CHƯƠNG 2

KIỂU DỮ LIỆU TRỪU TƯỢNG VÀ CÁC LỚP C + +

Mục đích của chương này là trình bày khái niệm lớp và các thành phần của lớp trong C + +. Sự trình bày sẽ không đi vào chi tiết, mà chỉ đề cập tới các vấn đề quan trọng liên quan tới các thành phần của lớp giúp cho bạn đọc dễ dàng hơn trong việc thiết kế các lớp khi cài đặt các KDLTT. Chương này cũng trình bày khái niệm lớp khuôn, lớp khuôn được sử dụng để cài đặt các lớp côngtơnơ. Cuối chương chúng ta sẽ giới thiệu các KDLTT quan trọng sẽ được nghiên cứu kỹ trong các chương sau.

2.1 LỚP VÀ CÁC THÀNH PHẦN CỦA LỚP

Các ngôn ngữ lập trình định hướng đối tượng, chẳng hạn C + +, cung cấp các phương tiện cho phép đóng gói CTDL và các hàm thao tác trên CTDL thành một đơn vị được gọi là lớp (class). Ví dụ, sau đây là định nghĩa lớp số phức:

```
class       Complex
{
 public :
        (1)  Complex (double a = 0.0 , double b = 0.0) ;
        (2)  Complex (const Complex & c);
        (3)  double GetReal ( )  const ;
        (4)  double GetImag ( )  const ;
        (5)  double GetAbs ( )  const ;
        (6)  friend Complex & operator + (const Complex & c1,
```

```
                                                const   Complex & c2) ;
        (7)   friend Complex & operator - (const Complex & c1,
                                                const Complex & c2) ;
        (8)   friend Complex & operator * (const Complex & c1,
                                                const Complex & c2) ;
        (9)   friend Complex & operator / (const Complex & c1,
                                                const Complex & c2) ;
       (10)   friend ostream & operator << (ostream & os,
                                                const Complex & c);
              // Các mẫu hàm cho các phép toán khác.
    private:
            double  real ;
            double  imag ;
};
```

Từ ví dụ đơn giản trên, chúng ta thấy rằng, một lớp bắt đầu bởi **đầu lớp**: đầu lớp gồm từ khoá class, rồi đến tên lớp. Phần còn lại trong định nghĩa lớp (nằm giữa cặp dấu { và }) là **danh sách thành phần**. Danh sách thành phần gồm các **thành phần dữ liệu (data member**), hay còn gọi là **biến thành phần (member variable**), chẳng hạn lớp Complex có hai biến thành phần là real và imag. Các thành phần (1) – (5) trong lớp Complex là các **hàm thành phần (member functions hoặc methods).**

Một lớp là một kiểu dữ liệu, ví dụ khai báo lớp Complex như trên, có nghĩa là người lập trình đã xác định một kiểu dữ liệu Complex. Các đối tượng dữ liệu thuộc một lớp được gọi là các **đối tượng (objects).**

Các thành phần của lớp điển hình được chia thành hai mục: mục public và mục private như trong định nghĩa lớp Complex. Trong chương trình, người lập trình có thể sử dụng trực tiếp các thành phần trong mục public để tiến hành các thao tác trên các đối tượng của lớp. Các thành phần trong mục private chỉ được phép sử dụng trong nội bộ lớp. Mục public (mục private) có thể chứa các hàm thành phần và các biến thành phần. Tuy nhiên,

khi cần thiết kế một lớp cài đặt một KDLTT, chúng ta nên đưa các biến thành phần mô tả CTDL vào mục private, còn các hàm biểu diễn các phép toán vào mục public. Trong định nghĩa lớp Complex cài đặt KDLTT số phức, chúng ta đã làm như thế.

Nên biết rằng, các thành phần của lớp có thể khai báo là tĩnh bằng cách đặt từ khoá static ở trước. Trong một lớp, chúng ta có thể khai báo các hằng tĩnh, các biến thành phần tĩnh, các hàm thành phần tĩnh. Chẳng hạn:

```
static const int CAPACITY = 50; // khai báo hằng tĩnh
static double static Var; // khai báo biến tĩnh
```

Các **thành phần tĩnh** là các thành phần được dùng chung cho tất cả các đối tượng của lớp. Trong lớp Complex không có thành phần nào cần phải là tĩnh.

Nếu khai báo của hàm trong một lớp bắt đầu bởi từ khoá friend, thì hàm được nói là bạn của lớp, chẳng hạn các hàm (6) – (10) trong lớp Complex. Một **hàm bạn (friend function**) không phải là hàm thành phần, song nó được phép truy cập tới các thành phần dữ liệu trong mục private của lớp.

Một hàm thành phần mà khai báo của nó có từ khoá const ở sau cùng được gọi là **hàm thành phần hằng (const member function**). Một hàm thành phần hằng có thể xem xét trạng thái của đối tượng, song không được phép thay đổi nó. Chẳng hạn, các hàm (3), (4), (5) trong lớp Complex. Các hàm này khi áp dụng vào một số phức, không làm thay đổi số phức mà chỉ cho ra phần thực, phần ảo và mođun của số phức, tương ứng.

2.2 CÁC HÀM THÀNH PHẦN

Trong mục này chúng ta sẽ xem xét một số đặc điểm của hàm thành phần.

2.2.1 Hàm kiến tạo và hàm huỷ

Một chương trình áp dụng sử dụng đến các lớp (cần nhớ rằng lớp là một kiểu dữ liệu) sẽ tiến hành một dãy các thao tác trên các đối tượng được khai báo và được tạo ra ban đầu. Do đó, trong một lớp cần có một số hàm thành phần thực hiện công việc khởi tạo ra các đối tượng. Các hàm thành phần này được gọi là **hàm kiến tạo (constructor**). Hàm kiến tạo có đặc điểm là tên của nó trùng với tên lớp và không có kiểu trả về, chẳng hạn hàm (1), (2) trong lớp Complex.

Nếu trong một lớp, bạn không định nghĩa một hàm kiến tạo, thì chương trình dịch sẽ tự động tạo ra một **hàm kiến tạo mặc định tự động (automatic default constructor**). Hàm này chỉ tạo ra đối tượng với tất cả các thành phần dữ liệu đều bằng 0. Nói chung, rất ít khi người ta thiết kế một lớp không có hàm kiến tạo. Đặc biệt khi bạn thiết kế một lớp có chứa thành phần dữ liệu là đối tượng của một lớp khác, thì nhất thiết bạn phải viết hàm kiến tạo.

Một loại hàm kiến tạo đặc biệt có tên gọi là **hàm kiến tạo copy** (**copy constructor**). Nhiệm vụ của hàm kiến tạo copy là khởi tạo ra một đối tượng mới là bản sao của một đối tượng đã có. Ví dụ, hàm (2) trong lớp Complex là hàm kiến tạo copy. Hàm kiến tạo copy chỉ có một tham biến tham chiếu hằng có kiểu là kiểu lớp đang định nghĩa.

Nếu bạn không đưa vào một hàm kiến tạo copy trong định nghĩa lớp, thì chương trình dịch sẽ tự động tạo ra một **hàm kiến tạo copy tự động** (**automatic copy constructor**). Nó thực hiện sao chép tất cả các thành phần dữ liệu của đối tượng đã có sang đối tượng đang khởi tạo. Nói chung, trong nhiều trường hợp chỉ cần sử dụng hàm kiến tạo copy tự động là đủ. Chẳng hạn, trong lớp Complex, thực ra không cần có hàm kiến tạo copy (2). Song trong trường hợp lớp chứa các biến thành phần là biến con trỏ, thì cần thiết phải thiết kế hàm kiến tạo copy cho lớp. (Tại sao?)

Sau đây là một số ví dụ sử dụng hàm kiến tạo trong khai báo các đối tượng thuộc lớp Complex:

Complex c_1; // khởi tạo số phức c_1 với c_1.real = 0.0 và c_1.imag = 0.0

Complex c_2(2.6); // khởi tạo số phức c_2 với c_2.real = 2.6
// và c_2.imag = 0.0
Complex c_3(5.4, 3.7); // khởi tạo số phức c_3 với c_3.real =5.4
// và c_3.imag = 3.7
Complex c_4 = c_2; // khởi tạo số phức c_4 là copy của c_2.

Ngược lại với hàm kiến tạo là **hàm huỷ (destructor**). Hàm huỷ thực hiện nhiệm vụ huỷ đối tượng (thu hồi vùng nhớ cấp phát cho đối tượng và trả lại cho hệ thống), khi đối tượng không cần thiết cho chương trình nữa. Hàm huỷ là hàm thành phần có tên trùng với tên lớp, không có tham biến và phía trước có dấu ngã ~. Hàm huỷ tự động được gọi khi đối tượng ra khỏi phạm vi của nó. Trong một định nghĩa lớp chỉ có thể có một hàm huỷ. Nói chung, trong một lớp không cần thiết phải đưa vào hàm huỷ (chẳng hạn, lớp Complex), trừ trường hợp lớp chứa thành phần dữ liệu là con trỏ trỏ tới vùng nhớ cấp phát động.

