Chương 7. Lớp

Chương này giới thiệu cấu trúc lớp C++ để định nghĩa các kiểu dữ liệu mới. Một kiểu dữ liệu mới gồm hai thành phần như sau:

- Đặc tả cụ thể cho các đối tượng của kiểu.
- Tập các thao tác để thực thi các đối tượng.

Ngoài các thao tác đã được chỉ định thì không có thao tác nào khác có thể điều khiển đối tượng. Về mặt này chúng ta thường nói rằng các thao tác mô tả kiểu, nghĩa là chúng quyết định cái gì có thể và cái gì không thể xảy ra trên các đối tượng. Cũng với cùng lý do này, các kiểu dữ liệu thích hợp như thế được gọi là **kiểu dữ liệu trừu tượng** (abstract data type) - trừu tượng bởi vì sự đặc tả bên trong của đối tượng được ẩn đi từ các thao tác mà không thuộc kiểu.

Một **định nghĩa lớp** gồm hai phần: phần đầu và phần thân. **Phần đầu** lớp chỉ định **tên lớp** và các **lớp cơ sở** (base class). (Lớp cơ sở có liên quan đến lớp dẫn xuất và được thảo luận trong chương 8). **Phần thân** lớp định nghĩa các **thành viên lớp**. Hai loại thành viên được hỗ trợ:

- **Dữ liệu thành viên** (member data) có cú pháp của định nghĩa biến và chỉ định các đại diện cho các đối tượng của lớp.
- **Hàm thành viên** (member function) có cú pháp của khai báo hàm và chỉ định các thao tác của lớp (cũng được gọi là các **giao diện** của lớp).

C++ sử dụng thuật ngữ dữ liệu thành viên và hàm thành viên thay cho thuộc tính và phương thức nên kể từ đây chúng ta sử dụng dụng hai thuật ngữ này để đặc tả các lớp và các đối tượng.

Các thành viên lớp được liệt kê vào một trong ba loại quyền truy xuất khác nhau:

- Các thành viên **chung** (public) có thể được truy xuất bởi tất cả các thành phần sử dụng lớp.
- Các thành viên **riêng** (private) chỉ có thể được truy xuất bởi các thành viên lớp.
- Các thành viên **được bảo vệ** (protected) chỉ có thể được truy xuất bởi các thành viên lớp và các thành viên của một lớp dẫn xuất.

Kiểu dữ liệu được định nghĩa bởi một lớp được sử dụng như kiểu có sẵn.

7.1. Lớp đơn giản

Danh sách 7.1 trình bày định nghĩa của một lớp đơn giản để đại diện cho các điểm trong không gian hai chiều.

Danh sách 7.1

```
1  class Point {
2          int  xVal, yVal;
3     public:
4          void SetPt (int, int);
5          void OffsetPt (int, int);
6  };
```

Chú giải

1 Hàng này chứa phần đầu của lớp và đặt tên cho lớp là Point. Một định nghĩa lớp luôn bắt đầu với từ khóa class và theo sau đó là tên lớp. Một dấu { (ngoặc mở) đánh dấu điểm bắt đầu của thân lớp.

2 Hàng này định nghĩa hai dữ liệu thành viên xVal và yVal, cả hai thuộc kiểu int. Quyền truy xuất mặc định cho một thành viên của lớp là riêng (private). Vì thế cả hai xVal và yVal là riêng.

3 Từ khóa này chỉ định rằng từ điểm này trở đi các thành viên của lớp là chung (public).

4-5 Hai hàng này là các hàm thành viên. Cả hai có hai tham số nguyên và một kiểu trả về void.

6 Dấu } (ngoặc đóng) này đánh dấu kết thúc phần thân lớp.

Thứ tự trình bày các dữ liệu thành viên và hàm thành viên của một lớp là không quan trọng lắm. Ví dụ lớp trên có thể được viết tương đương như thế này:

```
class Point {
public:
        void SetPt (int, int);
        void OffsetPt (int, int);
private:
        int     xVal, yVal;
};
```

Định nghĩa thật sự của các hàm thành viên thường không là bộ phận của lớp và xuất hiện một cách tách biệt. Danh sách 7.2 trình bày định nghĩa riêng biệt của SetPt và OffsetPt.

Danh sách 7.2

```
1   void Point::SetPt (int x, int y)
2   {
3       xVal = x;
4       yVal = y;
5   }

6   void Point::OffsetPt (int x, int y)
7   {
8       xVal += x;
9       yVal += y;
10  }
```

Chú giải

1 Định nghĩa của một hàm thành viên thì tương tự như là hàm bình thường. Tên hàm được chỉ rõ trước với tên lớp và một cặp dấu hai chấm kép. Điều này xem SetPt như một thành viên của Point. Giao diện hàm phải phù hợp với định nghĩa giao diện trước đó bên trong lớp (nghĩa là, lấy hai tham số nguyên và có kiểu trả về là void).

3-4 Chú ý là hàm SetPt (là thành viên của Point) có thể tự do tham khảo tới dữ liệu thành viên xVal và yVal. Các hàm không là hàm thành viên không có quyền này.

Một khi một lớp được định nghĩa theo cách này, tên của nó bao hàm một kiểu dữ liệu mới cho phép chúng ta định nghĩa các biến của kiểu đó. Ví dụ:

```
Point pt;              // pt là một đối tượng của lớp Point
pt.SetPt(10,20);       // pt được đặt tới (10,20)
pt.OffsetPt(2,2);  // pt trở thành (12,22)
```

Các hàm thành viên được sử dụng ký hiệu dấu chấm: pt.SetPt(10,20) gọi hàm SetPt của đối tượng pt, nghĩa là pt là một đối số ẩn của SetPt.

Bằng cách tạo ra các thành viên riêng xVal và yVal chúng ta phải chắc chắn rằng người sử dụng lớp không thể điều khiển trực tiếp chúng:

```
pt.xVal = 10;          // không hợp lệ
```

Điều này sẽ không biên dịch.

Ở giai đoạn này, chúng ta cần phân biệt rõ ràng giữa đối tượng và lớp. Một lớp biểu thị một kiểu duy nhất. Một đối tượng là một phần tử của một kiểu cụ thể (lớp). Ví dụ,

```
Point pt1, pt2, pt3;
```

định nghĩa tất cả ba đối tượng (pt1, pt2, và pt3) của cùng một lớp (Point). Các thao tác của một lớp được ứng dụng bởi các đối tượng của lớp đó nhưng không bao giờ được áp dụng trên chính lớp đó. Vì thế một lớp là một khái niệm không có sự tồn tại cụ thể mà chịu sự phản chiếu bởi các đối tượng của nó.

