Chương 5. Mảng, con trỏ, tham chiếu

Chương này giới thiệu về mảng, con trỏ, các kiểu dữ liệu tham chiếu và minh họa cách dùng chúng để định nghĩa các biến.

Mảng (array) gồm một tập các đối tượng (được gọi là **các phần tử**) tất cả chúng có cùng kiểu và được sắp xếp liên tiếp trong bộ nhớ. Nói chung chỉ có mảng là có tên đại diện chứ không phải là các phần tử của nó. Mỗi phần tử được xác định bởi một **chỉ số** biểu thị vị trí của phần tử trong mảng. Số lượng phần tử trong mảng được gọi là **kích thước** của mảng. Kích thước của mảng là cố định và phải được xác định trước; nó không thể thay đổi trong suốt quá trình thực hiện chương trình.

Mảng đại diện cho dữ liệu hỗn hợp gồm nhiều hạng mục riêng lẻ tương tự. Ví dụ: danh sách các tên, bảng các thành phố trên thế giới cùng với nhiệt độ hiện tại của các chúng, hoặc các giao dịch hàng tháng của một tài khoản ngân hàng.

Con trỏ (pointer) đơn giản là địa chỉ của một đối tượng trong bộ nhớ. Thông thường, các đối tượng có thể được truy xuất trong hai cách: trực tiếp bởi tên đại diện hoặc gián tiếp thông qua con trỏ. Các biến con trỏ được định nghĩa trỏ tới các đối tượng của một kiểu cụ thể sao cho khi con trỏ hủy thì vùng nhớ mà đối tượng chiếm giữ được thu hồi.

Các con trỏ thường được dùng cho việc tạo ra các **đối tượng động** trong thời gian thực thi chương trình. Không giống như các đối tượng bình thường (toàn cục và cục bộ) được cấp phát lưu trữ trên runtime stack, một đối tượng động được cấp phát vùng nhớ từ vùng lưu trữ khác được gọi là **heap**. Các đối tượng không tuân theo các luật phạm vi thông thường. Phạm vi của chúng được điều khiển rõ ràng bởi lập trình viên.

Tham chiếu (reference) cung cấp một tên tượng trưng khác gọi là **biệt hiệu** (alias) cho một đối tượng. Truy xuất một đối tượng thông qua một tham chiếu giống như là truy xuất thông qua tên gốc của nó. Tham chiếu nâng cao tính hữu dụng của các con trỏ và sự tiện lợi của việc truy xuất trực tiếp các đối tượng. Chúng được sử dụng để hỗ trợ các kiểu gọi thông qua tham chiếu của các tham số hàm đặc biệt khi các đối tượng lớn được truyền tới hàm.

5.1. Mảng (Array)

Biến mảng được định nghĩa bằng cách đặc tả kích thước mảng và kiểu các phần tử của nó. Ví dụ một mảng biểu diễn 10 thước đo chiều cao (mỗi phần tử là một số nguyên) có thể được định nghĩa như sau:

```
int heights[10];
```

Mỗi phần tử trong mảng có thể được truy xuất thông qua chỉ số mảng. Phần tử đầu tiên của mảng luôn có chỉ số 0. Vì thế, heights[0] và heights[9] biểu thị tương ứng cho phần tử đầu và phần tử cuối của mảng heights. Mỗi phần tử của mảng heights có thể được xem như là một biến số nguyên. Vì thế, ví dụ để đặt phần tử thứ ba tới giá trị 177 chúng ta có thể viết:

```
heights[2] = 177;
```

Việc cố gắng truy xuất một phần tử mảng không tồn tại (ví dụ, heights[-1] hoặc heights[10]) dẫn tới lỗi thực thi rất nghiêm trọng (được gọi là lỗi 'vượt ngoài biên').

Việc xử lý mảng thường liên quan đến một vòng lặp duyệt qua các phần tử mảng lần lượt từng phần tử một. Danh sách 5.1 minh họa điều này bằng việc sử dụng một hàm nhận vào một mảng các số nguyên và trả về giá trị trung bình của các phần tử trong mảng.

