
Chương 1. Mở đầu

Chương này giới thiệu những phần cơ bản của một chương trình C++. Chúng ta sử dụng những ví dụ đơn giản để trình bày cấu trúc các chương trình C++ và cách thức biên dịch chúng. Các khái niệm cơ bản như là hằng, biến, và việc lưu trữ chúng trong bộ nhớ cũng sẽ được thảo luận trong chương này. Sau đây là một đặc tả sơ bộ về khái niệm lập trình.

Lập trình

Máy tính số là một công cụ để giải quyết hàng loạt các **bài toán** lớn. Một lời giải cho một bài toán nào đó được gọi là một **giải thuật** (algorithm); nó mô tả một chuỗi các bước cần thực hiện để giải quyết bài toán. Một ví dụ đơn giản cho một bài toán và một giải thuật có thể là:

Bài toán: Sắp xếp một danh sách các số theo thứ tự tăng dần.

Giải thuật: Giả sử danh sách đã cho là *list1*; tạo ra một danh sách rỗng, *list2*, để lưu danh sách đã sắp xếp. Lặp đi lặp lại công việc, tìm số nhỏ nhất trong *list1*, xóa nó khỏi *list1*, và thêm vào phần tử kế tiếp trong danh sách *list2*, cho đến khi *list1* là rỗng.

Giải thuật được diễn giải bằng các thuật ngữ trừu tượng mang tính chất dễ hiểu. Ngôn ngữ thật sự được hiểu bởi máy tính là ngôn ngữ máy. Chương trình được diễn đạt bằng ngôn ngữ máy được gọi là **có thể thực thi**. Một chương trình được viết bằng bất kỳ một ngôn ngữ nào khác thì trước hết cần được dịch sang ngôn ngữ máy để máy tính có thể hiểu và thực thi nó.

Ngôn ngữ máy cực kỳ khó hiểu đối với lập trình viên vì thế họ không thể sử dụng trực tiếp ngôn ngữ máy để viết chương trình. Một sự trừu tượng khác là **ngôn ngữ assembly**. Nó cung cấp những tên dễ nhớ cho các lệnh và một ký hiệu dễ hiểu hơn cho dữ liệu. Bộ dịch được gọi là **assembler** chuyển ngôn ngữ assembly sang ngôn ngữ máy.

Ngay cả những ngôn ngữ assembly cũng khó sử dụng. Những ngôn ngữ cấp cao như C++ cung cấp các ký hiệu thuận tiện hơn nhiều cho việc thi hành các giải thuật. Chúng giúp cho các lập trình viên không phải nghĩ nhiều về các thuật ngữ cấp thấp, và giúp họ chỉ tập trung vào giải thuật. **Trình biên dịch** (compiler) sẽ đảm nhiệm việc dịch chương trình viết bằng ngôn ngữ cấp cao sang ngôn ngữ assembly. Mã assembly được tạo ra bởi trình biên dịch sau đó sẽ được tập hợp lại để cho ra một chương trình có thể thực thi.

1.1. Một chương trình C++ đơn giản

Danh sách 1.1 trình bày chương trình C++ đầu tiên. Chương trình này khi chạy sẽ xuất ra thông điệp Hello World.

Danh sách 1.1

```
1 #include <iostream.h>
2 int main (void)
3 {
4     cout << "Hello World\n";
5 }
```

Chú giải

- 1 Hàng này sử dụng chỉ thị tiền xử lý `#include` để chèn vào nội dung của tập tin header `iostream.h` trong chương trình. `iostream.h` là tập tin header chuẩn của C++ và chứa đựng các định nghĩa cho xuất và nhập.
- 2 Hàng này định nghĩa một hàm được gọi là `main`. Hàm có thể không có hay có nhiều **tham số** (parameters); các tham số này luôn xuất hiện sau tên hàm, giữa một cặp dấu ngoặc. Việc xuất hiện của từ `void` ở giữa dấu ngoặc chỉ định rằng hàm `main` không có tham số. Hàm có thể có **kiểu trả về**; kiểu trả về luôn xuất hiện trước tên hàm. Kiểu trả về cho hàm `main` là `int` (ví dụ: một số nguyên). Tất cả các chương trình C++ phải có một hàm `main` duy nhất. Việc thực thi chương trình luôn bắt đầu từ hàm `main`.
- 3 Dấu ngoặc nhọn bắt đầu thân của hàm `main`.
- 4 Hàng này là một **câu lệnh** (statement). Một lệnh là một sự tính toán để cho ra một giá trị. Kết thúc một lệnh thì luôn luôn được đánh dấu bằng dấu chấm phẩy (;). Câu lệnh này xuất ra **chuỗi** `"Hello World\n"` để gọi đến dòng xuất `cout`. Chuỗi là một dãy các ký tự được đặt trong cặp nháy kép. Ký tự cuối cùng trong chuỗi này (`\n`) là một ký tự xuống hàng (newline). Dòng là một đối tượng được dùng để thực hiện các xuất hoặc nhập. `cout` là dòng xuất chuẩn trong C++ (xuất chuẩn thường được hiểu là màn hình máy tính). Ký tự `<<` là toán tử xuất, nó xem dòng xuất như là toán hạng trái và xem biểu thức như là toán hạng phải, và tạo nên giá trị của biểu thức được gọi đến dòng xuất. Trong trường hợp này, kết quả là chuỗi `"Hello World\n"` được gọi đến dòng `cout`, làm cho nó được hiển thị trên màn hình máy tính.
- 5 Dấu ngoặc đóng kết thúc thân hàm `main`.

1.2. Biên dịch một chương trình C++

Bảng 1.1 trình bày chương trình trong danh sách 1.1 được biên dịch và chạy trong môi trường UNIX thông thường. Phần in đậm được xem như là đầu vào (input) của người dùng và phần in thường được xem như là đáp ứng của hệ thống. Dấu nhắc ở hàng lệnh UNIX xuất hiện như là ký tự dollar (\$).

Bảng 1.1

1	\$ CC hello.cc
2	\$ a.out
3	Hello World
4	\$

Chú giải

- 1 Lệnh đề triệu gọi bộ dịch AT&T của C++ trong môi trường UNIX là CC. Đối số cho lệnh này (hello.cc) là tên của tập tin chứa đựng chương trình. Theo qui định thì tên tập tin có phần mở rộng là .c, .C, hoặc là .cc. (Phần mở rộng này có thể là khác nhau đối với những hệ điều hành khác nhau)
- 2 Kết quả của sự biên dịch là một tập tin có thể thực thi mặc định là a.out. Để chạy chương trình, chúng ta sử dụng a.out như là lệnh.
- 3 Đây là kết quả được cung cấp bởi chương trình.
- 4 Dấu nhắc trở về hệ thống chỉ định rằng chương trình đã hoàn tất sự thực thi của nó.

Lệnh cc chấp nhận các phần tùy chọn. Mỗi tùy chọn xuất hiện như name, trong đó name là tên của tùy chọn (thường là một ký tự đơn). Một vài tùy chọn yêu cầu có đối số. Ví dụ tùy chọn xuất (-o) cho phép chỉ định rõ tập tin có thể được cung cấp bởi trình biên dịch thay vì là a.out. Bảng 1.2 minh họa việc sử dụng tùy chọn này bằng cách chỉ định rõ hello như là tên của tập tin có thể thực thi.

Bảng 1.2

1	\$ CC hello.cc -o hello
2	\$ hello
3	Hello World
4	\$

Mặc dù lệnh thực sự có thể khác phụ thuộc vào trình biên dịch, một thủ tục biên dịch tương tự có thể được dùng dưới môi trường MS-DOS. Trình biên dịch C++ dựa trên Windows đang tặng một môi trường thân thiện với người dùng mà việc biên dịch rất đơn giản bằng cách chọn lệnh từ menu. Qui định tên dưới MS-DOS và Windows là tên của tập tin nguồn C++ phải có phần mở rộng là .cpp.

1.3. Việc biên dịch C++ diễn ra như thế nào

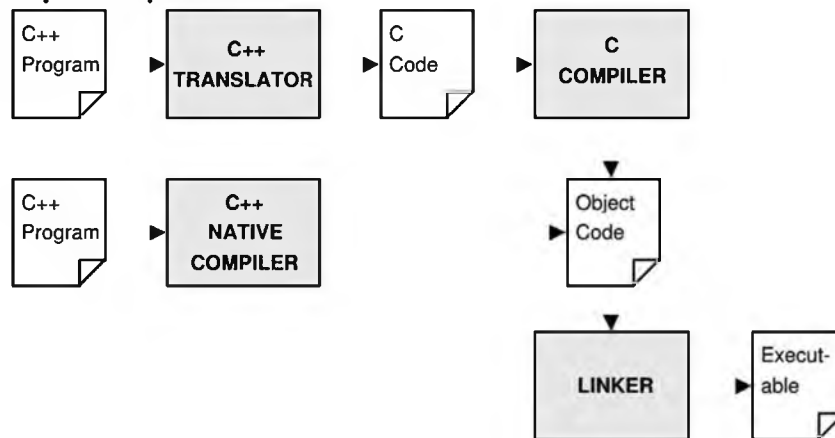
Biên dịch một chương trình C++ liên quan đến một số bước (hầu hết các bước là trong suốt với người dùng):

- Đầu tiên, **bộ tiền xử lý C++** xem qua mã trong chương trình và thực hiện các chỉ thị được chỉ định bởi các chỉ thị tiền xử lý (ví dụ, #include). Kết quả là một mã chương trình đã sửa đổi mà không còn chứa bất kỳ một chỉ thị tiền xử lý nào cả.

- Sau đó, **trình biên dịch C++** dịch các mã của chương trình. Trình biên dịch có thể là một trình biên dịch C++ thật sự phát ra mã assembly hay mã máy, hoặc chỉ là trình chuyển đổi dịch mã sang C. Ở trường hợp thứ hai, mã C sau khi được dịch ra sẽ tạo thành mã assembly hay mã máy thông qua trình biên dịch C. Trong cả hai trường hợp, đầu ra có thể không hoàn chỉnh vì chương trình tham khảo tới các thủ tục trong thư viện còn chưa được định nghĩa như một phần của chương trình. Ví dụ Danh sách 1.1 tham chiếu tới toán tử << mà thực sự được định nghĩa trong một thư viện IO riêng biệt.
- Cuối cùng, **trình liên kết** hoàn tất mã đối tượng bằng cách liên kết nó với mã đối tượng của bất kỳ các module thư viện mà chương trình đã tham khảo tới. Kết quả cuối cùng là một tập tin thực thi.

Hình 1.1 minh họa các bước trên cho cả hai trình chuyển đổi C++ và trình biên dịch C++. Thực tế thì tất cả các bước trên được triệu gọi bởi một lệnh đơn (như là `CC`) và người dùng thậm chí sẽ không thấy các tập tin được phát ra ngay lập tức.

Hình 1.1 Việc biên dịch C++



1.4. Biến

Biến là một tên tượng trưng cho một vùng nhớ mà dữ liệu có thể được lưu trữ trên đó hay là được sử dụng lại. Các biến được sử dụng để giữ các giá trị dữ liệu vì thế mà chúng có thể được dùng trong nhiều tính toán khác nhau trong một chương trình. Tất cả các biến có hai thuộc tính quan trọng:

- **Kiểu** được thiết lập khi các biến được định nghĩa (ví dụ như: integer, real, character). Một khi đã được định nghĩa, kiểu của một biến C++ không thể được chuyển đổi.

- **Giá trị** có thể được chuyển đổi bằng cách gán một giá trị mới cho biến. Loại giá trị của biến có thể nhận phụ thuộc vào kiểu của nó. Ví dụ, một biến số nguyên chỉ có thể giữ các giá trị nguyên (chẳng hạn, 2, 100, -12).

Danh sách 1.2 minh họa sử dụng một vài biến đơn giản.

Danh sách 1.2

```

1 #include <iostream.h>
2 int main (void)
3 {
4     int      workDays;
5     float  workHours, payRate, weeklyPay;
6
7     workDays = 5;
8     workHours = 7.5;
9     payRate = 38.55;
10    weeklyPay = workDays * workHours * payRate;
11    cout << "Weekly Pay = " << weeklyPay << '\n';
12 }
```

Chú giải

- Hàng này định nghĩa một biến int (kiểu số nguyên) tên là workDays, biến này đại diện cho số ngày làm việc trong tuần. Theo như luật chung, trước tiên một biến được định nghĩa bằng cách chỉ định kiểu của nó, theo sau đó là tên biến và cuối cùng là được kết thúc bởi dấu chấm phẩy.
- Hàng này định nghĩa ba biến float (kiểu số thực) lần lượt thay cho số giờ làm việc trong ngày, số tiền phải trả hàng giờ, và số tiền phải trả hàng tuần. Như chúng ta thấy ở hàng này, nhiều biến của cùng kiểu có thể định nghĩa một lượt qua việc dùng dấu phẩy để ngăn cách chúng.
- Hàng này là một câu lệnh gán. Nó gán giá trị 5 cho biến workDays. Vì thế, sau khi câu lệnh này được thực thi, workDays biểu thị giá trị 5.
- Hàng này gán giá trị 7.5 tới biến workHours.
- Hàng này gán giá trị 38.55 tới biến payRate.
- Hàng này tính toán số tiền phải trả hàng tuần từ các biến workDays, workHours, và payRate (* là toán tử nhân). Giá trị kết quả được lưu vào biến weeklyPay.
- Các hàng này xuất ba mục tuần tự là: chuỗi "Weekly Pay = ", giá trị của biến weeklyPay, và một ký tự xuống dòng.

Khi chạy, chương trình sẽ cho kết quả như sau:

```
Weekly Pay = 1445.625
```

Khi một biến được định nghĩa, giá trị của nó **không được định nghĩa** cho đến khi nó được gán cho một giá trị thật sự. Ví dụ, weeklyPay có một giá trị không được định nghĩa cho đến khi hàng 9 được thực thi. Việc gán giá trị cho một biến ở lần đầu tiên được gọi là **khởi tạo**. Việc chắc chắn rằng một

biến được khởi tạo trước khi nó được sử dụng trong bất kỳ công việc tính toán nào là rất quan trọng.

Một biến có thể được định nghĩa và khởi tạo cùng lúc. Điều này được xem như là một thói quen lập trình tốt bởi vì nó giành trước khả năng sử dụng biến trước khi nó được khởi tạo. Danh sách 1.3 là một phiên bản sửa lại của danh sách 1.2 mà có sử dụng kỹ thuật này. Trong mọi mục đích khác nhau thì hai chương trình là tương đương.

Danh sách 1.3

```
1 #include <iostream.h>
2 int main (void)
3 {
4     int      workDays = 5;
5     float workHours = 7.5;
6     float payRate = 38.55;
7     float weeklyPay = workDays * workHours * payRate;
8
9     cout << "Weekly Pay = ";
10    cout << weeklyPay;
11    cout << '\n';
12 }
```

1.5. Xuất/nhập đơn giản

Cách chung nhất mà một chương trình giao tiếp với thế giới bên ngoài là thông qua các thao tác xuất nhập hướng ký tự đơn giản. C++ cung cấp hai toán tử hữu dụng cho mục đích này là >> cho nhập và << cho xuất. Chúng ta đã thấy ví dụ của việc sử dụng toán tử xuất << rồi. Danh sách 1.4 sẽ minh họa thêm cho việc sử dụng toán tử nhập >>.

Danh sách 1.4

```
1 #include <iostream.h>
2 int main (void)
3 {
4     int      workDays = 5;
5     float workHours = 7.5;
6     float payRate, weeklyPay;
7
8     cout << "What is the hourly pay rate? ";
9     cin >> payRate;
10
11    weeklyPay = workDays * workHours * payRate;
12    cout << "Weekly Pay = ";
13    cout << weeklyPay;
14    cout << '\n';
15 }
```

Chú giải

- 7 Hàng này xuất ra lời nhắc nhở What is the hourly pay rate? để tìm dữ liệu nhập của người dùng.
- 8 Hàng này đọc giá trị nhập được gõ bởi người dùng và sao chép giá trị này tới biến payRate. Toán tử nhập >> lấy một dòng nhập như là toán hạng trái (cin là dòng nhập chuẩn của C++ mà tương ứng với dữ liệu được nhập vào từ bàn phím) và một biến (mà dữ liệu nhập được sao chép tới) như là toán hạng phải.

9-13 Phần còn lại của chương trình là như trước.

Khi chạy, chương trình sẽ xuất ra màn hình như sau (dữ liệu nhập của người dùng được in đậm):

```
What is the hourly pay rate? 33.55
Weekly Pay = 1258.125
```

Cả hai << và >> trả về toán hạng trái như là kết quả của chúng, cho phép nhiều thao tác nhập hay nhiều thao tác xuất được kết hợp trong một câu lệnh. Điều này được minh họa trong danh sách 1.5 với trường hợp cho phép nhập cả số giờ làm việc mỗi ngày và số tiền phải trả mỗi giờ.