7.2. Các hàm thành viên nội tuyến

Việc định nghĩa những hàm thành viên là nội tuyến cải thiện tốc độ đáng kể. Một hàm thành viên được định nghĩa là nội tuyến bằng cách chèn từ khóa inline trước định nghĩa của nó.

```
inline void Point::SetPt (int x,int y)
{
      xVal = x;
      yVal = y;
}
```

Một cách dễ hơn để định nghĩa các hàm thành viên là nội tuyến là chèn định nghĩa của các hàm này vào *bên trong* lớp.

```
class Point {
      int xVal, yVal;
public:
      void SetPt (int x,int y)          { xVal = x; yVal = y; }
      void OffsetPt (int x,int y)       { xVal += x; yVal += y; }
};
```

Chú ý rằng bởi vì thân hàm được chèn vào nên không cần dấu chấm phẩy sau khai báo hàm. Hơn nữa, các tham số của hàm phải được đặt tên.

7.3. Ví dụ: Lớp Set

Tập hợp (Set) là một tập các đối tượng không kể thứ tự và không lặp. Ví dụ này thể hiện rằng một tập hợp có thể được định nghĩa bởi một lớp như thế nào. Để đơn giản chúng ta giới hạn trên hợp các số nguyên với số lượng các phần tử là hữu hạn. Danh sách 7.3 trình bày định nghĩa lớp Set.

Danh sách 7.3

```
1   #include <iostream.h>

2   const maxCard = 100;
3   enum        Bool {false, true};

4   class Set {
5     public:
6           void  EmptySet      (void){ card = 0; }
7           Bool  Member        (const int);
8           void  AddElem       (const int);
9           void  RmvElem       (const int);
10          void  Copy          (Set&);
11          Bool  Equal         (Set&);
12          void  Intersect(Set&, Set&);
13          void  Union         (Set&, Set&);
14          void  Print         (void);
15    private:
16          int   elems[maxCard];        // cac phan tu cua tap hop
17          int   card;                  // so phan tu cua tap hop
18  };
```

Chú giải

2 maxCard biểu thị số lượng phần tử tối đa trong tập hợp.

6 EmptySet xóa nội dung tập hợp bằng cách đặt số phần tử tập hợp về 0.

7 Member kiểm tra một số cho trước có thuộc tập hợp hay không.

8 AddElem thêm một phần tử mới vào tập hợp. Nếu phần tử đã có trong tập hợp rồi thì không làm gì cả. Ngược lại thì thêm nó vào tập hợp. Trường hợp mà tập hợp đã tràn thì phần tử không được xen vào.

9 RmvElem xóa một phần tử trong tập hợp.

10 Copy sao chép tập hợp tới một tập hợp khác. Tham số cho hàm này là một tham chiếu tới tập hợp đích.

11 Equal kiểm tra hai tập hợp có bằng nhau hay không. Hai tập hợp là bằng nhau nếu chúng chứa đựng chính xác cùng số phần tử (thứ tự của chúng là không quan trọng).

12 Intersect so sánh hai tập hợp để cho ra tập hợp thứ ba chứa các phần tử là giao của hai tập hợp. Ví dụ, giao của {2,5,3} và {7,5,2} là {2,5}.

13 Union so sánh hai tập hợp để cho ra tập hợp thứ ba chứa các phần tử là hội của hai tập hợp. Ví dụ, hợp của {2,5,3} và {7,5,2} là {2,5,3,7}.

14 Print in một tập hợp sử dụng ký hiệu toán học theo qui ước. Ví dụ, một tập hợp gồm các số 5, 2, và 10 được in là {5,2,10}.

16 Các phần tử của tập hợp được biểu diễn bằng mảng elems.

17 Số phần tử của tập hợp được biểu thị bởi card. Chỉ có các đầu vào bản số đầu tiên trong elems được xem xét là các phần tử hợp lệ.

Việc định nghĩa tách biệt các hàm thành viên của một lớp đôi khi được biết tới như là **sự cài đặt** (implementation) của một lớp. Sự thi công lớp Set là như sau.

```
Bool Set::Member (const int elem)
{
    for (register i = 0; i < card; ++i)
        if (elems[i] == elem)
            return true;
    return false;
}

void Set::AddElem (const int elem)
{
    if (Member(elem))
        return;
    if (card < maxCard)
        elems[card++] = elem;
    else
    cout << "Set overflow\n";
}

void Set::RmvElem (const int elem)
{
    for (register i = 0; i < card; ++i)
```

```
        if (elems[i] == elem) {
            for (; i < card-1; ++i)      // dich cac phan tu sang trai
                elems[i] = elems[i+1];
            --card;
        }
}

void Set::Copy (Set &set)
{
    for (register i = 0; i < card; ++i)
        set.elems[i] = elems[i];
    set.card = card;
}

Bool Set::Equal (Set &set)
{
    if (card != set.card)
        return false;
    for (register i = 0; i < card; ++i)
        if (!set.Member(elems[i]))
            return false;
    return true;
}

void Set::Intersect (Set &set, Set &res)
{
    res.card = 0;
    for (register i = 0; i < card; ++i)
        if (set.Member(elems[i]))
            res.elems[res.card++] = elems[i];
}

void Set::Union (Set &set, Set &res)
{
    set.Copy(res);
    for (register i = 0; i < card; ++i)
        res.AddElem(elems[i]);
}

void Set::Print (void)
{
    cout << "{";
    for (int i = 0; i < card-1; ++i)
        cout << elems[i] << ",";
    if (card > 0)                    // khong co dau , sau phan tu cuoi cung
        cout << elems[card-1];
    cout << "}\n";
}
```

Hàm main sau đây tạo ra ba tập đối tượng Set và thực thi một vài hàm thành viên của nó.

```
int main (void)
{
    Set s1, s2, s3;

    s1.EmptySet();  s2.EmptySet();  s3.EmptySet();
    s1.AddElem(10); s1.AddElem(20); s1.AddElem(30); s1.AddElem(40);
    s2.AddElem(30); s2.AddElem(50); s2.AddElem(10); s2.AddElem(60);
```

```
    cout << "s1 = ";        s1.Print();
    cout << "s2 = ";        s2.Print();

    s2.RmvElem(50);
    cout << "s2 - {50} = ";
    s2.Print();
    if (s1.Member(20))
        cout << "20 is in s1\n";
    s1.Intersect(s2,s3);
    cout << "s1 intsec s2 = ";
    s3.Print();
    s1.Union(s2,s3);
        cout << "s1 union s2 = ";
    s3.Print();
    if (!s1.Equal(s2))
        cout << "s1 <> s2\n";
    return 0;
}
```

Khi chạy chương trình sẽ cho kết quả như sau:

```
s1 = {10,20,30,40}
s2 = {30,50,10,60}
s2 - {50} = {30,10,60}
20 is in s1
s1 intsec s2 = {10,30}
s1 union s2 = {30,10,60,20,40}
s1 <> s2
```

7.4. Hàm xây dựng (Constructor)

Hoàn toàn có thể định nghĩa và khởi tạo các đối tượng của một lớp ở cùng một thời điểm. Điều này được hỗ trợ bởi các hàm đặc biệt gọi là hàm xây dựng (constructor). Một hàm xây dựng luôn có cùng tên với tên lớp của nó. Nó không bao giờ có một kiểu trả về rõ ràng. Ví dụ,

```
class Point {
    int xVal, yVal;
public:
        Point (int x,int y) {xVal = x; yVal = y;} // constructor
    void OffsetPt (int,int);
};
```

là một định nghĩa có thể của lớp Point, trong đó SetPt đã được thay thế bởi một hàm xây dựng được định nghĩa nội tuyến.