2.2.2 Các tham biến của hàm

Các hàm thành phần của một lớp cũng như các hàm thông thường khác có một danh sách các tham biến (danh sách này có thể rỗng) được liệt kê sau tên hàm trong khai báo hàm. Các tham biến này được gọi là **tham biến hình thức (formal parameter**). Khi gọi hàm, các tham biến hình thức được thay thế bởi các **đối số (argument**) hay còn gọi là các **tham biến thực tế (actual parameter).**

Chúng ta xem xét ba loại tham biến:

- **Tham biến giá trị**: Tham biến giá trị (**value parameter**) được khai báo bằng cách viết tên kiểu theo sau là tên tham biến. Chẳng hạn, trong hàm kiến tạo của lớp Complex:

  Complex (double a = 0.0, double b = 0.0) ;

  thì a và b là các tham biến giá trị. Trong khai báo trên chúng ta đã xác định các đối số mặc định (default argument) cho các tham biến a

và b, chúng đều là 0.0. Khi chúng ta gọi hàm kiến tạo không đưa vào đối số, thì có nghĩa là đã gọi hàm kiến tạo với đối số mặc định. Ví dụ, khi ta khai báo Complex c ; thì số phức c được khởi tạo bằng gọi hàm kiến tạo với các đối số mặc định (số phức c có phần thực và phần ảo đều là 0.0).

- **Tham biến tham chiếu**: Tham biến tham chiếu (**reference parameter**) được khai báo bằng cách viết tên kiểu theo sau là dấu & rồi đến tên tham biến. Chẳng hạn, chúng ta có thể thiết kế hàm cộng hai số phức như sau:

void Add (Complex c_1, Complex c_2, Complex & c) ;

Trong hàm Add này, c_1 và c_2 là tham biến giá trị kiểu Complex, còn c là tham biến tham chiếu kiểu Complex.

Để hiểu được sự khác nhau giữa tham biến giá trị và tham biến tham chiếu, bạn cần biết cơ chế thực hiện một lời gọi hàm trong bộ nhớ của máy tính. Mỗi khi một hàm được gọi trong chương trình thì một vùng nhớ dành cho sự thực hiện hàm có tên gọi là **bản ghi hoạt động** (**activation record**) được tạo ra trên vùng nhớ ngăn xếp thời gian chạy (run-time staek). Bản ghi hoạt động ngoài việc chứa bộ nhớ cho các biến địa phương trong hàm, nó còn lưu bản sao của các đối số ứng với các tham biến giá trị và chỉ dẫn tới các đối số ứng với các tham biến tham chiếu. Như vậy, khi thực hiện một lời gọi hàm, các đối số ứng với tham biến giá trị sẽ được copy vào bản ghi hoạt động, còn các đối số ứng với tham biến tham biến thì không cần copy. Khi hoàn thành sự thực hiện hàm, thì bản ghi hoạt động được trả về cho ngăn xếp thời gian chạy. Do đó, sau khi thực hiện hàm, đối số ứng với tham biến giá trị không thay đổi giá trị vốn có của nó, còn đối số ứng với các tham biến tham chiếu vẫn lưu lại kết quả của các tính toán khi thực hiện hàm. Bởi vậy, các tham biến ghi lại kết quả của sự thực hiện hàm cần được khai báo là tham biến tham chiếu.

Trong hàm Add tham biến c ghi lại tổng của số phức c_1 và c_2, nên nó đã được khai báo là tham biến tham chiếu.

Trên đây là một cách sử dụng toán tử tham chiếu (&): nó được sử dụng để khai báo các tham biến tham chiếu. Một cách sử dụng khác của toán tử tham chiếu là để khai báo **kiểu trả về tham chiếu** (**reference return type**) cho một hàm. Ví dụ, chúng ta có thể thiết kế một hàm thực hiện phép cộng số phức một cách khác như sau:

Complex & Add (Complex c_1, Complex c_2) ;

Khai báo kiểu trả về của một hàm là kiểu trả về tham chiếu khi nào? Cần lưu ý rằng, khi thực hiện một hàm, giá trị trả về của hàm được lưu trong một biến địa phương, rồi mệnh đề return sẽ trả về một copy của biến này cho chương trình gọi hàm. Bởi vậy, khi đối tượng trả về của một hàm là lớn, để tránh phải copy từ ngăn xếp thời gian chạy, kiểu trả về của hàm đó nên được khai báo là kiểu trả về tham chiếu.

- **Tham biến tham chiếu hằng**: Như trên đã nói, tham biến tham chiếu ưu việt hơn tham biến giá trị ở chỗ khi thực hiện một hàm, đối số ứng với tham biến tham chiếu không cần phải copy vào ngăn xếp thời gian chạy, nhưng giá trị của nó có thể bị thay đổi, trong khi đó giá trị của đối số ứng với tham biến giá trị không thay đổi khi thực hiện hàm. Kết hợp tính hiệu quả của tham biến tham chiếu và tính an toàn của tham biến giá trị, người ta đưa vào loại tham biến tham chiếu hằng. Để xác định một tham biến tham chiếu hằng (**const reference parameter**), chúng ta chỉ cần đặt từ khoá const trước khai báo tham biến tham chiếu. Đối với tham biến tham chiếu hằng, trong thân hàm chúng ta chỉ có thể tham khảo nó, mọi hành động làm thay đổi giá trị của nó đều không được phép. Khi mà tham biến giá trị có kiểu dữ liệu lớn, để cho hiệu quả chúng ta có thể sử dụng tham biến tham chiếu hằng để thay thế.

Ví dụ, bạn có thể khai báo một hàm tính tổng của hai số phức như sau:

Complex & Add (const Complex & c_1, const Complex & c_2) ;

Trong hàm Add này, c_1 và c_2 là hai tham biến tham chiếu hằng, do đó trong thân của hàm chỉ được phép đọc c_1, c_2, không được phép làm thay đổi chúng.

2.2.3 Định nghĩa lại các phép toán

Giả sử trong định nghĩa lớp Complex, chúng ta xác định các hàm tính tổng và tích của hai số phức như sau:

Complex & Add (const Complex & c_1, const Complex & c_2) ; Complex & Multiply (const Complex & c_1, const Complex & c_2) ;

Khi đó trong chương trình muốn lấy số phức A cộng với tích của số phức B và số phức C, ta cần viết:

D = Add (A, Multiply (B, C)) ;

Cách viết này rất không sáng sủa, nhất là đối với các tính toán phức tạp hơn trên các số phức.

Chúng ta mong muốn biểu diễn các tính toán trên các số phức trong chương trình bởi các biểu thức toán học. Chẳng hạn, dòng lệnh trên, nếu được viết thành:

D = A + B * C ;

thì chương trình sẽ trở nên sáng sủa hơn, dễ đọc, dễ hiểu hơn. Sử dụng các công cụ mà C + + cung cấp, chúng ta có thể làm được điều đó.

Trong ngôn ngữ lập trình C + + có rất nhiều các phép toán (toán tử). Chẳng hạn, các phép toán số học +, - , * , / , % ; các phép toán so sánh = =, != , < , <= , > , >= , các toán tử gán và rất nhiều các phép toán khác. Các phép toán này có ngữ nghĩa đã được xác định trong ngôn ngữ. Chúng ta muốn sử dụng các ký hiệu phép toán trong C + +, nhưng với ngữ nghĩa hoàn

toàn mới, chẳng hạn chúng ta muốn sử dụng ký hiệu + để chỉ phép cộng số phức hoặc phép cộng vectơ hoặc phép cộng ma trận … Việc xác định lại ngữ nghĩa của các phép toán (toán tử) trên các lớp đối tượng dữ liệu mới sẽ được gọi là **định nghĩa lại các phép toán (operator overloading).**

Các phép toán được định nghĩa lại bởi các hàm có tên hàm bắt đầu bởi từ khoá operator và đi sau là ký hiệu phép toán, chúng ta sẽ gọi các hàm này là **hàm toán tử**. Ví dụ, chúng ta có thể định nghĩa lại phép toán + cho các số phức. Có ba cách định nghĩa phép toán cộng số phức bởi hàm toán tử operator +

- **Hàm toán tử không phải là hàm thành phần của lớp Complex**:

```
Complex & Operator + ( const Complex & c1, const Complex & c2);
{
   double  x , y ;
   x = c1. GetReal ( ) + c2. GetReal ( ) ;
   y = c1. GetImag ( ) + c2. GetImag ( ) ;
   Complex  c(x,y) ;
   return  c ;
}
```

Khi đó, trong chương trình muốn cộng hai số phức, ta có thể viết như sau:

```
   Complex A (3.5, 2.7) ;
   Complex B (-4.3, 5.8) ;
   Complex C ;
   C = A + B ;
```

Cũng có thể viết C = operator + (A, B), nhưng không nên sử dụng cách này.

- **Hàm toán tử là hàm thành phần của lớp Complex**

```
Complex & Complex :: operator + (const Complex & c)
{
  Complex temp ;
  temp.real = real + c.real ;
  temp.imag = imag + c. imag ;
  return temp ;
}
```

Trong cách này, khi ta viết C = A + B, thì toán hạng thứ nhất (số phức A) là đối tượng kích hoạt hàm toán tử, tức là

C = A.operator + (B).

- **Hàm toán tử là hàm bạn của lớp Complex**. Đây là cách mà chúng ta đã lựa chọn trong định nghĩa lớp Complex (xem mục 2.1.). Hàm bạn này được cài đặt như sau:

```
Complex & operator + (const Complex & c1, const Complex & c2);
{
  Complex  sum ;
  sum.real = c1.real + c2.real ;
  sum.imag = c1.imag + c2.imag ;
  return  sum ;
}
```

Sử dụng hàm toán tử là bạn giống như sử dụng hàm toán tử không phải thành phần của lớp.