Danh sách 1.5

```
1 #include <iostream.h>
2 int main (void)
3 {
4     int      workDays = 5;
5     float workHours, payRate, weeklyPay;
6
7     cout << "What are the work hours and the hourly pay rate? ";
8     cin >> workHours >> payRate;
9
10    weeklyPay = workDays * workHours * payRate;
11    cout << "Weekly Pay = " << weeklyPay << "\n";
12 }
```

Chú giải

- 7 Hàng này đọc hai giá trị nhập được nhập vào từ người dùng và chép tương ứng chúng tới hai biến workHours và payRate. Hai giá trị cần được tách biệt bởi một không gian trống (chẳng hạn, một hay là nhiều khoản trắng hay là các ký tự tab). Câu lệnh này tương đương với:

```
(cin >> workHours) >> payRate;
```

Vì kết quả của >> là toán hạng trái, (cin >> workHours) định giá cho cin mà sau đó được sử dụng như là toán hạng trái cho toán tử >> kế tiếp.

- 9 Hàng này là kết quả của việc kết hợp từ hàng 10 đến hàng 12 trong danh sách 1.4. Nó xuất "Weekly Pay = ", theo sau đó là giá trị của biến `weeklyPay`, và cuối cùng là một ký tự xuống dòng. Câu lệnh này tương đương với:

```
((cout << "Weekly Pay = ") << weeklyPay) << '\n';
```

Vì kết quả của `<<` là toán hạng trái, `(cout << "Weekly Pay = ")` định giá cho `cout` mà sau đó được sử dụng như là toán hạng trái của toán tử `<<` kế tiếp.

Khi chạy, chương trình sẽ hiển thị như sau:

```
What are the work hours and the hourly pay rate? 7.5 33.55
Weekly Pay=1258.125
```

1.6. Chú thích

Chú thích thường là một đoạn văn bản. Nó được dùng để giải thích một vài khía cạnh của chương trình. Trình biên dịch bỏ qua hoàn toàn các chú thích trong chương trình. Tuy nhiên các chú thích này là có ý nghĩa và đôi khi là rất quan trọng đối với người đọc (người xem các mã chương trình có sẵn) và người phát triển phần mềm. C++ cung cấp hai loại chú thích:

- Những gì sau `//` (cho đến khi kết thúc hàng mà nó xuất hiện) được xem như là một chú thích.
- Những gì đóng ngoặc trong cặp dấu `/*` và `*/` được xem như là một chú thích.

Danh sách 1.6 minh họa việc sử dụng cả hai hình thức này.

Danh sách 1.6

```
1 #include <iostream.h>
2 /* Chương trình này tính toán tổng số tiền phải trả hàng tuần cho một công nhân dựa trên tổng số giờ
3    làm việc và số tiền phải trả mỗi giờ. */
4
5 int main (void)
6 {
7     int      workDays = 5;        // số ngày làm việc trong tuần
8     float    workHours = 7.5;    // số giờ làm việc trong ngày
9     float    payRate = 33.50;    // số tiền phải trả mỗi giờ
10    float    weeklyPay;           // tổng số tiền phải trả mỗi tuần
11
12    weeklyPay = workDays * workHours * payRate;
13    cout << "Weekly Pay = " << weeklyPay << '\n';
14 }
```

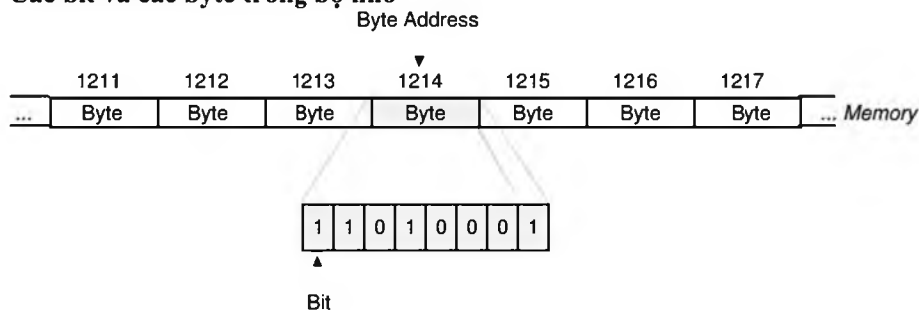
Các chú thích nên được sử dụng để tăng cường (không phải gây trở ngại) việc đọc một chương trình. Một vài điểm sau nên được chú ý:

- Chú thích nên dễ đọc và dễ hiểu hơn sự giải thích thông qua mã chương trình. Thà là không có chú thích nào còn hơn có một chú thích phức tạp dễ gây nhầm lẫn một cách không cần thiết.
- Sử dụng quá nhiều chú thích có thể dẫn đến khó đọc. Một chương trình chứa quá nhiều chú thích làm bạn khó có thể thấy mã thì không thể nào được xem như là một chương trình dễ đọc và dễ hiểu.
- Việc sử dụng các tên mô tả có ý nghĩa cho các biến và các thực thể khác trong chương trình, và những chỗ thụt vào của mã có thể làm giảm đi việc sử dụng chú thích một cách đáng kể, và cũng giúp cho lập trình viên dễ đọc và kiểm soát chương trình.

1.7. Bộ nhớ

Máy tính sử dụng bộ nhớ truy xuất ngẫu nhiên (RAM) để lưu trữ mã chương trình thực thi và dữ liệu mà chương trình thực hiện. Bộ nhớ này có thể được xem như là một chuỗi tuần tự các **bit nhị phân** (0 hoặc 1). Thông thường, bộ nhớ được chia thành những nhóm 8 bit liên tiếp (gọi là **byte**). Các byte được định vị liên tục. Vì thế mỗi byte có thể được chỉ định duy nhất bởi **địa chỉ** (xem Hình 1.2).

Hình 1.2 Các bit và các byte trong bộ nhớ



Trình biên dịch C++ phát ra mã có thể thực thi mà sắp xếp các thực thể dữ liệu tới các vị trí bộ nhớ. Ví dụ, định nghĩa biến

```
int salary = 65000;
```

làm cho trình biên dịch cấp phát một vài byte cho biến salary. Số byte cần được cấp phát và phương thức được sử dụng cho việc biểu diễn nhị phân của số nguyên phụ thuộc vào sự thi hành cụ thể của C++. Trình biên dịch sử dụng địa chỉ của byte đầu tiên của biến salary được cấp phát để tham khảo tới nó. Việc gán trên làm cho giá trị 65000 được lưu trữ như là một số nguyên bù hai trong hai byte được cấp phát (xem Hình 1.3).

Hình 1.3 Biểu diễn của một số nguyên trong bộ nhớ.

1211	1212	1213	1214	1215	1216	1217	
...	Byte	Byte	Byte	10110011	10110011	Byte	Byte
			salary				...

Bộ nhớ

một số nguyên 2 byte ở địa chỉ 1214

Trong khi việc biểu diễn nhị phân chính xác của một hạng mục dữ liệu là ít khi được các lập trình viên quan tâm tới thì việc tổ chức chung của bộ nhớ và sử dụng các địa chỉ để tham khảo tới các hạng mục dữ liệu là rất quan trọng.

1.8. Số nguyên

Biến số nguyên có thể được định nghĩa là kiểu `short`, `int`, hay `long`. Chỉ khác nhau là số `int` sử dụng nhiều hơn hoặc ít nhất bằng số byte như là một số `short`, và một số `long` sử dụng nhiều hơn hoặc ít nhất cùng số byte với một số `int`. Ví dụ, trên máy tính cá nhân thì một số `short` sử dụng 2 byte, một số `int` cũng 2 byte, và một số `long` là 4 byte.

```
short age=20;
int salary=65000;
long price=4500000;
```

Mặc định, một biến số nguyên được giả sử là có dấu (chẳng hạn, có một sự biểu diễn dấu để mà nó có thể biểu diễn các giá trị dương cũng như là các giá trị âm). Tuy nhiên, một số nguyên có thể được định nghĩa là không có dấu bằng cách sử dụng từ khóa `unsigned` trong định nghĩa của nó. Từ khóa `signed` cũng được cho phép nhưng hơi dư thừa.

```
unsigned short age=20;
unsigned int salary=65000;
unsigned long price=4500000;
```

Số nguyên (ví dụ, 1984) luôn luôn được giả sử là kiểu `int`, trừ khi có một hậu tố `L` hoặc `l` thì nó được hiểu là kiểu `long`. Một số nguyên cũng có thể được đặc tả sử dụng hậu tố là `U` hoặc `u`, ví dụ:

```
1984L 1984l 1984U 1984u 1984LU 1984ul
```

1.9. Số thực

Biến số thực có thể được định nghĩa là kiểu `float` hay `double`. Kiểu `double` sử dụng nhiều byte hơn và vì thế cho miền lớn hơn và chính xác hơn để biểu diễn các số thực. Ví dụ, trên các máy tính cá nhân một số `float` sử dụng 4 byte và một số `double` sử dụng 8 byte.

```
float interestRate=0.06;
double pi=3.141592654;
```

Số thực (ví dụ, 0.06) luôn luôn được giả sử là kiểu double, trừ phi có một hậu tố F hay f thì nó được hiểu là kiểu float, hoặc một hậu tố L hay l thì nó được hiểu là kiểu long double. Kiểu long double sử dụng nhiều byte hơn kiểu double cho độ chính xác tốt hơn (ví dụ, 10 byte trên các máy PC). Ví dụ:

```
0.06F      0.06f3.141592654L      3.141592654l
```

Các số thực cũng có thể được biểu diễn theo cách ký hiệu hóa khoa học. Ví dụ, 0.002164 có thể được viết theo cách ký hiệu hóa khoa học như sau:

```
2.164E-3      or      2.164e-3
```

Ký tự E (hay e) thay cho *số mũ* (exponent). Cách ký hiệu hóa khoa học được thông dịch như sau:

```
2.164E-3=2.164 × 10-3=0.002164
```

1.10.Ký tự

Biến ký tự được định nghĩa là kiểu char. Một biến ký tự chiếm một byte đơn để lưu giữ *mã* cho ký tự. Mã này là một giá trị số và phụ thuộc *hệ thống mã ký tự* đang được dùng (nghĩa là phụ thuộc máy). Hệ thống chung nhất là ASCII (American Standard Code for Information Interchange). Ví dụ, ký tự A có mã ASCII là 65, và ký tự a có mã ASCII là 97.

```
char ch='A';
```

Giống như số nguyên, biến ký tự có thể được chỉ định là có dấu hoặc không dấu. Mặc định (trong hầu hết các hệ thống) char nghĩa là signed char. Tuy nhiên, trên vài hệ thống thì nó có nghĩa là unsigned char. Biến ký tự có dấu có thể giữ giá trị số trong miền giá trị từ -128 tới 127. Biến ký tự không dấu có thể giữ giá trị số trong miền giá trị từ 0 tới 255. Kết quả là, cả hai thường được dùng để biểu diễn các số nguyên nhỏ trong chương trình (và có thể được đánh dấu các giá trị số như là số nguyên):

```
signed char    offset=-88;
unsigned char  row=2, column=26;
```

Ký tự được viết bằng cách đóng dấu ký tự giữa cặp nháy đơn (ví dụ, 'A'). Các ký tự mà không thể in ra được biểu diễn bằng việc sử dụng các mã escape. Ví dụ:

```
'\n'    // xuống hàng mới
'\r'    // phím xuống dòng
```

- '\t' // phím tab ngang
- '\v' // phím tab dọc
- '\b' // phím lùi

Các dấu nhảy đơn, nhảy đôi và ký tự gạch chéo ngược cũng có thể sử dụng ký hiệu escape:

" // trích dẫn đơn (')
 "" // trích dẫn đôi (")
 \\' // dấu gạch chéo ngược (\)

Ký tự cũng có thể được chỉ định rõ sử dụng giá trị mã số của chúng. Mã escape tổng quát \ooo (nghĩa là, 3 ký tự số cơ số 8 theo sau một dấu gạch chéo ngược) được sử dụng cho mục đích này. Ví dụ (giả sử ASCII):

"12' // hàng mới (mã thập phân = 10)
 "11' // tab ngang (mã thập phân = 9)
 "101' // 'A' (mã thập phân = 65)
 "0' // rỗng (mã thập phân = 0)

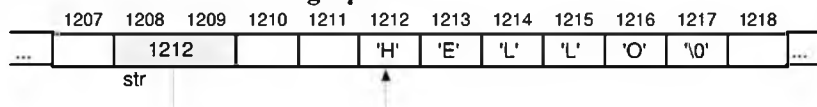
1.11.Chuỗi

Chuỗi là một dãy liên tiếp các ký tự được kết thúc bằng một ký tự null. **Biến chuỗi** được định nghĩa kiểu `char*` (nghĩa là, con trỏ ký tự). Con trỏ đơn giản chỉ là một vị trí trong bộ nhớ. (Các con trỏ sẽ được thảo luận trong chương 5). Vì thế biến chuỗi chứa đựng địa chỉ của ký tự đầu tiên trong chuỗi. Ví dụ, xem xét định nghĩa:

```
char *str="HELLO";
```

Hình 1.4 minh họa biến chuỗi và chuỗi "HELLO" có thể xuất hiện như thế nào trong bộ nhớ.

Hình 1.4 Chuỗi và biến chuỗi trong bộ nhớ



Chuỗi được viết bằng cách đóng ngoặc các ký tự của nó bên trong cặp dấu nháy kép (ví dụ, "HELLO"). Trình biên dịch luôn luôn thêm vào một ký tự null tới một hàng chuỗi để đánh dấu điểm kết thúc. Các ký tự chuỗi có thể được đặc tả sử dụng bất kỳ ký hiệu nào dùng để đặc tả các ký tự. Ví dụ:

"Name\tAddress\tTelephone"	// các từ phân cách
"ASCII character 65: \101"	// 'A' được đặc tả như '\101'

Chuỗi dài có thể nói rộng qua khỏi một hàng đơn, trong trường hợp này thì mỗi hàng trước phải được kết thúc bằng một dấu vạch chéo ngược. Ví dụ:

```
"Example to show \
the use of backslash for \
writing a long string"
```

Dấu \ trong ngữ cảnh này có nghĩa là phần còn lại của chuỗi được tiếp tục trên hàng kế tiếp. Chuỗi trên tương đương với chuỗi được viết trên hàng đơn như sau:

```
"Example to show the use of backslash for writing a long string"
```

Một lỗi lập trình chung thường xảy ra là lập trình viên thường nhầm lẫn một chuỗi ký tự đơn (ví dụ, "A") với một ký tự đơn (ví dụ, 'A'). Hai điều này là không tương đương. Chuỗi ký tự đơn gồm 2 byte (ký tự 'A' được theo sau là ký tự '\0'), trong khi ký tự đơn gồm chỉ một byte duy nhất.

Chuỗi ngắn nhất có thể có là chuỗi rỗng (""), chỉ chứa ký tự null.

1.12. Tên

Ngôn ngữ lập trình sử dụng tên để tham khảo tới các thực thể khác nhau dùng để tạo ra chương trình. Chúng ta cũng đã thấy các ví dụ của một loại các tên (nghĩa là tên biến) như thế. Các loại khác gồm: tên hàm, tên kiểu, và tên macro.

Sử dụng tên rất tiện lợi cho việc lập trình, nó cho phép lập trình viên tổ chức dữ liệu theo cách thức mà con người có thể hiểu được. Tên không được đưa vào mã có thể thực thi được tạo ra bởi trình biên dịch. Ví dụ, một biến `temperature` cuối cùng trở thành một vài byte bộ nhớ mà được tham khảo tới bởi các mã có thể thực thi thông qua địa chỉ của nó (không thông qua tên của nó).

C++ áp đặt những luật sau để xây dựng các tên hợp lệ (cũng được gọi là các **định danh**). Một tên chứa một hay nhiều ký tự, mỗi ký tự có thể là một chữ cái (nghĩa là, 'A'-'Z' và 'a'-'z'), một số (nghĩa là, '0'-'9'), hoặc một ký tự gạch dưới ('_'), ngoại trừ ký tự đầu tiên không thể là một số. Các ký tự viết hoa và viết thường là khác nhau. Ví dụ:

```
salary      // định danh hợp lệ
salary2     // định danh hợp lệ
2salary     // định danh không hợp lệ (bắt đầu với một số)
_salary     // định danh hợp lệ
Salary      // hợp lệ nhưng khác với salary
```

C++ không có giới hạn số ký tự của một định danh. Tuy nhiên, hầu hết thi công lại áp đặt sự giới hạn này nhưng thường đủ lớn để không gây bận tâm cho các lập trình viên (ví dụ 255 ký tự).

Một số từ được giữ bởi C++ cho một số mục đích riêng và không thể được dùng cho các định danh. Những từ này được gọi là **từ khóa** (keyword) và được tổng kết trong bảng 1.3:

Bảng 1.3 Các từ khóa C++.