Bây giờ chúng ta có thể định nghĩa các đối tượng kiểu Point và khởi tạo chúng một lượt. Điều này quả thật là ép buộc đối với những lớp chứa các hàm xây dựng đòi hỏi các đối số:

```
Point pt1 = Point(10,20);
Point pt2;                      // trái luật
```

Hàng thứ nhất có thể được đặc tả trong một hình thức ngắn gọn.

```
Point pt1(10,20);
```

Một lớp có thể có nhiều hơn một hàm xây dựng. Tuy nhiên, để tránh mơ hồ thì mỗi hàm xây dựng phải có một dấu hiệu duy nhất. Ví dụ,

```
class Point {
        int     xVal, yVal;
public:
                Point (int x, int y)        { xVal = x;  yVal = y; }
                Point (float, float);                         // các tọa độ cực
                Point (void)                 { xVal = yVal = 0; }  // gốc
        void OffsetPt (int, int);
};

Point::Point (float len, float angle)  // các tọa độ cực
{
        xVal = (int) (len * cos(angle));
        yVal = (int) (len * sin(angle));
}
```

có ba hàm xây dựng khác nhau. Một đối tượng có kiểu Point có thể được định nghĩa sử dụng bất kỳ hàm nào trong các hàm này:

```
Point pt1(10,20);         // tọa độ Đê-cát-tơ
Point pt2(60.3,3.14);   // tọa độ cực
Point pt3;                      // gốc
```

Lớp Set có thể được cải tiến bằng cách sử dụng một hàm xây dựng thay vì EmptySet:

```
class Set {
public:
        Set    (void)               { card = 0; }
        //...
};
```

Điều này tạo thuận lợi cho các lập trình viên không cần phải nhớ gọi EmptySet nữa. Hàm xây dựng đảm bảo rằng mọi tập hợp là rỗng vào lúc ban đầu.

Lớp Set có thể được cải tiến hơn nữa bằng cách cho phép người dùng điều khiển kích thước tối đa của tập hợp. Để làm điều này chúng ta định nghĩa elems như một con trỏ số nguyên hơn là mảng số nguyên. Hàm xây dựng sau đó có thể được cung cấp một đối số đặc tả kích thước tối đa mong muốn.

Nghĩa là maxCard sẽ không còn là hằng được dùng cho tất cả các đối tượng Set nữa mà chính nó trở thành một thành viên dữ liệu:

```
class Set {
public:
                Set (const int size);
        //...
private:
        int             *elems;                         // cac phan tu tap hop
    int         maxCard;                        // so phan tu toi da
        int             card;                           // so phan tu
};
```

Hàm xây dựng dễ dàng cấp phát một mảng động với kích thước mong muốn và khởi tạo giá trị phù hợp cho maxCard và card:

```
Set::Set (const int size)
{
    elems = new int[size];
    maxCard = size;
  card = 0;
}
```

Bây giờ có thể định nghĩa các tập hợp có các kích thước tối đa khác nhau:

```
Set ages(10), heights(20), primes(100);
```

Chúng ta cần lưu ý rằng một hàm xây dựng của đối tượng được ứng dụng khi đối tượng được tạo ra. Điều này phụ thuộc vào phạm vi của đối tượng. Ví dụ, một đối tượng toàn cục được tạo ra ngay khi sự thực thi chương trình bắt đầu; một đối tượng tự động được tạo ra khi phạm vi của nó được đăng ký; và một đối tượng động được tạo ra khi toán tử new được áp dụng tới nó.

7.5. Hàm hủy (Destructor)

Như là một hàm xây dựng được dùng để khởi tạo một đối tượng khi nó được tạo ra, một hàm hủy được dùng để dọn dẹp một đối tượng ngay trước khi nó được thu hồi. Hàm hủy luôn luôn có cùng tên với chính tên lớp của nó nhưng được đi đầu với ký tự ~. Không giống các hàm xây dựng, mỗi lớp chỉ có nhiều nhất một hàm hủy. Hàm hủy không nhận bất kỳ đối số nào và không có một kiểu trả về rõ ràng.

Thông thường các hàm hủy thường hữu ích và cần thiết cho các lớp chứa dữ liệu thành viên con trỏ. Các dữ liệu thành viên con trỏ trỏ tới các khối bộ nhớ được cấp phát từ lớp. Trong các trường hợp như thế thì việc giải phóng bộ nhớ đã được cấp phát cho các con trỏ thành viên là cực kỳ quan trọng trước khi đối tượng được thu hồi. Hàm hủy có thể làm công việc như thế.

Ví dụ, phiên bản sửa lại của lớp Set sử dụng một mảng được cấp phát động cho các thành viên elems. Vùng nhớ này nên được giải phóng bởi một hàm hủy:

```
class Set {
 public:
                Set   (const int size);
                ~Set  (void)        {delete elems;}   // destructor
     //...
 private:
     int        *elems;         // cac phan tu tap hop
   int      maxCard;        // so phan tu toi da
     int        card;       // so phan tu cua tap hop
};
```

Bây giờ hãy xem xét cái gì xảy ra khi một Set được định nghĩa và sử dụng trong hàm:

```
void Foo (void)
{
      Set s(10);
      //...
}
```

Khi hàm Foo được gọi, hàm xây dựng cho s được triệu tập, cấp phát lưu trữ cho s.elems và khởi tạo các thành viên dữ liệu của nó. Kế tiếp, phần còn lại của thân hàm Foo được thực thi. Cuối cùng, trước khi Foo trả về, hàm hủy cho cho s được triệu tập, xóa đi vùng lưu trữ bị chiếm bởi s.elems. Kể từ đây cho đến khi cấp phát lưu trữ được kể đến thì s ứng xử giống như là biến tự động của một kiểu có sẵn được tạo ra khi phạm vi của nó được biết đến và được hủy đi khi phạm vi của nó được rời khỏi.

Nói chung, hàm xây dựng của đối tượng được áp dụng trước khi đối tượng được thu hồi. Điều này phụ thuộc vào phạm vi của đối tượng. Ví dụ, một đối tượng toàn cục được thu hồi khi sự thực hiện của chương trình hoàn tất; một đối tượng tự động được thu hồi khi toán tử delete được áp dụng tới nó.