Danh sách 5.1

```
1  const int size = 3;

2  double Average (int nums[size])
3  {
4        double average = 0;

5        for (register i = 0; i < size; ++i)
6              average += nums[i];
7        return average/size;
8  }
```

Giống như các biến khác, một mảng có thể có một **bộ khởi tạo**. Các dấu ngoặc nhọn được sử dụng để đặc tả danh sách các giá trị khởi tạo được phân cách bởi dấu phẩy cho các phần tử mảng. Ví dụ,

```
int nums[3] = {5, 10, 15};
```

khởi tạo ba phần tử của mảng nums tương ứng tới 5, 10, và 15. Khi số giá trị trong bộ khởi tạo nhỏ hơn số phần tử thì các phần tử còn lại được khởi tạo tới 0:

```
int nums[3] = {5, 10};        // nums[2] khởi tạo tới 0
```

Khi bộ khởi tạo được sử dụng hoàn tất thì kích cỡ mảng trở thành dư thừa bởi vì số các phần tử là ẩn trong bộ khởi tạo. Vì thế định nghĩa đầu tiên của nums có thể viết tương đương như sau:

```
int nums[] = {5, 10, 15};      // không cần khai báo tường minh
                               // kích cỡ của mảng
```

Một tình huống khác mà kích cỡ có thể được bỏ qua đối với mảng tham số hàm. Ví dụ, hàm Average ở trên có thể được cải tiến bằng cách viết lại nó sao cho kích cỡ mảng nums không cố định tới một hằng mà được chỉ định bằng một tham số thêm vào. Danh sách 5.2 minh họa điều này.

Danh sách 5.2

```
1  double Average (int nums[], int size)
2  {
3        double average = 0;

4        for (register i = 0; i < size; ++i)
5              average += nums[i];
6        return average/size;
7  }
```

Một chuỗi C++ chỉ là một mảng các ký tự. Ví dụ,

```
char str[] = "HELLO";
```

định nghĩa chuỗi str là một mảng của 6 ký tự: năm chữ cái và một ký tự null. Ký tự kết thúc null được chèn vào bởi trình biên dịch. Trái lại,

```
char   str[] = {'H', 'E', 'L', 'L', 'O'};
```

định nghĩa str là mảng của 5 ký tự.

Kích cỡ của mảng có thể được tính một cách dễ dàng nhờ vào toàn tử sizeof. Ví dụ, với mảng ar đã cho mà kiểu phần tử của nó là Type thì kích cỡ của ar là:

```
sizeof(ar) / sizeof(Type)
```

5.2. Mảng đa chiều

Mảng có thể có hơn một chiều (nghĩa là, hai, ba, hoặc cao hơn.Việc tổ chức mảng trong bộ nhớ thì cũng tương tự không có gì thay đổi (một chuỗi liên tiếp các phần tử) nhưng cách tổ chức mà lập trình viên có thể lĩnh hội được thì lại khác. Ví dụ chúng ta muốn biểu diễn nhiệt độ trung bình theo từng mùa cho ba thành phố chính của Úc (xem Bảng 5.1).

Bảng 5.1 **Nhiệt độ trung bình theo mùa.**

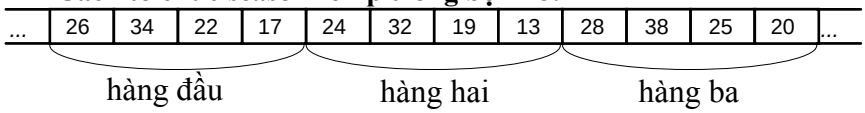
	Mùa xuân	Mùa hè	Mùa thu	Mùa đông
Sydney	26	34	22	17
Melbourne	24	32	19	13
Brisbane	28	38	25	20

Điều này có thể được biểu diễn bằng một mảng hai chiều mà mỗi phần tử mảng là một số nguyên:

```
int    seasonTemp[3][4];
```

Cách tổ chức mảng này trong bộ nhớ như là 12 phần tử số nguyên liên tiếp nhau. Tuy nhiên, lập trình viên có thể tưởng tượng nó như là một mảng gồm ba hàng với mỗi hàng có bốn phần tử số nguyên (xem Hình 5.1).

Hình 5.1 **Cách tổ chức seasonTemp trong bộ nhớ.**



Như trước, các phần tử được truy xuất thông qua chỉ số mảng. Một chỉ số riêng biệt được cần cho mỗi mảng. Ví dụ, nhiệt độ mùa hè trung bình của thành phố Sydney (hàng đầu tiên cột thứ hai) được cho bởi seasonTemp[0][1].

