ }
```

Mặc dù các thành viên riêng của một lớp lớp cơ sở được thừa kế bởi một lớp dẫn xuất nhưng chúng không thể được truy xuất. Ví dụ, lớp SmartDir thừa kế tất cả các thành viên riêng (và chung) của lớp ContactDir nhưng không được phép tham khảo trực tiếp tới các thành viên riêng của lớp ContactDir. Ý tưởng là các thành viên riêng nên được che dấu hoàn toàn sao cho chúng không thể bị can thiệp vào bởi các khách hàng (client) của lớp.

Sự giới hạn này có thể chứng tỏ chiều hướng ngăn cấm các lớp có khả năng là lớp cơ sở cho những lớp khác. Việc từ chối truy xuất của lớp dẫn xuất tới các thành viên riêng của lớp cơ sở vướng vào sự cài đặt nó hay thậm chí làm cho việc định nghĩa nó là không thực tế.

Sự giới hạn có thể được giải phóng bằng cách định nghĩa các thành viên riêng của lớp cơ sở như là được bảo vệ (protected). Đến khi các khách hàng của lớp được xem xét, một thành viên được bảo vệ thì giống như một thành viên riêng: nó không thể được truy xuất bởi các khách hàng lớp. Tuy nhiên,

một thành viên lớp cơ sở được bảo vệ có thể được truy xuất bởi bất kỳ lớp nào được dẫn xuất từ nó.

Ví dụ, các thành viên riêng của lớp ContactDir có thể được tạo ra là được bảo vệ bằng cách thay thế từ khóa protected cho từ khóa private:

```
class ContactDir {
      //...
 protected:
      int    Lookup (const char *name);
      Contact     **contacts;// danh sach cac doi tac
      int    dirSize;              // kich thuoc thu muc hien tai
      int    maxSize;              // kich thuoc thu muc toi da
};
```

Kết quả là, hàm Lookup và các thành viên dữ liệu của lớp ContactDir bây giờ có thể truy xuất bởi lớp SmartDir.

Các từ khóa truy xuất private, public, và protected có thể xuất hiện nhiều lần trong một định nghĩa lớp. Mỗi từ khóa truy xuất chỉ định các đặc điểm truy xuất của các thành viên theo sau nó cho đến khi bắt gặp một từ khóa truy xuất khác:

```
class Foo {
      public:
            // cac thanh vien chung...
      private:
            // cac thanh vien rieng...
      protected:
            // cac thanh vien duoc bao ve...
      public:
            // cac thanh vien chung nua...
      protected:
            // cac thanh vien duoc bao ve nua...
};
```

9.5. Lớp cơ sở riêng, chung, và được bảo vệ

Một lớp cơ sở có thể được chỉ định là riêng, chung, hay được bảo vệ. Nếu không được chỉ định như thế, lớp cơ sở được giả sử là riêng:

```
class A {
      private:    int x;        void Fx (void);
      public:           int y;        void Fy (void);
      protected:  int z;        void Fz (void);
};
class B : A {};                       // A la lop co so rieng cua B
class C : private A {};       // A la lop co so rieng cua C
class D : public A {};        // A la lop co so chung cua D
class E : protected A {};     // A la lop co so duoc bao ve cua E
```

Cư xử của những lớp này là như sau (xem Bảng 9.1 cho một tổng kết):

- Tất cả các thành viên của một lớp cơ sở riêng trở thành các thành viên *riêng* của lớp dẫn xuất. Vì thế tất cả x, Fx, y, Fy, z, và Fz trở thành các thành viên riêng của B và C.
- Các thành viên của lớp cơ sở chung giữ các đặc điểm truy xuất của chúng trong lớp dẫn xuất. Vì thế, x và Fx trở thành các thành viên riêng D, y và Fy trở thành các thành viên chung của D, và z và Fz trở thành các thành viên được bảo vệ của D.
- Các thành viên riêng của lớp cơ sở được bảo vệ trở thành các thành viên *riêng* của lớp dẫn xuất. Nhưng ngược lại, các thành viên chung và được bảo vệ của lớp cơ sở được bảo vệ trở thành các thành viên *được bảo vệ* của lớp dẫn xuất. Vì thế, x và Fx trở thành các thành viên riêng của E, và y, Fy, z, và Fz trở thành các thành viên được bảo vệ của E.

Bảng 9.1 Các qui luật thừa kế truy xuất lớp cơ sở.

Lớp cơ sở	Dẫn xuất riêng	Dẫn xuất chung	Dẫn xuất được bảo vệ
Private Member	*private*	*private*	*private*
Public Member	*private*	*public*	*protected*
Protected Member	*private*	*protected*	*protected*

Chúng ta cũng có thể miễn cho một thành viên riêng lẻ của lớp cơ sở từ những chuyển đổi truy xuất được đặc tả bởi một lớp dẫn sao cho nó vẫn giữ lại những đặc điểm truy xuất gốc của nó. Để làm điều này, các thành viên được miễn được đặt tên đầy đủ trong lớp dẫn xuất với đặc điểm truy xuất gốc của nó. Ví dụ:

