Chương 8. Tái định nghĩa

Chương này thảo luận về tái định nghĩa hàm và toán tử trong C++. Thuật ngữ *tái định nghĩa* (overloading) nghĩa là 'cung cấp nhiều định nghĩa'. Tái định nghĩa hàm liên quan đến việc định nghĩa các hàm riêng biệt chia sẻ cùng tên, mỗi hàm có một dấu hiệu duy nhất. Tái định nghĩa hàm thích hợp cho:

- Định nghĩa các hàm về bản chất là làm cùng công việc nhưng thao tác trên các kiểu dữ liệu khác nhau.
- Cung cấp các giao diện tới cùng hàm.

Tái định nghĩa hàm (function overloading) là một tiện lợi trong lập trình.

Giống như các hàm, các toán tử nhận các toán hạng (các đối số) và trả về một giá trị. Phần lớn các toán tử C++ có sẵn đã được tái định nghĩa rồi. Ví dụ, toán tử + có thể được sử dụng để cộng hai số nguyên, hai số thực, hoặc hai địa chỉ. Vì thế, nó có nhiều định nghĩa khác nhau. Các định nghĩa xây dựng sẵn cho các toán tử được giới hạn trên những kiểu có sẵn. Các định nghĩa thêm vào có thể được cung cấp bởi các lập trình viên sao cho chúng cũng có thể thao tác trên các kiểu người dùng định nghĩa. Mỗi định nghĩa thêm vào được cài đặt bởi một hàm.

Tái định nghĩa các toán tử sẽ được minh họa bằng cách sử dụng một số lớp đơn giản. Chúng ta sẽ thảo luận các qui luật chuyển kiểu có thể được sử dụng như thế nào để rút gọn nhu cầu cho nhiều tái định nghĩa của cùng toán tử. Chúng ta sẽ trình bày các ví dụ của tái định nghĩa một số toán tử phổ biến gồm << và >> cho xuất nhập, [] và () cho các lớp chứa, và các toán tử con trỏ. Chúng ta cũng sẽ thảo luận việc khởi tạo và gán tự động, tầm quan trọng của việc cài đặt chính xác chúng trong các lớp sử dụng các thành viên dữ liệu được cấp phát động.

Không giống như các hàm và các toán tử, các lớp không thể được tái định nghĩa; mỗi lớp phải có một tên duy nhất. Tuy nhiên, như chúng ta sẽ thấy trong chương 8, các lớp có thể được sửa đổi và mở rộng thông qua khả năng *thừa kế* (inheritance).

8.1. Tái định nghĩa hàm

Xem xét một hàm, GetTime, trả về thời gian hiện tại của ngày theo các tham số của nó, và giả sử rằng cần có hai biến thể của hàm này: một trả về thời gian theo giây tính từ nửa đêm, và một trả về thời gian theo giờ, phút, giây. Rõ ràng các hàm này phục vụ cùng mục đích nên không có lý do gì lại để cho chúng có những cái tên khác nhau.

C++ cho phép các hàm được tái định nghĩa, nghĩa là cùng hàm có thể có hơn một định nghĩa:

```
long GetTime (void);                    // số giây tính từ nửa đêm
void GetTime (int &hours, int &minutes, int &seconds);
```

Khi hàm GetTime được gọi, trình biên dịch so sánh số lượng và kiểu các đối số trong lời gọi với các định nghĩa của hàm GetTime và chọn một cái khớp với lời gọi. Ví dụ:

```
int h, m, s;
long t = GetTime();             // khớp với GetTime(void)
GetTime(h, m, s);               // khớp với GetTime(int&, int&, int&);
```

Để tránh nhầm lẫn thì mỗi định nghĩa của một hàm được tái định nghĩa phải có một dấu hiệu duy nhất.

Các hàm thành viên của một lớp cũng có thể được tái định nghĩa:

```
class Time {
      //...
      long GetTime (void);                  // số giây tính từ nửa đêm
      void GetTime (int &hours, int &minutes, int &seconds);
};
```

Tái định nghĩa hàm giúp ta thu được nhiều phiên bản đa dạng của hàm mà không thể có được bằng cách sử dụng đơn độc các đối số mặc định. Các hàm được tái định nghĩa cũng có thể có các đối số mặc định:

```
void Error (int errCode, char *errMsg = "");
void Error (char *errMsg);
```

8.2. Tái định nghĩa toán tử

C++ cho phép lập trình viên định nghĩa các ý nghĩa thêm vào cho các toán tử xác định trước của nó bằng cách tái định nghĩa chúng. Ví dụ, chúng ta có thể tái định nghĩa các toán tử + và – để cộng và trừ các đối tượng Point:

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

```
    Point operator + (Point &p)  {return Point(x + p.x,y + p.y);}
    Point operator - (Point &p)  {return Point(x - p.x,y - p.y);}
 private:
    int x, y;
};
```

Sau định nghĩa này thì + và – có thể được sử dụng để cộng và trừ các điểm giống như là chúng được sử dụng để cộng và trừ các số:

```
Point p1(10,20), p2(10,20);
Point p3 = p1 + p2;
Point p4 = p1 - p2;
```

Việc tái định nghĩa các toán tử + và – như trên sử dụng các hàm thành viên. Một khả năng khác là một toán tử có thể được tái định nghĩa toàn cục:

```
class Point {
 public:
    Point (int x, int y)        {Point::x = x; Point::y = y;}
    friend Point operator + (Point &p, Point &q)
                                    {return Point(p.x + q.x,p.y + q.y);}
    friend Point operator - (Point &p, Point &q)
                                    {return Point(p.x - q.x,p.y - q.y);}
 private:
    int x, y;
};
```

Sử dụng một toán tử đã tái định nghĩa tương đương với một lời gọi rõ ràng tới hàm thi công nó. Ví dụ:

```
operator+(p1, p2)            // tương đương với: p1 + p2
```

Thông thường, để định nghĩa một toán tử λ xác định trước thì chúng ta định nghĩa một hàm tên operator λ . Nếu λ là một toán tử nhị hạng:

- operator λ phải nhận chính xác một đối số nếu được định nghĩa như một thành viên của lớp, hoặc hai đối số nếu được định nghĩa toàn cục.

Tuy nhiên, nếu λ là một toán tử đơn hạng:

- operator λ phải nhận không đối số nếu được định nghĩa như một thành viên của lớp, hoặc một đối số nếu được định nghĩa toàn cục.