Mảng có thể được khởi tạo bằng cách sử dụng một bộ khởi tạo lồng nhau:

```
int seasonTemp[3][4] = {
        {26, 34, 22, 17},
        {24, 32, 19, 13},
        {28, 38, 25, 20}
};
```

Bởi vì điều này ánh xạ tới mảng một chiều gồm 12 phần tử trong bộ nhớ nên nó tương đương với:

```
int seasonTemp[3][4] = {
        26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 20
};
```

Bộ khởi tạo lồng nhau được ưa chuộng hơn bởi vì nó linh hoạt và dễ hiểu hơn. Ví dụ, nó có thể khởi tạo chỉ phần tử đầu tiên của mỗi hàng và phần còn lại mặc định là 0:

```
int seasonTemp[3][4] = {{26}, {24}, {28}};
```

Chúng ta cũng có thể bỏ qua chiều đầu tiên và để cho nó được dẫn xuất từ bộ khởi tạo:

```
int seasonTemp[][4] = {
        {26, 34, 22, 17},
        {24, 32, 19, 13},
```

```
        {28, 38, 25, 20}
    };
```

Xử lý mảng nhiều chiều thì tương tự như là mảng một chiều nhưng phải xử lý các vòng lặp lồng nhau thay vì vòng lặp đơn. Danh sách 5.3 minh họa điều này bằng cách trình bày một hàm để tìm nhiệt độ cao nhất trong mảng seasonTemp.

Danh sách 5.3

```
1   const int rows          = 3;
2   const int columns       = 4;

3   int seasonTemp[rows][columns] = {
4           {26, 34, 22, 17},
5           {24, 32, 19, 13},
6           {28, 38, 25, 20}
7   };

8   int HighestTemp (int temp[rows][columns])
9   {
10          int     highest = 0;

11          for (register i = 0; i < rows; ++i)
12          for (register j = 0; j < columns; ++j)
13                  if (temp[i][j] > highest)
14                  highest = temp[i][j];
15          return highest;
16  }
```

5.3. Con trỏ

Con trỏ đơn giản chỉ là *địa chỉ* của một vị trí bộ nhớ và cung cấp cách gián tiếp để truy xuất dữ liệu trong bộ nhớ. Biến con trỏ được định nghĩa để "trỏ tới" dữ liệu thuộc kiểu dữ liệu cụ thể. Ví dụ,

```
int         *ptr1;          // trỏ tới một int
char        *ptr2;          // trỏ tới một char
```

Giá trị của một biến con trỏ là địa chỉ mà nó trỏ tới. Ví dụ, với các định nghĩa đã có và

```
int         num;
```

chúng ta có thể viết:

```
ptr1 = &num;
```

Ký hiệu **&** là toán tử **lấy địa chỉ**; nó nhận một biến như là một đối số và trả về địa chỉ bộ nhớ của biến đó. Tác động của việc gán trên là địa chỉ của

num được khởi tạo tới ptr1. Vì thế, chúng ta nói rằng ptr1 trỏ tới num. Hình 5.2 minh họa sơ lược điều này.

Hình 5.2 Một con trỏ số nguyên đơn giản.

Với ptr1 trỏ tới num thì biểu thức *ptr1 nhận giá trị của biến ptr1 trỏ tới và vì thế nó tương đương với num. Ký hiệu * là toán tử **lấy giá trị**; nó nhận con trỏ như một đối số và trả về nội dung của vị trí mà con trỏ trỏ tới.

Thông thường thì kiểu con trỏ phải khớp với kiểu dữ liệu mà được trỏ tới. Tuy nhiên, một con trỏ kiểu void* sẽ hợp với tất cả các kiểu. Điều này thật thuận tiện để định nghĩa các con trỏ có thể trỏ đến dữ liệu của những kiểu khác nhau hay là các kiểu dữ liệu gốc không được biết.

Con trỏ có thể được **ép** (chuyển kiểu) thành một kiểu khác. Ví dụ,

```
ptr2 = (char*) ptr1;
```

chuyển con trỏ ptr1 thành con trỏ char trước khi gán nó tới con trỏ ptr2.

Không quan tâm đến kiểu của nó thì con trỏ có thể được gán tới giá trị null (gọi là con trỏ **null**). Con trỏ null được sử dụng để khởi tạo cho các con trỏ và tạo ra điểm kết thúc cho các cấu trúc dựa trên con trỏ (ví dụ, danh sách liên kết).

5.4. Bộ nhớ động

Ngoài vùng nhớ stack của chương trình (thành phần được sử dụng để lưu trữ các biến toàn cục và các khung stack cho các lời gọi hàm), một vùng bộ nhớ khác gọi là **heap** được cung cấp. Heap được sử dụng cho việc cấp phát động các khối bộ nhớ trong thời gian thực thi chương trình. Vì thế heap cũng được gọi là **bộ nhớ động** (dynamic memory). Vùng nhớ stack của chương trình cũng được gọi là **bộ nhớ tĩnh** (static memory).