7.6. Bạn (Friend)

Đôi khi chúng ta cần cấp quyền truy xuất cho một hàm tới các thành viên không là các thành viên chung của một lớp. Một truy xuất như thế được thực hiện bằng cách khai báo hàm như là bạn của lớp. Có hai lý do có thể cần đến truy xuất này là:

- Có thể là cách định nghĩa hàm chính xác.
- Có thể là cần thiết nếu như hàm cài đặt không hiệu quả.

Các ví dụ của trường hợp đầu sẽ được cung cấp trong chương 8 khi chúng ta thảo luận về tái định nghĩa các toán tử xuất/nhập. Một ví dụ của trường hợp thứ hai được thảo luận bên dưới.

Giả sử rằng chúng ta định nghĩa hai biến thể của lớp Set, một cho tập các số nguyên và một cho tập các số thực:

```
class IntSet {
 public:
      //...
 private:
      int elems[maxCard];
      int card;
};

class RealSet {
 public:
      //...
 private:
      float elems[maxCard];
      int   card;
};
```

Chúng ta muốn định nghĩa một hàm SetToReal để chuyển tập hợp số nguyên thành tập hợp số thực.Chúng ta có thể làm điều này bằng cách để cho hàm SetToReal là một thành viên của IntSet:

```
void IntSet::SetToReal (RealSet &set)
{
    set.EmptySet();
    for (register i = 0; i < card; ++i)
        set.AddElem((float) elems[i]);
}
```

Dẫu cho công việc này có thể thực hiện được nhưng tổn phí của việc gọi hàm AddElem cho mọi thành viên của tập hợp có thể là không thể chấp nhận. Công việc cài đặt có thể được cải thiện nếu chúng ta giành được truy xuất tới các dữ liệu riêng của cả hai IntSet và RealSet. Điều này có thể được giải quyết bằng cách khai báo hàm SetToReal như là bạn của lớp RealSet.

```
class RealSet {
    //...
    friend void IntSet::SetToReal (RealSet&);
};

void IntSet::SetToReal (RealSet &set)
{
    set.card = card;
    for (register i = 0; i < card; ++i)
        set.elems[i] = (float) elems[i];
}
```

Trường hợp để cho tất cả các hàm thành viên của lớp A như là bạn của một lớp B khác có thể được diễn giải trong một hình thức ngắn gọn như sau:

```
class A;
class B {
    //...
    friend class A;        // hình thức ngắn gọn
};
```

Cách khác của việc cài đặt hàm SetToReal là định nghĩa nó như là một hàm toàn cục mà là bạn của cả hai lớp:

```
class IntSet {
    //...
    friend void SetToReal (IntSet&, RealSet&);
};

class RealSet {
    //...
    friend void SetToReal (IntSet&, RealSet&);
};

void SetToReal (IntSet &iSet, RealSet &rSet)
{
    rSet.card = iSet.card;
    for (int i = 0; i < iSet.card; ++i)
        rSet.elems[i] = (float) iSet.elems[i];
}
```

Mặc dù khai báo bạn xuất hiện bên trong một lớp nhưng điều đó không làm cho hàm là một thành viên của lớp đó. Thông thường, vị trí của khai báo bạn trong một lớp là không quan trọng: dù cho nó xuất hiện trong phần chung, riêng, hay được bảo vệ thì đều có cùng nghĩa.

7.7. Đối số mặc định

Như là các hàm toàn cục, một hàm thành viên của một lớp có thể có các đối số mặc định. Ứng dụng luật tương tự, tất cả các đối số mặc định là các đối số ở phần đuôi (bên tay phải), và đối số có thể là một biểu thức gồm nhiều đối tượng được định nghĩa bên trong phạm vi mà lớp xuất hiện.

Ví dụ, một hàm xây dựng cho lớp Point có thể sử dụng các đối số mặc định để cung cấp nhiều cách thức khác nhau cho việc định nghĩa một đối tượng Point :

```
class Point {
        int     xVal, yVal;
public:
                Point (int x = 0, int y = 0);
        //...
};
```

Với hàm xây dựng đã có này thì các định nghĩa sau là hoàn toàn hợp lệ:

```
Point p1;                       // như là: p1(0, 0)
Point p2(10);                   // như là: p2(10, 0)
Point p3(10, 20);
```

Việc sử dụng cẩu thả các đối số mặc định có thể dẫn đến sự tối nghĩa không mong muốn. Ví dụ, với lớp đã cho

```
class Point {
        int     xVal, yVal;
public:
                Point (int x = 0, int y = 0);
                Point (float x = 0, float y = 0);       // tọa độ cực
        //...
};
```

thì định nghĩa sau được xem như là tối nghĩa bởi vì nó so khớp với cả hai hàm xây dựng:

```
Point p;                        // tối nghĩa hay mơ hồ
```

7.8. Đối số thành viên ẩn

Khi một hàm thành viên của lớp được gọi nó nhận một đối số ẩn biểu thị đối tượng cụ thể của lớp mà hàm được triệu gọi. Ví dụ, trong

```
Point pt(10,20);
pt.OffsetPt(2,2);
```

pt là một đối số ẩn cho OffsetPt. Bên trong thân của hàm thành viên tồn tại một con trỏ this tham khảo tới đối số ẩn này. This biểu thị một con trỏ tới đối tượng mà thành viên được triệu gọi. Sử dụng this hàm OffsetPt có thể được viết như sau:

```
Point::OffsetPt (int x, int y)
{
    this->xVal += x;        // tương đương với: xVal += x;
    this->yVal += y;        // tương đương với: yVal += y;
}
```

Việc sử dụng this trong trường hợp này là dư thừa. Tuy nhiên có những trường hợp lập trình trong đó sử dụng con trỏ this là cần thiết. Chúng ta sẽ thấy các ví dụ của những trường hợp như thế trong chương 7 khi thảo luận về tái định nghĩa các toán tử.

Con trỏ this có thể được sử dụng để tham khảo đến các hàm thành viên chính xác như là nó được sử dụng cho các dữ liệu thành viên. Tuy nhiên cần chú ý là con trỏ this được định nghĩa cho việc sử dụng bên trong các hàm thành viên của chỉ một lớp. Cụ thể hơn là nó không định nghĩa cho các hàm toàn cục (bao hàm cả các hàm bạn toàn cục).

7.9. Toán tử phạm vi

Khi gọi một hàm thành viên chúng ta thường sử dụng một cú pháp viết tắt. Ví dụ:

```
pt.OffsetPt(2,2);           // hình thức viết tắt
```

Điều này tương đương với hình thức viết đầy đủ:

```
pt.Point::OffsetPt(2,2);    // hình thức đầy đủ
```

Hình thức đầy đủ sử dụng toán tử phạm vi nhị hạng :: để chỉ định rằng hàm OffsetPt là một thành viên của lớp Point.

