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 vài 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 * tarrif1;
        else
                charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
} 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 hằng 1:
        các lệnh;
    ...
    case hằng n:
        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 thức$_1$ (thường được gọi là biểu thức khởi tạo) được ước lượng trước tiên. Mỗi vòng lặp *biểu thức*$_2$ được ước lượng. Nếu kết quả không là 0 (đúng) thì sau đó *lệnh* được thực thi và *biểu thức*$_3$ 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 thức 2) {
    lệnh;
    biểu thức 3;
}
```

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;           //                       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 các 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 gãy 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:

giaithua (*0*) = 1
giaithua (*n*) = *n* × *giaithua* (*n*-1)

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
```