Bảng 8.1 tổng kết các toán tử C++ có thể được tái định nghĩa. Năm toán tử còn lại không được tái định nghĩa là:

. .* :: ?: sizeof

Bảng 8.1 Các toán tử có thể tái định nghĩa.

Đơn hạng	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Nhị hạng	+	-	*	/	%	&	|	^	<<	>>	
	=	+=	-=	/=	%=	&=	|=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&	||	[]	()	,

Toán tử đơn hạng (ví dụ ~) không thể được tái định nghĩa như nhị hạng hoặc toán tử nhị hạng (ví dụ =) không thể được tái định nghĩa như toán tử đơn hạng.

C++ không hỗ trợ định nghĩa toán tử new bởi vì điều này có thể dẫn đến sự mơ hồ. Hơn nữa, luật ưu tiên cho các toán tử xác định trước cố định và không thể được sửa đổi. Ví dụ, dù cho bạn tái định nghĩa toán tử * như thế nào thì nó sẽ luôn có độ ưu tiên cao hơn toán tử +.

Các toán tử ++ và — có thể được tái định nghĩa như là tiền tố cũng như là hậu tố. Các luật tương đương không được áp dụng cho các toán tử đã tái định nghĩa. Ví dụ, tái định nghĩa + không ảnh hưởng tới += trừ phi toán tử += cũng được tái định nghĩa rõ ràng. Các toán tử ->, =, [], và () chỉ có thể được tái định nghĩa như các hàm thành viên, và không như toàn cục.

Để tránh sao chép các đối tượng lớn khi truyền chúng tới các toán tử đã tái định nghĩa thì các tham chiếu nên được sử dụng. Các con trỏ thì không thích hợp cho mục đích này bởi vì một toán tử đã được tái định nghĩa không thể thao tác toàn bộ trên con trỏ.

Ví dụ: Các toán tử trên tập hợp

Lớp Set được giới thiệu trong chương 6. Phần lớn các hàm thành viên của Set được định nghĩa như là các toán tử tái định nghĩa tốt hơn. Danh sách 8.1 minh họa.

Danh sách 8.1

```
1  #include <iostream.h>
2  const maxCard = 100;
3  enum       Bool {false, true};
4  class Set {
5   public:
6                              Set(void) { card = 0; }
7         friend Bool operator &  (const int, Set&);      // thanh vien
8         friend Bool operator == (Set&, Set&);           // bang
9         friend Bool operator != (Set&, Set&);           // khong bang
10        friend Set  operator *  (Set&, Set&);           // giao
11        friend Set  operator +  (Set&, Set&);           // hop
12        //...
13        void   AddElem(const int elem);
14        void   Copy   (Set &set);
15    void       Print  (void);
16  private:
17        int            elems[maxCard];                  // cac phan tu cua tap hop
18        int            card;                            // so phan tu cua tap hop
19 };
```

Ở đây, chúng ta phải quyết định định nghĩa các hàm thành viên toán tử như là bạn toàn cục. Chúng có thể được định nghĩa một cách dễ dàng như là hàm thành viên. Việc thi công các hàm này là như sau.

```
Bool operator & (const int elem, Set &set)
{
      for (register i = 0; i < set.card; ++i)
          if (elem == set.elems[i])
                return true;
      return false;
}

Bool operator == (Set &set1, Set &set2)
{
      if (set1.card != set2.card)
          return false;
      for (register i = 0; i < set1.card; ++i)
          if (!(set1.elems[i] & set2))      // sử dụng & đã tái định nghĩa
                return false;
      return true;
}

Bool operator != (Set &set1, Set &set2)
{
      return !(set1 == set2);              // sử dụng == đã tái định nghĩa
}

Set operator * (Set &set1, Set &set2)
{
      Set res;

      for (register i = 0; i < set1.card; ++i)
          if (set1.elems[i] & set2)        // sử dụng & đã tái định nghĩa
                res.elems[res.card++] = set1.elems[i];
```

```
    return res;
}

Set operator + (Set &set1, Set &set2)
{
        Set res;

        set1.Copy(res);
        for (register i = 0; i < set2.card; ++i)
            res.AddElem(set2.elems[i]);
        return res;
}
```

Cú pháp để sử dụng các toán tử này ngắn gọn hơn cú pháp của các hàm mà chúng thay thế như được minh họa bởi hàm main sau:

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

        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();

        if (20 & s1) cout << "20 thuoc s1\n";

        cout << "s1 giao s2 = ";     (s1 * s2).Print();
        cout << "s1 hop s2 = ";      (s1 + s2).Print();

        if (s1 != s2) cout << "s1 /= s2\n";
        return 0;
}
```

Khi chạy chương trình sẽ cho kết quả sau:

```
s1 = {10,20,30,40}
s2 = {30,50,10,60}
20 thuoc s1
s1 giao s2 = {10,30}
s1 hop s2 = {10,20,30,40,50,60}
s1 /= s2
```

8.3. Chuyển kiểu

Các luật chuyển kiểu thông thường có sẵn của ngôn ngữ cũng áp dụng tới các hàm và các toán tử đã tái định nghĩa. Ví dụ, trong

```
if ('a' & set)
        //...
```

toán hạng đầu của & (nghĩa là 'a') được chuyển kiểu ẩn từ char sang int, bởi vì toán tử & đã tái định nghĩa mong đợi toán hạng đầu của nó thuộc kiểu int.

Bất kỳ sự chuyển kiểu nào khác thêm vào phải được định nghĩa bởi lập trình viên. Ví dụ, giả sử chúng ta muốn tái định nghĩa toán tử + cho kiểu Point sao cho nó có thể được sử dụng để cộng hai điểm hoặc cộng một số nguyên tới cả hai tọa độ của một điểm:

```
class Point
        //...
        friend Point operator + (Point, Point);
        friend Point operator + (int, Point);
        friend Point operator + (Point, int);
};
```

Để làm cho toán tử + có tính giao hoán, chúng ta phải định nghĩa hai hàm để cộng một số nguyên với một điểm: một hàm đối với trường hợp số nguyên là toán hạng đầu tiên và một hàm đối với trường hợp số nguyên là toán hạng thứ hai. Quan sát rằng nếu chúng ta bắt đầu xem xét các kiểu khác thêm vào kiểu int thì tiếp cận này dẫn đến mức độ biến đổi khó kiểm soát của toán tử.

Một tiếp cận tốt hơn là sử dụng hàm xây dựng để chuyển đối tượng tới cùng kiểu như chính lớp sao cho một toán tử đã tái định nghĩa có thể điều khiển công việc. Trong trường hợp này, chúng ta cần một hàm xây dựng nhận một int đặc tả cả hai tọa độ của một điểm:

```
class Point {
        //...
        Point (int x) { Point::x = Point::y = x; }
        friend Point operator + (Point, Point);
};
```

Đối với các hàm xây dựng của một đối số thì không cần gọi hàm xây dựng một cách rõ ràng:

```
Point p = 10;                    // tương đương với: Point p(10);
```

Vì thế có thể viết các biểu thức liên quan đến các biến hoặc hằng thuộc kiểu Point và int bằng cách sử dụng toán tử +.