Trong một vài tình huống, sử dụng toán tử phạm vi là cần thiết. Ví dụ, trường hợp mà tên của thành viên lớp bị che dấu bởi biến cục bộ (ví dụ, tham số hàm thành viên) có thể được vượt qua bằng cách sử dụng toán tử phạm vi:

```
class Point {
public:
    Point (int x, int y)        { Point::x = x; Point::y = y; }
```

```
    //...
private:
    int x, y;
}
```

Ở đây x và y trong hàm xây dựng (phạm vi bên trong) che đi x và y trong lớp (phạm vi bên ngoài). x và y trong lớp được tham khảo rõ ràng là Point::x và Point::y.

7.10.Danh sách khởi tạo thành viên

Có hai cách khởi tạo các thành viên dữ liệu của một lớp. Tiếp cận đầu tiên liên quan đến việc khởi tạo các thành viên dữ liệu thông qua sử dụng các phép gán trong thân của hàm xây dựng. Ví dụ:

```
class Image {
 public:
        Image       (const int w, const int h);
 private:
    int    width;
    int    height;
  //...
};

Image::Image (const int w, const int h)
{
    width = w;
    height = h;
    //...
}
```

Tiếp cận thứ hai sử dụng một **danh sách khởi tạo thành viên** (member initialization list) trong định nghĩa hàm xây dựng. Ví dụ:

```
class Image {
 public:
        Image       (const int w, const int h);
 private:
    int    width;
    int    height;
  //...
};

Image::Image (const int w, const int h) : width(w), height(h)
{
    //...
}
```

Tác động của khai báo này là width được khởi tạo tới w và height được khởi tạo tới h. Chỉ khác nhau giữa tiếp cận này và tiếp cận trước đó là ở đây các thành viên được khởi tạo *trước khi* thân của hàm xây dựng được thực hiện.

Danh sách khởi tạo thành viên có thể được sử dụng để khởi tạo bất kỳ thành viên dữ liệu nào của một lớp. Nó luôn được đặt giữa phần đầu và phần thân của hàm xây dựng. Một dấu hai chấm (:) được sử dụng để phân biệt nó

với phần đầu. Nó gồm một danh sách các thành viên dữ liệu được phân biệt bằng dấu phẩy (,) mà giá trị khởi tạo của chúng xuất hiện bên trong một cặp dấu ngoặc đơn.

7.11.Thành viên hằng

Một thành viên dữ liệu của lớp có thể được định nghĩa như hằng. Ví dụ:

```
class Image {
        const int    width;
        const int    height;
   //...
};
```

Tuy nhiên, các hằng thành viên dữ liệu không thể được khởi tạo bằng cách sử dụng cùng cú pháp như là đối với các hằng khác:

```
class Image {
        const int    width = 256;         // khởi tạo trái luật
        const int    height = 168;    // khởi tạo trái luật
   //...
};
```

Cách chính xác để khởi tạo một hằng thành viên dữ liệu là thông qua một danh sách khởi tạo thành viên:

```
class Image {
public:
                        Image       (const int w, const int h);
private:
        const int    width;
        const int    height;
   //...
};

Image::Image (const int w, const int h) : width(w), height(h)
{
        //...
}
```

Như là một điều được mong đợi, không có hàm thành viên nào được cho phép gán tới một thành viên dữ liệu hằng.

Một thành viên dữ liệu hằng không thích hợp cho việc định nghĩa kích thước của một thành viên dữ liệu mảng. Ví dụ, trong

```
class Set {
 public:
                    Set(void) : maxCard(10)          { card = 0; }
        //...
 private:
        const maxCard;
        int             elems[maxCard];             // không đúng luật
        int             card;
};
```

mảng elems sẽ bị bát bỏ bởi trình biên dịch. Lý do là maxCard không được ràng buộc tới một giá trị trong thời gian biên dịch mà được ràng buộc khi chương trình chạy và hàm xây dựng được triệu gọi.

Các hàm thành viên cũng có thể được định nghĩa như là hằng. Điều này được sử dụng để đặc tả các hàm thành viên nào của lớp có thể được triệu gọi cho một đối tượng hằng. Ví dụ,

```
class Set {
 public:
                    Set(void){ card = 0; }
       Bool   Member(const int)    const;
       void    AddElem(const int);
       //...
};

Bool Set::Member (const int elem) const
{
       //...
}
```

định nghĩa hàm Member như là một hàm thành viên hằng. Để thực hiện điều đó khóa const được chèn sau phần đầu của hàm ở cả hai bên trong lớp và trong định nghĩa hàm.

Một đối tượng hằng chỉ có thể được sửa đổi bởi các hàm thành viên hằng của lớp:

```
const Set s;
s.AddElem(10);  // trái luật: AddElem không là thành viên hằng
s.Member(10);   // ok
```

Luật cho phép một hàm thành viên hằng được cho phép triệu gọi các đối tượng hằng, nhưng nếu nó cố gắng sửa đổi bất kỳ các thành viên dữ liệu nào của lớp là không đúng luật.

Hàm xây dựng và hàm hủy không bao giờ cần được định nghĩa như các thành viên hằng vì chúng có quyền thao tác trên các đối tượng hằng. Chúng cũng không bị tác động bởi luật trên và có thể gán tới một thành viên dữ liệu của một đối tượng hằng trừ phi thành viên dữ liệu chính nó là một hằng.

7.12.Thành viên tĩnh

Thành viên dữ liệu của một lớp có thể định nghĩa là tĩnh (static). Điều này đảm bảo rằng sẽ có chính xác một bản sao chép của thành viên được chia sẻ bởi tất cả các đối tượng của lớp. Ví dụ, xem xét lớp Window trên một trình bày bản đồ:

```
class Window {
       static Window    *first;// danh sách liên kết tất cả Window
       Window                 *next;        // con trỏ tới window kế tiếp
```

```
    //...
};
```

Ở đây, không quan tâm đến bao nhiêu đối tượng kiểu Window được định nghĩa, sẽ chỉ là một thể hiện của first. Giống như các biến tĩnh khác, một thành viên dữ liệu tĩnh mặc định được khởi tạo là 0. Nó có thể được khởi tạo tới một giá trị tùy ý trong cùng phạm vi mà định nghĩa hàm thành viên xuất hiện:

```
Window *Window::first = &myWindow;
```

Các hàm thành viên cũng có thể được định nghĩa là tĩnh. Về mặc ngữ nghĩa, một hàm thành viên tĩnh giống như là một hàm toàn cục mà là bạn của một lớp nhưng không thể truy xuất bên ngoài lớp. Nó không nhận một đối số ẩn và vì thế không thể tham khảo tới con trỏ this. Các hàm thành viên tĩnh là cần thiết để định nghĩa các thủ tục gọi lại (call-back routines) mà các danh sách tham số của nó được định trước và ngoài phạm vi điều khiển của lập trình viên.