Có sự khác nhau tinh tế giữa hàm toán tử thành phần và hàm toán tử bạn. Ví dụ, giả sử A và B là hai số phức, và hàm operator + là hàm bạn của lớp Complex. Khi đó, câu lệnh:

A = 1 + B ;

được chương trình dịch xem là:

A = operator+ (1,B) ;

và để thực hiện, 1 được chuyển thành đối tượng Complex với phần thực bằng 1, phần ảo bằng 0 bởi toán tử chuyển kiểu được xác định trong lớp Complex, rồi được cộng với số phức A.

Chúng ta xét xem điều gì sẽ xảy ra khi hàm toán tử operator + là hàm thành phần của lớp Complex. Trong trường hợp này, chương trình dịch sẽ minh họa A = 1 + B như là

A = 1. operator (B) ;

Nhưng 1 không phải là đối tượng của lớp Complex, do đó nó không thể kích hoạt một hàm thành phần của lớp Complex. Điều này dẫn tới lỗi!

Vì những lý do trên, khi thiết kế một lớp cài đặt một KDLTT thì các phép toán hai toán hạng (chẳng hạn, các phép cộng, trừ, nhân, chia số phức) nên được cài đặt bởi hàm toán tử bạn của lớp.

Trong một lớp cài đặt một KDLTT, nói chung ta cần đưa vào một hàm viết ra đối tượng dữ liệu trên các thiết bị ra chuẩn. C + + đã đưa vào toán tử << để viết ra các số nguyên, số thực, ký tự, … Chúng ta có thể định nghĩa lại toán tử << để viết ra các đối tượng dữ liệu phức hợp khác, chẳng hạn để viết ra các số phức trong lớp Complex. Trong lớp Complex, hàm operator << được thiết kế là hàm bạn với khai báo sau:

```
ostream &   operator << (ostream & os, const Complex & c) ;
```

trong đó ostream là lớp các luồng dữ liệu ra (output stream), ostream là thành viên của thư viện iostream.h, và cout (thiết bị ra chuẩn) là một đối tượng của lớp ostream.

Sau đây là cài đặt hàm toán tử bạn operator << trong lớp Complex:

```
ostream &   operator << (ostream & os, const Complex & c)
// Postcondition. Số phức c được viết ra luồng os, dưới dạng a + ib,
// trong đó a là phần thực và b là phần ảo của số phức c.
// Giá trị trả về là luồng ra os.
{
```

```
    os << c.real << " + i" << c.imag ;
    return os ;
}
```

Trên đây chúng ta đã xét cách cài đặt các hàm toán tử định nghĩa lại các phép toán + và << cho các số phức. Các ví dụ đó cũng cho bạn một phương pháp chung để khi thiết kế một lớp cài đặt một KDLTT, bạn có thể cài đặt một phép toán hai toán hạng bởi một hàm toán tử định nghĩa lại các phép toán trong C + +.

Hầu hết các phép toán, các toán tử trong C + + đều có thể định nghĩa lại. Tuy nhiên, khi thiết kế các lớp cài đặt các KDLTT, thông thường chúng ta chỉ cần đến định nghĩa lại các phép toán số học: +, - , * , / , các phép toán so sánh = = , ! = , < , < = , > , > = , các toán tử gán = , + = , - = , * = , / = .

2.3 PHÁT TRIỂN LỚP C + + CÀI ĐẶT KDLTT

Trong mục này chúng ta sẽ trình bày một ví dụ về lớp Complex, qua đó bạn đọc sẽ thấy cần phải làm gì để phát triển một lớp C + + cài đặt một KDLTT. Phần cuối của mục sẽ trình bày các hướng dẫn cài đặt KDLTT bởi lớp.

Một lớp khi được khai báo sẽ là một kiểu dữ liệu được xác định bởi người sử dụng. Vì vậy, bạn có thể khai báo một lớp trong chương trình và sử dụng nó trong chương trình giống như khai báo và sử dụng các kiểu dữ liệu quen thuộc khác. Hình 2.1 là một chương trình demo cho việc khai báo và sử dụng lớp Complex. Chú ý rằng, khi cài đặt các hàm thành phần của một lớp, bạn cần sử dụng toán tử định phạm vi để chỉ nó thuộc lớp đó ở đây bạn phải đặt Complex :: trước tên hàm.

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

```
class  Complex
{
public :
Complex (double a = 0, double b = 0) ;
// Tạo ra số phức có phần thực a, phần ảo b
double    GetReal( ) const ;
// Trả về phần thực của số phức.
double    GetImag ( ) const ;
// Trả về phần ảo của số phức.
double    GetAbs ( ) const ;
// Trả về giá trị tuyệt đối của số phức.
friend Complex & operator +(const Complex & c1,const Complex &c2);
// Trả về tổng của số phức c1 và c2.
friend Complex & operator -(const Complex & c1,const Complex & c2);
// Trả về hiệu của số phức c1 và c2.
friend Complex & operator *(const Complex & c1,const Complex & c2);
// Trả về tích của số phức c1 và c2.
friend Complex & operator /(const Complex & c1, const Complex & c2);
// Trả về thương của số phức c1 và c2.
friend ostream &   operator << (ostream & os, const Complex &c);
// In số phức c trên luồng ra os.
// Các mẫu hàm khác.
private :
      double real ;
      double imag ;
} ;

int main ( )
{
      Complex   A (3.2, 5.7) ;
      Complex   B (6.3, -4.5) ;
```

```
    cout << "Phần thực của số phức A:" << A.GetReal( ) << endl;
    cout << "Phần ảo của số phức A:" << A.GetImag ( ) << endl ;
    A = A + B ;
    cout << A << endl ;  // In ra số phức A.
    return   0 ;
}

// Sau đây là cài đặt các hàm đã khai báo trong lớp Complex
Complex :: Complex (double a = 0, double b = 0)
{
    real = a ;
    imag = b ;
}
double Complex :: GetReal ( )
{
    return real ;
}
// Cài đặt các hàm còn lại trong lớp Complex.
```

Hình 2.1. Chương trình demo về khai báo và sử dụng lớp.

Tuy nhiên chúng ta thiết kế một KDLTT và cài đặt nó bởi lớp C + + là để sử dụng trong một chương trình bất kỳ cần đến KDLTT đó, do đó khi phát triển một lớp cài đặt một KDLTT, chúng ta cần phải tổ chức thành hai file: file đầu và file cài đặt (tương tự như chúng ta đã làm khi cài đặt không định hướng đối tượng KDLTT, xem mục 1.4).

- **File đầu**: File đầu có tên kết thúc bởi .h. File đầu chứa tất cả các thông tin cần thiết mà người lập trình cần biết để sử dụng KDLTT trong chương trình của mình. Chúng ta sẽ tổ chức file đầu như sau: Đầu tiên là các chú thích về các hàm trong mục public của lớp. Mỗi chú thích về một hàm bao gồm mẫu hàm và các điều kiện trước, điều

kiện sau kèm theo mỗi hàm. Người sử dụng lớp chỉ cần đọc các thông tin trong phần chú thích này. Tiếp theo là định nghĩa lớp. Chú ý rằng, định nghĩa lớp cần đặt giữa các định hướng tiền xử lý # ifndef # define … # endif. Chẳng hạn, định nghĩa lớp Complex như sau:

```
# ifndef  COMPLEX_H
# define  COMPLEX_H
      class    Complex
      {
          // danh sách thành phần
       } ;
# endif
```

File đầu của lớp Complex được cho trong hình 2.2. Cần lưu ý rằng, trong lớp Complex đó, chúng ta mới chỉ đưa vào một số ít phép toán, để thuận lợi cho việc tiến hành các thao tác số phức, lớp Complex thực tế cần phải chứa rất nhiều phép toán khác, song để cho ngắn gọn, chúng ta đã không đưa vào.

```
// File đầu Complex.h
// Các hàm kiến tạo :
// Complex (double a = 0.0, double b = 0.0) ;
// Postcondition: số phức được tạo ra có phần thức là a, phần ảo là b.
// Các hàm thành phần khác:
// double GetReal ( ) const ;
// Trả về phần thực của số phức.
// double GetImag ( ) const ;
// Trả về phần ảo của số phức.
// double GetAbs ( ) const ;
// Trả về giá trị tuyệt đối của số phức.
// Các hàm bạn:
```

```
// friend Complex & operator + (const Complex & c1,
//                                                const Comple  & c2) ;
// Trả về tổng c1 + c2 của số phức c1 và c2.
// friend Complex & operator - (const Complex & c1,
//                                                const Complex  & c2);
// Trả về hiệu c1 – c2 của số phức c1 và c2.
// friend Complex & operator * (const Complex & c1,
//                                                const Complex & c2);
// Trả về tích c1 * c2 của số phức c1 và c2.
// friend Complex & operator / (const Complex & c1,
//                                                 const Complex &c2);
// Trả về thương c1 / c2 của số phức c1 và c2.
// friend ostream & operator << (ostream & os, const Complex &c);
// Postcondition: số phức c được in ra dưới dạng a + ib, trong đó a là
// phần thực, b là phần ảo của c.

# ifndef  COMPLEX_H
# define  COMPLEX_H
        class     Complex
        {
            public :
            Complex (double a = 0.0, double b = 0.0) ;
            double GetReal ( ) const ;
            double GetImag ( ) const ;
            double GetAbs ( ) const ;
            friend Complex & operator + (const Complex & c1, const
                                                Complex & c2) ;
            friend Complex & operator - (const Complex & c1,  const
                                                Complex & c2) ;
            friend Complex & operator * (const Complex & c1,  const
                                                Complex & c2) ;
```

```
        friend Complex & operator / (const Complex & c1,  const
                                                Complex & c2) ;
        friend ostream & operator << (ostream & os, const
                                                Complex & c) ;
        private :
           double   real ;
           double   imag ;
    } ;
# endif
```