```
Point p(10,20), q = 0;
q = p + 5;                       // tương đương với: q = p + Point(5);
```

Ở đây, 5 được chuyển tạm thời thành đối tượng Point và sau đó được cộng vào p. Đối tượng tạm sau đó sẽ được hủy đi. Tác động toàn bộ là một chuyển kiểu không tường minh từ int thành Point. Vì thế giá trị cuối của q là (15,25).

Cái gì xảy ra nếu chúng ta muốn thực hiện chuyển kiểu ngược lại từ kiểu lớp thành một kiểu khác? Trong trường hợp này các hàm xây dựng không thể được sử dụng bởi vì chúng luôn trả về một đối tượng của lớp mà chúng thuộc về. Để thay thế, một lớp có thể định nghĩa một hàm thành viên mà chuyển rõ ràng một đối tượng thành một kiểu mong muốn.

Ví dụ, với lớp Rectangle đã cho chúng ta có thể định nghĩa một hàm chuyển kiểu thực hiện chuyển một hình chữ nhật thành một điểm bằng cách tái định nghĩa toán tử kiểu Point trong lớp Rectangle:

```
class Rectangle {
 public:
      Rectangle (int left, int top, int right, int bottom);
  Rectangle (Point &p, Point &q);
      //...
  operator Point ()      {return botRight - topLeft;}

 private:
      Point topLeft;
      Point botRight;
};
```

Toán tử này được định nghĩa để chuyển một hình chữ nhật thành một điểm mà tọa độ của nó tiêu biểu cho độ rộng và chiều cao của hình chữ nhật. Vì thế, trong đoạn mã

```
Point       p(5,5);
Rectangle  r(10,10,20,30);
r+p;
```

trước hết hình chữ nhật r được chuyển *không tường minh* thành một đối tượng Point bởi toán tử chuyển kiểu và sau đó được cộng vào p.

Chuyển kiểu Point cũng có thể được áp dụng *tường minh* bằng cách sử dụng ký hiệu ép kiểu thông thường. Ví dụ:

```
Point(r);          // ép kiểu tường minh thành Point
(Point)r;          // ép kiểu tường minh thành Point
```

Thông thường với kiểu người dùng định nghĩa X đã cho và kiểu Y khác (có sẵn hay người dùng định nghĩa) thì:

- Hàm xây dựng được định nghĩa cho X nhận một đối số đơn kiểu Y sẽ chuyển không tường minh các đối tượng Y thành các đối tượng X khi được cần.
- Tái định nghĩa toán tử Y trong X sẽ chuyển không tường minh các đối tượng X thành các đối tượng Y khi được cần.

```
class X {
      //...
      X (Y&);              // chuyển Y thành X
      operator Y ();     // chuyển X thành Y
};
```

Một trong những bất lợi của các phương thức chuyển kiểu do người dùng định nghĩa là nếu chúng không được sử dụng một cách hạn chế thì chúng có thể làm cho các hoạt động của chương trình là khó có thể tiên đoán. Cũng có sự rủi ro thêm vào của việc tạo ra sự mơ hồ. Sự mơ hồ xảy ra khi trình biên

dịch có hơn một chọn lựa cho nó để áp dụng các qui luật chuyển kiểu người dùng định nghĩa và vì thế không thể chọn được. Tất cả những trường hợp như thế được báo cáo như những lỗi bởi trình biên dịch.

Để minh họa cho các mơ hồ có thể xảy ra, giả sử rằng chúng ta cũng định nghĩa một hàm chuyển kiểu cho lớp Rectangle (nhận một đối số Point) cũng như là tái định nghĩa các toán tử + và -:

```
class Rectangle {
 public:
      Rectangle (int left, int top, int right, int bottom);
   Rectangle (Point &p, Point &q);
      Rectangle (Point &p);

   operator Point () {return botRight - topLeft;}
      friend Rectangle operator + (Rectangle &r, Rectangle &t);
      friend Rectangle operator - (Rectangle &r, Rectangle &t);

 private:
      Point topLeft;
      Point botRight;
};
```

Bây giờ, trong

```
Point       p(5,5);
Rectangle  r(10,10,20,30);
r + p;
```

r + p có thể được thông dịch theo hai cách. Hoặc là

```
r + Rectangle(p)            // cho ra một Rectangle
```

hoặc là:

```
Point(r) + p                // cho ra một Point
```

Nếu lập trình viên không giải quyết sự mơ hồ bởi việc chuyển kiểu tường minh thì trình biên dịch sẽ từ chối.

Ví dụ: Lớp Số Nhị Phân

Danh sách 8.2 định nghĩa một lớp tiêu biểu cho các số nguyên nhị phân như là một chuỗi các ký tự 0 và 1.

Danh sách 8.2

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

3  int const binSize = 16;          // chieu dai so nhi phan la 16

4  class Binary {
5   public:
6                  Binary (const char*);
7                      Binary (unsigned int);
8     friend     Binary       operator + (const Binary, const Binary);
9                      operator int ();              // chuyen kieu
10         void  Print (void);
11  private:
12               char   bits[binSize];                  // cac bit nhi phan
13 };
```

Chú giải

6 Hàm xây dựng này cung cấp một số nhị phân từ mẫu bit của nó.

7 Hàm xây dựng này chuyển một số nguyên dương thành biểu diễn nhị phân tương đương của nó.

8 Toán tử + được tái định nghĩa để cộng hai số nhị phân. Phép cộng được làm từng bit một. Để đơn giản thì những lỗi tràn được bỏ qua.