Có hai toán tử được sử dụng cho việc cấp phát và thu hồi các khối bộ nhớ trên heap. Toán tử new nhận một kiểu như là một đối số và được cấp phát một khối bộ nhớ cho một đối tượng của kiểu đó. Nó trả về một con trỏ tới khối đã được cấp phát. Ví dụ,

```
int *ptr = new int;
char *str = new char[10];
```

cấp phát tương ứng một khối cho lưu trữ một số nguyên và một khối đủ lớn cho lưu trữ một mảng 10 ký tự.

Bộ nhớ được cấp phát từ heap không tuân theo luật phạm vi như các biến thông thường. Ví dụ, trong

```
void Foo (void)
{
    char *str = new char[10];
    //...
}
```

khi Foo trả về các biến cục bộ str được thu hồi nhưng các khối bộ nhớ được trỏ tới bởi str thì không. Các khối bộ nhớ vẫn còn cho đến khi chúng được giải phóng rõ ràng bởi các lập trình viên.

Toán tử delete được sử dụng để giải phóng các khối bộ nhớ đã được cấp phát bởi new. Nó nhận một con trỏ như là đối số và giải phóng khối bộ nhớ mà nó trỏ tới. Ví dụ:

```
delete ptr;          // xóa một đối tượng
delete [] str;       // xóa một mảng các đối tượng
```

Chú ý rằng khi khối nhớ được xóa là một mảng thì một cặp dấu [] phải được chèn vào để chỉ định công việc này. Sự quan trọng sẽ được giải thích sau đó khi chúng ta thảo luận về lớp.

Toán tử delete nên được áp dụng tới con trỏ mà trỏ tới bất cứ thứ gì vì một đối tượng được cấp phát động (ví dụ, một biến trên stack), một lỗi thực thi nghiêm trọng có thể xảy ra. Hoàn toàn vô hại khi áp dụng delete tới một biến không là con trỏ.

Các đối tượng động được sử dụng để tạo ra dữ liệu kéo dài tới khi lời gọi hàm tạo ra chúng. Danh sách 5.4 minh họa điều này bằng cách sử dụng một hàm nhận một tham số chuỗi và trả về *bản sao* của một chuỗi.

Danh sách 5.4

```
1  #include <string.h>

2  char* CopyOf (const char *str)
3  {
4      char *copy = new char[strlen(str) + 1];

5      strcpy(copy, str);
6      return copy;
7  }
```

Chú giải

1 Đây là tập tin header chuỗi chuẩn khai báo các dạng hàm cho thao tác trên chuỗi.

4 Hàm strlen (được khai báo trong thư viện string.h) đếm các ký tự trong đối số chuỗi của nó cho đến (nhưng không vượt quá) ký tự null sau cùng. Bởi vì ký tự null không được tính vào trong việc đếm nên chúng ta cộng thêm 1 tới tổng và cấp phát một mảng ký tự của kích thước đó.

5 Hàm strcpy (được khai báo trong thư viện string.h) sao chép đối số thứ hai đến đối số thứ nhất của nó theo từng ký tự một bao gồm luôn cả ký tự null sau cùng.

Vì tài nguyên bộ nhớ là có giới hạn nên rất có thể bộ nhớ động có thể bị cạn kiệt trong thời gian thực thi chương trình, đặc biệt là khi nhiều khối lớn được cấp phát và không có giải phóng. Toán tử new không thể cấp phát một khối có kích thước được yêu cầu thì nó trả về 0. Chính lập trình viên phải chịu trách nhiệm giải quyết những vấn đề này. Cơ chế điều khiển ngoại lệ của C++ cung cấp một cách thức thực tế giải quyết những vấn đề như thế.

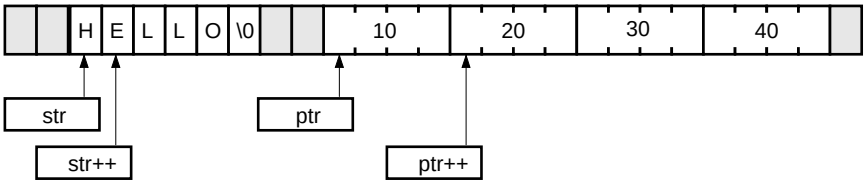
5.5. Tính toán con trỏ

Trong C++ chúng ta có thể thực hiện cộng hay trừ số nguyên trên con trỏ. Điều này thường xuyên được sử dụng bởi các lập trình viên được gọi là các tính toán con trỏ. Tính toán con trỏ thì không giống như là tính toán số nguyên bởi vì kết quả phụ thuộc vào kích thước của đối tượng được trỏ tới. Ví dụ, một kiểu int được biểu diễn bởi 4 byte. Bây giờ chúng ta có

```
char *str = "HELLO";
int nums[] = {10, 20, 30, 40};
int *ptr = &nums[0];              // trỏ tới phần tử đầu tiên
```