```
class C : private A {
      //...
public:          A::Fy;      // lam cho Fy la mot thanh vien chung cua C
protected:  A::z; // lam cho z la mot thanh vien duoc bao ve
                              // cua C
};
```

9.6. Hàm ảo

Xem xét sự thay đổi khác của lớp ContactDir được gọi là SortedDir, mà đảm bảo rằng các đối tác mới được xen vào phần còn lại của danh sách đã được sắp xếp. Thuận lợi rõ ràng của điều này là tốc độ tìm kiếm có thể được cải thiện bằng cách sử dụng giải thuật tìm kiếm nhị phân thay vì tìm kiếm tuyến tính.

Việc tìm kiếm trong thực tế được thực hiện bởi hàm thành viên Lookup. Vì thế chúng ta cần định nghĩa lại hàm này trong lớp SortedDir sao cho nó sử dụng giải thuật tìm kiếm nhị phân. Tuy nhiên, tất cả các hàm thành viên khác tham khảo tới ContactDir::Lookup. Chúng ta cũng có thể định nghĩa các hàm này sao cho chúng tham khảo tới SortedDir::Lookup. Nếu chúng ta theo tiếp cận này, giá trị của thừa kế trở nên đáng ngờ hơn bởi vì thực tế chúng ta có thể phải định nghĩa lại toàn bộ lớp.

Thực sự cái mà chúng ta muốn làm là tìm cách để biểu diễn điều này: hàm Lookup nên được liên kết tới *kiểu* của đối tượng mà triệu gọi nó. Nếu đối tượng thuộc kiểu SortedDir sau đó triệu gọi Lookup (từ bất kỳ chỗ nào, thậm chí từ bên trong các hàm thành viên của ContactDir) có nghĩa là SortedDir::Lookup. Tương tự, nếu đối tượng thuộc kiểu ContactDir sau đó gọi Lookup (từ bất kỳ chỗ nào) có nghĩa là ContactDir::Lookup.

Điều này có thể được thực thi thông qua **liên kết động** (dynamic binding) của hàm Lookup: sự quyết định chọn phiên bản nào của hàm Lookup để gọi được tạo ra ở thời gian chạy phụ thuộc vào kiểu của đối tượng.

Trong C++, liên kết động được hỗ trợ thông qua các hàm thành viên ảo. Một hàm thành viên được khai báo như là ảo bằng cách chèn thêm từ khóa virtual trước nguyên mẫu (prototype) của nó trong lớp cơ sở. Bất kỳ hàm thành viên nào, kể cả hàm xây dựng và hàm hủy, có thể được khai báo như ảo. Hàm Lookup nên được khai báo như ảo trong lớp ContactDir:

```
class ContactDir {
        //...
public:
        virtual     int    Lookup (const char *name);
        //...
};
```

Chỉ các hàm thành viên không tĩnh có thể được khai báo như là ảo. Một hàm thành viên ảo được định nghĩa lại trong một lớp dẫn xuất phải có chính xác cùng tham số và kiểu trả về như một hàm thành viên trong lớp cơ sở. Các hàm ảo có thể được tái định nghĩa giống như các thành viên khác.

Danh sách 9.3 trình bày định nghĩa của lớp SortedDir như lớp dẫn xuất của lớp ContactDir.

Danh sách 9.3

```
1  class SortedDir : public ContactDir {
2  public:
3                          SortedDir  (const int max) : ContactDir(max) {}
4  public:
5        virtual     int    Lookup           (const char *name);
6  };
```

Chú giải

3 Hàm xây dựng đơn giản chỉ gọi hàm xây dựng lớp cơ sở.

5 Hàm Lookup được khai báo lại như là ảo để cho phép bất kỳ lớp nào được dẫn xuất từ lớp SortedDir định nghĩa lại nó.

Định nghĩa mới của hàm Lookup như sau:

```
int SortedDir::Lookup (const char *name)
{
        int bot = 0;
        int top = dirSize - 1;
```

```
    int pos = 0;
    int  mid, cmp;

    while (bot <= top) {
        mid = (bot + top) / 2;
        if ((cmp = strcmp(name, contacts[mid]->Name())) == 0)
            return mid;
        else if (cmp < 0)
            pos = top = mid - 1;      // gioi han tim tren nua thap hon
        else
            pos = bot = mid + 1;      // gioi han tim tren nua cao hon
    }
    return pos < 0 ? 0 : pos;
}
```

Đoạn mã sau minh họa rằng hàm SortedDir::Lookup được gọi bởi hàm ContactDir::Insert khi được triệu gọi thông qua đối tượng SortedDir:

```
SortedDir  dir(10);
dir.Insert(Contact("Mary", "11 South Rd", "282 1324"));
dir.Insert(Contact("Peter", "9 Port Rd", "678 9862"));
dir.Insert(Contact("Jane", "321 Yara Ln", "982 6252"));
dir.Insert(Contact("Jack", "42 Wayne St", "663 2989"));
dir.Insert(Contact("Fred", "2 High St", "458 2324"));
cout << dir;
```

Nó sẽ cho ra kết quả sau:

```
(Fred , 2 High St , 458 2324)
(Jack , 42 Wayne St , 663 2989)
(Jane , 321 Yara Ln , 982 6252)
(Mary , 11 South Rd , 282 1324)
(Peter , 9 Port Rd , 678 9862)
```

9.7. Đa thừa kế

Các lớp dẫn xuất mà chúng ta đã bắt gặp đến thời điểm này trong chương này chỉ là biểu diễn **đơn thừa kế**, bởi vì mỗi lớp thừa kế các thuộc tính của nó từ một lớp cơ sở đơn. Một tiếp cận có thể khác, một lớp dẫn xuất có thể có nhiều lớp cơ sở. Điều này được biết đến như là **đa thừa kế** (multiple inheritance).

Ví dụ, chúng ta phải định nghĩa hai lớp tương ứng để biểu diễn các danh sách của các tùy chọn và các cửa sổ điểm ảnh:

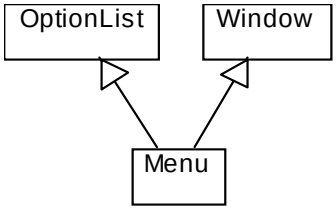
```
class OptionList {
 public:
      OptionList (int n);
      ~OptionList       (void);
      //...
};

class Window {
 public:
      Window    (Rect &bounds);
      ~Window  (void);
   //...
};
```

Một menu là một danh sách của các tùy chọn được hiển thị ở bên trong cửa sổ của nó. Vì thế nó có thể định nghĩa Menu bằng cách dẫn xuất từ lớp OptionList và lớp Window:

```
class Menu : public OptionList, public Window {
 public:
      Menu       (int n, Rect &bounds);
  ~Menu   (void);
      //...
};
```

Với đa thừa kế, một lớp dẫn xuất thừa kế tất cả các thành viên của các lớp cơ sở của nó. Như trước, mỗi thành viên của lớp cơ sở có thể là riêng, chung, hay là được bảo vệ. Áp dụng cùng các nguyên lý truy xuất thành viên lớp cơ sở. Hình 9.4 minh họa thứ bậc lớp cho Menu..

Hình 9.4 Thứ bậc lớp cho Menu



Vì các lớp cơ sở của lớp Menu có các hàm xây dựng yêu cầu các đối số nên hàm xây dựng cho lớp dẫn xuất nên triệu gọi những hàm xây dựng trong danh sách khởi tạo thành viên của nó:

```
Menu::Menu (int n, Rect &bounds) : OptionList(n), Window(bounds)
{
      //...
}
```

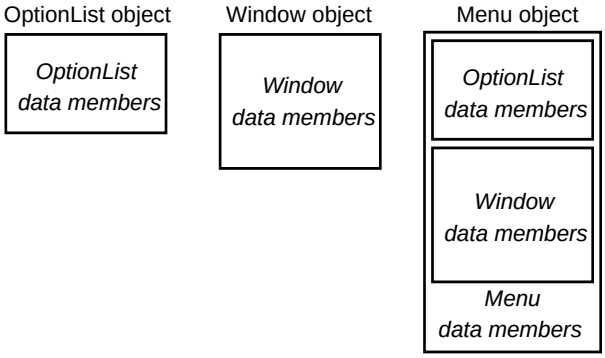
Thứ tự mà các hàm xây dựng được triệu gọi cùng với thứ tự mà chúng được đặc tả trong phần đầu của lớp dẫn xuất (*không* theo thứ tự mà chúng xuất hiện trong danh sách khởi tạo thành viên của các hàm xây dựng lớp dẫn xuất). Ví dụ, với Menu hàm xây dựng cho lớp OptionList được triệu gọi trước khi hàm xây dựng cho lớp Window, thậm chí nếu chúng ta chuyển đổi thứ tự trong hàm xây dựng:

```
Menu::Menu (int n, Rect &bounds) : Window(bounds), OptionList(n)
{
      //...
}
```

Các hàm hủy được ứng dụng theo thứ tự ngược lại: ~Menu, kế đó là ~Window, và cuối cùng là ~OptionList.

Sự cài đặt rõ ràng của đối tượng lớp dẫn xuất là chứa đựng một đối tượng từ mỗi đối tượng của các lớp cơ sở của nó. Hình 9.5 minh họa mối quan hệ giữa một đối tượng lớp Menu và các đối tượng lớp cơ sở.

Hình 9.5 Các đối tượng lớp dẫn xuất và cơ sở.



Thông thường, một lớp dẫn xuất có thể có số lượng các lớp cơ sở bất kỳ, tất cả chúng phải phân biệt:

```
class X : A, B, A {          // không hợp qui tắc: xuất hiện hai lần
      //...
};
```

9.8. Sự mơ hồ

Đa thừa kế làm phức tạp thêm nữa các qui luật để tham khảo tới các thành viên của một lớp. Ví dụ, giả sử cả hai lớp OptionList và Window có một hàm thành viên gọi là Highlight để làm nổi bậc một phần cụ thể của kiểu đối tượng này hay kiểu kia:

```
class OptionList {
 public:
      //...
      void Highlight (int part);
};

class Window {
 public:
      //...
      void Highlight (int part);
};
```

Lớp dẫn xuất Menu sẽ thừa kế cả hai hàm này. Kết quả là, lời gọi

```
m.Highlight(0);
```

(trong đó m là một đối tượng Menu) là mơ hồ và sẽ không biên dịch bởi vì nó không rõ ràng, nó tham khảo tới hoặc là OptionList::Highlight hoặc là Window::Highlight. Sự mơ hồ được giải quyết bằng cách làm cho lời gọi rõ ràng:

```
m.Window::Highlight(0);
```

Một khả năng chọn lựa khác, chúng ta có thể định nghĩa một hàm thành viên Highlight cho lớp Menu gọi các thành viên Highlight của các lớp cơ sở:

```
class Menu : public OptionList, public Window {
 public:
      //...
      void Highlight (int part);
};

void Menu::Highlight (int part)
{
      OptionList::Highlight(part);
      Window::Highlight(part);
}
```

9.9. Chuyển kiểu

Đối với bất kỳ lớp dẫn xuất nào có một sự chuyển kiểu không tường minh từ lớp dẫn xuất tới bất kỳ lớp cơ sở *chung* của nó. Điều này có thể được sử dụng để chuyển một đối tượng lớp dẫn xuất thành một đối tượng lớp cơ sở như là một đối tượng thích hợp, một tham chiếu, hoặc một con trỏ:

```
Menu        menu(n, bounds);
Window   win = menu;
Window   &wRef = menu;
Window   *wPtr = &menu;
```

Những chuyển đổi như thế là an toàn bởi vì đối tượng lớp dẫn xuất luôn chứa đựng tất cả các đối tượng lớp cơ sở của nó. Ví dụ, phép gán đầu tiên làm cho thành phần Window của menu được gán tới win.

Ngược lại, không có sự chuyển đổi từ lớp cơ sở thành lớp dẫn xuất. Lý do một sự chuyển kiểu như thế có khả năng nguy hiểm vì thực tế đối tượng lớp dẫn xuất có thể có các dữ liệu thành viên không có mặt trong đối tượng lớp cơ sở. Vì thế các thành viên dữ liệu phụ kết thúc bởi các giá trị không thể tiên toán. Tất cả chuyển kiểu như thế phải được ép kiểu rõ ràng để xác nhận ý định của lập trình viên:

```
Menu      &mRef = (Menu&) win;          // cẩn thận!
Menu      *mPtr = (Menu*) &win;         // cẩn thận!
```

Một đối tượng lớp cơ sở không thể được gán tới một đối tượng lớp cơ sở trừ phi có một hàm xây dựng chuyển kiểu trong lớp dẫn xuất được định nghĩa cho mục đích này. Ví dụ, với

```
class Menu : public OptionList, public Window {
public:
      //...
      Menu (Window&);
};
```

thì câu lệnh gán sau là hợp lệ và có thể sử dụng hàm xây dựng để chuyển đổi win thành đối tượng Menu trước khi gán:

```
menu = win;              // triệu gọi Menu::Menu(Window&)
```

9.10.Lớp cơ sở ảo

Trở lại lớp Menu và giả sử rằng hai lớp cơ sở của nó cũng được dẫn xuất từ nhiều lớp khác:

```
class OptionList : public Widget, List                    { /*...*/ };
class Window     : public Widget, Port                    { /*...*/ };
class Menu       : public OptionList, public Window { /*...*/ };
```

Vì lớp Widget là lớp cơ sở cho cả hai lớp OptionList và Window nên mỗi đối tượng menu sẽ có hai đối tượng widget (xem Hình 9.6a). Điều này là không mong muốn (bởi vì một menu được xem xét là một widget đơn) và có thể dẫn đến mơ hồ. Ví dụ, khi áp dụng hàm thành viên widget tới một đối tượng menu, thật không rõ ràng như áp dụng tới một trong hai đối tượng widget. Vấn đề này được khắc phục bằng cách làm cho lớp Widget là một **lớp cơ sở ảo** của lớp OptionList và Window. Một lớp cơ sở được làm cho ảo bằng cách đặt từ khóa virtual trước tên của nó trong phần đầu lớp dẫn xuất:

```
class OptionList : virtual public Widget, List        { /*...*/ };
class Window     : virtual public Widget, Port        { /*...*/ };
```

Điều này đảm bảo rằng một đối tượng Menu sẽ chứa đựng vừa đúng một đối tượng Widget. Nói cách khác, lớp OptionList và lớp Window sẽ chia sẻ cùng đối tượng Widget.

Một đối tượng của một lớp mà được dẫn xuất từ một lớp cơ sở ảo không chứa đựng trực tiếp đối tượng của lớp cơ sở ảo mà chỉ là một con trỏ tới nó (xem Hình 9.6b và 9.6c). Điều này cho phép nhiều hành vi của một lớp ảo trong một hệ thống cấp bậc được ghép lại thành một (xem Hình 9.6d).

Nếu ở trong một hệ thống cấp bậc lớp một vài thể hiện của lớp cơ sở X được khai báo như là ảo và các thể hiện khác như là không ảo thì sau đó đối tượng lớp dẫn xuất sẽ chứa đựng một đối tượng X cho mỗi thể hiện không ảo của X, và một đối tượng đơn X cho tất cả sự xảy ra ảo của X.

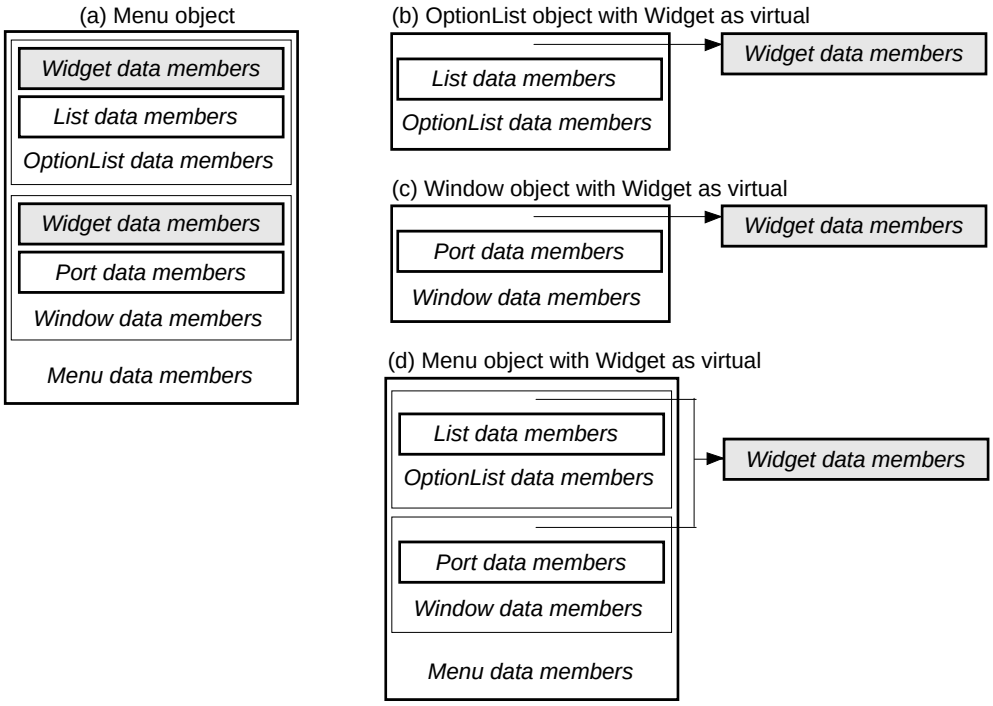
Một đối tượng lớp cơ sở ảo không được khởi tạo bởi lớp dẫn xuất trực tiếp của nó mà được khởi tạo bởi lớp dẫn xuất xa nhất dưới hệ thống cấp bậc lớp. Luật này đảm bảo rằng đối tượng lớp cơ sở ảo được khởi tạo chỉ một lần. Ví dụ, trong một đối tượng menu, đối tượng widget được khởi tạo bởi hàm

xây dựng Menu (ghi đè lời gọi hàm của hàm xây dựng Widget bởi OptionList hoặc Window):