9 Toán tử chuyển kiểu này được sử dụng để chuyển một đối tượng Binary thành đối tượng int .

10 Hàm này đơn giản chỉ in mẫu bit của số nhị phân.

12 Mảng này được sử dụng để giữ các bit 0 và 1 của số lượng 1 bit như là các ký tự.

Cài đặt các hàm này là như sau:

```
Binary::Binary (const char *num)
{
      int iSrc = strlen(num) - 1;
      int iDest = binSize - 1;

      while (iSrc >= 0 && iDest >= 0)          // sao chep cac bit
        bits[iDest--] = (num[iSrc--] == '0' ? '0' : '1');
      while (iDest >= 0)                                  // dat cac bit trai ve 0
        bits[iDest--] = '0';
}

Binary::Binary (unsigned int num)
{
      for (register i = binSize - 1; i >= 0; --i) {
        bits[i] = (num % 2 == 0 ? '0' : '1');
        num >>= 1;
      }
}

Binary operator + (const Binary n1, const Binary n2)
{
      unsigned carry = 0;
```

```
        unsigned value;
        Binary res = "0";

        for (register i = binSize - 1; i >= 0; --i) {
          value = (n1.bits[i] == '0' ? 0 : 1) +
                (n2.bits[i] == '0' ? 0 : 1) + carry;
          res.bits[i] = (value % 2 == 0 ? '0' : '1');
          carry = value >> 1;
        }
        return res;
    }

    Binary::operator int ()
    {
        unsigned value = 0;

        for (register i = 0; i < binSize; ++i)
          value = (value << 1) + (bits[i] == '0' ? 0 : 1);
        return value;
    }

    void Binary::Print (void)
    {
        char str[binSize + 1];
        strncpy(str, bits, binSize);
        str[binSize] = '\0';
        cout << str << '\n';
    }
```

Hàm main sau tạo ra hai đối tượng kiểu Binary và kiểm tra toán tử + .

```
main ()
{
    Binary n1 = "01011";
    Binary n2 = "11010";
    n1.Print();
    n2.Print();
    (n1 + n2).Print();
    cout << n1 + Binary(5) << '\n';   // cong va chuyen thanh int
    cout << n1 - 5 << '\n';         // chuyen n2 thanh int và tru
}
```

Hai hàng cuối của hàm main ứng xử hoàn toàn khác nhau. Hàng đầu của hai hàng này chuyển 5 thành Binary, thực hiện cộng, và sau đó chuyển kết quả Binary thành int trước khi gởi nó đến dòng xuất cout. Điều này tương đương với:

```
cout << (int) Binary::operator+(n2,Binary(5)) << '\n';
```

Hàng thứ hai trong hai hàng này chuyển n1 thành int (bởi vì toán tử - không được định nghĩa cho Binary), thực hiện trừ, và sau đó gởi kết quả đến dòng xuất cout. Điều này tương đương với:

```
cout << ((int) n2) - 5 << '\n';
```

Trong trường hợp này thì toán tử chuyển kiểu được áp dụng không tường minh. Kết quả cho bởi chương trình là bằng chứng cho các chuyển kiểu được thực hiện chính xác:

```
0000000000001011
0000000000011010
0000000000100101
16
6
```

8.4. Tái định nghĩa toán tử xuất <<

Việc xuất đồng bộ và đơn giản cho các kiểu có sẵn được mở rộng dễ dàng cho các kiểu người dùng định nghĩa bằng cách tái định nghĩa thêm nửa toán tử <<. Đối với bất kỳ kiểu người dùng định nghĩa T, chúng ta có thể định nghĩa một hàm operator<< để xuất các đối tượng kiểu T:

```
ostream& operator<<(ostream&, T&);
```

Tham số đầu phải là một tham chiếu tới dòng xuất ostream sao cho có nhiều sử dụng của << có thể nối vào nhau. Tham số thứ hai không cần là một tham chiếu nhưng điều này lại hiệu quả cho các đối tượng có kích thước lớn.

Ví dụ, thay vì hàm thành viên Print của lớp Binary chúng ta có thể tái định nghĩa toán tử << cho lớp. Bởi vì toán hạng đầu của toán tử << phải là một đối tượng ostream nên nó không thể được tái định nghĩa như là một hàm thành viên. Vì thế nó được định nghĩa như là hàm toàn cục:

```
class Binary {
    //...
    friend      ostream& operator<<(ostream&, Binary&);
};

ostream& operator<<(ostream &os, Binary &n)
{
    char str[binSize + 1];
    strncpy(str, n.bits, binSize);
    str[binSize] = '\0';
    cout << str;
    return os;
}
```

Từ định nghĩa đã cho, << có thể được định nghĩa cho xuất các số nhị phân theo cách giống như các kiểu có sẵn. Ví dụ:

```
Binary n1 = "01011", n2 = "11010";
cout << n1 << " + " << n2 << " = " << n1 + n2 << '\n';
```

sẽ cho ra kết quả sau:

```
0000000000001011 + 0000000000011010 = 0000000000100101
```

Với cách thức đơn giản, kiểu xuất này loại bỏ đi gánh nặng của việc nhớ tên hàm xuất đối với mỗi kiểu người dùng định nghĩa. Trong trường hợp không sử dụng tái định nghĩa << thì ví dụ cuối có thể được viết như sau: (giả sử rằng \n đã được xóa từ hàm Print):

```
Binary n1 = "01011", n2 = "11010";
n1.Print();    cout << " + ";   n2.Print();
cout << " = "; (n1 + n2).Print(); cout << '\n';
```

8.5. Tái định nghĩa toán tử nhập >>

Việc nhập các kiểu người dùng định nghĩa được làm cho dễ dàng bằng cách tái định nghĩa toán tử >> theo cùng cách với << được tái định nghĩa. Đối với bất kỳ kiểu người dùng định nghĩa T chúng ta có thể định nghĩa một hàm operator>> nhập các đối tượng kiểu T:

```
istream& operator >> (istream&, T&);
```

Tham số đầu tiên phải là một tham chiếu tới dòng nhập istream sao cho sử dụng nhiều >> có thể được nối vào nhau. Tham số thứ hai phải là một tham chiếu vì nó sẽ được sửa đổi bởi hàm.