Hình 2.2. File đầu của lớp Complex

- **File cài đặt**. Hầu hết các chương trình dịch đòi hỏi file cài đặt có tận cùng là .cpp hoặc .c. Trong file cài đặt trước hết cần có mệnh đề # include "tên file đầu" và các mệnh đề # include khác, chẳng hạn # include <math.h>, …, khi mà các file thư viện chuẩn này cần thiết cho sự cài đặt các hàm trong lớp. Một điều cần lưu ý là, khi viết định nghĩa mỗi hàm thành phần, bạn cần sử dụng toán tử định phạm vi để chỉ nó thuộc lớp nào. Trong ví dụ đang xét, bạn cần đặt Complex :: ngay trước tên hàm thành phần. File cài đặt Complex.cpp được cho trong hình 2.3.

```
// File cài đặt Complex.cpp
# include "Complex.h"
# include <math.h>
# include <iostream.h>
  Complex :: Complex (double a = 0.0, double b = 0.0)
  {
        real = a ;
        imag = b ;
  }
```

```
double Complex :: GetReal ( ) const
{
        return  real ;
}
double Complex :: GetImag ( ) const
{
        return  imag ;
}
double Complex :: GetAbs ( ) const
{
        return sqrt (real * real + imag * imag) ;
}
Complex & operator + (const Complex & c1, const Complex & c2)
{
        Complex  c;
        c.real = c1.real + c2.real ;
        c.imag = c1.imag +c2.imag ;
        return  c ;
}
// Các hàm toán tử -, *, /
   ostream & operator << (ostream & os, const Complẽ &c)
   {
         os << c.real << "+i" << c.imag ;
         return  os ;
   }
```

Hình 2.3. File cài đặt Complex.cpp

Hướng dẫn xây dựng lớp cài đặt KDLTT.

Xây dựng một lớp tốt cài đặt một KDLTT là một nhiệm vụ khó khăn. Ứng với một KDLTT có thể có nhiều cách cài đặt khác nhau. Điều đó trước hết là do một loại đối tượng dữ liệu có thể được biểu diễn bởi nhiều CTDL khác nhau. Sự lựa chọn CTDL để cài đặt đối tượng dữ liệu cần phải sao cho các hàm thực hiện các phép toán trên dữ liệu là hiệu quả.

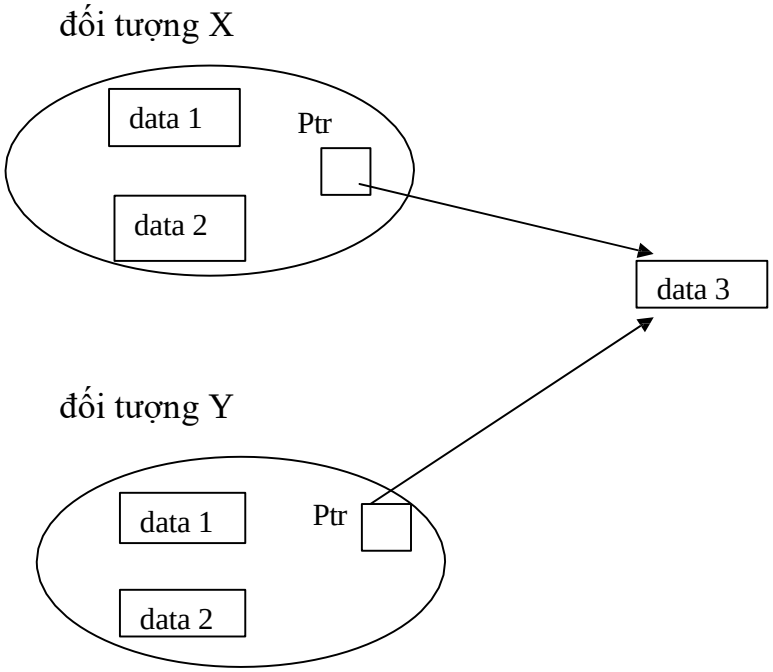
Sau khi đã lựa chọn CTDL, nhiệm vụ tiếp theo là thiết kế lớp: lớp cần chứa các hàm thành phần, hàm bạn nào? Các hàm đó cần được thiết kế như thế nào? (Tức là các hàm đó cần có mẫu hàm như thế nào?). Các hướng dẫn sau đây sẽ giúp bạn dễ dàng hơn khi phát triển một lớp cài đặt KDLTT. Các hướng dẫn này cũng nhằm mục đích để người lập trình có thể sử dụng KDLTT một cách thuận tiện, an toàn và hiệu quả.

1. Cần nhớ rằng, không phải trong đặc tả KDLTT có bao nhiêu phép toán thì trong lớp chỉ có bấy nhiêu hàm tương ứng với các phép toán đó. Thông thường ngoài các hàm tương ứng với các phép toán, chúng ta cần đưa vào lớp nhiều hàm thành phần (hoặc hàm bạn) khác giúp cho người sử dụng tiến hành dễ dàng các thao tác trên dữ liệu trong chương trình, chẳng hạn các hàm kiến tạo, hàm huỷ, các loại toán tử gán, các hàm đọc dữ liệu, viết dữ liệu, hàm chuyển kiểu, …
2. Cần cung cấp một số hàm kiến tạo thích hợp để khởi tạo ra các đối tượng của lớp (rất ít khi người ta thiết kế một lớp không có hàm kiến tạo). Đặc biệt cần lưu ý đến hàm kiến tạo mặc định (hàm kiến tạo không có tham số) và hàm kiến tạo copy.

Nếu bạn không cung cấp cho lớp hàm kiến tạo mặc định, thì chương trình dịch sẽ tạo ra hàm kiến tạo mặc định tự động. Tuy nhiên hàm kiến tạo mặc định được cung cấp bởi chương trình dịch có hạn chế: nó chỉ khởi tạo các thành phần dữ liệu có hàm kiến tạo mặc định, còn các thành phần dữ liệu khác thì không được khởi tạo.

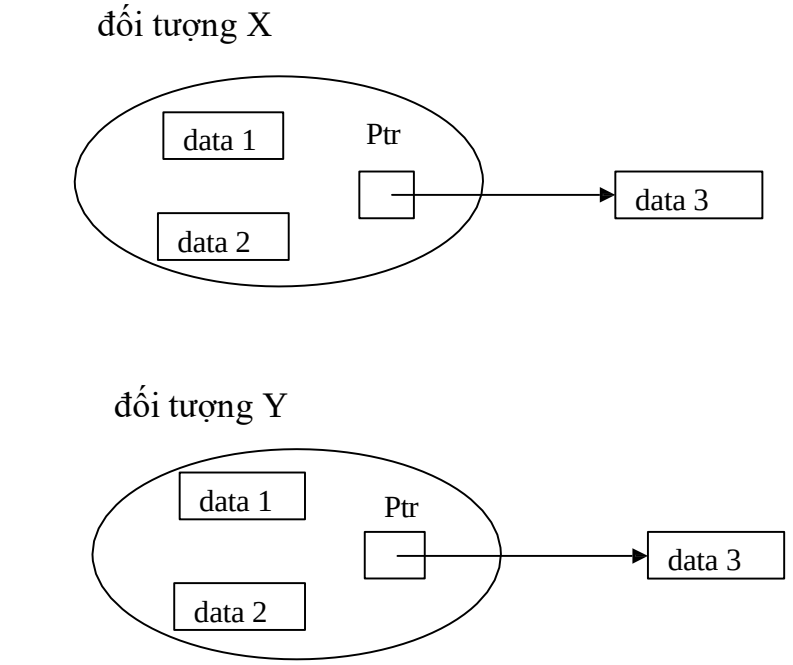
Cần biết rằng, hàm kiến tạo copy sẽ được tự động gọi bởi chương trình dịch khi mà đối tượng được truyền bởi giá trị trong một lời gọi hàm.

Nếu ta không cung cấp cho lớp một hàm kiến tạo copy, thì một hàm kiến tạo copy tự động sẽ được chương trình dịch cung cấp. Hàm này sẽ copy từng thành phần dữ liệu của đối tượng bị copy sang cho đối tượng copy. Chúng ta sẽ xét trường hợp lớp chứa thành phần dữ liệu là biến con trỏ Ptr. Giả sử X là đối tượng đã có của lớp này. Ta muốn khởi tạo ra đối tượng mới Y là bản sao của X bởi khai báo Y(X) (hoặc Y = X). Nếu trong lớp không cung cấp hàm kiến tạo copy, thì kết quả của lệnh trên sẽ được minh hoạ trong hình 2.4. Điều này sẽ kéo theo hậu quả là bất kỳ sự thay đổi dữ liệu data3 của đối tượng Y cũng kéo theo sự thay đổi dữ liệu data3 của đối tượng X và ngược lại. Đó là điều mà chúng ta không muốn có.



Hình 2.4. Đối tượng Y là copy của đối tượng X, khi lớp không được cung cấp hàm kiến tạo copy.