str++ tăng str lên một char (nghĩa là 1 byte) sao cho nó trỏ tới ký tự thứ hai của chuỗi "HELLO" nhưng ngược lại ptr++ tăng ptr lên một int (nghĩa là 4 bytes) sao cho nó trỏ tới phần tử thứ hai của nums. Hình 5.3 minh họa sơ lược điều này.

Hình 5.3 Tính toán con trỏ.



Vì thế, các phần tử của chuỗi "HELLO" có thể được tham khảo tới như *str, *(str + 1), *(str + 2), vâng vâng. Tương tự, các phần tử của nums có thể được tham khảo tới như *ptr, *(ptr + 1), *(ptr + 2), và *(ptr + 3).

Một hình thức khác của tính toán con trỏ được cho phép trong C++ liên quan đến trừ hai con trỏ của cùng kiểu. Ví dụ:

```
int *ptr1 = &nums[1];
int *ptr2 = &nums[3];
int n = ptr2 - ptr1;              // n trở thành 2
```

Tính toán con trỏ cần khéo léo khi xử lý các phần tử của mảng. Danh sách 5.5 trình bày ví dụ một hàm sao chép chuỗi tương tự như hàm định nghĩa sẵn strcpy.

Danh sách 5.5

```
1  void CopyString (char *dest, char *src)
2  {
3        while (*dest++ = *src++)    ;
4  }
```

Chú giải

3 Điều kiện của vòng lặp này gán nội dung của chuỗi src cho nội dung của chuỗi dest và sau đó tăng cả hai con trỏ. Điều kiện này trở thành 0 khi ký tự null kết thúc của chuỗi src được chép tới chuỗi dest.

Một biến mảng (như nums) chính nó là địa chỉ của phần tử đầu tiên của mảng mà nó đại diện. Vì thế các phần tử của mảng nums cũng có thể được tham khảo tới bằng cách sử dụng tính toán con trỏ trên nums, nghĩa là nums[i] tương đương với *(nums + i). Khác nhau giữa nums và ptr ở chỗ nums là một hằng vì thế nó không thể được tạo ra để trỏ tới bất cứ thứ gì nữa trong khi ptr là một biến và có thể được tạo ra để trỏ tới các số nguyên bất kỳ.

Danh sách 5.6 trình bày hàm HighestTemp (đã được trình bày trước đó trong Danh sách 5.3) có thể được cải tiến như thế nào bằng cách sử dụng tính toán con trỏ.

Danh sách 5.6

```
1  int HighestTemp (const int *temp, const int rows, const int columns)
2  {
3        int    highest = 0;

4        for (register i = 0; i < rows; ++i)
5        for (register j = 0; j < columns; ++j)
6                  if (*(temp + i * columns + j) > highest)
7                        highest = *(temp + i * columns + j);
8        return highest;
9  }
```

Chú giải

1 Thay vì truyền một mảng tới hàm, chúng ta truyền một con trỏ int và hai tham số thêm vào đặc tả kích cỡ của mảng. Theo cách này thì hàm không bị hạn chế tới một kích thước mảng cụ thể.

6 Biểu thức *(temp + i * columns + j) tương đương với temp[i][j] trong phiên bản hàm trước.

Hàm HighestTemp có thể được đơn giản hóa hơn nữa bằng cách xem temp như là một mảng một chiều của row * column số nguyên. Điều này được trình bày trong Danh sách 5.7.

