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.

Phần 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.

Phần 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      for (int i = 0; i < exponent; ++i)
5          result *= base;
6      return result;
7  }
```

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 va exponent, và sau đó thân hàm được ước lượng.

Danh sách 4.2

```
1  #include <iostream.h>

2  main (void)
3  {
4      cout << "2 ^ 8 = " << Power(2,8) << '\n';
5  }
```

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 bao ham
3  main (void)
4  {
5        cout << "2 ^ 8 = " << Power(2,8) << '\n';
6  }
7  int Power (int base, unsigned int exponent)
8  {
9        int    result = 1;
10       for (int i = 0; i < exponent; ++i)
11             result *= base;
12       return result;
13 }
```

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 cí 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 xuất 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  int main (int argc, const char *argv[])
4  {
5        double sum = 0;
6        for (int i = 1; i < argc; ++i)
7        sum += atof(argv[i]);
8        cout << sum << '\n';
9        return 0;
10 }
```

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.