```
Menu::Menu (int n, Rect &bounds) :    Widget(bounds),
                                          OptionList(n),
                                          Window(bounds)
{ //... }
```

Không quan tâm vị trí nó xuất hiện trong một hệ thống cấp bậc lớp, một đối tượng lớp cơ sở ảo luôn được xây dựng *trước* các đối tượng không ảo trong cùng hệ thống cấp bậc.

Hình 9.6 Các lớp cơ sở ảo và không ảo



Nếu trong một hệ thống cấp bậc lớp một cơ sở ảo được khai báo với các phạm vi truy xuất đối lập (nghĩa là, bất kỳ sự kết hợp của riêng, được bảo vệ, và chung) sau đó khả năng truy xuất lớn nhất sẽ thống trị. Ví dụ, nếu Widget được khai báo là một lớp cơ sở ảo riêng của lớp OptionList và là một lớp cơ sở ảo chung của lớp Window thì sau đó nó sẽ vẫn còn là một lớp cơ sở ảo chung của lớp Menu.

9.11.Các toán tử được tái định nghĩa

Ngoại trừ toán tử gán, một lớp dẫn xuất thừa kế tất cả các toán tử đã tái định nghĩa của lớp cơ sở của nó. Toán tử được tái định nghĩa bởi chính lớp dẫn xuất che giấu đi việc tái định nghĩa của cùng toán tử bởi các lớp cơ sở (giống như là các hàm thành viên của một lớp dẫn xuất che giấu đi các hàm thành viên của các lớp cơ sở).

Phép gán và khởi tạo memberwise (xem Chương 7) mở rộng tới lớp dẫn xuất. Đối với bất kỳ lớp Y dẫn xuất từ X, khởi tạo memberwise được điều khiển bởi một hàm xây dựng phát ra tự động (hay do người dùng định nghĩa) ở hình thức:

```
Y::Y (const Y&);
```

Tương tự, phép gán memberwise được điều khiển bởi tái định nghĩa toán tử = được phát ra tự động (hay do người dùng định nghĩa):

```
Y& Y::operator = (Y&)
```

Khởi tạo memberwise (hoặc gán) của đối tượng lớp dẫn xuất liên quan đến khởi tạo memberwise (hoặc gán) của các lớp cơ sở của nó cũng như là các thành viên đối tượng lớp của nó.

Cần sự quan tâm đặc biệt khi một lớp dẫn xuất nhờ vào tái định nghĩa các toán tử new và delete cho lớp cơ sở của nó. Ví dụ, trở lại việc tái định nghĩa hai toán tử này cho lớp Point trong Chương 7, và giả sử rằng chúng ta muốn sử dụng chúng cho một lớp dẫn xuất:

```
class Point3D : public Point {
  public:
        //...
  private:
        int depth;
};
```

Bởi vì sự cài đặt của Point::operator new giả sử rằng khối được cần sẽ có kích thước của đối tượng Point, việc thừa kế của nó bởi lớp Point3D dẫn tới một vấn đề: không thể giải thích tại sao dữ liệu thành viên của lớp Point3D (nghĩa là, depth) lại cần không gian phụ.

Để tránh vấn đề này, tái định nghĩa của toán tử new cố gắng cấp phát vừa đúng tổng số lượng lưu trữ được chỉ định bởi tham số kích thước của nó hơn là một kích thước giới hạn trước. Tương tự, tái định nghĩa của delete nên chú ý vào kích cỡ được chỉ định bởi tham số thứ hai của nó và cố gắng giải phóng vừa đúng các byte này.

Bài tập cuối chương 9

9.1 Xem xét lớp Year chia các ngày trong năm thành các ngày làm việc và các ngày nghỉ. Bởi vì mỗi ngày có một giá trị nhị phân nên lớp Year dễ dàng được dẫn xuất từ BitVec:

```
enum Month {
        Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};
class Year : public BitVec {
 public:
        Year        (const short year);
        void        WorkDay  (const short day);// dat ngay nhu ngay lam viec
        void   OffDay    (const short day);// dat ngay nhu ngay nghi
        Bool   Working  (const short day);// true neu la ngay lam viec
        short       Day          (const short day, // chuyen date thanh day
                                  const Month month, const short year);
protected:
        short       year; // nam theo lich
};
```

Các ngày được đánh số từ điểm bắt đầu của năm, bắt đầu từ ngày 1 tháng 1 năm 1. Hoàn tất lớp Year bằng cách thi công các hàm thành viên của nó.

9.2 Các bảng liệt kê được giới thiệu bởi một khai báo enum là một tập con nhỏ của các số nguyên. Trong một vài ứng dụng chúng ta có thể cần xây dựng các tập hợp của các bảng liệt kê như thế. Ví dụ, trong một bộ phân tích cú pháp, mỗi hàm phân tích có thể được truyền một tập các ký hiệu mà sẽ được bỏ qua khi bộ phân tích cú pháp cố gắng phục hồi từ một lỗi hệ thống. Các ký hiệu này thông thường được dành riêng những từ của ngôn ngữ:

```
enum Reserved {classSym, privateSym, publicSym, protectedSym,
                friendSym, ifSym, elseSym, switchSym,...};
```

Với những thứ đã cho có thể có nhiều nhất *n* phần tử (*n* là một số nhỏ) tập hợp có thể được trình bày có hiệu quả như một vectơ bit của *n* phần tử. Dẫn xuất một lớp đặt tên là EnumSet từ BitVec để làm cho điều này dễ dàng. Lớp EnumSet nên tái định nghĩa các toán tử sau:

- Toán tử + để hợp tập hợp.
- Toán tử - để hiệu tập hợp.
- Toán tử * để giao tập hợp.
- Toán tử % để kiểm tra một phần tử có là thành viên của tập hợp.
- Các toán tử <= và >= để kiểm tra một tập hợp có là một tập con của tập khác hay không.
- Các toán tử >> và << để thêm một phần tử tới tập hợp và xóa một phần tử từ tập hợp.

9.3 **Lớp trừu tượng** là một lớp mà không bao giờ được sử dụng trực tiếp nhưng cung cấp một khung cho các lớp khác được dẫn xuất từ nó. Thông thường, tất

cả các hàm thành viên của một lớp trừu tượng là ảo. Bên dưới là một ví dụ đơn giản của một lớp trừu tượng:

```
class Database {
  public:
        virtual   void Insert(Key, Data)      {}
        virtual   void Delete      (Key) {}
        virtual   Data Search      (Key) {return 0;}
};
```

Nó cung cấp một khung cho các lớp giống như cơ sở dữ liệu. Các ví dụ của loại lớp có thể được dẫn xuất từ cơ sở dữ liệu gồm: danh sách liên kết, cây nhị phân, và B-cây. Trước tiên dẫn xuất lớp B-cây từ lớp Database và sau đó dẫn xuất lớp B*-cây từ lớp B-cây:

```
class BTree : public Database { /*...*/ };
class BStar : public BTree    { /*...*/ };
```

Trong bài tập này sử dụng kiểu có sẵn int cho Key và double cho Data.