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Bài tập cuối chương 1

- 1.1 Viết chương trình cho phép nhập vào một số đo nhiệt độ theo độ Fahrenheit và xuất ra nhiệt độ tương đương của nó theo độ Celsius, sử dụng công thức chuyển đổi:

$$^{\circ}C = \frac{5}{9} (^{\circ}F - 32)$$

Biên dịch và chạy chương trình. Việc thực hiện của nó giống như thế này:

Nhiệt độ theo độ Fahrenheit: 41
41 độ Fahrenheit = 5 độ Celsius

- 1.2 Hàng nào trong các hàng sau biểu diễn việc định nghĩa biến là không hợp lệ?

```
int n=-100;
unsigned int i=-100;
signed int = 2.9;
long m = 2, p = 4;
int 2k;
double x = 2 * m;
float y = y * 2;
unsigned double z = 0.0;
double d = 0.67F;
float f = 0.52L;
signed char = -1786;
char c = '$' + 2;
sign char h = '\111';
```

```
char *name = "Peter Pan";  
unsigned char *num = "276811";
```

1.3 Các định danh nào sau đây là không hợp lệ?

```
identifier  
seven_11  
_unique_  
gross-income  
gross$income  
2by2  
default  
average_weight_of_a_large_pizza  
variable  
object.oriented
```

1.4 Định nghĩa các biến để biểu diễn các mục sau đây:

- Tuổi của một người.
- Thu nhập của một nhân viên.
- Số từ trong một từ điển.
- Một ký tự alphabet.
- Một thông điệp chúc mừng.

Chương 2. Biểu thức

Chương này giới thiệu các toán tử xây dựng sẵn cho việc soạn thảo các biểu thức. Một biểu thức là bất kỳ sự tính toán nào mà cho ra một giá trị.

Khi thảo luận về các biểu thức, chúng ta thường sử dụng thuật ngữ **ước lượng**. Ví dụ, chúng ta nói rằng một biểu thức ước lượng một giá trị nào đó. Thường thì giá trị sau cùng chỉ là lý do cho việc ước lượng biểu thức. Tuy nhiên, trong một vài trường hợp, biểu thức cũng có thể cho các kết quả phụ. Các kết quả này là sự thay đổi lâu dài trong trạng thái của chương trình. Trong trường hợp này, các biểu thức C++ thì khác với các biểu thức toán học.

C++ cung cấp các toán tử cho việc soạn thảo các biểu thức toán học, quan hệ, luận lý, trên bit, và điều kiện. Nó cũng cung cấp các toán tử cho ra các kết quả phụ hữu dụng như là gán, tăng, và giảm. Chúng ta sẽ xem xét lần lượt từng loại toán tử. Chúng ta cũng sẽ thảo luận về các luật ưu tiên mà ảnh hưởng đến thứ tự ước lượng của các toán tử trong một biểu thức có nhiều toán tử.

2.1. Toán tử toán học

C++ cung cấp 5 toán tử toán học cơ bản. Chúng được tổng kết trong Bảng 2.1.

Bảng 2.1 **Các toán tử toán học.**

Toán tử	Tên	Ví dụ
+	Cộng	12 + 4.9 // cho 16.9
-	Trừ	3.98 - 4 // cho -0.02
*	Nhân	2 * 3.4 // cho 6.8
/	Chia	9 / 2.0 // cho 4.5
%	Lấy phần dư	13 % 3 // cho 1

Ngoại trừ toán tử lấy phần dư (%) thì tất cả các toán tử toán học có thể chấp nhận pha trộn các toán hạng số nguyên và toán hạng số thực. Thông thường, nếu cả hai toán hạng là số nguyên sau đó kết quả sẽ là một số

nguyên. Tuy nhiên, một hoặc cả hai toán hạng là số thực thì sau đó kết quả sẽ là một số thực (real hay double).

Khi cả hai toán hạng của toán tử chia là số nguyên thì sau đó phép chia được thực hiện như là một phép chia số nguyên và không phải là phép chia thông thường mà chúng ta sử dụng. Phép chia số nguyên luôn cho kết quả nguyên (có nghĩa là luôn được làm tròn). Ví dụ:

```
9/2      // được 4, không phải là 4.5!
-9/2     // được -5, không phải là -4!
```

Các phép chia số nguyên không xác định thường là các lỗi lập trình chung. Để thu được một phép chia số thực khi cả hai toán hạng là số nguyên, bạn cần ép một trong hai số nguyên về số thực:

```
int      cost = 100;
int      volume = 80;
double   unitPrice = cost / (double) volume;    // được 1.25
```

Toán tử lấy phần dư (%) yêu cầu cả hai toán hạng là số nguyên. Nó trả về phần dư còn lại của phép chia. Ví dụ 13%/3 được tính toán bằng cách chia số nguyên 13 đi 3 để được 4 và phần dư là 1; vì thế kết quả là 1.

Có thể có trường hợp một kết quả của một phép toán toán học quá lớn để lưu trữ trong một biến nào đó. Trường hợp này được gọi là **tràn**. Hậu quả của tràn là phụ thuộc vào máy vì thế nó không được định nghĩa. Ví dụ:

```
unsigned char k = 10 * 92;    // tràn: 920 > 255
```

Chia một số cho 0 là hoàn toàn không đúng luật. Kết quả của phép chia này là một lỗi run-time gọi là lỗi *division-by-zero* thường làm cho chương trình kết thúc.

2.2. Toán tử quan hệ

C++ cung cấp 6 toán tử quan hệ để so sánh các số. Các toán tử này được tổng kết trong Bảng 2.2. Các toán tử quan hệ ước lượng về 1 (thay cho kết quả đúng) hoặc 0 (thay cho kết quả sai).

Bảng 2.2 Các toán tử quan hệ.

Toán tử	Tên	Ví dụ
<code>==</code>	So sánh bằng	<code>5 == 5</code> // cho 1
<code>!=</code>	So sánh không bằng	<code>5 != 5</code> // cho 0
<code><</code>	So sánh nhỏ hơn	<code>5 < 5.5</code> // cho 1
<code><=</code>	So sánh nhỏ hơn hoặc bằng	<code>5 <= 5</code> // cho 1
<code>></code>	So sánh lớn hơn	<code>5 > 5.5</code> // cho 0
<code>>=</code>	So sánh lớn hơn hoặc bằng	<code>6.3 >= 5</code> // cho 1

Chú ý rằng các toán tử \leq và \geq chỉ được hỗ trợ trong hình thức hiển thị. Nói riêng cả hai \leq và \geq đều không hợp lệ và không mang ý nghĩa gì cả.

Các toán hạng của một toán tử quan hệ phải ước lượng về một số. Các ký tự là các toán hạng hợp lệ vì chúng được đại diện bởi các giá trị số. Ví dụ (giả sử mã ASCII):

```
'A' < 'F'           // được 1 (giống như là 65 < 70)
```

Các toán tử quan hệ không nên được dùng để so sánh chuỗi bởi vì điều này sẽ dẫn đến các *địa chỉ* của chuỗi được so sánh chứ không phải là nội dung chuỗi. Ví dụ, biểu thức

```
"HELLO" < "BYE"
```

làm cho địa chỉ của chuỗi "HELLO" được so sánh với địa chỉ của chuỗi "BYE". Vì các địa chỉ này được xác định bởi trình biên dịch, kết quả có thể là 0 hoặc có thể là 1, cho nên chúng ta có thể nói kết quả là không được định nghĩa.

C++ cung cấp các thư viện hàm (ví dụ, `strcmp`) để thực hiện so sánh chuỗi.

2.3. Toán tử luận lý

C++ cung cấp ba toán tử luận lý cho việc kết nối các biểu thức luận lý. Các toán tử này được tổng kết trong Bảng 2.3. Giống như các toán tử quan hệ, các toán tử luận lý ước lượng tới 0 hoặc 1.

Bảng 2.3 Các toán tử luận lý.

Toán tử	Tên	Ví dụ
!	Phủ định luận lý	!(5 == 5) // được 0
&&	Và luận lý	5 < 6 && 6 < 6 // được 0
	Hoặc luận lý	5 < 6 6 < 5 // được 1

Phủ định luận lý là một toán tử đơn hạng chỉ phủ định giá trị luận lý toán hạng đơn của nó. Nếu toán hạng của nó không là 0 thì được 0, và nếu nó là không thì được 1.

Và luận lý cho kết quả 0 nếu một hay cả hai toán hạng của nó ước lượng tới 0. Ngược lại, nó cho kết quả 1. Hoặc luận lý cho kết quả 0 nếu cả hai toán hạng của nó ước lượng tới 0. Ngược lại, nó cho kết quả 1.

Chú ý rằng ở đây chúng ta nói các toán hạng là 0 và khác 0. Nói chung, bất kỳ giá trị không là 0 nào có thể được dùng để đại diện cho đúng (true), trong khi chỉ có giá trị 0 là đại diện cho sai (false). Tuy nhiên, tất cả các hàng sau đây là các biểu thức luận lý hợp lệ:

```

!20                // được 0
10 && 5             // được 1
10 || 5.5           // được 1
10 && 0             // được 0

```

C++ không có kiểu boolean xây dựng sẵn. Vì lẽ đó mà ta có thể sử dụng kiểu `int` cho mục đích này. Ví dụ:

```

int sorted = 0;    // false
int balanced = 1;  // true

```

2.4. Toán tử trên bit

C++ cung cấp 6 toán tử trên bit để điều khiển các bit riêng lẻ trong một số lượng số nguyên. Chúng được tổng kết trong Bảng 2.4.

Bảng 2.4 Các toán tử trên bit.

Toán tử	Tên	Ví dụ
~	Phủ định bit	~'011' // được '366'
&	Và bit	'011' & '027' // được '001'
	Hoặc bit	'011' '027' // được '037'
^	Hoặc exclusive bit	'011' ^ '027' // được '036'
<<	Dịch trái bit	'011' << 2 // được '044'
>>	Dịch phải bit	'011' >> 2 // được '002'

Các toán tử trên bit mong đợi các toán hạng của chúng là các số nguyên và xem chúng như là một chuỗi các bit. *Phủ định bit* là một toán tử đơn hạng thực hiện đảo các bit trong toán hạng của nó. *Và bit* so sánh các bit tương ứng của các toán hạng của nó và cho kết quả là 1 khi cả hai bit là 1, ngược lại là 0. *Hoặc bit* so sánh các bit tương ứng của các toán hạng của nó và cho kết quả là 0 khi cả hai bit là 0, ngược lại là 1. *XOR bit* so sánh các bit tương ứng của các toán hạng của nó và cho kết quả 0 khi cả hai bit là 1 hoặc cả hai bit là 0, ngược lại là 1.

Cả hai toán tử *dịch trái bit* và *dịch phải bit* lấy một chuỗi bit làm toán hạng trái của chúng và một số nguyên dương n làm toán hạng phải. Toán tử dịch trái cho kết quả là một chuỗi bit sau khi thực hiện dịch n bit trong chuỗi bit của toán hạng trái về phía trái. Toán tử dịch phải cho kết quả là một chuỗi bit sau khi thực hiện dịch n bit trong chuỗi bit của toán hạng trái về phía phải. Các bit trống sau khi dịch được đặt tới 0.

Bảng 2.5 minh họa chuỗi các bit cho các toán hạng ví dụ và kết quả trong Bảng 2.4. Để tránh lo lắng về bit dấu (điều này phụ thuộc vào máy) thường thì khai báo chuỗi bit như là một số không dấu:

```

unsigned char x = '011';
unsigned char y = '027';

```

Bảng 2.5 Các bit được tính toán như thế nào.

Ví dụ	Giá trị cơ số 8	Chuỗi bit							
x	011	0	0	0	0	1	0	0	1
y	027	0	0	0	1	0	1	1	1
~x	366	1	1	1	1	0	1	1	0
x & y	001	0	0	0	0	0	0	0	1
x y	037	0	0	0	1	1	1	1	1
x ^ y	036	0	0	0	1	1	1	1	0
x << 2	044	0	0	1	0	0	1	0	0
x >> 2	002	0	0	0	0	0	0	1	0

2.5. Toán tử tăng/giảm

Các toán tử *tăng một* (++) và *giảm một* (--) cung cấp các tiện lợi tương ứng cho việc cộng thêm 1 vào một biến số hay trừ đi 1 từ một biến số. Các toán tử này được tổng kết trong Bảng 2.6. Các ví dụ giả sử đã định nghĩa biến sau:

```
int k=5;
```

Bảng 2.6 Các toán tử tăng và giảm.

Toán tử	Tên	Ví dụ
++	Tăng một (tiền tố)	++k + 10 // được 16
++	Tăng một (hậu tố)	k++ + 10 // được 15
--	Giảm một (tiền tố)	--k + 10 // được 14
--	Giảm một (hậu tố)	k-- + 10 // được 15

Cả hai toán tử có thể được sử dụng theo hình thức tiền tố hay hậu tố là hoàn toàn khác nhau. Khi được sử dụng theo hình thức tiền tố thì toán tử được áp dụng trước và kết quả sau đó được sử dụng trong biểu thức. Khi được sử dụng theo hình thức hậu tố thì biểu thức được ước lượng trước và sau đó toán tử được áp dụng.

Cả hai toán tử có thể được áp dụng cho biến nguyên cũng như là biến thực mặc dù trong thực tế thì các biến thực hiếm khi được dùng theo hình thức này.

2.6. Toán tử khởi tạo

Toán tử khởi tạo được sử dụng để lưu trữ một biến. Toán hạng trái nên là một giá trị trái và toán hạng phải có thể là một biểu thức bất kỳ. Biểu thức được ước lượng và kết quả được lưu trữ trong vị trí được chỉ định bởi giá trị trái.

Giá trị trái là bất kỳ thứ gì chỉ định rõ vị trí bộ nhớ lưu trữ một giá trị. Chỉ một loại của giá trị trái mà chúng ta được biết cho đến thời điểm này là

biến. Các loại khác của giá trị trái (dựa trên con trỏ và tham chiếu) sẽ được thảo luận sau.

Toán tử khởi tạo có một số biến thể thu được bằng cách kết nối nó với các toán tử toán học và các toán tử trên bit. Chúng được tổng kết trong Bảng 2.7. Các ví dụ giả sử rằng n là một biến số nguyên.

Bảng 2.7 Các toán tử khởi tạo.

Toán tử	Ví dụ	Tương đương với
<code>=</code>	<code>n = 25</code>	
<code>+=</code>	<code>n += 25</code>	<code>n = n + 25</code>
<code>-=</code>	<code>n -= 25</code>	<code>n = n - 25</code>
<code>*=</code>	<code>n *= 25</code>	<code>n = n * 25</code>
<code>/=</code>	<code>n /= 25</code>	<code>n = n / 25</code>
<code>%=</code>	<code>n %= 25</code>	<code>n = n % 25</code>
<code>&=</code>	<code>n &= 0xF2F2</code>	<code>n = n & 0xF2F2</code>
<code> =</code>	<code>n = 0xF2F2</code>	<code>n = n 0xF2F2</code>
<code>^=</code>	<code>n ^= 0xF2F2</code>	<code>n = n ^ 0xF2F2</code>
<code><<=</code>	<code>n <<= 4</code>	<code>n = n << 4</code>
<code>>>=</code>	<code>n >>= 4</code>	<code>n = n >> 4</code>

Phép toán khởi tạo chính nó là một biểu thức mà giá trị của nó là giá trị được lưu trong toán hạng trái của nó. Vì thế một phép toán khởi tạo có thể được sử dụng như là toán hạng phải của một phép toán khởi tạo khác. Bất kỳ số lượng khởi tạo nào có thể được kết nối theo cách này để hình thành một biểu thức. Ví dụ:

```
int m, n, p;
m = n = p = 100;           // nghĩa là: n = (m = (p = 100));
m = (n = p = 100) + 2;     // nghĩa là: m = (n = (p = 100)) + 2;
```

Việc này có thể ứng dụng tương tự cho các hình thức khởi tạo khác. Ví dụ:

```
m = 100;
m += n = p = 10;           // nghĩa là: m = m + (n = p = 10);
```

2.7. Toán tử điều kiện

Toán tử điều kiện yêu cầu 3 toán hạng. Hình thức chung của nó là:

toán hạng 1 ? toán hạng 2 : toán hạng 3

Toán hạng đầu tiên được ước lượng và được xem như là một điều kiện. Nếu kết quả không là 0 thì toán hạng 2 được ước lượng và giá trị của nó là kết quả sau cùng. Ngược lại, toán hạng 3 được ước lượng và giá trị của nó là kết quả sau cùng. Ví dụ:

```
int m=1, n=2;
int min=(m<n ? m : n); // min nhận giá trị 1
```

Chú ý rằng trong các toán hạng thứ 2 và toán hạng thứ 3 của toán tử điều kiện thì chỉ có một toán hạng được thực hiện. Điều này là quan trọng khi một hoặc cả hai chứa hiệu ứng phụ (nghĩa là, việc ước lượng của chúng làm chuyển đổi giá trị của biến). Ví dụ, với $m=1$ và $n=2$ thì trong

```
int min=(m<n ? m++ : n++);
```

m được tăng lên bởi vì $m++$ được ước lượng nhưng n không tăng vì $n++$ không được ước lượng.

Bởi vì chính phép toán điều kiện cũng là một biểu thức nên nó có thể được sử dụng như một toán hạng của phép toán điều kiện khác, có nghĩa là các biểu thức điều kiện có thể được lồng nhau. Ví dụ:

```
int m=1, n=2, p=3;
int min=(m<n ? (m<p ? m : p)
           : (n<p ? n : p));
```

2.8. Toán tử phẩy

Nhiều biểu thức có thể được kết nối vào cùng một biểu thức sử dụng toán tử phẩy. Toán tử phẩy yêu cầu 2 toán hạng. Đầu tiên nó ước lượng toán hạng trái sau đó là toán hạng phải, và trả về giá trị của toán hạng phải như là kết quả sau cùng. Ví dụ:

```
int m=1, n=2, min;
int mCount=0, nCount=0;
//...
min=(m<n ? mCount++, m : nCount++, n);
```

Ở đây khi m nhỏ hơn n , $mCount++$ được ước lượng và giá trị của m được lưu trong min . Ngược lại, $nCount++$ được ước lượng và giá trị của n được lưu trong min .