Ví dụ, lớp Window có thể sử dụng một hàm gọi lại để sơn các vùng lộ ra của cửa sổ:

```
class Window {
    //...
    static void PaintProc (Event *event);    // gọi lại
};
```

Bởi vì các hàm tĩnh được chia sẻ và không nhờ vào con trỏ this nên chúng được tham khảo tốt nhất nhờ vào sử dụng cú pháp *class::member*. Ví dụ, first và PaintProc sẽ được tham khảo như Window::first và Window::PaintProc. Các thành viên tĩnh chúng có thể được tham khảo tới thông qua sử dụng cú pháp này bởi các hàm không là thành viên (ví dụ, các hàm toàn cục).

7.13.Thành viên tham chiếu

Thành viên dữ liệu của lớp có thể được định nghĩa như là tham chiếu. Ví dụ:

```
class Image {
    int   width;
    int   height;
    int   &widthRef;
  //...
};
```

Tương tự các hằng thành viên dữ liệu, một tham chiếu thành viên dữ liệu không thể được khởi tạo bằng cách sử dụng cùng cú pháp như đối với các tham chiếu khác:

```
class Image {
    int   width;
    int   height;
    int   &widthRef = width;        // trái luật
  //...
};
```

Cách chính xác để khởi tạo một tham chiếu thành viên dữ liệu là thông qua một danh sách khởi tạo thành viên:

```
class Image {
 public:
      Image(const int w, const int h);
 private:
      int    width;
      int    height;
      int    &widthRef;
  //...
};

Image::Image (const int w, const int h) : widthRef(width)
{
      //...
}
```

Điều này làm cho widthRef trở thành một tham chiếu cho thành viên width.

7.14.Thành viên là đối tượng của một lớp

Thành viên dữ liệu của một lớp có thể là kiểu người dùng định nghĩa, có nghĩa là một đối tượng của một lớp khác. Ví dụ, lớp Rectangle có thể được định nghĩa bằng cách sử dụng hai thành viên dữ liệu Point đại diện cho góc trên bên trái và góc dưới bên phải của hình chữ nhật:

```
class Rectangle {
 public:
              Rectangle (int left, int top, int right, int bottom);
      //...
 private:
      Point topLeft;
      Point botRight;
};
```

Hàm xây dựng cho lớp Rectangle cũng có thể khởi tạo hai thành viên đối tượng của lớp. Giả sử rằng lớp Point có một hàm xây dựng thì điều này được thực hiện bằng cách thêm topLeft và botRight vào danh sách khởi tạo thành viên của hàm xây dựng cho lớp Rectangle:

```
Rectangle::Rectangle (int left, int top, int right, int bottom)
      : topLeft(left,top), botRight(right,bottom)
{
}
```

Nếu hàm xây dựng của lớp Point không có tham số hoặc nếu nó có các đối số mặc định cho tất cả tham số của nó thì danh sách khởi tạo thành viên ở trên có thể được bỏ qua.

Thứ tự khởi tạo thì luôn là như sau. Trước hết hàm xây dựng cho topLeft được triệu gọi và theo sau là hàm xây dựng cho botRight, và cuối cùng là hàm xây dựng cho chính lớp Rectangle. Hàm hủy đối tượng luôn theo hướng ngược

lại. Trước tiên là hàm xây dựng cho lớp Rectangle (nếu có) được triệu gọi, theo sau là hàm hủy cho botRight, và cuối cùng là cho topLeft. Lý do mà topLeft được khởi tạo trước botRight không phải vì nó xuất hiện trước trong danh khởi tạo thành viên mà vì nó xuất hiện trước botRight trong chính lớp đó. Vì thế, định nghĩa hàm xây dựng như sau sẽ không thay đổi thứ tự khởi tạo (hoặc hàm hủy):

```
Rectangle::Rectangle (int left, int top, int right, int bottom)
        : botRight(right,bottom), topLeft(left,top)
{
}
```

7.15.Mảng các đối tượng

Mảng các kiểu người dùng định nghĩa được định nghĩa và sử dụng nhiều theo cùng phương thức như mảng các kiểu xây dựng sẵn. Ví dụ, hình ngũ giác có thể được định nghĩa như mảng của 5 điểm:

```
Point pentagon[5];
```

Định nghĩa này giả sử rằng lớp Point có một hàm xây dựng không đối số (nghĩa là một hàm xây dựng có thể được triệu gọi không cần đối số). Hàm xây dựng được áp dụng tới mỗi phần tử của mảng.

Mảng cũng có thể được khởi tạo bằng cách sử dụng bộ khởi tạo mảng thông thường. Mỗi mục trong danh sách khởi tạo có thể triệu gọi hàm xây dựng với các đối số mong muốn. Khi bộ khởi tạo có ít mục hơn kích thước mảng, các phần tử còn lại được khởi tạo bởi hàm xây dựng không đối số. Ví dụ,

```
Point pentagon[5] = {
        Point(10,20), Point(10,30), Point(20,30), Point(30,20)
};
```

khởi tạo bốn phần tử của mảng pentagon tới các điểm cụ thể, và phần tử sau cùng được khởi tạo tới (0,0).

Khi hàm xây dựng có thể được triệu gọi với một đối số đơn, nó vừa đủ để đặc tả đối số. Ví dụ,

```
Set sets[4] = {10, 20, 20, 30};
```

là một phiên bản ngắn gọn của:

```
Set sets[4] = {Set(10), Set(20), Set(20), Set(30)};
```

Mảng các đối tượng cũng có thể được tạo ra động bằng cách sử dụng toán tử new:

```
Point *petagon = new Point[5];
```

Sau cùng, khi mảng được xóa bằng cách sử dụng toán tử delete thì một cặp dấu ngoặc vuông ([]) nên được chèn vào:

```
delete [] pentagon;        // thu hồi tất cả các phần tử của mảng
```

Nếu không sử dụng cặp [] được chèn vào thì toán tử delete sẽ không có cách nào biết rằng pentagon biểu thị một mảng các điểm chứ không phải là một mảng đơn. Hàm hủy (nếu có) được ứng dụng tới các phần tử của mảng theo thứ tự ngược lại trước khi mảng được xóa. Việc loại bỏ cặp [] sẽ làm cho hàm hủy được áp dụng chỉ tới phần tử đầu tiên của mảng.

```
delete pentagon;           // thu hồi chỉ phần tử đầu tiên!
```

Vì các đối tượng của mảng động không thể được khởi tạo rõ ràng ở thời điểm tạo ra, lớp phải có một hàm xây dựng không đối số để điều khiển việc khởi tạo không tường minh. Khi việc khởi tạo không tường minh này không đủ thông tin thì sau đó lập trình viên có thể khởi tạo lại cụ thể cho từng phần tử của mảng:

```
pentagon[0].Point(10, 20);
pentagon[1].Point(10, 30);
//...
```

Mảng các đối tượng động được sử dụng trong các tình huống mà chúng ta không thể biết trước kích thước của mảng. Ví dụ, một lớp đa giác tổng quát không có cách nào biết được một hình đa giác có chính xác bao nhiêu đỉnh:

```
class Polygon {
public:
    //...
private:
    Point *vertices;        // các đỉnh
    int      nVertices;     // số các đỉnh
};
```