Tiếp theo lớp Binary chúng ta tái định nghĩa toán tử >> để nhập vào một chuỗi các bit. Nhắc lại, bởi vì toán hạng đầu tiên của toán tử >> phải là một đối tượng istream nên nó không thể được tái định nghĩa như là một hàm thành viên:

```
class Binary {
      //...
      friend       istream& operator >> (istream&, Binary&);
};

istream& operator >> (istream &is, Binary &n)
{
      char str[binSize + 1];
      cin >> str;
  n = Binary(str);              // use the constructor for simplicity
      return is;
}
```

Với định nghĩa đã cho này thì toán tử >> có thể được sử dụng để nhập vào các số nhị phân theo cách của các kiểu dữ liệu có sẵn. Ví dụ,

```
Binary n;
cin >> n;
```

sẽ đọc một số nhị phân từ bàn phím tới n.

8.6. Tái định nghĩa []

Danh sách 8.3 định nghĩa một lớp vectơ kết hợp đơn giản. Một vectơ kết hợp là một mảng một chiều mà các phần tử có thể được tìm kiếm bằng nội dung của chúng hơn là vị trí của chúng trong mảng. Trong AssocVec thì mỗi phần tử có một tên dạng chuỗi (thông qua đó nó có thể được tìm kiếm) và một giá trị số nguyên kết hợp.

Danh sách 8.3

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

3  class AssocVec {
4   public:
5                   AssocVec  (const int dim);
6                   ~AssocVec       (void);
7       int& operator [] (const char *idx);
8   private:
9       struct VecElem {
10          char  *index;
11          int           value;
12      }   *elems;                 // cac phan tu cua vecto
13      int   dim;                  // kich thuoc cua vecto
14      int   used;                 // cac phan tu duoc su dung toi hien tai
15 };
```

Chú giải

5 Hàm xây dựng tạo ra một vectơ kết hợp có kích cỡ được chỉ định bởi tham số của nó.

7 Toán tử [] đã tái định nghĩa được sử dụng để truy xuất các phần tử của vectơ. Hàm tái định nghĩa [] phải có chính xác một tham số. Với một chuỗi đã cho nó tìm kiếm phần tử tương ứng chứa trong vectơ. Nếu một việc so khớp chỉ số được tìm thấy thì sau đó một tham chiếu tới giá trị kết hợp với nó được trả về. Ngược lại, một phần tử mới được tạo ra và một tham chiếu tới giá trị này được trả về.

12 Các phần tử vectơ được biểu diễn bởi một mảng động của các cấu trúc VecElem. Mỗi phần tử của vectơ gồm một chuỗi (được biểu thị bởi index) và một giá trị số nguyên (được biểu thị bởi value).

Thi công của các hàm này như sau:

```
AssocVec::AssocVec (const int dim)
{
        AssocVec::dim = dim;
        used = 0;
        elems = new VecElem[dim];
}

AssocVec::~AssocVec (void)
{
```

```
    for (register i = 0; i < used; ++i)
        delete elems[i].index;
    delete [] elems;
}

int& AssocVec::operator [] (const char *idx)
{
    for (register i = 0; i < used; ++i)  // tim phan tu ton tai
        if (strcmp(idx,elems[i].index) == 0)
            return elems[i].value;

    if (used < dim &&                    // tao ra phan tu moi
      (elems[used].index = new char[strlen(idx)+1]) != 0) {
        strcpy(elems[used].index,idx);
        elems[used].value = used + 1;
        return elems[used++].value;
    }
    static int dummy = 0;
    return dummy;
}
```

Chú ý rằng bởi vì AssocVec::operator[] phải trả về một tham chiếu hợp lệ, một tham chiếu tới một số nguyên tĩnh giả được trả về khi vectơ đầy hay toán tử new thất bại.

Một biểu thức tham chiếu là một giá trị trái và vì thế có thể xuất hiện trên cả hai phía của một phép gán. Nếu một hàm trả về một tham chiếu sau đó một lời gọi hàm tới hàm đó có thể được gán tới. Điều này là tại sao kiểu trả về của AssocVec::operator[] được định nghĩa là một tham chiếu.

Sử dụng AssocVec chúng ta bây giờ có thể tạo ra các vectơ kết hợp mà xử lý rất giống các vectơ bình thường:

```
AssocVec count(5);
count["apple"] = 5;
count["orange"] = 10;
count["fruit"] = count["apple"] + count["orange"];
```

Điều này sẽ đặt count["fruit"] tới 15.

8.7. Tái định nghĩa ()

Danh sách 8.4 định nghĩa một lớp ma trận. Một ma trận là một bảng các giá trị (mảng hai chiều) mà kích thước của nó được biểu thị bởi số hàng và số cột trong bảng. Một ví dụ của ma trận đơn giản 2 x 3 sẽ là:

$$M = \begin{bmatrix} 10 & 20 & 30 \\ 21 & 52 & 19 \end{bmatrix}$$

Ký hiệu toán học chuẩn để tham khảo các phần tử của ma trận là các dấu ngoặc. Ví dụ phần tử 20 của ma trận *M* (nghĩa là trong hàng đầu và cột thứ hai) được tham khảo tới như là *M*(1,2). Đại số học của ma trận cung cấp một tập các thao tác để cài đặt ma trận bao gồm cộng, trừ, nhân, và chia.

Danh sách 8.4

```
1   #include <iostream.h>

2   class Matrix {
3       public:
4           Matrix      (const short rows, const short cols);
5           ~Matrix     (void)      {delete elems;}
6           double&     operator () (const short row, const short col);
7         friend    ostream&   operator << (ostream&, Matrix&);
8         friend  Matrix   operator  + (Matrix&, Matrix&);
9         friend  Matrix   operator  - (Matrix&, Matrix&);
10        friend  Matrix   operator  * (Matrix&, Matrix&);