Danh sách 5.7

```
1  int HighestTemp (const int *temp, const int rows, const int columns)
2  {
3      int    highest = 0;

4      for (register i = 0; i < rows * columns; ++i)
5          if (*(temp + i) > highest)
6              highest = *(temp + i);
7      return highest;
8  }
```

5.6. Con trỏ hàm

Chúng ta có thể lấy địa chỉ một hàm và lưu vào trong một con trỏ hàm. Sau đó con trỏ có thể được sử dụng để gọi gián tiếp hàm. Ví dụ,

```
int (*Compare)(const char*, const char*);
```

định nghĩa một con trỏ hàm tên là Compare có thể giữ địa chỉ của bất kỳ hàm nào nhận hai con trỏ ký tự hằng như là các đối số và trả về một số nguyên. Ví dụ hàm thư viện so sánh chuỗi strcmp thực hiện như thế. Vì thế:

```
Compare = &strcmp;              // Compare trỏ tới hàm strcmp
```

Toán tử & không cần thiết và có thể bỏ qua:

```
Compare = strcmp;               // Compare trỏ tới hàm strcmp
```

Một lựa chọn khác là con trỏ có thể được định nghĩa và khởi tạo một lần:

```
int (*Compare)(const char*, const char*) = strcmp;
```

Khi địa chỉ hàm được gán tới con trỏ hàm thì hai kiểu phải khớp với nhau. Định nghĩa trên là hợp lệ bởi vì hàm strcmp có một nguyên mẫu hàm khớp với hàm.

```
int strcmp(const char*, const char*);
```

Với định nghĩa trên của Compare thì hàm strcmp hoặc có thể được gọi trực tiếp hoặc có thể được gọi gián tiếp thông qua Compare. Ba lời gọi hàm sau là tương đương:

```
strcmp("Tom", "Tim");           // gọi trực tiếp
(*Compare)("Tom", "Tim");       // gọi gián tiếp
Compare("Tom", "Tim");          // gọi gián tiếp (ngắn gọn)
```

Cách sử dụng chung của con trỏ hàm là truyền nó như một đối số tới một hàm khác; bởi vì thông thường các hàm sau yêu cầu các phiên bản khác nhau của hàm trước trong các tình huống khác nhau. Một ví dụ dễ hiểu là hàm tìm

kiếm nhị phân thông qua một mảng sắp xếp các chuỗi. Hàm này có thể sử dụng một hàm so sánh (như là strcmp) để so sánh chuỗi tìm kiếm ngược lại chuỗi của mảng. Điều này có thể không thích hợp đối với tất cả các trường hợp. Ví dụ, hàm strcmp là phân biệt chữ hoa hay chữ thường. Nếu chúng ta thực hiện tìm kiếm theo cách không phân biệt dạng chữ sau đó một hàm so sánh khác sẽ được cần.

Như được trình bày trong Danh sách 5.8 bằng cách để cho hàm so sánh một tham số của hàm tìm kiếm, chúng ta có thể làm cho hàm tìm kiếm độc lập với hàm so sánh.

Danh sách 5.8

```
1  int BinSearch (char *item, char *table[], int n,
2                 int (*Compare)(const char*, const char*))
3  {
4      int bot = 0;
5      int top = n - 1;
6      int mid, cmp;

7      while (bot <= top) {
8          mid = (bot + top) / 2;
9          if ((cmp = Compare(item,table[mid])) == 0)
10             return mid;          // tra ve chi so hangg muc
11         else if (cmp < 0)
12             top = mid - 1;       // gioi hạn tim kiem toi nua thap hon
13         else
14             bot = mid + 1;       // gioi han tim kiem toi nua cao hon
15     }
16     return -1;                   // khong tim thay
17 }
```

Chú giải

1 Tìm kiếm nhị phân là một giải thuật nổi tiếng để tìm kiếm thông qua một danh sách các hạng mục đã được sắp xếp. Danh sách tìm kiếm được biểu diễn bởi table – một mảng các chuỗi có kích thước n. Hạng mục tìm kiếm được biểu thị bởi item.

2 Compare là con trỏ hàm được sử dụng để so sánh item với các phần tử của mảng.

7 Ở mỗi vòng lặp, việc tìm kiếm được giảm đi phân nữa. Điều này được lặp lại cho tới khi hai đầu tìm kiếm giao nhau (được biểu thị bởi bot và top) hoặc cho tới khi một so khớp được tìm thấy.

9 Hạng mục được so sánh với mục ở giữa của mảng.

10 Nếu item khớp với hạng mục giữa thì trả về chỉ mục của phần sau.

11 Nếu item nhỏ hơn hạng mục giữa thì sau đó tìm kiếm được giới hạn tới nữa thấp hơn của mảng.

14 Nếu item lớn hơn hạng mục giữa thì sau đó tìm kiếm được giới hạn tới nữa cao hơn của mảng..

16 Trả về -1 để chỉ định rằng không có một hạng mục so khớp.

Ví dụ sau trình bày hàm BinSearch có thể được gọi với strcmp được truyền như hàm so sánh như thế nào:

```
char *cities[] = {"Boston", "London", "Sydney", "Tokyo"};
cout << BinSearch("Sydney", cities, 4, strcmp) << '\n';
```

Điều này sẽ xuất ra 2 như được mong đợi.