Trong trường hợp lớp có chứa biến thành phần là biến con trỏ, chúng ta cần đưa vào lớp hàm kiến tạo copy. Hàm này cần phải tạo ra đối tượng mới Y từ đối tượng X cũ như trong hình 2.5.



Hình 2.5. Đối tượng Y là copy của đối tượng X, khi trong lớp có hàm kiến tạo copy

3. Nếu bạn không đưa vào lớp hàm huỷ, thì chương trình dịch sẽ tạo ra hàm huỷ tự động. Song hàm huỷ này chỉ thu hồi vùng nhớ đã cấp cho tất cả các biến thành phần của lớp, vùng nhớ cấp phát động mà các biến thành phần kiểu con trỏ trỏ tới thì không bị thu hồi. Vì vậy, trong trường hợp lớp chứa các thành phần dữ liệu là biến con trỏ thì nhất thiết bạn phải thiết kế cho lớp một hàm huỷ, hàm này thực hiện nhiệm vụ thu hồi tất cả các vùng nhớ liên quan tới đối tượng để trả về cho hệ thống, khi mà đối tượng không còn cần thiết nữa.

4. Trong chương trình, toán tử gán được sử dụng thường xuyên trên các đối tượng dữ liệu. Do đó trong lớp, chúng ta cần đưa vào hàm toán tử operator =, nó định nghĩa lại toán tử gán truyền thống trong C + +, nó tương tự như hàm kiến tạo copy, chỉ có điều nó được sử dụng để gán, chứ không phải để khởi tạo ra đối tượng mới. Nếu bạn không cung cấp cho lớp toán tử gán thì toán tử gán tự động sẽ được chương trình dịch cung cấp. Nó chỉ làm được một việc: copy từng thành phần dữ liệu của đối tượng ở bên phải toán tử gán tới các thành phần dữ liệu tương ứng của đối tượng ở bên trái.

5. Các phép toán một toán hạng, hai toán hạng trên các đối tượng dữ liệu của KDLTT cần được thiết kế là các hàm toán tử của lớp (các hàm này định nghĩa lại các phép toán số học +, -, *, /, …tuỳ theo ngữ nghĩa của chúng). Các hàm toán tử có thể thiết kế như là hàm thành phần hoặc như là hàm bạn của lớp.

2.4 LỚP KHUÔN

Trong mục này chúng ta sẽ trình bày khái niệm **lớp khuôn (template class).** Lớp khuôn là một công cụ quan trọng trong C + + được sử dụng để cài đặt các lớp phụ thuộc tham biến kiểu dữ liệu. Các KDLTT quan trọng mà chúng ta nghiên cứu trong các chương sau đều được cài đặt bởi lớp khuôn. Trước hết chúng ta xét một ví dụ về lớp côngtơnơ và cách cài đặt không sử dụng lớp khuôn.

2.4.1 Lớp côngtơnơ

Trong nhiều ứng dụng, chúng ta cần sử dụng các KDLTT mà mỗi đối tượng dữ liệu của nó là một bộ sưu tập các phần tử dữ liệu cùng kiểu nào đó. Lớp cài đặt các KDLTT như thế được gọi là **lớp côngtơnơ (container class**). Như vậy, lớp côngtơnơ là một thuật ngữ để chỉ các lớp mà mỗi đối tượng của lớp là một "côngtơnơ" chứa một bộ sưu tập các dữ liệu cùng kiểu.

Các lớp danh sách, hàng đội, ngăn xếp, … được nghiên cứu sau này đều là lớp côngtơnơ. Trong một chương trình của mình, bạn có thể cần đến lớp côngtơnơ các số nguyên, người khác lại cần sử dụng chính lớp côngtơnơ đó, chỉ khác một điều là các côngtơnơ của anh ta không phải là côngtơnơ các số nguyên mà là côngtơnơ các số thực, hoặc côngtơnơ các ký tự. Do đó, vấn đề đặt ra cho việc thiết kế các lớp côngtơnơ là: lớp cần được thiết kế như lớp phụ thuộc tham biến kiểu dữ liệu Item sao cho trong một chương trình bất kỳ, bạn có thể dễ dàng sử dụng lớp bằng cách chỉ ra Item được thay bởi một kiểu dữ liệu cụ thể, chẳng hạn int, double hoặc char… Sau đây là một ví dụ về một lớp côngtơnơ và cách cài đặt sử dụng mệnh đề định nghĩa kiểu typedef.

Ví dụ (KDLTT Túi). Trước hết chúng ta đặc tả KDLTT Túi. Mỗi đối tượng dữ liệu là một "túi" chứa một bộ sưu tập các phần tử cùng kiểu. Chẳng hạn, đối tượng dữ liệu có thể là túi bi, một túi có thể chứa 3 bi xanh, 1 bi đỏ, 2 bi vàng, … Một ví dụ khác, đối tượng dữ liệu có thể là côngtơnơ áo sơ mi, mỗi côngtơnơ có thể chứa nhiều áo thuộc cùng một loại áo (mỗi loại áo đặc trưng bởi kiểu dáng, loại vải, cỡ áo).

Sau đây là các phép toán có thể thực hiện trên các túi, Trong các phép toán này, B là ký hiệu một túi, element là một phần tử cùng kiểu với kiểu của các phần tử trong túi.

1. Sum (B). Hàm trả về số phần tử trong túi B.
2. Insert (B, element). Thêm phần tử element vào túi B.
3. Remove (B, element). Lấy ra một phần tử là bản sao của element khỏi túi B.
4. Occurr (B, element). Hàm trả về số phần tử là bản sao của element trong túi B.
5. Union (B_1, B_2). Hợp của túi B_1và túi B_2. Hàm trả về một túi chứa tất cả các phần tử của B_1 và tất cả các phần tử của túi B_2.

Dưới đây chúng ta đưa ra một cách cài đặt KDLTT Túi đã đặc tả ở trên.

Chúng ta sẽ sử dụng một mảng (tĩnh) data có cỡ là MAX để lưu các phần tử của túi. Các phần tử của túi được lưu trong các thành phần của mảng: data[0], data[1], …, data[size-1], trong đó size là số phần tử của túi. Chẳng hạn, một túi chứa 5 số nguyên: 2 số nguyên 3, 1 số nguyên 5, 2 số nguyên 6 có thể lưu trong mảng như sau:

0 1 2 3 4 MAX-1

Trong khai báo mảng data, cỡ của mảng MAX cần phải là hằng, MAX là số tối đa các phần tử mà túi có thể chứa, và nó là chung cho tất cả các đối tượng của lớp. Vì vậy, chúng ta sẽ khai báo MAX là hằng tĩnh trong lớp. Chẳng hạn, static const int MAX = 50 ;

Mục tiêu thiết kế lớp túi (class bag) của chúng ta là sao cho người lập trình sử dụng lớp túi trong các hoàn cảnh khác nhau một cách thuận tiện: chỉ cần một vài thay đổi nhỏ, không cần viết lại toàn bộ mã của lớp. Trong một chương trình, người lập trình có thể sử dụng lớp túi số nguyên, trong chương trình khác lại có thể sử dụng lớp túi bi hoặc lớp túi áo sơ mi, … Các phép toán trên các đối tượng của các loại túi khác nhau là hoàn toàn như nhau, không phụ thuộc gì vào kiểu dữ liệu của các phần tử trong túi. Do đó, chúng ta đưa vào lớp mệnh đề định nghĩa kiểu, mệnh đề này xác định kiểu dữ liệu Item, Item là kiểu dữ liệu của các phần tử trong túi . Chẳng hạn, trong định nghĩa lớp túi số nguyên, bạn đưa vào mệnh đề:

typedef int Item ;

Lớp túi chứa hai biến thành phần: mảng data lưu các phần tử của túi và biến size lưu số phần tử trong túi. Thiết kế ban đầu của lớp túi như sau:

```
class Bag
{
  public :
```

```
        static const int MAX = 50 ;
         typedef int Item ;
         // Các hàm thành phần
    private :
        Item data[MAX] ;
        int size ;
} ;
```

Trong các hàm thành phần của lớp Bag, bất kỳ chỗ nào có mặt Item, chương trình dịch sẽ hiểu đó là int. Với cách thiết kế này, nếu chương trình của bạn cần đến lớp túi mà các đối tượng của nó chứa các ký tự, bạn chỉ cần thay int trong mệnh đề typedef bởi char. Còn nếu bạn muốn sử dụng các túi có cỡ lớn hơn, bạn chỉ cần xác định lại hằng MAX. Cần lưu ý rằng, hằng MAX và kiểu Item được xác định trong phạm vi lớp Bag. Vì vậy, ở ngoài phạm vi lớp Bag, để truy cập tới chúng, bạn cần sử dụng toán tử định phạm vi. Chẳng hạn, trong chương trình để in ra cỡ tối đa của túi, bạn cần viết

```
cout << "cỡ tối đa của túi là:" << Bag :: MAX << endl ;
```

Bây giờ chúng ta thiết kế các hàm thành phần của lớp Bag. Trước hết, cần có hàm kiến tạo mặc định sau

```
Bag ( ) ;
```

Hàm này khởi tạo ra một túi rỗng.