11      private:
12          const short rows;  // số hàng của ma trận
13          const short cols;   // số cột của ma trận
14          double              *elems;      // các phần tử của ma trận
15  };
```

Chú giải

4 Hàm xây dựng tạo ra một ma trận có kích cỡ được chỉ định bởi các tham số của nó, tất cả các phần tử của nó được khởi tạo là 0.

6 Toán từ() đã tái định nghĩa được sử dụng để truy xuất các phần tử của ma trận. Hàm tái định nghĩa toán tử () có thể không có hay có nhiều tham số. Nó trả về một tham chiếu tới giá trị của phần tử được chỉ định.

7 Toán tử << đã tái định nghĩa được sử dụng để in một ma trận theo hình thức bảng.

8-10 Các toán tử đã tái định nghĩa này cung cấp các thao tác trên ma trận.

14 Các phần tử của ma trận được biểu diễn bởi một mảng động kiểu double.

Việc cài đặt của ba hàm đầu tiên như sau:

```
Matrix::Matrix (const short r, const short c) : rows(r), cols(c)
{
    elems = new double[rows * cols];
}

double& Matrix::operator () (const short row, const short col)
{
    static  double dummy = 0.0;
    return (row >= 1 && row <= rows && col >= 1 && col <= cols)
            ? elems[(row - 1)*cols + (col - 1)]
            : dummy;
}

ostream& operator << (ostream &os, Matrix &m)
{
```

```
    for (register r = 1; r <= m.rows; ++r) {
        for (int c = 1; c <= m.cols; ++c)
            os << m(r,c) << " ";
        os << '\n';
    }
    return os;
}
```

Như trước bởi vì Matrix::operator() phải trả về một tham chiếu hợp lệ, một tham chiếu tới một số thực double tĩnh giả được trả về khi phần tử được chỉ định không tồn tại. Đoạn mã sau minh họa rằng các phần tử của ma trận là các giá trị trái:

```
Matrix m(2,3);
m(1,1) = 10;        m(1,2) = 20;        m(1,3) = 30;
m(2,1) = 15;        m(2,2) = 25;        m(2,3) = 35;
cout << m << '\n';
```

Điều này sẽ cho kết quả sau:

```
10   20   30
15   25   35
```

8.8. Khởi tạo ngầm định

Hãy xem xét định nghĩa của toán tử + đã tái định nghĩa cho lớp Matrix sau:

```
Matrix operator + (Matrix &p, Matrix &q)
{
    Matrix m(p.rows, p.cols);
    if (p.rows == q.rows && p.cols == q.cols)
        for (register r = 1; r <= p.rows; ++r)
            for (register c = 1; c <= p.cols; ++c)
                m(r,c) = p(r,c) + q(r,c);
    return m;
}
```

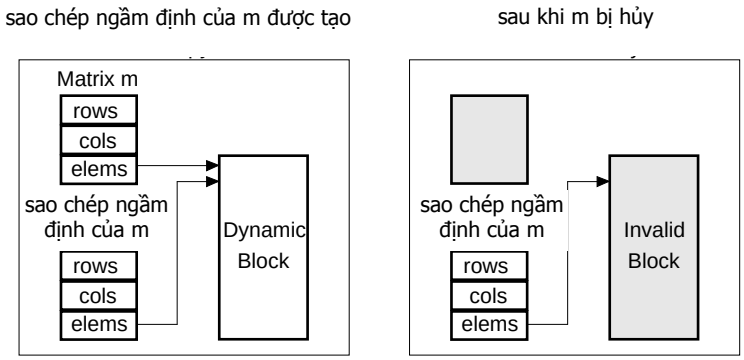
Hàm sau trả về một đối tượng Matrix được khởi tạo tới m. Việc khởi tạo được điều khiển bởi một hàm xây dựng bên trong do trình biên dịch tự động phát ra cho lớp Matrix:

```
Matrix::Matrix (const Matrix &m) : rows(m.rows), cols(m.cols)
{
    elems = m.elems;
}
```

Hình thức khởi tạo này được gọi là **khởi tạo ngầm định** bởi vì hàm xây dựng đặc biệt khởi tạo từng thành viên một của đối tượng. Nếu chính các thành viên dữ liệu của đối tượng đang được khởi tạo lại là các đối tượng của lớp khác thì sau đó chúng cũng được khởi tạo ngầm định.

Kết quả của việc khởi tạo ngầm định là các thành viên dữ liệu elems của cả hai đối tượng sẽ trỏ tới cùng khối đã được cấp phát động. Tuy nhiên m được hủy nhờ vào trả về của hàm. Do đó các hàm hủy xóa đi khối đã được trỏ tới bởi m.elems bỏ lại thành viên dữ liệu của đối tượng đã trả về đang trỏ tới một khối không hợp lệ! Cuối cùng điều này dẫn đến một thất bại trong khi thực thi chương trình. Hình 8.2 minh họa.

Hình 8.2 Lỗi của việc khởi tạo ngầm định



Khởi tạo ngầm định xảy ra trong các tình huống sau:

- Khi định nghĩa và khởi tạo một đối tượng trong một câu lệnh khai báo mà sử dụng đối tượng khác như là bộ khởi tạo của nó, ví dụ lệnh khởi tạo Matrix n = m trong hàm Foo bên dưới.
- Khi truyền một đối số là đối tượng đến một hàm (không có thể dùng được đối số con trỏ hay tham chiếu), ví dụ m trong hàm Foo bên dưới.
- Khi trả về một giá trị đối tượng từ một hàm (không có thể dùng được đối số con trỏ hay tham chiếu), ví dụ return n trong hàm Foo bên dưới.

```
Matrix Foo (Matrix m) // sao chép ngầm định tới m
{
    Matrix n = m;     // sao chép ngầm định tới n
    //...
    return n;         // sao chép ngầm định n và trả về sao chép
}
```

Rõ ràng việc khởi tạo ngầm định là thích hợp cho các lớp không có các thành viên dữ liệu con trỏ (ví dụ, lớp Point). Các vấn đề gây ra bởi khởi tạo ngầm định của các lớp khác có thể được tránh bằng cách định nghĩa các hàm xây dựng phụ trách công việc khởi tạo ngầm định một cách rõ ràng. Hàm xây dựng này còn được gọi là hàm xây dựng sao chép. Đối với bất kỳ lớp X đã cho thì hàm xây dựng sao chép luôn có hình thức:

```
X::X (const X&);
```

Ví dụ với lớp Matrix thì điều này có thể được định nghĩa như sau:

```
class Matrix {
```

```
    Matrix (const Matrix&);
    //...
};