5.7. Tham chiếu

Một tham chiếu (reference) là một biệt hiệu (alias) cho một đối tượng. Ký hiệu được dùng cho định nghĩa tham chiếu thì tương tự với ký hiệu dùng cho con trỏ ngoại trừ & được sử dụng thay vì *. Ví dụ,

```
double num1 = 3.14;
double &num2 = num1;          // num2 là một tham chiếu tới num1
```

định nghĩa num2 như là một tham chiếu tới num1. Sau định nghĩa này cả hai num1 và num2 tham khảo tới cùng một đối tượng như thể chúng là cùng biến. Cần biết rõ là một tham chiếu không tạo ra một bản sao của một đối tượng mà chỉ đơn thuần là một biệt hiệu cho nó. Vì vậy, sau phép gán

```
num1 = 0.16;
```

cả hai num1 và num2 sẽ biểu thị giá trị 0.16.

Một tham chiếu phải luôn được khởi tạo khi nó được định nghĩa: nó là một biệt danh cho cái gì đó. Việc định nghĩa một tham chiếu rồi sau đó mới khởi tạo nó là không đúng luật.

```
double &num3;  // không đúng luật: tham chiếu không có khởi tạo
num3 = num1;
```

Bạn cũng có thể khởi tạo tham chiếu tới một hằng. Trong trường hợp này, một bản sao của hằng được tạo ra (sau khi bất kỳ sự chuyển kiểu cần thiết nào đó) và tham chiếu được thiết lập để tham chiếu tới bản sao đó.

```
int &n = 1;              // n tham khảo tới bản sao của 1
```

Lý do mà n lại tham chiếu tới bản sao của 1 hơn là tham chiếu tới chính 1 là sự an toàn. Bạn hãy xem xét điều gì sẽ xảy ra trong trường hợp sau:

```
int &x = 1;
++x;
int y = x + 1;
```

1 ở hàng đầu tiên và 1 ở hàng thứ ba giống nhau là cùng đối tượng (hầu hết các trình biên dịch thực hiện tối ưu hằng và cấp phát cả hai 1 trong cùng một vị trí bộ nhớ). Vì thế chúng ta mong đợi y là 3 nhưng nó có thể chuyển thành

4. Tuy nhiên, bằng cách ép buộc x là một bản sao của 1 nên trình biên dịch đảm bảo rằng đối tượng được biểu thị bởi x sẽ khác với cả hai 1.

Việc sử dụng chung nhất của tham chiếu là cho các tham số của hàm. Các tham số của hàm thường làm cho dễ dàng kiểu truyền-bằng-tham chiếu, trái với kiểu truyền-bằng-giá trị mà chúng ta sử dụng đến thời điểm này. Để quan sát sự khác nhau hãy xem xét ba hàm swap trong Danh sách 5.9.

Danh sách 5.9

```
1  void Swap1 (int x, int y)          // truyền bằng trị (đối tượng)
2  {
3        int temp = x;
4        x = y;
5        y = temp;
6  }

7  void Swap2 (int *x, int *y)        // truyền bằng địa chỉ (con trỏ)
8  {
9        int temp = *x;
10       *x = *y;
11       *y = temp;
12 }

13 void Swap3 (int &x, int &y)        // truyền bằng tham chiếu
14 {
15       int temp = x;
16       x = y;
17       y = temp;
18 }
```

Chú giải

1 Mặc dù Swap1 chuyển đổi x và y, điều này không ảnh hưởng tới các đối số được truyền tới hàm bởi vì Swap1 nhận một *bản sao* của các đối số. Những thay đổi trên bản sao thì không ảnh hưởng đến dữ liệu gốc.

7 Swap2 vượt qua vấn đề của Swap1 bằng cách sử dụng các tham số con trỏ để thay thế. Thông qua giải tham khảo (dereferencing) các con trỏ Swap2 lấy giá trị gốc và chuyển đổi chúng.

13 Swap3 vượt qua vấn đề của Swap1 bằng cách sử dụng các tham số tham chiếu để thay thế. Các tham số trở thành các biệt danh cho các đối số được truyền tới hàm và vì thế chuyển đổi chúng khi cần.