2.9. Toán tử lấy kích thước

C++ cung cấp toán tử hữu dụng, `sizeof`, để tính toán kích thước của bất kỳ hạng mục dữ liệu hay kiểu dữ liệu nào. Nó yêu cầu một toán hạng duy nhất có thể là tên kiểu (ví dụ, `int`) hay một biểu thức (ví dụ, `100`) và trả về kích thước của những thực thể đã chỉ định theo byte. Kết quả hoàn toàn phụ thuộc vào máy. Danh sách 2.1 minh họa việc sử dụng toán tử `sizeof` cho các kiểu có sẵn mà chúng ta đã gặp cho đến thời điểm này.

Danh sách 2.1

```

1 #include <iostream.h>
2 int main (void)
3 {
4     cout << "char size=" << sizeof(char) << " bytes\n";
5     cout << "char* size=" << sizeof(char*) << " bytes\n";
6     cout << "short size=" << sizeof(short) << " bytes\n";
7     cout << "int size=" << sizeof(int) << " bytes\n";
8     cout << "long size=" << sizeof(long) << " bytes\n";
9     cout << "float size=" << sizeof(float) << " bytes\n";
10    cout << "double size=" << sizeof(double) << " bytes\n";
11
12    cout << "1.55 size=" << sizeof(1.55) << " bytes\n";
13    cout << "1.55L size=" << sizeof(1.55L) << " bytes\n";
14    cout << "HELLO size=" << sizeof("HELLO") << " bytes\n";
15 }

```

Khi chạy, chương trình sẽ cho kết quả sau (trên máy tính cá nhân):

```

char size=1 bytes
char* size=2 bytes
short size=2 bytes
int size=2 bytes
long size=4 bytes
float size=4 bytes
double size=8 bytes
1.55 size=8 bytes
1.55L size=10 bytes
HELLO size=6 bytes

```

2.10.Độ ưu tiên của các toán tử

Thứ tự mà các toán tử được ước lượng trong một biểu thức là rất quan trọng và được xác định theo các luật ưu tiên. Các luật này chia các toán tử C++ ra thành một số mức độ ưu tiên (xem Bảng 2.8). Các toán tử ở mức cao hơn sẽ có độ ưu tiên cao hơn các toán tử có độ ưu tiên thấp hơn.

Bảng 2.8 Độ ưu tiên của các toán tử.

Mức	Toán tử						Loại	Thứ tự
Cao nhất	::						Đơn hạng	Cả hai
	()	[]	->	.			Nhị hạng	Trái tới phải
	+	++	!	*	new	sizeof	Đơn hạng	Phải tới trái
	-	--	~	&	delete	()		
	->*	*.					Nhị hạng	Trái tới phải
	*	/	%				Nhị hạng	Trái tới phải
	+	-					Nhị hạng	Trái tới phải
	<<	>>					Nhị hạng	Trái tới phải
	<	<=	>	>=			Nhị hạng	Trái tới phải
	=	!=					Nhị hạng	Trái tới phải
	&						Nhị hạng	Trái tới phải

	^					Nhị hạng	Trái tới phải
						Nhị hạng	Trái tới phải
	&&					Nhị hạng	Trái tới phải
						Nhị hạng	Trái tới phải
	?:					Tam hạng	Trái tới phải
	=	+=	*=	^=	&=	<<=	Phải tới trái
		-=	/=	%=	=	>>=	
Thấp nhất	,					Nhị hạng	Trái tới phải

Ví dụ, trong biểu thức
 $a = b + c * d$

$c * d$ được ước lượng trước bởi vì toán tử $*$ có độ ưu tiên cao hơn toán tử $+$ và $=$. Sau đó kết quả được cộng tới b bởi vì toán tử $+$ có độ ưu tiên cao hơn toán tử $=$, và sau đó $=$ được ước lượng. Các luật ưu tiên có thể được cho quyền cao hơn thông qua việc sử dụng các dấu ngoặc. Ví dụ, viết lại biểu thức trên như sau

$$a = (b + c) * d$$

sẽ làm cho toán tử $+$ được ước lượng trước toán tử $*$.

Các toán tử với cùng mức độ ưu tiên được ước lượng theo thứ tự được ước lượng trong cột cuối cùng trong Bảng 2.8. Ví dụ, trong biểu thức

$$a = b += c$$

thứ tự ước lượng là từ phải sang trái, vì thế $b += c$ được ước lượng trước và kết quả là $a = b$.

2.11. Chuyển kiểu đơn giản

Một giá trị thuộc về những kiểu xây dựng sẵn mà chúng ta biết đến thời điểm này đều có thể được chuyển về bất kỳ một kiểu nào khác. Ví dụ:

```
(int) 3.14    // chuyển 3.14 sang int để được 3
(long) 3.14   // chuyển 3.14 sang long để được 3L
(double) 2    // chuyển 2 sang double để được 2.0
(char) 122    // chuyển 122 sang char có mã là 122
(unsigned short) 3.14 // được 3 như là một unsigned short
```

Như đã được trình bày trong các ví dụ, các định danh kiểu xây dựng sẵn có thể được sử dụng như **các toán tử kiểu**. Các toán tử kiểu là đơn hạng (nghĩa là chỉ có một toán hạng) và xuất hiện bên trong các dấu ngoặc về bên trái toán hạng của chúng. Điều này được gọi là **chuyển kiểu tường minh**. Khi tên kiểu chỉ là một từ thì có thể đặt dấu ngoặc xung quanh toán hạng:

```
int(3.14)    // như là: (int) 3.14
```


Trong một vài trường hợp, C++ cũng thực hiện **chuyển kiểu không tường minh**. Điều này xảy ra khi các giá trị của các kiểu khác nhau được trộn lẫn trong một biểu thức. Ví dụ:

```
double    d=1;           // d nhận 1.0
int       i=10.5;        // i nhận 10
i=i+d;     // nghĩa là: i=int(double(i)+d)
```

Trong ví dụ cuối, $i + d$ bao hàm các kiểu không hợp nhau, vì thế trước tiên i được chuyển thành *double* (*thăng cấp*) và sau đó được cộng vào d . Kết quả là *double* không hợp kiểu với i trên phía trái của phép gán, vì thế nó được chuyển thành *int* (*hạ cấp*) trước khi được gán cho i .

Luật trên đại diện cho một vài trường hợp chung đơn giản để chuyển kiểu. Các trường hợp phức tạp hơn sẽ được trình bày ở phần sau của giáo trình sau khi chúng ta thảo luận các kiểu dữ liệu khác.

Bài tập cuối chương 2

2.1 Viết các biểu thức sau đây:

- Kiểm tra một số n là chẵn hay không.
- Kiểm tra một ký tự c là một số hay không.
- Kiểm tra một ký tự c là một mẫu tự hay không.
- Thực hiện kiểm tra: n là lẻ và dương hoặc n chẵn và âm.
- Đặt lại k bit của một số nguyên n tới 0.
- Đặt k bit của một số nguyên n tới 1.
- Cho giá trị tuyệt đối của một số n .
- Cho số ký tự trong một chuỗi s được kết thúc bởi ký tự null.

2.2 Thêm các dấu ngoặc phụ vào các biểu thức sau để hiển thị rõ ràng thứ tự các toán tử được ước lượng:

```
(n <= p + q && n >= p - q || n == 0)
(++n * q - ++p - q)
(n | p & q ^ p << 2 + q)
(p < q ? n < p ? q * n - 2 : q / n + 1 : q - n)
```

2.3 Cho biết giá trị của mỗi biến sau đây sau khi khởi tạo nó:

```
double    d=2 * int(3.14);
long      k=3.14 - 3;
char      c='a' + 2;
char      c='p' + 'A' - 'a';
```

2.4 Viết một chương trình cho phép nhập vào một số nguyên dương n và xuất ra giá trị của n mũ 2 và 2 mũ n .

- 2.5 Viết một chương trình cho phép nhập ba số và xuất ra thông điệp Sorted nếu các số là tăng dần và xuất ra Not sorted trong trường hợp ngược lại.

Chương 3. Lệnh

Chương này giới thiệu các hình thức khác nhau của các câu lệnh C++ để soạn thảo chương trình. Các lệnh trình bày việc xây dựng các khối ở mức độ thấp nhất của một chương trình. Nói chung mỗi lệnh trình bày một bước tính toán có một tác động chính yếu. Bên cạnh đó cũng có thể có các tác động phụ khác. Các lệnh là hữu dụng vì tác dụng chính yếu mà nó gây ra, sự kết nối của các lệnh cho phép chương trình phục vụ một mục đích cụ thể (ví dụ, sắp xếp một danh sách các tên).

Một chương trình đang chạy dành toàn bộ thời gian để thực thi các câu lệnh. Thứ tự mà các câu lệnh được thực hiện được gọi là **dòng điều khiển** (flow control). Thuật ngữ này phản ánh việc các câu lệnh đang thực thi hiện thời có *sự điều khiển* của CPU, khi CPU hoàn thành sẽ được chuyển giao tới một lệnh khác. Đặc trưng dòng điều khiển trong một chương trình là tuần tự, lệnh này đến lệnh kế, nhưng có thể chuyển hướng tới đường dẫn khác bởi các lệnh rẽ nhánh. Dòng điều khiển là một sự xem xét trọng yếu bởi vì nó quyết định lệnh nào được thực thi và lệnh nào không được thực thi trong quá trình chạy, vì thế làm ảnh hưởng đến kết quả toàn bộ của chương trình.

Giống nhiều ngôn ngữ thủ tục khác, C++ cung cấp những hình thức khác nhau cho các mục đích khác nhau. Các lệnh khai báo được sử dụng cho định nghĩa các biến. Các lệnh như gán được sử dụng cho các tính toán đại số đơn giản. Các lệnh rẽ nhánh được sử dụng để chỉ định đường dẫn của việc thực thi phụ thuộc vào kết quả của một điều kiện luận lý. Các lệnh lặp được sử dụng để chỉ định các tính toán cần được lặp cho tới khi một điều kiện luận lý nào đó được thỏa. Các lệnh điều khiển được sử dụng để làm chuyển đường dẫn thực thi tới một đường dẫn khác của chương trình. Chúng ta sẽ lần lượt thảo luận tất cả những vấn đề này.

3.1. Lệnh đơn và lệnh phức

Lệnh đơn là một sự tính toán được kết thúc bằng dấu chấm phẩy. Các định nghĩa biến và các biểu thức được kết thúc bằng dấu chấm phẩy như trong ví dụ sau:

```
int i;           // lệnh khai báo
++i;            // lệnh này có một tác động chính yếu
double d = 10.5; // lệnh khai báo
d + 5;          // lệnh không hữu dụng
```

Ví dụ cuối trình bày một lệnh không hữu dụng bởi vì nó không có tác động chính yếu (d được cộng với 5 và kết quả bị vứt bỏ).

Lệnh đơn giản nhất là lệnh rỗng chỉ gồm dấu chấm phẩy mà thôi.

```
;
```

// lệnh rỗng

Mặc dầu lệnh rỗng không có tác động chính yếu nhưng nó có một vai việc dùng xác thật.

Nhiều lệnh đơn có thể kết nối lại thành một **lệnh phức** bằng cách rào chúng bên trong các dấu ngoặc xoắn. Ví dụ:

```
{ int min, i = 10, j = 20;
  min = (i < j ? i : j);
  cout << min << '\n';
}
```

Bởi vì một lệnh phức có thể chứa các định nghĩa biến và định nghĩa một phạm vi cho chúng, nó cũng được gọi **một khối**. Phạm vi của một biến C++ được giới hạn bên trong khối trực tiếp chứa nó. Các khối và các luật phạm vi sẽ được mô tả chi tiết hơn khi chúng ta thảo luận về hàm trong chương kế.

3.2. Lệnh if

Đôi khi chúng ta muốn làm cho sự thực thi một lệnh phụ thuộc vào một điều kiện nào đó cần được thỏa. Lệnh if cung cấp cách để thực hiện công việc này, hình thức chung của lệnh này là:

```
if (biểu thức)
    lệnh;
```

Trước tiên *biểu thức* được ước lượng. Nếu kết quả khác 0 (đúng) thì sau đó *lệnh* được thực thi. Ngược lại, không làm gì cả.

Ví dụ, khi chia hai giá trị chúng ta muốn kiểm tra rằng mẫu số có khác 0 hay không.

```
if(count != 0)
```

```
average = sum / count;
```

Để làm cho nhiều lệnh phụ thuộc trên cùng điều kiện chúng ta có thể sử dụng lệnh phức:

```
if (balance > 0) {  
    interest = balance * creditRate;  
    balance += interest;  
}
```

Một hình thức khác của lệnh if cho phép chúng ta chọn một trong hai lệnh: một lệnh được thực thi nếu như điều kiện được thỏa và lệnh còn lại được thực hiện nếu như điều kiện không thỏa. Hình thức này được gọi là lệnh if-else và có hình thức chung là:

```
if (biểu thức)  
    lệnh 1;  
else  
    lệnh 2;
```

Trước tiên *biểu thức* được ước lượng. Nếu kết quả khác 0 thì *lệnh 1* được thực thi. Ngược lại, *lệnh 2* được thực thi.

Ví dụ:

```
if (balance > 0) {  
    interest = balance * creditRate;  
    balance += interest;  
} else {  
    interest = balance * debitRate;  
    balance += interest;  
}
```

Trong cả hai phần có sự giống nhau ở lệnh `balance += interest` vì thế toàn bộ câu lệnh có thể viết lại như sau:

```
if (balance > 0)  
    interest = balance * creditRate;  
else  
    interest = balance * debitRate;  
balance += interest;
```

Hoặc đơn giản hơn bằng việc sử dụng biểu thức điều kiện:

```
interest = balance * (balance > 0 ? creditRate : debitRate);  
balance += interest;
```

Hoặc chỉ là:

```
balance += balance * (balance > 0 ? creditRate : debitRate);
```

Các lệnh if có thể được lồng nhau bằng cách để cho một lệnh if xuất hiện bên trong một lệnh if khác. Ví dụ:

```

if (callHour > 6) {
    if (callDuration <= 5)
        charge = callDuration * tariff1;
    else
        charge = 5 * tariff1 + (callDuration - 5) * tariff2;
} else
    charge = flatFee;

```

Một hình thức được sử dụng thường xuyên của những lệnh if lồng nhau liên quan đến phần else gồm có một lệnh if-else khác. Ví dụ:

```

if (ch >= '0' && ch <= '9')
    kind = digit;
else {
    if (ch >= 'A' && ch <= 'Z')
        kind = upperLetter;
    else {
        if (ch >= 'a' && ch <= 'z')
            kind = lowerLetter;
        else
            kind = special;
    }
}

```

Để cho dễ đọc có thể sử dụng hình thức sau:

```

if (ch >= '0' && ch <= '9')
    kind = digit;
else if (ch >= 'A' && ch <= 'Z')
    kind = capitalLetter;
else if (ch >= 'a' && ch <= 'z')
    kind = smallLetter;
else
    kind = special;

```

3.3. Lệnh switch

Lệnh switch cung cấp phương thức lựa chọn giữa một tập các khả năng dựa trên giá trị của biểu thức. Hình thức chung của câu lệnh switch là:

```

switch (biểu thức) {
    case hằng1:
        các lệnh;
    ...
    case hằngn:
        các lệnh;
    default:
        các lệnh;
}

```

Biểu thức (gọi là thẻ switch) được ước lượng trước tiên và kết quả được so sánh với mỗi hằng số (gọi là các nhãn) theo thứ tự chúng xuất hiện cho đến khi một so khớp được tìm thấy. *Lệnh* ngay sau khi so khớp được thực hiện

sau đó. Chú ý số nhiều: mỗi case có thể được theo sau bởi không hay nhiều lệnh (không chỉ là một lệnh). Việc thực thi tiếp tục cho tới khi hoặc là bắt gặp một lệnh break hoặc tất cả các lệnh xen vào đến cuối lệnh switch được thực hiện. Trường hợp default ở cuối cùng là một tùy chọn và được thực hiện nếu như tất cả các case trước đó không được so khớp.

Ví dụ, chúng ta phải phân tích cú pháp một phép toán toán học nhị hạng thành ba thành phần của nó và phải lưu trữ chúng vào các biến operator, operand1, và operand2. Lệnh switch sau thực hiện phép toán và lưu trữ kết quả vào result.

```
switch (operator) {
    case '+': result = operand1 + operand2;
              break;
    case '-': result = operand1 - operand2;
              break;
    case '*': result = operand1 * operand2;
              break;
    case '/': result = operand1 / operand2;
              break;
    default: cout << "unknown operator: " << operator << '\n';
              break;
}
```

Như đã được minh họa trong ví dụ, chúng ta cần thiết chèn một lệnh break ở cuối mỗi case. Lệnh break ngắt câu lệnh switch bằng cách nhảy đến điểm kết thúc của lệnh này. Ví dụ, nếu chúng ta mở rộng lệnh trên để cho phép x cũng có thể được sử dụng như là toán tử nhân, chúng ta sẽ có:

```
switch (operator) {
    case '+': result = operand1 + operand2;
              break;
    case '-': result = operand1 - operand2;
              break;
    case 'x':
    case '*': result = operand1 * operand2;
              break;
    case '/': result = operand1 / operand2;
              break;
    default: cout << "unknown operator: " << operator << '\n';
              break;
}
```

Bởi vì case 'x' không có lệnh break nên khi case này được thỏa thì sự thực thi tiếp tục thực hiện các lệnh trong case kế tiếp và phép nhân được thi hành.