Matrix::Matrix (const Matrix &m) : rows(m.rows), cols(m.cols)
{
    int n = rows * cols;
    elems = new double[n];                    // cùng kích thước
    for (register i = 0; i < n; ++i)          // sao chép các phần tử
        elems[i] = m.elems[i];
}
```

8.9. Gán ngầm định

Các đối tượng thuộc cùng lớp được gán tới một lớp khác bởi một tái định nghĩa toán tử gán bên trong mà được phát ra tự động bởi trình biên dịch. Ví dụ để điều khiển phép gán trong

```
Matrix m(2,2), n(2,2);
//...
m = n;
```

trình biên dịch tự động phát ra một hàm bên trong như sau:

```
Matrix& Matrix::operator = (const Matrix &m)
{
    rows = m.rows;
    cols = m.cols;
    elems = m.elems;
}
```

Điều này giống y hệt như trong việc khởi tạo ngầm định và được gọi là **gán ngầm định**. Nó cũng có cùng vấn đề như trong khởi tạo ngầm định và có thể khắc phục bằng cách tái định nghĩa toán tử = một cách rõ ràng. Ví dụ đối với lớp Matrix thì việc tái định nghĩa toán tử = sau đây là thích hợp:

```
Matrix& Matrix::operator = (const Matrix &m)
{
    if (rows == m.rows && cols == m.cols) {     // phải khớp
        int n = rows * cols;
        for (register i = 0; i < n; ++i)   // sao chép các phần tử
            elems[i] = m.elems[i];
    }
    return *this;
}
```

Thông thường, đối với bất kỳ lớp X đã cho thì toán tử = được tái định nghĩa bằng thành viên sau của X:

```
X& X::operator = (X&)
```

Toán tử = chỉ có thể được tái định nghĩa như là thành viên và không thể được định nghĩa toàn cục.

8.10.Tái định nghĩa new và delete

Các đối tượng khác nhau thường có kích thước và tần số sử dụng khác nhau. Kết quả là chúng có những yêu cầu bộ nhớ khác nhau. Cụ thể các đối tượng nhỏ không được điều khiển một cách hiệu quả bởi các phiên bản mặc định của toán tử new và delete. Mọi khối được cấp phát bởi toán tử new giữ một vài phí được dùng cho mục đích quản lý. Đối với các đối tượng lớn thì điều này không đáng kể nhưng đối với các đối tượng nhỏ thì phí này có thể lớn hơn chính các khối. Hơn nữa, có quá nhiều khối nhỏ có thể làm chậm chạp dữ dội cho các cấp phát và thu hồi theo sau. Hiệu suất của chương trình bằng cách tạo ra nhiều khối nhỏ tự động có thể được cải thiện đáng kể bởi việc sử dụng một chiến lược quản lý bộ nhớ đơn giản hơn cho các đối tượng này.

Các toán tử quản lý lưu trữ động new và delete có thể được tái định nghĩa cho một lớp bằng cách viết chồng lên định nghĩa toàn cục của các toán tử này khi được sử dụng cho các đối tượng của lớp đó.

Ví dụ giả sử chúng ta muốn tái định nghĩa toán tử new và delete cho lớp Point sao cho các đối tượng Point được cấp phát từ một mảng:

```
#include <stddef.h>
#include <iostream.h>

const int maxPoints = 512;

class Point {
 public:
      //...
      void* operator new          (size_t bytes);
      void  operator delete    (void *ptr, size_t bytes);
 private:
      int xVal, yVal;

      static union Block {
            int      xy[2];
            Block *next;
  }                 *blocks;    // tro toi cac luu tru ranh
      static Block *freeList;  // ds ranh cua cac khoi da lien ket
      static int     used;              // cac khoi duoc dung
};
```

Tên kiểu size_t được định nghĩa trong stddef.h.. Toán tử new sẽ luôn trả về void*. Tham số của new là kích thước của khối được cấp phát (tính theo byte). Đối số tương ứng luôn được truyền một cách tự động tới trình biên dịch. Tham số đầu của toán tử delete là khối được xóa. Tham số hai (tùy chọn) là

kích thước khối đã cấp phát. Các đối số được truyền một cách tự động tới trình biên dịch.

Vì các khối, freeList và used là tĩnh nên chúng không ảnh hưởng đến kích thước của đối tượng Point. Những khối này được khởi tạo như sau:

```
Point::Block *Point::blocks = new Block[maxPoints];
Point::Block *Point::freeList = 0;
int Point::used = 0;
```

Toán tử new nhận khối có sẵn kế tiếp từ blocks và trả về địa chỉ của nó. Toán tử delete giải phóng một khối bằng cách chèn nó trước danh sách liên kết được biểu diễn bởi freeList. Khi used đạt tới maxPoints, new trả về 0 khi danh sách liên kết là rỗng, ngược lại new trả về khối đầu tiên trong danh sách liên kết.

```
void* Point::operator new (size_t bytes)
{
    Block *res = freeList;
    return used < maxPoints
                ? &(blocks[used++])
                : (res == 0 ? 0
                            : (freeList = freeList->next, res));
}

void Point::operator delete (void *ptr, size_t bytes)
{
    ((Block*) ptr)->next = freeList;
    freeList = (Block*) ptr;
}
```

Point::operator new và Point::operator delete được triệu gọi chỉ cho các đối tượng Point. Lời gọi new với bất kỳ đối số kiểu khác sẽ triệu gọi định nghĩa toàn cục của new, thậm chí nếu lời gọi xảy ra bên trong một hàm thành viên của Point. Ví dụ:

```
Point *pt = new Point(1,1);     // gọi Point::operator new
char  *str = new char[10];      // gọi ::operator new
delete pt;                          // gọi Point::operator delete
delete str;                         // gọi ::operator delete
```

Khi new và delete được tái định nghĩa cho một lớp, new và delete toàn cục cũng có thể được sử dụng khi tạo và hủy mảng các đối tượng:

```
Point *points = new Point[5];   // gọi ::operator new
//...
delete [] points;               // gọi ::operator delete
```

Toán tử new được triệu gọi trước khi đối tượng được xây dựng trong khi toán tử delete được gọi sau khi đối tượng đã được hủy.

8.11.Tái định nghĩa ++ và --

Các toán tử tăng và giảm một cũng có thể được tái định nghĩa theo cả hai hình thức tiền tố và hậu tố. Để phân biệt giữa hai hình thức này thì phiên bản hậu tố được đặc tả để nhận một đối số nguyên phụ. Ví dụ, các phiên bản tiền tố và hậu tố của toán tử ++ có thể được tái định nghĩa cho lớp Binary như sau:

```
class Binary {
      //...
      friend      Binary      operator ++      (Binary&);              // tien to
      friend      Binary      operator ++      (Binary&, int);         // hau to
};
```

Mặc dù chúng ta phải chọn định nghĩa các phiên bản này như là các hàm bạn toàn cục nhưng chúng cũng có thể được định nghĩa như là các hàm thành viên. Cả hai được định nghĩa dễ dàng theo thuật ngữ của toán tử + đã được định nghĩa trước đó:

```
Binary operator ++ (Binary &n)              // tien to
{
      return n = n + Binary(1);
}

Binary operator ++ (Binary &n, int)         // hau to
{
      Binary m = n;
      n = n + Binary(1);
      return m;
}
```

Chú ý rằng chúng ta đơn giản đã phớt lờ tham số phụ của phiên bản hậu tố. Khi toán tử này được sử dụng thì trình biên dịch tự động cung cấp một đối số mặc định cho nó.

Đoạn mã sau thực hiện cả hai phiên bản của toán tử:

```
Binary n1 = "01011";
Binary n2 = "11010";
cout << ++n1 << '\n';
cout << n2++ << '\n';
cout << n2  << '\n';
```

Nó sẽ cho kết quả sau:

```
0000000000001100
0000000000011010
0000000000011011
```

Các phiên bản tiền tố và hậu số của toán tử -- có thể được tái định nghĩa theo cùng cách này.

Bài tập cuối chương 8

8.1 Viết các phiên bản tái định nghĩa của hàm Max để so sánh hai số nguyên, hai số thực, hoặc hai chuỗi, và trả về thành phần lớn hơn.

8.2 Tái định nghĩa hai toán tử sau cho lớp Set:

- Toán tử - cho hiệu của các tập hợp (ví dụ, *s* - *t* cho một tập hợp gồm các phần tử thuộc *s* mà không thuộc *t*).
- Toán tử <= kiểm tra một tập hợp có chứa trong một tập hợp khác hay không (ví dụ, *s* <= *t* là true nếu tất cả các phần tử thuộc *s* cũng thuộc *t*).

8.3 Tái định nghĩa hai toán tử sau đây cho lớp Binary:

- Toán tử - cho hiệu của hai giá trị nhị phân. Để đơn giản, giả sử rằng toán hạng đầu tiên luôn lớn hơn toán hạng thứ hai.
- Toán tử [] lấy chỉ số một bit thông qua vị trí của nó và trả về giá trị của nó như là một số nguyên 0 hoặc 1.

8.4 Các ma trận thưa được sử dụng trong một số phương thức số (ví dụ, phân tích phần tử có hạn). Một ma trận thưa là một ma trận có đại đa số các phần tử của nó là 0. Trong thực tế, các ma trận thưa có kích thước lên đến 500 × 500 là bình thường. Trên một máy sử dụng biểu diễn 64 bit cho các số thực, lưu trữ một ma trận như thế như một mảng sẽ yêu cầu 2 megabytes lưu trữ. Một biểu diễn kinh tế hơn sẽ chỉ cần ghi nhận các phần tử khác 0 cùng với các vị trí của chúng trong ma trận. Định nghĩa một lớp SparseMatrix sử dụng một danh sách liên kết để ghi nhận chỉ các phần tử khác 0, và tái định nghĩa các toán tử +, -, và * cho nó. Cũng định nghĩa một hàm xây dựng khởi tạo ngầm định và một toán tử khởi tạo ngầm định cho lớp.

8.5 Hoàn tất việc cài đặt của lớp String. Chú ý rằng hai phiên bản của hàm xây dựng ngầm định và toán tử = ngầm định được đòi hỏi, một cho khởi tạo hoặc gán tới một chuỗi bằng cách sử dụng char*, và một cho khởi tạo hoặc gán ngầm định. Toán tử [] nên chỉ mục một ký tự chuỗi bằng cách sử dụng vị trí của nó. Toán tử + cho phép nối hai chuỗi vào nhau.

```
class String {
    public:
        String      (const char*);
        String      (const String&);
        String      (const short);
        ~String     (void);

        String&     operator =(const char*);
        String&     operator =(const String&);
        char&       operator [](const short);
        int         Length(void)    {return(len);}
        friend      String operator +(const String&, const String&);
      friend  ostream& operator << (ostream&, String&);
```

```
    private:
        char        *chars;    // cac ky tu chuoi
        short len;             // chieu dai cua chuoi
};
```

8.6 Một véctơ bit là một véctơ với các phần tử nhị phân, nghĩa là mỗi phần tử có giá trị hoặc là 0 hoặc là 1. Các véctơ bit nhỏ được biểu diễn thuận tiện bằng các số nguyên không dấu. Ví dụ, một unsigned char có thể bằng một véctơ bit 8 phần tử. Các véctơ bit lớn hơn có thể được định nghĩa như mảng của các véctơ bit nhỏ hơn. Hoàn tất sự thi công của lớp Bitvec, như được định nghĩa bên dưới. Nên cho phép các véctơ bit của bất kỳ kích thước được tạo ra và được thao tác bằng cách sử dụng các toán tử kết hợp.

```
enum Bool {false, true};
typedef unsigned char uchar;

class BitVec {
 public:
                BitVec                  (const short dim);
                BitVec                  (const char* bits);
                BitVec                  (const BitVec&);
                ~BitVec             (void){ delete vec; }
    BitVec& operator =          (const BitVec&);
    BitVec& operator &=         (const BitVec&);
    BitVec& operator |= (const BitVec&);
    BitVec& operator ^=         (const BitVec&);
    BitVec& operator <<=        (const short);
    BitVec& operator >>=        (const short);
    int   operator [] (const short idx);
    void        Set                 (const short idx);
    void        Reset                   (const short idx);

    BitVec  operator ~          (void);
    BitVec  operator &          (const BitVec&);
    BitVec  operator |          (const BitVec&);
    BitVec  operator ^          (const BitVec&);
    BitVec  operator << (const short n);
    BitVec  operator >> (const short n);
    Bool        operator ==         (const BitVec&);
    Bool        operator !=             (const BitVec&);

  friend  ostream& operator << (ostream&, BitVec&);
 private:
    uchar     *vec;
    short       bytes;
};
```