7.16.Phạm vi lớp

Một lớp mở đầu **phạm vi lớp** rất giống với cách một hàm (hay khối) mở đầu một phạm vi cục bộ. Tất cả các thành viên của lớp phụ thuộc vào phạm vi lớp và ẩn đi các thực thể với các tên giống hệt trong phạm vi.Ví dụ, trong

```
int fork (void);           // fork hệ thống

class Process {
    int fork (void);
    //...
};
```

hàm thành viên fork ẩn đi hàm hệ thống toàn cục fork. Hàm thành viên có thể tham khảo tới hàm hệ thống toàn cục bằng cách sử dụng toán tử phạm vi đơn hạng:

```
int Process::fork (void)
```

```
{
    int pid = ::fork();   // sử dụng hàm fork hệ thống toàn cục
    //...
}
```

Lớp chính nó có thể được định nghĩa ở bất kỳ một trong ba phạm vi có thể:

- Ở phạm vi toàn cục. Điều này dẫn tới một **lớp toàn cục** bởi vì nó có thể được tham khảo tới bởi tất cả phạm vi khác. Đại đa số các lớp C++ (kể cả tất cả các ví dụ được trình bày đến thời điểm này) được định nghĩa ở phạm vi toàn cục.
- Ở phạm vi lớp của lớp khác. Điều này dẫn tới một **lớp lồng nhau** trong đó lớp được chứa đựng bởi lớp khác.
- Ở phạm vi cục bộ của một khối hay một hàm. Điều này dẫn đến một **lớp cục bộ** trong đó lớp được chứa đựng hoàn toàn bởi một khối hoặc một hàm.

Lớp lồng nhau là hữu dụng khi một lớp được sử dụng chỉ bởi một lớp khác. Ví dụ,

```
class Rectangle {              // một lớp lồng nhau
 public:
                Rectangle (int, int, int, int);
    //..
 private:
    class Point {
    public:
                Point (int, int);
  private:
        int x, y;
    };
    Point topLeft, botRight;
};
```

định nghĩa lớp Point lồng bên trong lớp Rectangle. Các hàm thành viên của lớp Point có thể được định nghĩa hoặc nội tuyến (inline) ở bên trong lớp Point hoặc ở phạm vi toàn cục. Phạm vi toàn cục sẽ đòi hỏi thêm các tên của hàm thành viên bằng cách đặt trước chúng với Rectangle::

```
Rectangle::Point::Point (int x, int y)
{
    //...
}
```

Một lớp lồng nhau vẫn còn có thể được truy xuất bên ngoài lớp bao bọc của nó bằng cách chỉ định đầy đủ tên lớp. Ví dụ sau là hợp lệ ở bất kỳ phạm vi nào (giả sử rằng Point được tạo ra chung (public) ở bên trong Rectangle):

```
Rectangle::Point pt(1,1);
```

Lớp cục bộ hữu dụng khi một lớp được sử dụng chỉ bởi một hàm – hàm toàn cục hay hàm thành viên – hoặc thậm chí chỉ là một khối. Ví dụ,

```
void Render (Image &image)
{
      class ColorTable {
  public:
            ColorTable      (void)                    {/* ... */}
            AddEntry  (int r, int g, int b)  {/* ... */}
            //...
      };

      ColorTable colors;
      //...
}
```

định nghĩa ColorTable như là một lớp cục bộ tới Render.

Không giống như các lớp lồng nhau, một lớp cục bộ không thể truy xuất bên ngoài phạm vi nó được định nghĩa. Vì thế hàng sau là không hợp lệ ở phạm vi toàn cục:

```
ColorTable ct;          // không được định nghĩa!
```

Một lớp cục bộ phải được định nghĩa đầy đủ bên trong phạm vi mà nó xuất hiện. Vì thế, tất cả các hàm thành viên của nó cần được định nghĩa nội tuyến ở bên trong lớp. Điều này ngụ ý rằng một phạm vi cục bộ không phù hợp cho định nghĩa bất cứ cái gì ngoại trừ các lớp thật là đơn giản.

7.17.Cấu trúc và hợp

Cấu trúc (structure) là tất cả các thành viên của nó được định nghĩa mặc định là chung (public). (Nhớ rằng tất cả các thành viên của lớp được định nghĩa mặc định là riêng (private)). Các cấu trúc được định nghĩa bằng cách sử dụng cùng cú pháp như các lớp ngoại trừ từ khóa struct được sử dụng thay vì class. Ví dụ,

```
struct Point {
            Point(int, int);
      void OffsetPt(int, int);
      int x, y;
};
```

đương đương với:

```
class Point {
  public:
                Point(int, int);
        void OffsetPt(int, int);
        int  x, y;
};
```

Cấu trúc struct được bắt nguồn từ ngôn ngữ C, nó chỉ có thể chứa đựng các thành viên dữ liệu. Nó đã được giữ lại cho khả năng tương thích về sau. Trong C, một cấu trúc có thể có một bộ khởi tạo với cú pháp tương tự như là cú pháp của một mảng. C++ cho phép các bộ khởi tạo như thế dành cho các

cấu trúc và các lớp mà tất cả các thành viên dữ liệu của chúng là chung (public):

```
class Employee {
    public:
        char      *name;
        int       age;
        double    salary;
};

Employee emp = {"Jack", 24, 38952.25};
```

Bộ khởi tạo gồm các giá trị được gán cho các thành viên dữ liệu của cấu trúc (hoặc lớp) theo thứ tự chúng xuất hiện. Các kiểu khởi tạo này phần lớn được thay thế bằng các hàm xây dựng. Vả lại, nó không thể được sử dụng với lớp mà có hàm xây dựng.

Hợp (union) là một lớp mà tất cả các thành viên dữ liệu của nó được ánh xạ tới cùng địa chỉ ở bên trong đối tượng của nó (hơn là liên tiếp như trong trường hợp của lớp). Vì thế kích thước đối tượng của một hợp là kích thước thành viên dữ liệu lớn nhất của nó.

Hợp được sử dụng chủ yếu cho các tình huống mà một đối tượng có thể chiếm lấy các giá trị của các kiểu khác nhưng chỉ một giá trị ở một thời điểm. Ví dụ, xem xét một trình thông dịch cho một ngôn ngữ lập trình đơn giản được gọi là P hỗ trợ cho một số kiểu dữ liệu như là: số nguyên, số thực, chuỗi, và danh sách. Một giá trị trong ngôn ngữ lập trình này có thể được định nghĩa kiểu:

```
union Value {
    long    integer;
    double  real;
    char    *string;
    Pair    list;
    //...
};
```

trong đó Pair chính nó là một kiểu người dùng định nghĩa cho việc tạo ra các danh sách:

```
class Pair {
    Value     *head;
    Value     *tail;
    //...
};
```

Giả sử rằng kiểu long là 4 byte, kiểu double là 8 byte, và con trỏ là 4 byte, đối tượng thuộc kiểu Value có thể chính xác 8 byte, nghĩa là cùng kích thước với kiểu double hay đối tượng kiểu Pair (bằng với hai con trỏ).