Chúng ta có thể quan sát rằng bất kỳ lệnh switch nào cũng có thể được viết như nhiều câu lệnh if-else. Ví dụ, lệnh trên có thể được viết như sau:

```

if(operator == '+')
    result = operand1 + operand2;
else if(operator == '-')
    result = operand1 - operand2;
else if(operator == 'x' || operator == '*')
    result = operand1 * operand2;
else if(operator == '/')
    result = operand1 / operand2;
else
    cout << "unknown operator: " << ch << '\n';

```

người ta có thể cho rằng phiên bản switch là rõ ràng hơn trong trường hợp này. Tiếp cận if-else nên được dành riêng cho tình huống mà trong đó switch không thể làm được công việc (ví dụ, khi các điều kiện là phức tạp không thể đơn giản thành các đẳng thức toán học hay khi các nhãn cho các case không là các hằng số).

3.4. Lệnh while

Lệnh while (cũng được gọi là **vòng lặp while**) cung cấp phương thức lặp một lệnh cho tới khi một điều kiện được thỏa. Hình thức chung của lệnh lặp là:

```

while (biểu thức)
    lệnh;

```

Biểu thức (cũng được gọi là **điều kiện lặp**) được ước lượng trước tiên. Nếu kết quả khác 0 thì sau đó *lệnh* (cũng được gọi là **thân vòng lặp**) được thực hiện và toàn bộ quá trình được lặp lại. Ngược lại, vòng lặp được kết thúc.

Ví dụ, chúng ta muốn tính tổng của tất cả các số nguyên từ 1 tới n. Điều này có thể được diễn giải như sau:

```

i = 1;
sum = 0;
while(i <= n){
    sum += i;
    i++;
}

```

Trường hợp n là 5, Bảng 3.1 cung cấp bảng phát họa vòng lặp bằng cách liệt kê các giá trị của các biến có liên quan và điều kiện lặp.

Bảng 3.1 Vết của vòng lặp while.

Vòng lặp	i	n	i <= n	sum += i++
Một	1	5	1	1
Hai	2	5	1	3
Ba	3	5	1	6
Bốn	4	5	1	10
Năm	5	5	1	15
Sáu	6	5	0	

Đôi khi chúng ta có thể gặp vòng lặp `while` có thân rỗng (nghĩa là một câu lệnh `null`). Ví dụ vòng lặp sau đặt `n` tới thừa số lẻ lớn nhất của nó.

```
while(n%2 == 0 && n != 2)    ;
```

Ở đây điều kiện lặp cung cấp tất cả các tính toán cần thiết vì thế không thật sự cần một thân cho vòng lặp. Điều kiện vòng lặp không những kiểm tra `n` là chẵn hay không mà nó còn chia `n` cho 2 và chắc chắn rằng vòng lặp sẽ dừng.

3.5. Lệnh `do - while`

Lệnh `do` (cũng được gọi là **vòng lặp `do`**) thì tương tự như lệnh `while` ngoại trừ thân của nó được thực thi trước tiên và sau đó điều kiện vòng lặp mới được kiểm tra. Hình thức chung của lệnh `do` là:

```
do
    lệnh;
while (biểu thức);
```

Lệnh được thực thi trước tiên và sau đó *biểu thức* được ước lượng. Nếu kết quả của biểu thức khác 0 thì sau đó toàn bộ quá trình được lặp lại. Ngược lại thì vòng lặp kết thúc.

Vòng lặp `do` ít được sử dụng thường xuyên hơn vòng lặp `while`. Nó hữu dụng trong những trường hợp khi chúng ta cần thân vòng lặp thực hiện ít nhất một lần mà không quan tâm đến điều kiện lặp. Ví dụ, giả sử chúng ta muốn thực hiện lặp đi lặp lại công việc đọc một giá trị và in bình phương của nó, và dừng khi giá trị là 0. Điều này có thể được diễn giải trong vòng lặp sau đây:

```
do {
    cin >> n;
    cout << n * n << '\n';
} while (n != 0);
```

Không giống như vòng lặp `while`, vòng lặp `do` ít khi được sử dụng trong những tình huống mà nó có một thân rỗng. Mặc dù vòng lặp `do` với thân rỗng có thể là tương đương với một vòng lặp `while` tương tự nhưng vòng lặp `while` thì luôn dễ đọc hơn.

3.6. Lệnh `for`

Lệnh `for` (cũng được gọi là **vòng lặp `for`**) thì tương tự như vòng lặp `while` nhưng có hai thành phần thêm vào: một biểu thức được ước lượng chỉ một lần trước hết và một biểu thức được ước lượng mỗi lần ở cuối mỗi lần lặp. Hình thức tổng quát của lệnh `for` là:

```
for (biểu thức1; biểu thức2; biểu thức3)
    lệnh;
```

Biểu thức₁ (thường được gọi là biểu thức khởi tạo) được ước lượng trước tiên. Mỗi vòng lặp *biểu thức₂* được ước lượng. Nếu kết quả không là 0 (đúng) thì sau đó *lệnh* được thực thi và *biểu thức₃* được ước lượng. Ngược lại, vòng lặp kết thúc. Vòng lặp for tổng quát thì tương đương với vòng lặp while sau:

```
biểu thức1;
while (biểu thức2) {
    lệnh;
    biểu thức3;
}
```

Vòng lặp for thường được sử dụng trong các trường hợp mà có một biến được tăng hay giảm ở mỗi lần lặp. Ví dụ, vòng lặp for sau tính toán tổng của tất cả các số nguyên từ 1 tới n.

```
sum = 0;
for (i = 1; i <= n; ++i)
    sum += i;
```

Điều này được ưa chuộng hơn phiên bản của vòng lặp while mà chúng ta thấy trước đó. Trong ví dụ này i thường được gọi là **biến lặp**.

C++ cho phép biểu thức đầu tiên trong vòng lặp for là một định nghĩa biến. Ví dụ trong vòng lặp trên thì i có thể được định nghĩa bên trong vòng lặp:

```
for (int i = 1; i <= n; ++i)
    sum += i;
```

Trái với sự xuất hiện, phạm vi của i không ở trong thân của vòng lặp mà là chính vòng lặp. Xét trên phạm vi thì ở trên tương đương với:

```
int i;
for (i = 1; i <= n; ++i)
    sum += i;
```

Bất kỳ biểu thức nào trong 3 biểu thức của vòng lặp for có thể rỗng. Ví dụ, xóa biểu thức đầu và biểu thức cuối cho chúng ta dạng giống như vòng lặp while:

```
for (; i != 0;)           // tương đương với: while (i != 0)
    something;           //
```

Xóa tất cả các biểu thức cho chúng ta một vòng lặp vô hạn. Điều kiện của vòng lặp này được giả sử luôn luôn là đúng.

```
for (;;)                 // vòng lặp vô hạn
    something;
```

Trường hợp vòng lặp với nhiều biến lặp thì hiếm dùng. Trong những trường hợp như thế, toán tử phẩy (,) được sử dụng để phân cách các biểu thức của chúng:

```
for(i=0,j=0; i+j<n; ++i, ++j)
    something;
```

Bởi vì các vòng lặp là các lệnh nên chúng có thể xuất hiện bên trong các vòng lặp khác. Nói cách khác, các vòng lặp có thể lồng nhau. Ví dụ,

```
for(int i=1; i<=3; ++i)
    for(int j=1; j<=3; ++j)
        cout<<'('<<i<<' '<<j<<")\n";
```

cho tích số của tập hợp {1,2,3} với chính nó, kết quả như sau:

```
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
(3,1)
(3,2)
(3,3)
```

3.7. Lệnh continue

Lệnh continue dừng lần lặp hiện tại của một vòng lặp và nhảy tới lần lặp kế tiếp. Nó áp dụng tức thì cho vòng lặp gần với lệnh continue. Sử dụng lệnh continue bên ngoài vòng lặp là lỗi.

Trong vòng lặp while và vòng lặp do-while, vòng lặp kế tiếp mở đầu từ điều kiện lặp. Trong vòng lặp for, lần lặp kế tiếp khởi đầu từ biểu thức thứ ba của vòng lặp. Ví dụ, một vòng lặp thực hiện đọc một số, xử lý nó nhưng bỏ qua những số âm, và dừng khi số là 0, có thể diễn giải như sau:

```
do {
    cin >> num;
    if (num < 0) continue;
    // xử lý số ở đây ...
} while (num != 0);
```

Điều này tương đương với:

```
do {
    cin >> num;
    if (num >= 0) {
        // xử lý số ở đây ...
    }
} while (num != 0);
```

Một biến thể của vòng lặp này để đọc chính xác một số n lần (hơn là cho tới khi số đó là 0) có thể được diễn giải như sau:

```
for (i=0; i<n; ++i) {
    cin >> num;
    if (num<0) continue;           // làm cho nhảy tới: ++i
    // xử lý số ở đây ...
}
```

Khi lệnh `continue` xuất hiện bên trong vòng lặp được lồng vào thì nó áp dụng trực tiếp lên vòng lặp gần nó chứ không áp dụng cho vòng lặp bên ngoài. Ví dụ, trong một tập các vòng lặp được lồng nhau sau đây, lệnh `continue` áp dụng cho vòng lặp `for` và không áp dụng cho vòng lặp `while`:

```
while (more) {
    for (i=0; i<n; ++i) {
        cin >> num;
        if (num<0) continue;       // làm cho nhảy tới: ++i
        // process num here...
    }
    //etc...
}
```

3.8. Lệnh `break`

Lệnh `break` có thể xuất hiện bên trong vòng lặp (`while`, `do`, hay `for`) hoặc một lệnh `switch`. Nó gây ra bước nhảy ra bên ngoài những lệnh này và vì thế kết thúc chúng. Giống như lệnh `continue`, lệnh `break` chỉ áp dụng cho vòng lặp hoặc lệnh `switch` gần nó. Sử dụng lệnh `break` bên ngoài vòng lặp hay lệnh `switch` là lỗi.

Ví dụ, chúng ta đọc vào một mật khẩu người dùng nhưng không cho phép một số hữu hạn lần thử:

```
for (i=0; i< attempts; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password)) // kiểm tra mật khẩu đúng hay sai
        break;           // thoát khỏi vòng lặp
    cout << "Incorrect!\n";
}
```

Ở đây chúng ta phải giả sử rằng có một hàm được gọi `Verify` để kiểm tra một mật khẩu và trả về `true` nếu như mật khẩu đúng và ngược lại là `false`.

Chúng ta có thể viết lại vòng lặp mà không cần lệnh `break` bằng cách sử dụng một biến luận lý được thêm vào (`verified`) và thêm nó vào điều kiện vòng lặp:

```
verified=0;
for (i=0; i< attempts && !verified; ++i) {
```

```

        cout << "Please enter your password: ";
        cin >> password;
        verified = Verify(password));
        if (!verified)
            cout << "Incorrect!\n";
    }

```

Người ta cho rằng phiên bản của `break` thì đơn giản hơn nên thường được ưa chuộng hơn.

3.9. Lệnh `goto`

Lệnh `goto` cung cấp mức thấp nhất cho việc nhảy. Nó có hình thức chung là:

```
goto nhãn;
```

trong đó *nhãn* là một định danh được dùng để đánh dấu đích cần nhảy tới. Nhãn cần được theo sau bởi một dấu hai chấm (:) và xuất hiện trước một lệnh bên trong hàm như chính lệnh `goto`.

Ví dụ, vai trò của lệnh `break` trong vòng lặp `for` trong phần trước có thể viết lại bởi một lệnh `goto`.

```

for (i = 0; i < attempts; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    if (Verify(password)) // check password for correctness
        goto out;        // drop out of the loop
    cout << "Incorrect!\n";
}
out:
//etc...

```

Bởi vì lệnh `goto` cung cấp một hình thức nhảy tự do không có cấu trúc (không giống như lệnh `break` và `continue`) nên dễ làm gây đổ chương trình. Phần lớn các lập trình viên ngày nay tránh sử dụng nó để làm cho chương trình rõ ràng. Tuy nhiên, `goto` có một vài (dù cho hiếm) sử dụng chính đáng. Vì sự phức tạp của những trường hợp như thế mà việc cung cấp những ví dụ được trình bày ở những phần sau.

3.10. Lệnh `return`

Lệnh `return` cho phép một hàm trả về một giá trị cho thành phần gọi nó. Nó có hình thức tổng quát:

```
return biểu thức;
```

trong đó *biểu thức* chỉ rõ giá trị được trả về bởi hàm. Kiểu của giá trị này nên hợp với kiểu của hàm. Trường hợp kiểu trả về của hàm là void, *biểu thức* nên rỗng:

```
return;
```

Hàm mà được chúng ta thảo luận đến thời điểm này chỉ có hàm main, kiểu trả về của nó là kiểu int. Giá trị trả về của hàm main là những gì mà chương trình trả về cho hệ điều hành khi nó hoàn tất việc thực thi. Chẳng hạn dưới UNIX qui ước là trả về 0 từ hàm main khi chương trình thực thi không có lỗi. Ngược lại, một mã lỗi khác 0 được trả về. Ví dụ:

```
int main(void)
{
    cout << "Hello World\n";
    return 0;
}
```

Khi một hàm có giá trị trả về không là void (như trong ví dụ trên), nếu không trả về một giá trị sẽ mang lại một cảnh báo trình biên dịch. Giá trị trả về thực sự sẽ không được định nghĩa trong trường hợp này (nghĩa là, nó sẽ là bất cứ giá trị nào được giữ trong vị trí bộ nhớ tương ứng của nó tại thời điểm đó).

Bài tập cuối chương 3

- 3.1 Viết chương trình nhập vào chiều cao (theo centimet) và trọng lượng (theo kilogram) của một người và xuất một trong những thông điệp: underweight, normal, hoặc overweight, sử dụng điều kiện:

```
Underweight:  weight < height/2.5
Normal:       height/2.5 <= weight <= height/2.3
Overweight:   height/2.3 < weight
```

- 3.2 Giả sử rằng n là 20, đoạn mã sau sẽ xuất ra cái gì khi nó được thực thi?

```
if(n >= 0)
    if(n < 10)
        cout << "n is small\n";
    else
        cout << "n is negative\n";
```

- 3.3 Viết chương trình nhập một ngày theo định dạng dd/mm/yy và xuất nó theo định dạng month dd, year. Ví dụ, 25/12/61 trở thành:

```
Thang muoi hai 25, 1961
```

- 3.4 Viết chương trình nhập vào một giá trị số nguyên, kiểm tra nó là dương hay không và xuất ra giai thừa của nó, sử dụng công thức:

$$\begin{aligned}giaithua(0) &= 1 \\giaithua(n) &= n \times giaithua(n-1)\end{aligned}$$

- 3.5 Viết chương trình nhập vào một số cơ số 8 và xuất ra số thập phân tương đương. Ví dụ sau minh họa các công việc thực hiện của chương trình theo mong đợi:

Nhap vao so bat phan: **214**
 BatPhan(214)=ThapPhan(140)

- 3.6 Viết chương trình cung cấp một bảng cửu chương đơn giản của định dạng sau cho các số nguyên từ 1 tới 9:

1 x 1 = 1
 1 x 2 = 2
 ...
 9 x 9 = 81

Chương 4. Hàm

Chương này mô tả những hàm do người dùng định nghĩa như là một trong những khối chương trình C++. Hàm cung cấp một phương thức để đóng gói quá trình tính toán một cách dễ dàng để được sử dụng khi cần. **Định nghĩa hàm** gồm hai phần: giao diện và thân.

Phản giao diện hàm (cũng được gọi là **khai báo hàm**) đặc tả hàm có thể được sử dụng như thế nào. Nó gồm ba phần:

- **Tên hàm.** Đây chỉ là một định danh duy nhất.
- **Các tham số của hàm.** Đây là một tập của không hay nhiều định danh đã định kiểu được sử dụng để truyền các giá trị tới và từ hàm.
- **Kiểu trả về của hàm.** Kiểu trả về của hàm đặc tả cho kiểu của giá trị mà hàm trả về. Hàm không trả về bất kỳ kiểu nào thì nên trả về kiểu void.

Phản thân hàm chứa đựng các bước tính toán (các lệnh).

Sử dụng một hàm liên quan đến việc gọi nó. Một **lời gọi hàm** gồm có tên hàm, theo sau là cặp dấu ngoặc đơn '()', bên trong cặp dấu ngoặc là không, một hay nhiều **đối số** được tách biệt nhau bằng dấu phẩy. Số các đối số phải khớp với số các tham số của hàm. Mỗi đối số là một biểu thức mà kiểu của nó phải khớp với kiểu của tham số tương ứng trong khai báo hàm.