Các phép toán (1)-(4) của KDLTT Túi được cài đặt bởi các hàm thành phần tương ứng của lớp Bag như sau:

```
int  Sum (   )  const ;
void  Insert (const Item & element) ;
void  Remove (const Item & element) ;
int  Occurr (const Item & element) ;
```

Phép toán (5) được cài đặt bởi hàm toán tử bạn:

```
friend  Bag& operator + (const Bag & B1, const Bag & B2);
```

Với hàm toán tử này, chúng ta đã xác định phép + trên các túi. Do đó, để thuận tiện khi cần cộng thêm một túi vào một túi khác, chúng ta đưa vào lớp Bag một hàm toán tử thành phần operator += như sau:

void operator += (const Bag & B1) ;

Không cần đưa vào lớp Bag hàm kiến tạo copy, hàm toán tử gán, vì chỉ cần sử dụng hàm kiến tạo copy tự động, toán tử gán tự động là đủ.

Đến đây chúng ta có thể viết ra file đầu của lớp Bag. File đầu được cho trong hình 2.6.

```
// File đầu bag.h
// Sau đây là các thông tin người lập trình cần biết
// khi sử dụng lớp Bag.
// MAX là số tối đa các phần tử mà một Bag có thể chứa.
// Item là kiểu dữ liệu của các phần tử trong túi.
// Item có thể là một kiểu bất kỳ có sẵn trong C + + (chẳng hạn int,
// double, char, …), hoặc có thể là một lớp trong đó được cung cấp
// hàm kiến tạo mặc định, các hàm toán tử =, == và !=
// Sau đây là các hàm thành phần của lớp Bag:
// Bag( ) ;
// Postcondition: một Bag rỗng được khởi tạo.
// int  Sum ( )  const ;
// Postcondition: Trả về tổng số phần tử của túi.
// void  Insert(const Item& element) ;
// Hàm thực hiện thêm element vào túi B.
// Precondition: B.Sum ( ) < MAX
// Postcondition: một bản sao của element được thêm vào túi B.
// void Remove (const Item& element) ;
// Hàm loại một phần tử khỏi túi B.
// Postcondition: một phần tử là bản sao của element bị loại khỏi túi B,
// nếu trong B có chứa, và B không thay đổi nếu trong B không chứa.
// int  Occurr (const Item& element) ;
```

```
// Postcondition: Trả về số lần xuất hiện của element trong túi B.
// void operator += (const Bag& B1)
// Hàm thực hiện cộng túi B1 vào túi B, B += B1
// Postcondition: tất cả các phần tử của túi B1 được thêm vào túi B.
// Hàm toán tử bạn của lớp Bag:
// friend Bag& operator + (const Bag& B1, const Bag& B2) ;
// Precondition: B1.Sum ( ) + B2.Sum( ) <= MAX
// Postcondition: Trả về túi B là hợp của túi B1 và B2.
// Bạn có thể sử dụng toán tử gán = và hàm kiến tạo copy cho các
// đối tượng của lớp Bag.
# ifdef BAG_H
# define BAG_H
    class Bag
    {
      public:
          static const int MAX = 50 ;
          typdef int Item ;
          Bag ( ) {size = 0 ;}
          int Sum( ) const
          {return size ;}
          int Occurr (const Item& element) const ;
          void Insert (const Item& element) ;
          void Remove (const Item& element) ;
          void operator += (const Bag & B1) ;
          friend Bag & operator + (const Bag & B1,const Bag & B2);
      private :
          Item data[MAX];
          int size ;
    } ;
# endif
```

Hình 2.6. File đầu của lớp Bag.

Bây giờ chúng ta tiến hành cài đặt các hàm đã khai báo trong định nghĩa lớp Bag.

- **Hàm Insert**: Trước hết cần kiểm tra túi không đầy. Các phần tử của túi được lưu trong các thành phần của mảng data: data[0], data[1], …, data[size - 1], do đó nếu túi không đầy, phần tử element được lưu vào thành phần data[size], tức là

```
if  (size < MAX)
    {
        data[size] = element ;
        size + + ;
     }
```

- **Hàm Remove**: Để loại được phần tử element khỏi túi, trước hết ta tìm vị trí đầu tiên trong mảng data, tại đó có lưu element. Điều này được thực hiện bằng cách cho biến i chạy từ 0 tới size – 1, nếu gặp một thành phần của mảng mà data[i] == element thì dừng lại, còn nếu i chạy tới size thì điều đó có nghĩa là túi không chứa element. Bước tìm vị trí đầu tiên lưu element được thực hiện bởi vòng lặp:

```
For ( i = 0; ( i < size ) && (data[i] != element ); i + + ) ;
```

Việc loại element ở vị trí thứ i trong mảng data được thực hiện bằng cách chuyển phần tử lưu ở vị trí cuối cùng data[size - 1] tới vị trí i, và giảm số phần tử của túi i đi 1. Tức là:

```
data[i] = data[size - 1] ;
size -- ;
```

- **Hàm Occurr**: Chúng ta cần đếm số lần xuất hiện của phần tử element trong túi. Muốn vậy, chúng ta sử dụng một biến đếm count, cho biến chỉ số i chạy từ đầu mảng tới vị trí size – 1, cứ mỗi lần gặp phần tử element thì biến đếm count được tăng lên 1.

```
count = 0 ;
for (i = 0; i < size; i + +)
```

```
if ( data[i] = = element)
    count + +;
```

- **Hàm toán tử operator +=** : Nhiệm vụ của hàm là "đổ" nội dung của túi B1 vào túi B. Chỉ có thể thực hiện được điều đó khi mà tổng số phần tử của hai túi không vượt quá dung lượng MAX. Nếu điều kiện đó đúng, chúng ta sao chép lần lượt các phần tử của túi B1 vào các thành phần mảng data của túi B, bắt đầu từ vị trí size. Hành động đó được thực hiện bởi vòng lặp:

```
for (i =0; i < B1.size; i + +)
{
    data[size] = B1.data[i] ;
    size + + ;
 }
```

- **Hàm toán tử bạn operator +**. Muốn gộp hai túi thành một túi, đương nhiên cũng cần điều kiện tổng số phần tử của hai túi không lớn hơn MAX. Chúng ta khởi tạo một túi B rỗng, rồi áp dụng toán tử += đổ lần lượt túi B1, B2 vào túi B, tức là:

```
Bag B;
B  +=  B1;
B  +=  B2;
```

Sử dụng hàm tiện ích assert.

Trong nhiều hàm, để hàm thực hiện được nhiệm vụ của mình, các dữ liệu vào cần thoả mãn các điều kiện được liệt kê trong phần chú thích Precondition ở ngay sau mẫu hàm. Khi cài đặt các hàm, chúng ta có thể sử dụng mệnh đề if (Precondition) ở đầu thân hàm, như chúng ta đã làm khi nói về hàm Insert. Một cách lựa chọn khác là sử dụng hàm assert trong thư viện chuẩn assert.h. Hàm assert có một đối số, đó là một biểu thức nguyên, hoặc thông thường là một biểu thức logic. Hàm assert sẽ đánh giá biểu thức, nếu biểu thức có giá trị true, các lệnh tiếp theo trong thân hàm sẽ được thực hiện. Nếu biểu thức có giá trị false, assert sẽ in ra một thông báo lỗi và chương

trình sẽ bị treo. Vì vậy, khi cài đặt một hàm, chúng ta nên sử dụng hàm assert thay cho mệnh đề if.

File cài đặt của lớp Bag được cho trong hình 3.7. Chú ý rằng, trong các hàm có sử dụng hàm assert, nên ở đầu file chúng ta cần đưa vào mệnh đề # include < assert.h >

```
// File : bag.cpp
# include  <assert.h>
# include  "bag.h"
int  Bag :: Occurr (const Item & element) const
{
        int  count = 0 ;
        int i ;
        for (i=0; i < size ; i + +)
           if (data[i] == element)
                count + + ;
        return  count ;
}
void  Bag :: Insert (const Item & element)
{
        assert (Sum( ) < MAX) ;
        data[size] = element ;
        size + + ;
}
void  Bag :: Remove (const Item & element)
{
        int  i;
        for (i = 0; (i < size) && (data[i]! =element); i + +) ;
        if (i == size)
          return ;
        data[i] = data[size - 1] ;
```

```
        size -- ;
}
void  Bag :: operator += (const Bag & B1)
{
        int  i;
        int  a = B1.size ;
        assert (Sum( ) + B1.Sum( ) <= MAX) ;
        for (i = 0; i < a; i + +)
        {
           data[size] = B1.data[i] ;
           size + + ;
        }
}
Bag & operator + (const Bag & B1, const Bag & B2)
{
        Bag  B;
        assert (B1.Sum( ) + B2.Sum( ) <= Bag :: MAX) ;
        B += B1 ;
        B += B2 ;
        return  B ;
}
```

Hình 2.7. File cài đặt của lớp Bag.