Một đối tượng trong ngôn ngữ P có thể được biểu diễn bởi lớp,

```
class Object {
    private:
        enum ObjType {intObj, realObj, strObj, listObj};
```

```
        ObjType type;       // kiểu đối tượng
        Value   val;        // giá trị của đối tượng
        //...
};
```

trong đó type cung cấp cách thức ghi nhận kiểu của giá trị mà đối tượng giữ hiện tại. Ví dụ, khi type được đặt tới strObj, val.string được sử dụng để tham khảo tới giá trị của nó.

Bởi vì chỉ có một cách duy nhất mà các thành viên dữ liệu được ánh xạ tới bộ nhớ nên một hợp không thể có thành viên dữ liệu tĩnh hay thành viên dữ liệu mà yêu cầu một hàm xây dựng.

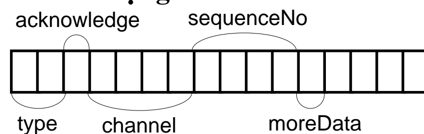
Giống như cấu trúc, tất cả các thành viên của hợp được định nghĩa mặc định là chung (public). Các từ khóa private, public, và protected có thể được sử dụng bên trong struct hoặc union chính xác theo cùng cách mà chúng được sử dụng bên trong một lớp để định nghĩa các thành viên riêng, chung, và được bảo vệ.

7.18.Các trường bit

Đôi khi chúng ta muốn điều khiển trực tiếp một đối tượng ở mức bit sao cho nhiều hạng mục dữ liệu riêng có thể được đóng gói thành một dòng bit mà không còn lo lắng về các biên của từ hay byte.

Ví dụ trong truyền dữ liệu, dữ liệu được truyền theo từng đơn vị rời rạc gọi là các gói tin (packets). Ngoài phần dữ liệu cần truyền thì mỗi gói tin còn chứa đựng một phần header gồm các thông tin về mạng hỗ trợ cho việc quản lý và truyền các gói tin qua mạng. Để làm giảm thiểu chi phí truyền nhận chúng ta mong muốn giảm thiểu không gian chiếm bởi phần header. Hình 7.1 minh họa các trường của header được đóng gói thành các bit gần kề để đạt được mục đích này.

Hình 7.1 Các trường header của một gói.



Các trường này có thể được biểu diễn thành các thành viên dữ liệu **trường bit** của một lớp Packet. Một trường bit có thể được định nghĩa thuộc kiểu int hoặc kiểu unsigned int:

```
typedef unsigned int Bit;

class Packet {
    Bit   type          : 2;     // rộng 2 bit
    Bit acknowledge: 1;          // rộng 1 bit
    Bit channel      : 4;        // rộng 4 bit
    Bit   sequenceNo    : 4;     // rộng 4 bit
```

```
        Bit    moreData        : 1;          // rộng 1 bit
        //...
};
```

Một trường bit được tham khảo giống như là tham khảo tới bất kỳ thành viên dữ liệu nào khác. Bởi vì một trường bit không nhất thiết bắt đầu trên một biến của byte nên việc lấy địa chỉ của nó là không hợp lệ. Với lý do này, một trường bit không được định nghĩa là tĩnh (static).

Sử dụng bảng liệt kê có thể dễ dàng làm việc với các trường bit hơn. Ví dụ, từ bảng liệt kê cho trước

```
enum PacketType {dataPack, controlPack, supervisoryPack};
enum Bool       {false, true};
```

chúng ta có thể viết:

```
Packet p;
p.type = controlPack;
p.acknowledge = true;
```

Bài tập cuối chương 7

7.1 Giải thích tại sao các tham số của các hàm thành viên Set được khai báo như là các tham chiếu.

7.2 Định nghĩa một lớp có tên là Complex để biểu diễn các số phức. Một số phức có hình thức tổng quát là a + bi, trong đó a là phần thực và b là phần ảo (i thay cho ảo). Các quy luật toán học trên số phức như sau:

(a + bi) + (c + di) = (a + c) + (b + d)i
(a + bi) – (c + di) = (a + c) – (b + d)i
(a + bi) * (c + di) = (ac – bd) + (bc + ad)i

Định nghĩa các thao tác này như là các hàm thành viên của lớp Complex.

7.3 Định nghĩa một lớp có tên là Menu sử dụng danh sách liên kết của các chuỗi để biểu diễn menu với nhiều chọn lựa. Sử dụng một lớp lồng nhau tên là Option để biểu diễn tập hợp các phần tử. Định nghĩa một hàm xây dựng, hàm hủy, và các hàm thành viên sau cho lớp Menu:

- Insert chèn một chọn lựa mới vào một vị trí cho trước. Cung cấp một đối số mặc định sao cho mục chọn được nối vào ở điểm cuối.
- Delete xóa một chọn lựa tồn tại.
- Choose hiển thị menu và mời người dùng chọn một chọn lựa.

7.4 Định nghĩa lại lớp Set như là một danh sách liên kết sao cho không có giới hạn về số lượng các phần tử một tập hợp có thể có. Sử dụng một lớp lồng nhau tên là Element để biểu diễn tập hợp các phần tử.

7.5 Định nghĩa một lớp tên là Sequence để lưu trữ các chuỗi đã được sắp xếp. Định nghĩa một hàm xây dựng, một hàm hủy, và các hàm thành viên sau cho lớp Sequence:

- Insert chèn một chuỗi mới vào vị trí sắp xếp của nó.
- Delete xóa một chuỗi hiện có.
- Find tìm tuần tự với một chuỗi cho trước và trả về true nếu tìm được và false nếu không tìm được.
- Print in ra các chuỗi tuần tự.

7.6 Định nghĩa lớp tên là BinTree để lưu trữ các chuỗi đã được sắp xếp như là một cây nhị phân. Định nghĩa cùng tập các hàm thành viên như đối với lớp Sequence ở bài tập trước.

7.7 Định nghĩa một hàm thành viên cho lớp BinTree để chuyển một chuỗi thành cây nhị phân như là bạn của lớp Sequence. Sử dụng hàm này để định nghĩa một hàm xây dựng cho lớp BinTree nhận một chuỗi làm đối số.

7.8 Thêm một thành viên dữ liệu ID là số nguyên vào lớp Menu (Bài tập 7.3) sao cho tất cả các đối tượng menu được đánh số tuần tự bắt đầu từ 0. Định nghĩa một hàm thành viên nội tuyến trả về số ID. Bạn sẽ theo dõi ID cuối cùng được cấp phát như thế nào?

7.9 Sửa đổi lớp Menu sao cho chọn lựa chính nó có thể là một menu, bằng cách ấy cho phép các menu lồng nhau.