Khi lời gọi hàm được thực thi, các đối số được ước lượng trước tiên và các giá trị kết quả của chúng được gán tới các tham số tương ứng. Sau đó thân hàm được thực hiện. Cuối cùng giá trị trả về của hàm được truyền tới thành phần gọi hàm.

Vì một lời gọi tới một hàm mà kiểu trả về không là void sẽ mang lại một giá trị trả về nên lời gọi là một biểu thức và có thể được sử dụng trong các biểu thức khác. Ngược lại một lời gọi tới một hàm mà kiểu trả về của nó là void thì lời gọi là một lệnh.

4.1. Hàm đơn giản

Danh sách 4.1 trình bày định nghĩa của một hàm đơn giản để tính lũy thừa của một số nguyên.

Danh sách 4.1

```
1 int Power (int base, unsigned int exponent)
2 {
3     int    result = 1;
4
5     for (int i = 0; i < exponent; ++i)
6         result *= base;
7     return result;
}
```

Chú giải

- 1 Dòng này định nghĩa giao diện hàm. Nó bắt đầu với kiểu trả về của hàm (là int trong trường hợp này). Kế tiếp là tên hàm, theo sau là danh sách các tham số. Power có hai tham số (base và exponent) thuộc kiểu int và unsigned int tương ứng. Chú ý là cú pháp cho các tham số là tương tự như cú pháp cho định nghĩa biến: định danh kiểu được theo sau bởi tên tham số. Tuy nhiên, không thể theo sau định danh kiểu với nhiều tham số phân cách bởi dấu phẩy:
`int Power (int base, exponent) // Sai!`
- 2 Dấu ngoặc này đánh dấu điểm bắt đầu của thân hàm.
- 3 Dòng này là định nghĩa một **biến cục bộ**.
- 4-5 Vòng lặp for này tăng cơ số base lên lũy thừa của exponent và lưu trữ kết quả vào trong result.
- 6 Hàng này trả result về như là kết quả của hàm.
- 7 Dấu ngoặc này đánh dấu điểm kết thúc của thân hàm.

Danh sách 4.2 minh họa hàm được gọi như thế nào. Tác động của lời gọi hàm này là đầu tiên các giá trị 2 và 8 tương ứng được gán cho các tham số base và exponent, và sau đó thân hàm được ước lượng.

Danh sách 4.2

```
1 #include <iostream.h>
2
3 main (void)
4 {
5     cout << "2 ^ 8 =" << Power(2,8) << '\n';
}
```

Khi chạy chương trình này xuất ra kết quả sau:

$2^8 = 256$

Nói chung, một hàm phải được khai báo trước khi sử dụng nó. **Khai báo hàm** (function declaration) đơn giản gồm có mẫu ban đầu của hàm gọi là nguyên mẫu hàm (function prototype) chỉ định tên hàm, các kiểu tham số, và kiểu trả về. Hàng 2 trong Danh sách 4.3 trình bày hàm Power có thể được khai báo như thế nào cho chương trình trên. Nhưng một hàm cũng có thể được khai báo mà không cần tên các tham số của nó,

```
int Power (int, unsigned int);
```

tuy nhiên chúng ta không nên làm điều đó trừ phi vai trò của các tham số là rõ ràng.

Danh sách 4.3

```
1 #include <iostream.h>
2 int Power (int base, unsigned int exponent); // khai báo hàm
3
4 main (void)
5 {
6     cout << "2 ^ 8 =" << Power(2,8) << '\n';
7 }
8
9 int Power (int base, unsigned int exponent)
10 {
11     int result = 1;
12     for (int i = 0; i < exponent; ++i)
13         result *= base;
14     return result;
15 }
```

Bởi vì một định nghĩa hàm chứa đựng một nguyên mẫu (prototype) nên nó cũng được xem như là một khai báo. Vì thế nếu định nghĩa của một hàm xuất hiện trước khi sử dụng nó thì không cần khai báo thêm vào. Tuy nhiên việc sử dụng các nguyên mẫu hàm là khuyến khích cho mọi trường hợp. Tập hợp của nhiều hàm vào một tập tin header riêng biệt cho phép những lập trình viên khác truy xuất nhanh chóng tới các hàm mà không cần phải đọc toàn bộ các định nghĩa của chúng.

4.2. Tham số và đối số

C++ hỗ trợ hai kiểu tham số: giá trị và tham chiếu. **Tham số giá trị** nhận một sao chép giá trị của đối số được truyền tới nó. Kết quả là, nếu hàm có bất kỳ chuyển đổi nào tới tham số thì vẫn không tác động đến đối số. Ví dụ, trong

```
#include <iostream.h>

void Foo (int num)
{
    num = 0;
}
```

```

        cout << "num - " << num << '\n';
    }

    int main (void)
    {
        int x = 10;

        Foo(x);
        cout << "x = " << x << '\n';
        return 0;
    }

```

thì tham số duy nhất của hàm Foo là một tham số giá trị. Đến lúc mà hàm này được thực thi thì num được sử dụng như là một biến cục bộ bên trong hàm. Khi hàm được gọi và x được truyền tới nó, num nhận một sao chép giá trị của x. Kết quả là mặc dù num được đặt về 0 bởi hàm nhưng vẫn không có gì tác động lên x. Chương trình cho kết quả như sau:

```

num = 0;
x = 10;

```

Trái lại, **tham số tham chiếu** nhận các đối số được truyền tới nó và làm trực tiếp trên đối số đó. Bất kỳ chuyển đổi nào được tạo ra bởi hàm tới tham số tham chiếu đều tác động trực tiếp lên đối số.

Bên trong ngữ cảnh của các lời gọi hàm, hai kiểu truyền đối số tương ứng được gọi là **truyền-bằng-giá trị** và **truyền-bằng-tham chiếu**. Thật là hoàn toàn hợp lệ cho một hàm truyền-bằng-giá trị đối với một vài tham số và truyền-bằng-tham chiếu cho một vài tham số khác. Trong thực tế thì truyền-bằng-giá trị thường được sử dụng nhiều hơn.

4.3. Phạm vi cục bộ và toàn cục

Mọi thứ được định nghĩa ở mức phạm vi chương trình (nghĩa là bên ngoài các hàm và các lớp) được hiểu là có một **phạm vi toàn cục** (global scope). Các hàm ví dụ mà chúng ta đã thấy cho đến thời điểm này đều có một phạm vi toàn cục. Các biến cũng có thể định nghĩa ở phạm vi toàn cục:

```

int year = 1994;           // biến toàn cục
int Max (int, int);        // hàm toàn cục
int main (void)            // hàm toàn cục
{
    //...
}

```

Các biến toàn cục không được khởi tạo, sẽ được khởi tạo tự động là 0.

Vì các đầu vào toàn cục là có thể thấy được ở mức chương trình nên chúng cũng phải là duy nhất ở mức chương trình. Điều này nghĩa là cùng các biến hoặc hàm toàn cục có thể không được định nghĩa nhiều hơn một lần ở

mức toàn cục. (Tuy nhiên chúng ta sẽ thấy sau này một tên hàm có thể được sử dụng lại). Thông thường các biến hay hàm toàn cục có thể được truy xuất từ mọi nơi trong chương trình.

Mỗi khối trong một chương trình định nghĩa một **phạm vi cục bộ**. Thật vậy, thân của một hàm trình bày một phạm vi cục bộ. Các tham số của một hàm có cùng phạm vi như là thân hàm. Các biến được định nghĩa ở bên trong một phạm vi cục bộ có thể nhìn thấy tới chỉ phạm vi đó. Do đó một biến chỉ cần là duy nhất ở trong phạm vi của chính nó. Các phạm vi cục bộ có thể lồng nhau, trong trường hợp này các phạm vi bên trong chồng lên các phạm vi bên ngoài. Ví dụ trong

```
int xyz;                // xyz là toàn cục
void Foo (int xyz)      // xyz là cục bộ cho thân của Foo
{
    if (xyz > 0) {
        double xyz;    // xyz là cục bộ cho khối này
        //...
    }
}
```

có ba phạm vi riêng biệt, mỗi phạm vi chứa đựng một xyz riêng.

Thông thường, thời gian sống của một biến bị giới hạn bởi phạm vi của nó. Vì thế, ví dụ các biến toàn cục tồn tại suốt thời gian thực hiện chương trình trong khi các biến cục bộ được tạo ra khi phạm vi của chúng bắt đầu và mất đi khi phạm vi của chúng kết thúc. Không gian bộ nhớ cho các biến toàn cục được dành riêng trước khi sự thực hiện của chương trình bắt đầu nhưng ngược lại không gian bộ nhớ cho các biến cục bộ được cấp phát ở thời điểm thực hiện chương trình.

4.4. Toán tử phạm vi

Bởi vì phạm vi cục bộ ghi chồng lên phạm vi toàn cục nên một biến cục bộ có cùng tên với biến toàn cục làm cho biến toàn cục không thể truy xuất được tới phạm vi cục bộ. Ví dụ, trong

```
int error;

void Error (int error)
{
    //...
}
```

biến toàn cục `error` là không thể truy xuất được bên trong hàm `Error` bởi vì nó được ghi chồng bởi tham số `error` cục bộ.

Vấn đề này được giải quyết nhờ vào sử dụng toán tử phạm vi đơn hạng (`::`), toán tử này lấy đầu vào toàn cục như là đối số:

```

int error;

void Error(int error)
{
    //...
    if (::error != 0)           // tham khảo tới error toàn cục
        //...
}

```

4.5. Biến tự động

Bởi vì thời gian sống của một biến cục bộ là có giới hạn và được xác định hoàn toàn tự động nên những biến này cũng được gọi là **tự động**. Bộ xác định lớp lưu trữ `auto` có thể được dùng để chỉ định rõ ràng một biến cục bộ là tự động. Ví dụ:

```

void Foo(void)
{
    auto int xyz;           // như là: int xyz;
    //...
}

```

Điều này ít khi được sử dụng bởi vì tất cả các biến cục bộ mặc định là tự động.

4.6. Biến thanh ghi

Như được đề cập trước đó, nói chung các biến biểu thị các vị trí bộ nhớ nơi mà giá trị của biến được lưu trữ tới. Khi mã chương trình tham khảo tới một biến (ví dụ, trong một biểu thức), trình biên dịch phát ra các mã máy truy xuất tới vị trí bộ nhớ được biểu thị bởi các biến. Đối với các biến dùng thường xuyên (ví dụ như các biến vòng lặp), hiệu suất chương trình có thể thu được bằng cách giữ biến trong một thanh ghi, bằng cách này có thể tránh được truy xuất bộ nhớ tới biến đó.

Bộ lưu trữ thanh ghi có thể được sử dụng để chỉ định cho trình biên dịch biến có thể được lưu trữ trong một thanh ghi nếu có thể. Ví dụ:

```

for (register int i = 0; i < n; ++i)
    sum += i;

```

Ở đây mỗi vòng lặp `i` được sử dụng ba lần: một lần khi nó được so sánh với `n`, một lần khi nó được cộng vào `sum`, và một lần khi nó được tăng. Vì thế việc giữ biến `i` trong thanh ghi trong suốt vòng lặp `for` là có ý nghĩa trong việc cải thiện hiệu suất chương trình.

Chú ý rằng thanh ghi chỉ là một gợi ý cho trình biên dịch, và trong một vài trường hợp trình biên dịch có thể chọn không sử dụng thanh ghi khi nó được

yêu cầu làm điều đó. Một lý do để lý giải là bất kỳ máy tính nào cũng có một số hữu hạn các thanh ghi và nó có thể ở trong trường hợp tất cả các thanh ghi đang được sử dụng.

Thậm chí khi lập trình viên không khai báo thanh ghi, nhiều trình biên dịch tối ưu cố gắng thực hiện một dự đoán thông minh và sử dụng các thanh ghi mà chúng muốn để cải thiện hiệu suất của chương trình.

Ý tưởng sử dụng khai báo thanh ghi thường được đề xuất sau cùng; nghĩa là sau khi viết mã chương trình hoàn tất lập trình viên có thể xem lại mã và chèn các khai báo thanh ghi vào những nơi cần thiết.

4.7. Hàm nội tuyến

Giả sử một chương trình thường xuyên yêu cầu tìm giá trị tuyệt đối của một số các số nguyên. Cho một giá trị được biểu thị bởi n , điều này có thể được giải thích như sau:

$$(n > 0 ? n : -n)$$

Tuy nhiên, thay vì tái tạo biểu thức này tại nhiều vị trí khác nhau trong chương trình, tốt hơn hết là nên định nghĩa nó trong một hàm như sau:

```
int Abs(int n)
{
    return n > 0 ? n : -n;
}
```

Phiên bản hàm có một số các thuận lợi. Thứ nhất, nó làm cho chương trình dễ đọc. Thứ hai, nó có thể được sử dụng lại. Và thứ ba, nó tránh được hiệu ứng phụ không mong muốn khi đối số chính nó là một biểu thức có các hiệu ứng phụ.

Tuy nhiên, bất lợi của phiên bản hàm là việc sử dụng thường xuyên có thể dẫn tới sự bất lợi về hiệu suất đáng kể vì các tổn phí dành cho việc gọi hàm. Ví dụ, nếu hàm Abs được sử dụng trong một vòng lặp được lặp đi lặp lại một ngàn lần thì sau đó nó sẽ có một tác động trên hiệu suất. Tổn phí có thể được tránh bằng cách định nghĩa hàm Abs như là hàm nội tuyến (inline):

```
inline int Abs(int n)
{
    return n > 0 ? n : -n;
}
```

Hiệu quả của việc sử dụng hàm nội tuyến là khi hàm Abs được gọi, trình biên dịch thay vì phát ra mã để gọi hàm Abs thì mở rộng và thay thế thân của hàm Abs vào nơi gọi. Trong khi về bản chất thì cùng tính toán được thực hiện nhưng không có liên quan đến lời gọi hàm vì thế mà không có cấp phát stack.

Bởi vì các lời gọi tới hàm nội tuyến được mở rộng nên không có vết của chính hàm được đưa vào trong mã đã biên dịch. Vì thế, nếu một hàm được định nghĩa nội tuyến ở trong một tập tin thì nó không sẵn dùng cho các tập tin khác. Do đó, các hàm nội tuyến thường được đặt vào trong các tập tin header để mà chúng có thể được chia sẻ.

Giống như từ khóa register, inline là một gợi ý cho trình biên dịch thực hiện. Nói chung, việc sử dụng inline nên có hạn chế chỉ cho các hàm đơn giản được sử dụng thường xuyên mà thôi. Việc sử dụng inline cho các hàm dài và phức tạp quá thì chắc chắn bị bỏ qua bởi trình biên dịch.

4.8. Đệ qui

Một hàm gọi chính nó được gọi là **đệ qui**. Đệ qui là một kỹ thuật lập trình tổng quát có thể ứng dụng cho các bài toán mà có thể định nghĩa theo thuật ngữ của chính chúng. Chẳng hạn bài toán giai thừa được định nghĩa như sau:

- Giai thừa của 0 là 1.
- Giai thừa của một số n là n lần giai thừa của $n-1$.

Hàng thứ hai rõ ràng cho biết giai thừa được định nghĩa theo thuật ngữ của chính nó và vì thế có thể được biểu diễn như một hàm đệ qui:

```
int Factorial(unsigned int n)
{
    return n == 0 ? 1 : n * Factorial(n-1);
}
```

Cho n bằng 3, Bảng 4.1 cung cấp vết của các lời gọi Factorial. Các khung stack cho các lời gọi này xuất hiện tuần tự từng cái một trên runtime stack.

Bảng 4.1 Vết thực thi của Factorial(3).

Call	n	n == 0	n * Factorial(n-1)	Returns
Thứ nhất	3	0	3 * Factorial(2)	6
Thứ hai	2	0	2 * Factorial(1)	2
Thứ ba	1	0	1 * Factorial(0)	1
Thứ tư	0	1		1

Một hàm đệ qui phải có ít nhất một **điều kiện dừng** có thể được thỏa. Ngược lại, hàm sẽ gọi chính nó vô hạn định cho tới khi tràn stack. Ví dụ hàm Factorial có điều kiện dừng là $n == 0$. (Chú ý đối với trường hợp n là số âm thì điều kiện sẽ không bao giờ thỏa và Factorial sẽ thất bại).

4.9. Đối số mặc định

Đối số mặc định là một thuận lợi lập trình để bỏ bớt đi gánh nặng phải chỉ định các giá trị đối số cho tất cả các tham số hàm. Ví dụ, xem xét hàm cho việc báo cáo lỗi:

```
void Error(char *message, int severity=0);
```

Ở đây thì severity có một đối số mặc định là 0; vì thế cả hai lời gọi sau đều hợp lệ:

```
Error("Division by zero", 3); // severity đặt tới 3
Error("Round off error");    // severity đặt tới 0
```

Như là lời gọi hàm đầu tiên minh họa, một đối số mặc định có thể được ghi chồng bằng cách chỉ định rõ ràng một đối số.