Nhận xét. Lớp Bag mà chúng ta đã cài đặt có các đặc điểm sau: Trước hết, mỗi đối tượng của lớp là túi chỉ chứa được tối đa 50 phần tử, do khai báo hằng MAX ở trong lớp:

```
        static  const  int  MAX = 50;
```

Mặt khác, mệnh đề

```
        typedef  int  Item;
```

xác định rằng, các phần tử trong túi là các số nguyên. Trong một chương trình, nếu bạn muốn sử dụng Bag với cỡ túi lớn hơn, chẳng hạn 100, và các phần tử trong túi có kiểu khác, chẳng hạn char, bạn chỉ cần thay 50 bằng 100 trong mệnh đề khai báo hằng MAX và thay int bằng char trong mệnh đề định nghĩa Item. Song nếu như trong một chương trình, bạn cần đến cả túi số nguyên, cả túi ký tự thì sao? Bạn không thể định nghĩa hai lần lớp Bag trong một chương trình! **Lớp khuôn (template class**) sẽ giải quyết được khó khăn này. Lớp khuôn trong C + + là một công cụ cho phép ta thiết kế các lớp phụ thuộc tham biến kiểu dữ liệu. Trước khi trình bày khái niệm lớp khuôn, chúng ta hãy nói về các **hàm khuôn (template function**)

2.4.2 Hàm khuôn

Xét vấn đề tìm giá trị lớn nhất (nhỏ nhất) trong một mảng. Thuật toán đơn giản là đọc tuần tự các thành phần của mảng và lưu lại vết của phần tử lớn nhất (nhỏ nhất). Rõ ràng, logic của thuật toán này chẳng phụ thuộc gì vào kiểu dữ liệu của các phần tử trong mảng. Các thuật toán sắp xếp mảng cũng có đặc trưng đó. Một ví dụ khác: vấn đề trao đổi giá trị của hai biến. Thuật toán trao đổi giá trị của hai biến mà chúng ta đã quen biết cũng không phụ thuộc vào kiểu dữ liệu của hai biến đó. Chúng ta mong muốn viết ra các hàm cài đặt các thuật toán đó sao cho hàm có thể sử dụng được cho các loại kiểu dữ liệu khác nhau.

Một cách lựa chọn là sử dụng mệnh đề định nghĩa kiểu typedef. Chẳng hạn, hàm trao đổi giá trị của hai biến có thể viết như sau:

```
typedef  int  Item ;
void  Swap (Item & x, Item & y)
{  Item  temp;
   temp = x ;
   x = y ;
```

```
  y = temp ;
}
```

Hàm trên để trao đổi hai biến nguyên. Nếu cần trao đổi hai biến ký tự, bạn thay int bởi char trong mệnh đề typedef. Nhưng cách tiếp cận này có vấn đề, khi trong cùng một chương trình bạn vừa cần trao đổi hai biến nguyên, vừa cần trao đổi hai biến ký tự, bởi vì bạn không thể hai lần định nghĩa kiểu Item trong cùng một chương trình.

Một cách tiếp cận khác khắc phục được hạn chế của cách tiếp cận trên là sử dụng hàm khuôn. Chẳng hạn, hàm khuôn Swap được viết như sau:

```
template  <class Item>
void  Swap (Item & x, Item & y)
// Item cần phải là kiểu bất kỳ có sẵn trong C + + (int, char, …)
// hoặc là lớp có hàm kiến tạo mặc định và toán tử gán.
{
  Item  temp ;
  temp = x ;
  x = y ;
  y = temp;
}
```

Để xác định một hàm khuôn, bạn chỉ cần đặt biểu thức **template <class Item>** ngay trước mẫu hàm. Biểu thức đó nói rằng, ở mọi chỗ Item xuất hiện trong định nghĩa hàm, Item là một kiểu dữ liệu nào đó chưa được xác định. Nó sẽ được hoàn toàn xác định khi ta sử dụng hàm (gọi hàm). Chương trình đơn giản sau minh họa cách sử dụng hàm khuôn Swap.

```
  int main(  )
  {
      int  a = 3;
```

```
    int  b = 5;
    double  x = 4.2;
    double  y = 3.6;
    Swap(a,b); // Item được thay thế bởi int
    Swap(x,y); // Item được thay thế bởi double
    return   0 ;
}
```

Một ví dụ khác về hàm khuôn: hàm tìm giá trị lớn nhất trong mảng

```
template  <class  Item>
Item & FindMax (Item data[  ], int n )
// Item cần phải là kiểu bất kỳ có sẵn trong C + + (int, char, …)
// hoặc là lớp có phép toán so sánh < ,
// Precondition: data là mảng có ít nhất n thành phần, n > 0.
// Postcondition: giá trị trả về là phần tử lớn nhất trong n phần tử
// data[0], …,data[n -1].
{
  int  i ;
  int index = 0; // chỉ số của phần tử lớn nhất.
  assert( n>0)
  for (i = 1; i < n; i + +)
    if (data [index] < data [i])
        index = i;
  return data [index];
}
```

2.4.3 Lớp khuôn

Hàm khuôn là hàm phụ thuộc tham biến kiểu dữ liệu. Tương tự, nhờ lớp khuôn chúng ta xây dựng được lớp phụ thuộc tham biến kiểu dữ liệu.

Định nghĩa lớp khuôn

Xác định một lớp khuôn cũng tương tự như xác định một hàm khuôn. Cần đặt biểu thức template <class Item> ngay trước định nghĩa lớp, Item là tham biến kiểu của lớp. Trong định nghĩa lớp, chỗ nào có mặt Item, chúng ta chỉ cần hiểu Item là một kiểu dữ liệu nào đó. Item là kiểu cụ thể gì (là int hay double, …) sẽ được xác định khi chúng ta sử dụng lớp trong chương trình.

Ví dụ (Lớp khuôn Bag). Từ lớp Bag được xây dựng bằng cách sử dụng mệnh đề định nghĩa kiểu typedef int Item (xem hình 2.6), chúng ta có thể xây dựng lớp khuôn Bag như sau:

```
template <class Item>
class Bag
{
  // Bỏ mệnh đề typedef int Item;
} ;
```

Trong định nghĩa lớp khuôn Bag, tất cả các hàm thành phần của nó là các hàm thành phần của lớp Bag cũ (lớp trong hình 2.6), không có gì thay đổi. Riêng hàm operator +, nó không phải là hàm thành phần của lớp, nên trong mẫu hàm của nó, Bag (với tư cách là tên kiểu dữ liệu) cần phải được viết là Bag <Item>. Cụ thể là:

```
friend Bag<Item> & operator + (const Bag <Item> & B1,
                               const Bag <Item> & B2);
```

Tổng quát, ở bên ngoài định nghĩa lớp khuôn Bag (chẳng hạn, ở trong file cài đặt lớp khuôn Bag), bất kỳ chỗ nào Bag xuất hiện với tư cách là tên lớp (trong toán tử định phạm vi Bag ::). hoặc với tư cách là tên kiểu dữ liệu, chúng ta cần phải viết là **Bag <Item>** để chỉ rằng, Bag là lớp phụ thuộc tham biến kiểu Item.

Cài đặt các hàm. Tất cả các hàm thao tác trên các đối tượng của lớp khuôn (bao gồm các hàm thành phần, các hàm bạn, các hàm không thành

phần) đều phải được cài đặt là hàm khuôn. Chẳng hạn, hàm toán tử += của lớp Bag cần được cài đặt như sau:

```
template  <class Item>
void  Bag <Item> :: operator += (const Bag <Item> & B1)
{
    int  i;
    int  a = B1. size;
    assert (Sum( ) + B1. Sum( ) < = MAX);
    for (i = 0; i < a; i + +)
    {
      data [size] = B1.data [i];
      size + + ;
    }
}
```

Tổ chức các file. Như khi cài đặt lớp bình thường, cài đặt lớp khuôn cũng được tổ chức thành hai file: file đầu và file cài đặt. Chỉ có một điều cần lưu ý là, hầu hết các chương trình dịch đòi hỏi phải đưa tên file cài đặt vào trong file đầu bởi mệnh đề # include "tên file cài đặt". Hình 2.8 là file đầu của lớp khuôn Bag, file cài đặt ở trong hình 2.9.

```
// File đầu: bagt.h
// Các thông tin cần thiết để sử dụng lớp Bag, các chú thích về các
//  hàm hoàn toàn giống như trong file đầu bag.h (xem hình 2.6).
# ifndef    BAG_H
# define    BAG_H
   template  <class Item>
   class    Bag
   {
     public :
```

```
        static const int MAX = 50;
        Bag( ) {size = 0; ]
        int Sum( ) const
        { return size; }
        int  Occurr (const Item & element) const;
        void  Insert (const Item & element);
        void  Remove (const Item & element);
        void  operator += (const Bag & B1);
         friend  Bag< Item> & operator + (const Bag<Item> & B1,
                                          const Bag<Item> & B2);
    private :
        Item data[MAX];
        int size;
    };
# include "bag.template"
# endif
```

Hình 2.8. File đầu của lớp khuôn Bag.

```
// File cài đặt : bag.template
# include <assert.h>
template <class Item>
int Bag<Item> :: Occurr (const Item & element) const
{
  int count = 0;
  int i ;
  for (i = 0; i < size; i + +)
      if (data[i] == element)
          count + +;
```

```
  return count ;
}

template  <class Item>
void Bag<Item> :: Insert (const Item & element)
{
  assert ( Sum( ) < MAX) ;
  data[size] = element;
  size + +;
}

template  <class Item>
void Bag<Item> :: Remove (const Item & element)
{
  int i;
  for (i = 0; (i < size) && (data[i]!= element); i + +);
    if ( i == size)
       return;
  data[i] = data[size - 1];
  size - - ;
}

template  <class Item>
void  Bag<Item> :: operator += (const Bag<Item> & B1)
{
  int  i;
  int a = B1. size;
  assert ( Sum( ) + B1. Sum( ) <= MAX);
  for (i = 0; i < a; i + +)
  {
     data[size] = B1.data[i];
```

```
      size + +;
    }
  }

template  <class Item>
Bag<Item> & operator + (const Bag<Item> & B1,
                        const Bag<Item> & B2)
{
  Bag<Item>  B;
  assert (B1.Sum( ) + B2.Sum( ) <= Bag<Item> :: MAX);
  B += B1;
  B += B2;
  return  B;
}
```

Hình 2.9. File cài đặt lớp khuôn Bag.