Swap3 có thuận lợi thêm, cú pháp gọi của nó giống như Swap1 và không có liên quan đến định địa chỉ (addressing) hay là giải tham khảo (dereferencing). Hàm main sau minh họa sự khác nhau giữa các lời gọi hàm Swap1, Swap2, và Swap3.

```
int main (void)
{
      int i = 10, j = 20;
      Swap1(i, j);              cout << i << ", " << j << '\n';
      Swap2(&i, &j);  cout << i << ", " << j << '\n';
```

```
    Swap3(i, j);            cout << i << ", " << j << '\n';
}
```

Khi chạy chương trình sẽ cho kết quả sau:

```
10, 20
20, 10
20, 10
```

5.8. Định nghĩa kiểu

Typedef là cú pháp để mở đầu cho các tên tượng trưng cho các kiểu dữ liệu. Như là một tham chiếu định nghĩa một biệt danh cho một đối tượng, một typedef định nghĩa một biệt danh cho một kiểu. Mục đích cơ bản của nó là để đơn giản hóa các khai báo kiểu phức tạp khác như một sự trợ giúp để cải thiện khả năng đọc. Ở đây là một vài ví dụ:

```
typedef char *String;
typedef char Name[12];
typedef unsigned int uint;
```

Tác dụng của các định nghĩa này là String trở thành một biệt danh cho char*, Name trở thành một biệt danh cho một mảng gồm 12 char, và uint trở thành một biệt danh cho unsigned int. Vì thế:

```
String    str;        // thì tương tự như: char *str;
Name      name;             // thì tương tự như: char name[12];
uint   n;             // thì tương tự như: unsigned int n;
```

Khai báo phức tạp của Compare trong Danh sách 5.8 là một minh họa tốt cho typedef:

```
typedef int (*Compare)(const char*, const char*);

int BinSearch (char *item, char *table[], int n, Compare comp)
{
    //...
        if ((cmp = comp(item, table[mid])) == 0)
            return mid;
    //...
}
```

typedef mở đầu Compare như là một tên kiểu mới cho bất kỳ hàm với nguyên mẫu (prototype) cho trước. Người ta cho rằng điều này làm cho dấu hiệu của BinSearch đơn giản hơn.

Bài tập cuối chương 5

5.1 Định nghĩa hai hàm tương ứng thực hiện nhập vào các giá trị cho các phần tử của mảng và xuất các phần tử của mảng:

```
void ReadArray  (double nums[], const int size);
void WriteArray (double nums[], const int size);
```

5.2 Định nghĩa một hàm đảo ngược thứ tự các phần tử của một mảng số thực:

```
void Reverse (double nums[], const int size);
```

5.3 Bảng sau đặc tả các nội dung chính của bốn loại hàng của các ngũ cốc điểm tâm. Định nghĩa một mảng hai chiều để bắt dữ liệu này:

	Sơ	Đường	Béo	Muối
Top Flake	12g	25g	16g	0.4g
Cornabix	22g	4g	8g	0.3g
Oatabix	28g	5g	9g	0.5g
Ultrabran	32g	7g	2g	0.2g

Viết một hàm xuất bảng này từng phần tử một.

5.4 Định nghĩa một hàm để nhập vào danh sách các tên và lưu trữ chúng như là các chuỗi được cấp phát động trong một mảng và một hàm để xuất chúng:

```
void ReadNames (char *names[], const int size);
void WriteNames (char *names[], const int size);
```

Viết một hàm khác để sắp xếp danh sách bằng cách sử dụng giải thuật sắp xếp nổi bọt (bubble sort):

```
void BubbleSort (char *names[], const int size);
```

Sắp xếp nổi bọt liên quan đến việc quét lặp lại danh sách, trong đó trong khi thực hiện quét các hạng mục kề nhau được so sánh và đổi chỗ nếu không theo thứ tự. Quét mà không liên quan đến việc đổi chỗ chỉ ra rằng danh sách đã được sắp xếp thứ tự.

5.5 Viết lại hàm sau bằng cách sử dụng tính toán con trỏ:

```
char* ReverseString (char *str)
{
        int len = strlen(str);
        char *result = new char[len + 1];

        for (register i = 0; i < len; ++i)
                result[i] = str[len - i - 1];
        result[len] = '\0';
        return result;
}
```

5.6 Viết lại giải thuật BubbleSort (từ bài 5.4) sao cho nó sử dụng một con trỏ hàm để so sánh các tên.

5.7 Viết lại các mã sau bằng cách sử dụng định nghĩa kiểu:

```
void (*Swap)(double, double);
char *table[];
char *&name;
usigned long *values[10][20];
```