Các đối số mặc định là thích hợp cho các trường hợp mà trong đó các tham số nào đó của hàm (hoặc tất cả) thường xuyên lấy cùng giá trị. Ví dụ trong hàm Error, severity 0 lỗi thì phổ biến hơn là những trường hợp khác và vì thế là một ứng cử viên tốt cho đối số mặc định. Một cách dùng các đối số ít phù hợp có thể là:

```
int Power(int base, unsigned int exponent=1);
```

Bởi vì 1 (hoặc bất kỳ giá trị nào khác) thì không chắc xảy ra thường xuyên trong tình huống này.

Để tránh mơ hồ, tất cả đối số mặc định phải là các đối số theo đuôi. Vì thế khai báo sau là không theo luật:

```
void Error(char *message="Bomb", int severity); // Trái qui tắc
```

Một đối số mặc định không nhất thiết là một hằng. Các biểu thức tùy ý có thể được sử dụng miễn là các biến được dùng trong các biểu thức là có sẵn cho phạm vi định nghĩa hàm (ví dụ, các biến toàn cục).

Qui ước được chấp nhận dành cho các đối số mặc định là chỉ định chúng trong các khai báo hàm chứ không ở trong định nghĩa hàm.

4.10. Đối số hàng lệnh

Khi một chương trình được thực thi dưới một hệ điều hành (như là DOS hay UNIX) nó có thể nhận không hay nhiều đối số từ dòng lệnh. Các đối số này xuất hiện sau tên chương trình có thể thực thi và được phân cách bởi các khoảng trắng. Bởi vì chúng xuất hiện trên cùng hàng nơi mà các lệnh của hệ điều hành phát ra nên chúng được gọi là các **đối số hàng lệnh**.

Ví dụ như xem xét một chương trình được đặt tên là `sum` để in ra tổng của tập hợp các số được cung cấp tới nó như là các đối số hàng lệnh. Hộp thoại 4.1 minh họa hai số được truyền như là các đối số tới hàm `sum` như thế nào (\$ là dấu nhắc UNIX).

Hộp thoại 4.1

```
1 $ sum 10.4 12.5
2 22.9
3 $
```

Các đối số hàng lệnh được tạo ra sẵn cho một chương trình C++ thông qua hàm `main`. Có hai cách định nghĩa một hàm `main`:

```
int main(void);
int main(int argc, const char* argv[]);
```

Cách sau được sử dụng khi chương trình được dự tính để chấp nhận các đối số hàng lệnh. Tham số đầu, `argc`, biểu thị số các đối số được truyền tới chương trình (bao gồm cả tên của chính chương trình). Tham số thứ hai, `argv`, là một mảng của các hằng chuỗi đại diện cho các đối số. Ví dụ từ hàng lệnh đã cho trong hộp thoại 4.1, chúng ta có:

```
argc      is      3
argv[0]    is      "sum"
argv[1]    is      "10.4"
argv[2]    is      "12.5"
```

Danh sách 4.4 minh họa một thi công đơn giản cho chương trình tính tổng `sum`. Các chuỗi được chuyển đổi sang số thực sử dụng hàm `atof` được định nghĩa trong thư viện `stdlib.h`.

Danh sách 4.4

```
1 #include <iostream.h>
2 #include <stdlib.h>
3
4 int main (int argc, const char *argv[])
5 {
6     double sum = 0;
7     for (int i = 1; i < argc; ++i)
8         sum += atof(argv[i]);
9     cout << sum << '\n';
10    return 0;
11 }
```

Bài tập cuối chương 4

4.1 Viết chương trình trong bài tập 1.1 và 3.1 sử dụng hàm.

4.2 Chúng ta có định nghĩa của hàm Swap sau

```
void Swap (int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

cho biết giá trị của x và y sau khi gọi hàm:

```
x = 10;
y = 20;
Swap(x, y);
```

4.3 Chương trình sau xuất ra kết quả gì khi được thực thi?

```
#include <iostream.h>
char *str = "global";

void Print (char *str)
{
    cout << str << '\n';
    {
        char *str = "local";
        cout << str << '\n';
        cout << ::str << '\n';
    }
    cout << str << '\n';
}

int main (void)
{
    Print("Parameter");
    return 0;
}
```

4.4 Viết hàm xuất ra tất cả các số nguyên tố từ 2 đến n (n là số nguyên dương):

```
void Primes (unsigned int n);
```

Một số là số nguyên tố nếu như nó chỉ chia hết cho chính nó và 1.

4.5 Định nghĩa một bảng liệt kê gọi là Month cho tất cả các tháng trong năm và sử dụng nó để định nghĩa một hàm nhận một tháng như là một đối số và trả về nó như là một hằng chuỗi.

4.6 Định nghĩa một hàm inline IsAlpha, hàm trả về khác 0 khi tham số của nó là một ký tự và trả về 0 trong các trường hợp khác.

4.7 Định nghĩa một phiên bản đệ qui của hàm Power đã được trình bày trong chương này.

4.8 Viết một hàm trả về tổng của một danh sách các giá trị thực

```
double Sum (int n, double val ...);
```

trong đó n biểu thị số lượng các giá trị trong danh sách.

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 toán 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.**

...	26	34	22	17	24	32	19	13	28	38	25	20	...
	hàng đầu				hàng hai				hàng ba				

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

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.

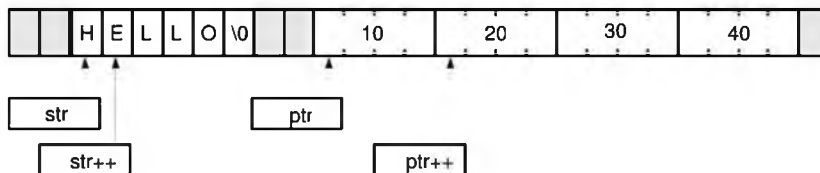
5.5. Tính toán con trỏ

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

- 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

- 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.
- 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

kiểm 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
8      while (bot <= top) {
9          mid = (bot + top) / 2;
10         if ((cmp = Compare(item, table[mid])) == 0)
11             return mid;           // trả về chỉ số hàng mục
12         else if (cmp < 0)
13             top = mid - 1;         // giới hạn tìm kiếm tới nửa thấp hơn
14         else
15             bot = mid + 1;         // giới hạn tìm kiếm tới nửa cao hơn
16     }
17     return -1;                   // không tìm thấy