Sử dụng lớp khuôn. Trong một chương trình, để sử dụng một lớp khuôn, trước hết bạn cần đưa vào mệnh đề # include "tên file đầu", chẳng hạn # include "bag.h". Khi khai báo các đối tượng của lớp, bạn cần thay thế tham biến kiểu bởi kiểu thực tế. Chẳng hạn, biểu thức Bag<Item> nói rằng, Bag là lớp khuôn với tham biến kiểu Item, nên trong chương trình nếu bạn muốn sử dụng đối tượng A là túi số nguyên, bạn cần khai báo:

```
Bag<int>   A;
```

Chương trình demo sử dụng lớp khuôn Bag được cho trong hình 2.10.

```
# include <iostream.h>
# include "bag.h"
int  main( )
{
    Bag<int> A; // A là túi số nguyên.
```

```
    Bag<char>  B; // B là túi ký tự.
    char c; // ký tự bạn đưa vào.
    cout << "Please type 10 chars \ n"
         << "Press the return key after typing \ n" ;
    for (int i = 1; i <= 10; i + +)
    {
      cin >> c;
      B. Insert(c);
      A. Insert(int(c));
    }
    cout << "The total of integers 65 in the bag A:"
         << A. Occurr(65);
    return 0;
}
```

Hình 2.10. Sử dụng lớp khuôn.

2.5 CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG QUAN TRỌNG

Trong mục này chúng ta sẽ giới thiệu các KDLTT quan trọng nhất. Các KDLTT này được sử dụng thường xuyên trong thiết kế các thuật toán. Các KDLTT này sẽ được lần lượt nghiên cứu trong các chương tiếp theo.

Trong chương 1 chúng ta đã thảo luận về tầm quan trọng của sự trừu tượng hoá dữ liệu trong thiết kế thuật toán. Sự trừu tượng hoá dữ liệu được thực hiện bằng cách xác định các KDLTT. Một KDLTT là một tập các đối tượng dữ liệu cùng với một tập phép toán có thể thực hiện trên các đối tượng dữ liệu đó. Các đối tượng dữ liệu trong thế giới hiện thực rất đa dạng và có thể rất phức tạp. Để mô tả chính xác các đối tượng dữ liệu, chúng ta cần sử dụng các khái niệm toán học, các mô hình toán học.

Tập hợp là khái niệm cơ bản trong toán học, nó là cơ sở để xây dựng nên nhiều khái niệm toán học khác. Tuy nhiên, tập hợp trong toán học là cố định. Còn trong các chương trình, thông thường chúng ta cần phải lưu một tập các phần tử dữ liệu, tập này sẽ thay đổi theo thời gian trong quá trình xử lý, do chúng ta thêm các phần tử dữ liệu mới vào và loại các phần tử dữ liệu nào đó khỏi tập. Chúng ta sẽ gọi các tập như thế là **tập động (dynamic set**). Giả sử rằng, các phần tử của tập động chứa một trường được gọi là khoá (key) và trên các giá trị khoá có quan hệ thứ tự (chẳng hạn khoá là số nguyên, số thực, ký tự,…). Chúng ta cũng giả thiết rằng, các phần tử khác nhau của tập động có khoá khác nhau.

KDLTT tập động (**dynamic set ADT**) được xác định như sau: Mỗi đối tượng dữ liệu của kiểu này là một tập động. Dưới đây là các phép toán của KDLTT tập động, trong các phép toán đó, S ký hiệu một tập, k là một giá trị khoá và x là một phần tử dữ liệu.

1. Insert(S, x). Thêm phần tử x vào tập S.
2. Delete(S, k). Loại khỏi tập S phần tử có khoá k.
3. Search(S, k). Tìm phần tử có khoá k trong tập S.
4. Max(S). Trả về phần tử có khoá lớn nhất trong tập S.
5. Min(S). Trả về phần tử có khoá nhỏ nhất trong tập S.

Ngoài các phép toán chính trên, còn có một số phép toán khác trên tập động.

Trong nhiều áp dụng, chúng ta chỉ cần đến ba phép toán Insert, Delete và Search. Các tập động với ba phép toán này tạo thành **KDLTT từ điển (dictionary ADT).**

Một khái niệm quan trọng khác trong giải tích toán học là khái niệm dãy (a_1, …, a_i, …). Dãy khác tập hợp ở chỗ, các phần tử của dãy được sắp xếp theo một thứ tự xác định: a_1 là phần tử đầu tiên, a_i là phần tử ở vị trí thứ i (i = 1, 2, …), một phần tử có thể xuất hiện ở nhiều vị trí khác nhau trong dãy. Chúng ta sẽ gọi một dãy hữu hạn là một **danh sách.** Sử dụng danh sách với tư cách là đối tượng dữ liệu, chúng ta cũng cần đến các thao tác Insert, Delete. Tuy nhiên, các phép toán Insert và Delete trên danh sách có khác với

các phép toán này trên tập động. Nếu ký hiệu L là danh sách thì phép toán Insert (L, x, i) có nghĩa là xen phần tử x và vị trí thứ i trong danh sách L, còn Delete (L, i) là loại phần tử ở vị trí thứ i khỏi danh sách L. Trên danh sách còn có thể tiến hành nhiều phép toán khác. Các danh sách cùng với các phép toán trên danh sách tạo thành **KDLTT danh sách (list ADT**). KDLTT danh sách sẽ được nghiên cứu trong chương 4 và 5.

Trong nhiều trường hợp, khi thiết kế thuật toán, chúng ta chỉ cần sử dụng hạn chế hai phép toán Insert và Delete trên danh sách. Các danh sách với hai phép toán Insert và Delete chỉ được phép thực hiện ở một đầu danh sách lập ra một KDLTT mới: **KDLTT ngăn xếp (stack ADT**). Nếu chúng ta xét các danh sách với phép toán Insert chỉ được phép thực hiện ở một đầu danh sách, còn phép toán Delete chỉ được phép thực hiện ở một đầu khác của danh sách, chúng ta có **KDLTT hàng đợi (queue ADT**). Ngăn xếp được nghiên cứu trong chương 6, hàng đợi trong chương 7.

Cây là một tập hợp với cấu trúc phân cấp. Một dạng cây đặc biệt là cây nhị phân, trong cây nhị phân mỗi đỉnh chỉ có nhiều nhất hai đỉnh con được phân biệt là đỉnh con trái và đỉnh con phải. **KDLTT cây nhị phân** (**binary tree ADT**) sẽ được nghiên cứu trong chương 9.

KDLTT hàng ưu tiên (priority queue ADT) là một biến thể của KDLTT từ điển. Hàng ưu tiên là một tập động với ba phép toán Insert, FindMin, DeleteMin. Hàng ưu tiên sẽ được trình bày trong chương 11.

Các KDLTT trên đây đóng vai trò cực kỳ quan trọng trong thiết kế chương trình, đặc biệt KDLTT từ điển, bởi vì trong phần lớn các chương trình chúng ta cần lưu một tập dữ liệu rồi thì tiến hành tìm kiếm dữ liệu và cập nhật tập dữ liệu đó (bởi xen, loại dữ liệu).

Như chúng ta đã nhấn mạnh trong chương 2, một KDLTT có thể có nhiều cách cài đặt. Chúng ta sẽ cài đặt các KDLTT bởi các lớp C + +, và sẽ phân tích, đánh gía hiệu quả của các phép toán trong mỗi cách cài đặt. Chúng ta có thể cài đặt danh sách bởi mảng tĩnh, bởi mảng động hoặc bởi CTDL danh sách liên kết (chương 4).

Có nhiều cách cài đặt tập động. Có thể cài đặt tập động bởi danh sách, danh sách được sắp (chương 4). Cài đặt tập động bởi **cây tìm kiếm nhị phân** (**binary search tree**) sẽ được trình bày trong chương 9. Đó là một trong các cách cài đặt tập động đơn giản và hiệu quả.

Bảng băm (hashing table) là một CTDL đặc biệt thích hợp cho cài đặt từ điển. Cài đặt từ điển bởi bảng băm là nội dung của chương 10.

BÀI TẬP

1. Hãy nói rõ sự khác nhau giữa hai mục public và private của một lớp.
2. Hãy mô tả vai trò của hàm kiến tạo và hàm huỷ? Có các loại hàm kiến tạo nào?
3. Sự khác nhau giữa hàm kiến tạo copy và toán tử gán?
4. Nếu ta không cung cấp cho lớp hàm kiến tạo và hàm huỷ thì kết quả sẽ như thế nào?
5. Khi nào trong một lớp nhất thiết phải đưa vào hàm huỷ, hàm kiến tạo copy, toán tử gán?
6. Sự khác nhau giữa hàm thành phần và hàm bạn của một lớp?
7. Hãy cài đặt lớp xâu ký tự theo các chỉ dẫn sau. Xâu ký tự được biểu diễn bởi mảng động. Lớp cần chứa các hàm thực hiện các công việc sau: xác định độ dài của xâu, truy cập tới ký tự thứ i trong xâu, kết nối hai xâu thành một xâu, các phép toán so sánh hai xâu, đọc và viết ra xâu. Tất cả các hàm cần phải cài đặt là hàm toán tử, chỉ trừ hàm xác định độ dài của xâu.