```

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 nửa. 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 số 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 nửa thấp hơn của mảng.
- 14 Nếu item lớn hơn hạng mục giữa thì sau đó tìm kiếm được giới hạn tới nửa 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
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
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;  
unsigned long *values[10][20];
```

Chương 6. Lập trình hướng đối tượng

Chương này giới thiệu những khái niệm cơ bản trong lập trình hướng đối tượng. Các khái niệm cơ bản như lớp, đối tượng, thuộc tính, phương thức, thông điệp, và quan hệ của chúng sẽ được thảo luận trong phần này. Thêm vào đó là sự trình bày của những đặc điểm quan trọng trong lập trình hướng đối tượng như tính bao gói, tính thừa kế, tính đa hình,.. nhằm giúp người học có cái nhìn tổng quát về lập trình hướng đối tượng.

6.1. Giới thiệu

Hướng đối tượng (object orientation) cung cấp một kiểu mới để xây dựng phần mềm. Trong kiểu mới này, các đối tượng (object) và các lớp (class) là những khối xây dựng trong khi các phương thức (method), thông điệp (message), và sự thừa kế (inheritance) cung cấp các cơ chế chủ yếu.

Lập trình hướng đối tượng (OOP- Object-Oriented Programming) là một cách tư duy mới, tiếp cận hướng đối tượng để giải quyết vấn đề bằng máy tính. Thuật ngữ OOP ngày càng trở nên thông dụng trong lĩnh vực công nghệ thông tin.

Khái niệm 6.1

Lập trình hướng đối tượng (OOP) là một phương pháp thiết kế và phát triển phần mềm dựa trên kiến trúc lớp và đối tượng.

Nếu bạn chưa bao giờ sử dụng một ngôn ngữ OOP thì trước tiên bạn nên nắm vững các khái niệm của OOP hơn là viết các chương trình. Bạn cần hiểu được đối tượng (object) là gì, lớp (class) là gì, chúng có quan hệ với nhau như thế nào, và làm thế nào để các đối tượng trao đổi thông điệp (message) với nhau, vâng vâng.

OOP là tập hợp các kỹ thuật quan trọng mà có thể dùng để làm cho việc triển khai chương trình hiệu quả hơn. Quá trình tiến hóa của OOP như sau:

- Lập trình tuyến tính
- Lập trình có cấu trúc
- Sự trừu tượng hóa dữ liệu
- Lập trình hướng đối tượng

6.2. Trừu tượng hóa (Abstraction)

Trừu tượng hóa là một kỹ thuật chỉ trình bày những các đặc điểm cần thiết của vấn đề mà không trình bày những chi tiết cụ thể hay những lời giải thích phức tạp của vấn đề đó. Hay nói khác hơn nó là một kỹ thuật tập trung vào thứ cần thiết và phớt lờ đi những thứ không cần thiết.

Ví dụ những thông tin sau đây là các đặc tính gắn kết với con người:

- Tên
- Tuổi
- Địa chỉ
- Chiều cao
- Màu tóc

Giả sử ta cần phát triển ứng dụng khách hàng mua sắm hàng hóa thì những chi tiết thiết yếu là tên, địa chỉ còn những chi tiết khác (tuổi, chiều cao, màu tóc, ..) là không quan trọng đối với ứng dụng. Tuy nhiên, nếu chúng ta phát triển một ứng dụng hỗ trợ cho việc điều tra tội phạm thì những thông tin như chiều cao và màu tóc là thiết yếu.

Sự trừu tượng hóa đã không ngừng phát triển trong các ngôn ngữ lập trình, nhưng chỉ ở mức dữ liệu và thủ tục. Trong OOP, việc này được nâng lên ở mức cao hơn – mức đối tượng. Sự trừu tượng hóa được phân thành sự trừu tượng hóa dữ liệu và trừu tượng hóa chương trình.

Khái niệm 6.2

Trừu tượng hóa dữ liệu (data abstraction) là tiến trình xác định và nhóm các thuộc tính và các hành động liên quan đến một thực thể đặc thù trong ứng dụng đang phát triển.

Trừu tượng hóa chương trình (program abstraction) là một sự trừu tượng hóa dữ liệu mà làm cho các dịch vụ thay đổi theo dữ liệu.

6.3. Đối tượng (object)

Các đối tượng là chìa khóa để hiểu được kỹ thuật hướng đối tượng. Bạn có thể nhìn xung quanh và thấy được nhiều đối tượng trong thế giới thực như: con chó, cái bàn, quyển vở, cây viết, tivi, xe hơi ... Trong một hệ thống hướng đối tượng, mọi thứ đều là đối tượng. Một bảng tính, một ô trong bảng tính, một biểu đồ, một bảng báo cáo, một con số hay một số điện thoại, một tập tin, một thư mục, một máy in, một câu hoặc một từ, thậm chí một ký tự, tất cả chúng là những ví dụ của một đối tượng. Rõ ràng chúng ta viết một chương trình hướng đối tượng cũng có nghĩa là chúng ta đang xây dựng một mô hình

của một vài bộ phận trong thế giới thực. Tuy nhiên các đối tượng này có thể được biểu diễn hay mô hình trên máy tính.

Một đối tượng thế giới thực là một thực thể cụ thể mà thông thường bạn có thể sờ, nhìn thấy hay cảm nhận được. Tất cả các đối tượng trong thế giới thực đều có **trạng thái** (state) và **hành động** (behaviour). Ví dụ:

	Trạng thái	Hành động
Con chó	<ul style="list-style-type: none">TênMàuGiốngVui sướng	<ul style="list-style-type: none">SủaVẫy taiChạyĂn
Xe đạp	<ul style="list-style-type: none">Bánh răngBàn đạpDây xíchBánh xe	<ul style="list-style-type: none">Tăng tốcGiảm tốcChuyển bánh răng

Các **đối tượng phần mềm** (software object) có thể được dùng để biểu diễn các đối tượng thế giới thực. Chúng được mô hình sau khi các đối tượng thế giới thực có cả trạng thái và hành động. Giống như các đối tượng thế giới thực, các đối tượng phần mềm cũng có thể có trạng thái và hành động. Một đối tượng phần mềm có biến (variable) hay trạng thái (state) mà thường được gọi là **thuộc tính** (attribute; property) để duy trì trạng thái của nó và **phương thức** (method) để thực hiện các hành động của nó. Thuộc tính là một hạng mục dữ liệu được đặt tên bởi một định danh (identifier) trong khi phương thức là một chức năng được kết hợp với đối tượng chứa nó.

OOP thường sử dụng hai thuật ngữ mà sau này Java cũng sử dụng là thuộc tính (attribute) và phương thức (method) để đặc tả tương ứng cho trạng thái (state) hay biến (variable) và hành động (behavior). Tuy nhiên C++ lại sử dụng hai thuật ngữ **dữ liệu thành viên** (member data) và **hàm thành viên** (member function) thay cho các thuật ngữ này.

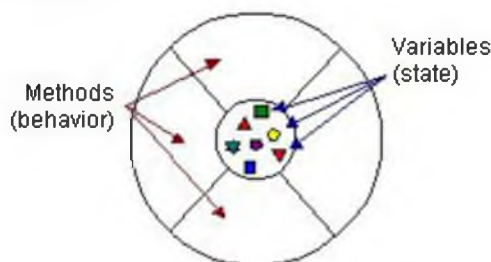
Xét một cách đặc biệt, chỉ một đối tượng riêng rẽ thì chính nó không hữu dụng. Một chương trình hướng đối tượng thường gồm có hai hay nhiều hơn các đối tượng phần mềm tương tác lẫn nhau như là sự tương tác của các đối tượng trong thế giới thực.

Khái niệm 6.3

Đối tượng (object) là một thực thể phần mềm bao bọc các thuộc tính và các phương thức liên quan.

Kể từ đây, trong giáo trình này chúng ta sử dụng thuật ngữ **đối tượng** (object) để chỉ một đối tượng phần mềm. Hình 6.1 là một minh họa của một đối tượng phần mềm:

Hình 6.1 Một đối tượng phần mềm



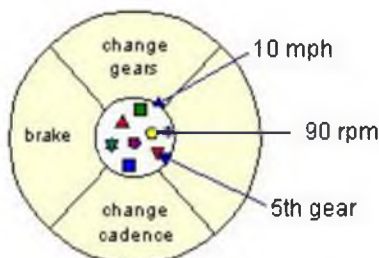
Mọi thứ mà đối tượng phần mềm biết (trạng thái) và có thể làm (hành động) được thể hiện qua các thuộc tính và các phương thức. Một đối tượng phần mềm mô phỏng cho chiếc xe đạp sẽ có các thuộc tính để xác định các trạng thái của chiếc xe đạp như: tốc độ của nó là 10 km trên giờ, nhịp bàn đạp là 90 vòng trên phút, và bánh răng hiện tại là bánh răng thứ 5. Các thuộc tính này thông thường được xem như **thuộc tính thể hiện** (instance attribute) bởi vì chúng chứa đựng các trạng thái cho một đối tượng xe đạp cụ thể. Trong kỹ thuật hướng đối tượng thì một đối tượng cụ thể được gọi là một **thể hiện** (instance).

Khái niệm 6.4

Một đối tượng cụ thể được gọi là một **thể hiện** (instance).

Hình 6.2 minh họa một xe đạp được mô hình như một đối tượng phần mềm:

Hình 6.2 Đối tượng phần mềm xe đạp



Đối tượng xe đạp phần mềm cũng có các phương thức để thắng lại, tăng nhịp đạp hay là chuyển đổi bánh răng. Nó không có phương thức để thay đổi tốc độ vì tốc độ của xe đạp có thể tính ra từ hai yếu tố số vòng quay và bánh răng hiện tại. Những phương thức này thông thường được biết như là các **phương thức thể hiện** (instance method) bởi vì chúng tác động hay thay đổi trạng thái của một đối tượng cụ thể.

6.4. Lớp (Class)

Trong thế giới thực thông thường có nhiều loại đối tượng cùng loại. Chẳng hạn chiếc xe đạp của bạn chỉ là một trong hàng tỉ chiếc xe đạp trên thế giới. Tương tự, trong một chương trình hướng đối tượng có thể có nhiều đối tượng cùng loại và chia sẻ những đặc điểm chung. Sử dụng thuật ngữ hướng đối tượng, chúng ta có thể nói rằng chiếc xe đạp của bạn là một thể hiện của lớp xe đạp. Các xe đạp có một vài trạng thái chung (bánh răng hiện tại, số vòng quay hiện tại, hai bánh xe) và các hành động (chuyển bánh răng, giảm tốc). Tuy nhiên, trạng thái của mỗi xe đạp là độc lập và có thể khác với các trạng thái của các xe đạp khác. Trước khi tạo ra các xe đạp, các nhà sản xuất thường thiết lập một bản thiết kế (blueprint) mô tả các đặc điểm và các yếu tố cơ bản của xe đạp. Sau đó hàng loạt xe đạp sẽ được tạo ra từ bản thiết kế này. Không hiệu quả nếu như tạo ra một bản thiết kế mới cho mỗi xe đạp được sản xuất.

Trong phần mềm hướng đối tượng cũng có thể có nhiều đối tượng cùng loại chia sẻ những đặc điểm chung như là: các hình chữ nhật, các mẫu tin nhân viên, các đoạn phim, ... Giống như là các nhà sản xuất xe đạp, bạn có thể tạo ra một bản thiết kế cho các đối tượng này. Một bản thiết kế phần mềm cho các đối tượng được gọi là **lớp** (class).

Khái niệm 6.5

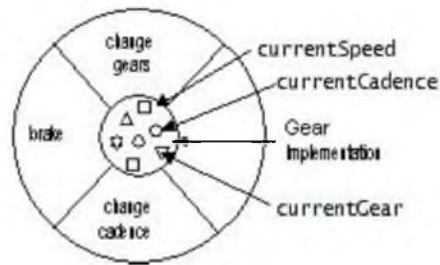
Lớp (class) là một thiết kế (blueprint) hay một mẫu ban đầu (prototype) định nghĩa các thuộc tính và các phương thức chung cho tất cả các đối tượng của cùng một loại nào đó.

Một đối tượng là một thể hiện cụ thể của một lớp.

Trở lại ví dụ về xe đạp chúng ta thấy rằng một lớp Xedap là một bản thiết kế cho hàng loạt các đối tượng xe đạp được tạo ra. Mỗi đối tượng xe đạp là một thể hiện của lớp Xedap và trạng thái của nó có thể khác với các đối tượng xe đạp khác. Ví dụ một xe đạp hiện tại có thể là ở bánh răng thứ 5 trong khi một chiếc khác có thể là ở bánh răng thứ 3.

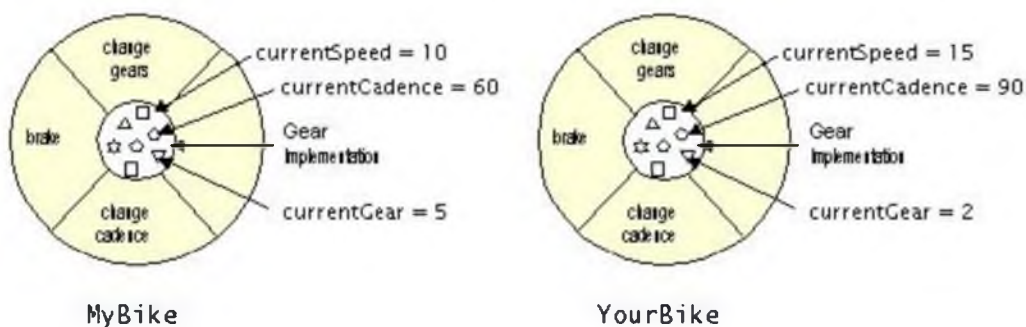
Lớp Xedap sẽ khai báo các thuộc tính thể hiện cần thiết để chứa đựng bánh răng hiện tại, số vòng quay hiện tại, .. cho mỗi đối tượng xe đạp. Lớp Xedap cũng khai báo và cung cấp những thi công cho các phương thức thể hiện để cho phép người đi xe đạp chuyển đổi bánh răng, phanh lại, chuyển đổi số vòng quay, .. như Hình 6.3.

Hình 6.3 Khai báo cho lớp Xedap



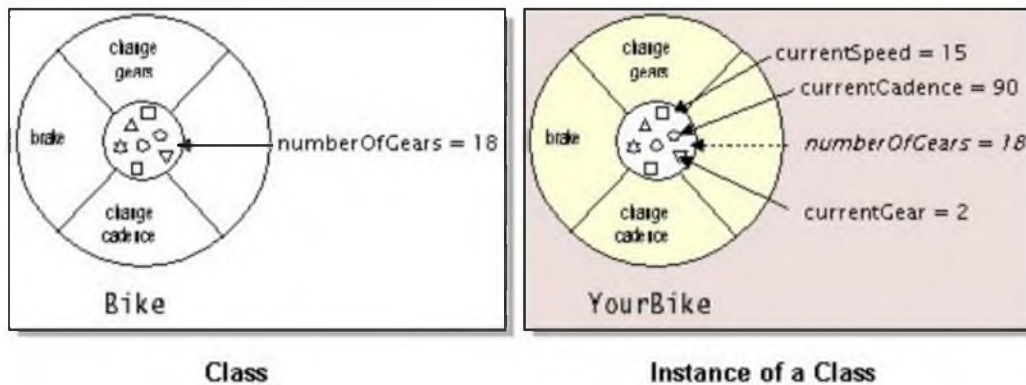
Sau khi bạn đã tạo ra lớp xe đạp, bạn có thể tạo ra bất kỳ đối tượng xe đạp nào từ lớp này. Khi bạn tạo ra một thể hiện của lớp, hệ thống cấp phát đủ bộ nhớ cho đối tượng và tất cả các thuộc tính thể hiện của nó. Mỗi thể hiện sẽ có vùng nhớ riêng cho các thuộc tính thể hiện của nó. Hình 6.4 minh họa hai đối tượng xe đạp khác nhau được tạo ra từ cùng lớp Xedap:

Hình 6.4 Hai đối tượng của lớp Xedap



Ngoài các thuộc tính thể hiện, các lớp có thể định nghĩa các **thuộc tính lớp** (class attribute). Một thuộc tính lớp chứa đựng các thông tin mà được chia sẻ bởi tất cả các thể hiện của lớp. Ví dụ, tất cả xe đạp có cùng số lượng bánh răng. Trong trường hợp này, định nghĩa một thuộc tính thể hiện để giữ số lượng bánh răng là không hiệu quả bởi vì tất cả các vùng nhớ của các thuộc tính thể hiện này đều giữ cùng một giá trị. Trong những trường hợp như thế bạn có thể định nghĩa một thuộc tính lớp để chứa đựng số lượng bánh răng của xe đạp. Tất cả các thể hiện của lớp Xedap sẽ chia thuộc tính này. Một lớp cũng có thể khai báo các **phương thức lớp** (class methods). Bạn có thể triệu gọi một phương thức lớp trực tiếp từ lớp nhưng ngược lại bạn phải triệu gọi các phương thức thể hiện từ một thể hiện cụ thể nào đó.

Hình 6.5 Lớp và thể hiện của lớp



Khái niệm 6.6

Thuộc tính lớp (class attribute) là một hạng mục dữ liệu liên kết với một lớp cụ thể mà không liên kết với các thể hiện của lớp. Nó được định nghĩa bên trong định nghĩa lớp và được chia sẻ bởi tất cả các thể hiện của lớp.

Phương thức lớp (class method) là một phương thức được triệu gọi mà không tham khảo tới bất kỳ một đối tượng nào. Tất cả các phương thức lớp ảnh hưởng đến toàn bộ lớp chứ không ảnh hưởng đến một lớp riêng rẽ nào.

6.5. Thuộc tính (Attribute)

Các thuộc tính trình bày trạng thái của đối tượng. Các thuộc tính nắm giữ các giá trị dữ liệu trong một đối tượng, chúng định nghĩa một đối tượng đặc thù.

Khái niệm 6.7

Thuộc tính (attribute) là dữ liệu trình bày các đặc điểm về một đối tượng.

Một thuộc tính có thể được gán một giá trị chỉ sau khi một đối tượng dựa trên lớp ấy được tạo ra. Một khi các thuộc tính được gán giá trị chúng mô tả một đối tượng. Mọi đối tượng của một lớp phải có cùng các thuộc tính nhưng giá trị của các thuộc tính thì có thể khác nhau. Một thuộc tính của đối tượng có thể nhận các giá trị khác nhau tại những thời điểm khác nhau.

6.6. Phương thức (Method)

Các phương thức thực thi các hoạt động của đối tượng. Các phương thức là nhân tố làm thay đổi các thuộc tính của đối tượng.

Khái niệm 6.8

Phương thức (method) có liên quan tới những thứ mà đối tượng có thể làm. Một phương thức đáp ứng một chức năng tác động lên dữ liệu của đối tượng (thuộc tính).

Các phương thức xác định cách thức hoạt động của một đối tượng và được thực thi khi đối tượng cụ thể được tạo ra. Ví dụ, các hoạt động chung của một đối tượng thuộc lớp Chó là sủa, vẫy tai, chạy, và ăn. Tuy nhiên, chỉ khi một đối tượng cụ thể thuộc lớp Chó được tạo ra thì các phương thức sủa, vẫy tai, chạy, và ăn mới được thực thi.

Các phương thức mang lại một cách nhìn khác về đối tượng. Khi bạn nhìn vào đối tượng Cửa ra vào vào bên trong môi trường của bạn (môi trường thế giới thực), một cách đơn giản bạn có thể thấy nó là một đối tượng bất động không có khả năng suy nghĩ. Trong tiếp cận hướng đối tượng cho phát triển hệ thống, Cửa ra vào có thể được liên kết tới phương thức được giả sử là có thể được thực hiện. Ví dụ, Cửa ra vào có thể mở, nó có thể đóng, nó có thể khóa, hoặc nó có thể mở khóa. Tất cả các phương thức này gắn kết với đối tượng Cửa ra vào và được thực hiện bởi Cửa ra vào chứ không phải một đối tượng nào khác.

6.7. Thông điệp (Message)

Một chương trình hay ứng dụng lớn thường chứa nhiều đối tượng khác nhau. Các đối tượng phần mềm tương tác và giao tiếp với nhau bằng cách gửi các **thông điệp** (message). Khi đối tượng A muốn đối tượng B thực hiện các phương thức của đối tượng B thì đối tượng A gửi một thông điệp tới đối tượng B.

Ví dụ đối tượng người đi xe đạp muốn đối tượng xe đạp thực hiện phương thức chuyển đổi bánh răng của nó thì đối tượng người đi xe đạp cần phải gửi một thông điệp tới đối tượng xe đạp.

Đôi khi đối tượng nhận cần thông tin nhiều hơn để biết chính xác thực hiện công việc gì. Ví dụ khi bạn chuyển bánh răng trên chiếc xe đạp của bạn thì bạn phải chỉ rõ bánh răng nào mà bạn muốn chuyển. Các thông tin này được truyền kèm theo thông điệp và được gọi là các **tham số** (parameter).

Một thông điệp gồm có:

- Đối tượng nhận thông điệp
- Tên của phương thức thực hiện
- Các tham số mà phương thức cần

Khái niệm 6.9

Một **thông điệp** (message) là một lời yêu cầu một hoạt động.

Một thông điệp được truyền khi một đối tượng triệu gọi một hay nhiều phương thức của đối tượng khác để yêu cầu thông tin.

Khi một đối tượng nhận được một thông điệp, nó thực hiện một phương thức tương ứng. Ví dụ đối tượng xe đạp nhận được thông điệp là chuyển đổi bánh răng nó sẽ thực hiện việc tìm kiếm phương thức chuyển đổi bánh răng tương ứng và thực hiện theo yêu cầu của thông điệp mà nó nhận được.

6.8. Tính bao gói (Encapsulation)

Trong đối tượng xe đạp, giá trị của các thuộc tính được chuyển đổi bởi các phương thức. Phương thức `changeGear()` chuyển đổi giá trị của thuộc tính `currentGear`. Thuộc tính `speed` được chuyển đổi bởi phương thức `changeGear()` hoặc `changRpm()`.

Trong OOP thì các thuộc tính là trung tâm, là hạt nhân của đối tượng. Các phương thức bao quanh và che giấu đi hạt nhân của đối tượng từ các đối tượng khác trong chương trình. Việc bao gói các thuộc tính của một đối tượng bên trong sự che chở của các phương thức của nó được gọi là sự **đóng gói** (encapsulation) hay là đóng gói dữ liệu.

Đặc tính đóng gói dữ liệu là ý tưởng của các nhà thiết kế các hệ thống hướng đối tượng. Tuy nhiên, việc áp dụng trong thực tế thì có thể không hoàn toàn như thế. Vì những lý do thực tế mà các đối tượng đôi khi cần phải phơi bày ra một vài thuộc tính này và che giấu đi một vài phương thức kia. Tùy thuộc vào các ngôn ngữ lập trình hướng đối tượng khác nhau, chúng ta có các điều khiển các truy xuất dữ liệu khác nhau.

Khái niệm 6.10

Đóng gói (encapsulation) là tiến trình che giấu việc thực thi chi tiết của một đối tượng.

Một đối tượng có một giao diện chung cho các đối tượng khác sử dụng để giao tiếp với nó. Do đặc tính đóng gói mà các chi tiết như: các trạng thái

được lưu trữ như thế nào hay các hành động được thi công ra sao có thể được che giấu đi từ các đối tượng khác. Điều này có nghĩa là các chi tiết riêng của đối tượng có thể được chuyển đổi mà hoàn toàn không ảnh hưởng tới các đối tượng khác có liên hệ với nó. Ví dụ, một người đi xe đạp không cần biết chính xác cơ chế chuyển bánh răng trên xe đạp thực sự làm việc như thế nào nhưng vẫn có thể sử dụng nó. Điều này được gọi là che giấu thông tin.

Khái niệm 6.11

Che giấu thông tin (information hiding) là việc ẩn đi các chi tiết của thiết kế hay thi công từ các đối tượng khác.

6.9. Tính thừa kế (Inheritance)

Hệ thống hướng đối tượng cho phép các lớp được định nghĩa kế thừa từ các lớp khác. Ví dụ, lớp xe đạp leo núi và xe đạp đua là những lớp con (subclass) của lớp xe đạp. Như vậy ta có thể nói lớp xe đạp là lớp cha (superclass) của lớp xe đạp leo núi và xe đạp đua.

Khái niệm 6.12

Thừa kế (inheritance) nghĩa là các hành động (phương thức) và các thuộc tính được định nghĩa trong một lớp có thể được thừa kế hoặc được sử dụng lại bởi lớp khác.

Khái niệm 6.13

Lớp cha (superclass) là lớp có các thuộc tính hay hành động được thừa hưởng bởi một hay nhiều lớp khác.

Lớp con (subclass) là lớp thừa hưởng một vài đặc tính chung của lớp cha và thêm vào những đặc tính riêng khác.

Các lớp con thừa kế thuộc tính và hành động từ lớp cha của chúng. Ví dụ, một xe đạp leo núi không những có bánh răng, số vòng quay trên phút và tốc độ giống như mọi xe đạp khác mà còn có thêm một vài loại bánh răng vì thế mà nó cần thêm một thuộc tính là gearRange (loại bánh răng).

Các lớp con có thể định nghĩa lại các phương thức được thừa kế để cung cấp các thi công riêng biệt cho các phương thức này. Ví dụ, một xe đạp leo núi sẽ cần một phương thức đặc biệt để chuyển đổi bánh răng.

Các lớp con cung cấp các phiên bản đặc biệt của các lớp cha mà không cần phải định nghĩa lại các lớp mới hoàn toàn. Ở đây, mã lớp cha có thể được sử dụng lại nhiều lần.

6.10. Tính đa hình (Polymorphism)

Một khái niệm quan trọng khác có liên quan mật thiết với truyền thông điệp là **đa hình** (polymorphism). Với đa hình, nếu cùng một hành động (phương thức) ứng dụng cho các đối tượng thuộc các lớp khác nhau thì có thể đưa đến những kết quả khác nhau.

Khái niệm 6.14

Đa hình (polymorphism) nghĩa là “nhiều hình thức”, hành động cùng tên có thể được thực hiện khác nhau đối với các đối tượng/các lớp khác nhau.

Chúng ta hãy xem xét các đối tượng Cửa Sổ và Cửa Cái. Cả hai đối tượng có một hành động chung có thể thực hiện là đóng. Nhưng một đối tượng Cửa Cái thực hiện hành động đó có thể khác với cách mà một đối tượng Cửa Sổ thực hiện hành động đó. Cửa Cái khép cánh cửa lại trong khi Cửa Sổ hạ các thanh cửa xuống. Thật vậy, hành động đóng có thể thực hiện một trong hai hình thức khác nhau. Một ví dụ khác là hành động hiển thị. Tùy thuộc vào đối tượng tác động, hành động ấy có thể hiển thị một chuỗi, hoặc vẽ một đường thẳng, hoặc là hiển thị một hình.

Đa hình có sự liên quan tới việc truyền thông điệp. Đối tượng yêu cầu cần biết hành động nào để yêu cầu và yêu cầu từ đối tượng nào. Tuy nhiên đối tượng yêu cầu không cần lo lắng về một hành động được hoàn thành như thế nào.

Bài tập cuối chương 6

- 6.1 Trình bày các định nghĩa của các thuật ngữ:
- Lập trình hướng đối tượng
 - Trừu tượng hóa
 - Đối tượng
 - Lớp
 - Thuộc tính
 - Phương thức
 - Thông điệp

- 6.2 Phân biệt sự khác nhau giữa lớp và đối tượng, giữa thuộc tính và giá trị, giữa thông điệp và truyền thông điệp.
- 6.3 Trình bày các đặc điểm của OOP.
- 6.4 Những lợi ích có được thông qua thừa kế và bao gói.
- 6.5 Những thuộc tính và phương thức cơ bản của một cái máy giặt.
- 6.6 Những thuộc tính và phương thức cơ bản của một chiếc xe hơi.
- 6.7 Những thuộc tính và phương thức cơ bản của một hình tròn.
- 6.8 Chỉ ra các đối tượng trong hệ thống rút tiền tự động ATM.
- 6.9 Chỉ ra các lớp có thể kế thừa từ lớp điện thoại, xe hơi, và động vật.