# CHAPTER 1: HƯỚNG DẪN LÀM QUEN

Chúng ta hãy bắt đầu làm làm quen nhanh với ngôn ngữ C nhé. Mục tiêu của chúng ta là thấy được bản chất các yếu tố của ngôn ngữ này trong những chương trình thực tế nhưng không đi sâu vào những chi tiết, các quy tắc và những ngoại lệ. Ở thời điểm này, chúng ta sẽ không cố hoàn thiện nó hoặc thậm chí chính xác tuyệt đối. Chúng tôi muốn đưa bạn đến điểm mà bạn có thể tự viết được các chương trình có ích một cách nhanh nhất, và để làm được điều đó chúng ta cần phải tập trung vào những cơ bản: biến và hằng, toán tử, kĩ thuật điều khiển luồng, hàm, và những nguyên tắc cơ bản vào/ra (INPUT/OUTPUT). Sau đó sẽ là những phần quan trọng hơn trong ngôn ngữ C để từ đó có thể viết được các chương trình lớn hơn, thực tế hơn. Nó bao gồm: con trỏ, cấu trúc, tập hợp phong phú nhất về các toán tử của ngôn ngữ C, một vài câu lệnh điều khiển luồng và thư viện chuẩn.

Cách tiếp cận sau đây có một vài hạn chế. Đáng chú ý nhất là sẽ không có sự tường thuật chi tiết nào về các đặc điểm của ngôn ngữ này ở đây, và các hướng dẫn, nói một cách ngắn gọn, có thể sẽ gây hiểu sai. Và bởi vì các ví dụ này không sử dụng hết sức mạnh của ngôn ngữ C, chúng cũng có thể không được tối ưu và khá là "hack não" cho người mới. Chúng tôi đã cố gắng giảm thiểu những điểm này, nhưng hãy cẩn trọng cố gắng. Một hạn chế nữa là các chapter sau đó sẽ có thể lặp lại một số điểm ở chapter này khi cần thiết. Chúng tôi mong rằng sự lặp lại này sẽ giúp cho bạn thông hơn là hack não lần nữa.

Trong bất kì trường hợp nào, các lập trình viên có kinh nghiệm có thể ngoại suy từ các đặc điểm ở chapter này để trau dồi cho những điều cần thiết trong kĩ năng lập trình của họ. Những người mới nên cố gắng viết thêm các chương trình nhỏ tương tự cho mình. Cả hai nhóm đều có thể thông qua chapter này để làm sườn cho những mô tả chi tiết bắt đầu từ chapter 2.

## 1.1 Mở đầu

Cách duy nhất để học một ngôn ngữ lập trình mới là phải viết các chương trình bằng ngôn ngữ đó. Chương trình đầu tiên sau đây đều tương tự với các ngôn ngữ lập trình khác là:

*In ra màn hình dòng*

```
Hello, world
```

Đây là một rào cản lớn cho những ai mới học ngôn ngữ lập trình nói chung; để có thể vượt qua được rào cản này thì bạn phải có nơi để có thể soạn thảo chương trình chữ, biên dịch nó thành công, tải, chạy nó và tìm hiểu xem đầu ra (OUTPUT) của bạn ở đâu. Nếu đã thành thạo những chi tiết này thì công việc sẽ rất là dễ dàng.

Ở trong ngôn ngữ C, đoạn chương trình để in dòng "`Hello, world`" là

```
#include <stdio.h>

main()

{
```

```
        printf("Hello, wolrd\n");
}
```

Còn bây giờ là việc chạy chương trình, nó phụ thuộc vào hệ điều hành bạn đang sử dụng. Ví dụ nếu bạn sử dụng hệ điều hành UNIX thì bạn phải soạn đoạn chương trình trên vào một tệp có kết thúc đuôi là **`.c`**, như **`hello.c`**, và rồi biên dịch nó với câu lệnh

```
cc hello.c
```

Nếu không có lỗi gì xảy ra, như thiếu dấu chấm phẩy ở cuối câu lệnh printf hay là thiếu các dấu ngoặc, thì công việc biên dịch sẽ chạy ẩn và tạo ra một tệp thực thi tên là **`a.out`**. Nếu bạn chạy **`a.out`** bằng cách viết dòng lệnh

```
a.out
```

thì nó sẽ in ra màn hình

```
Hello, world
```

Đối với các hệ điều hành khác, các quy tắc sẽ khác nhau. Nhưng hiện tại đa phần chúng ta sử dụng hệ điều hành windows, nên việc soạn chương trình và biên dịch sẽ được thực hiện trong các phần mềm thông dụng như Borland C, C-free, Visual Studio, Code:Block,… Việc này sẽ để các bạn tự tìm hiểu, nhưng mọi quy tắc trong ngôn ngữ C đều như nhau.

Bây giờ sẽ là một số lời giải thích về chương trình trên. Một chương trình viết theo ngôn ngữ C, dù bất cứ dung lượng lớn hay nhỏ, đều bao gồm các hàm và các biến. Một hàm chứa các câu lệnh để ra lệnh cho máy tính thực hiện, và chứa các biến dùng để lưu trữ giá trị trong việc tính toán. Các hàm trong C giống như các chương trình con của ngôn ngữ FORTRAN, cũng như trong PASCAL. Ví dụ như `main` là một hàm. Bình thường thì bạn có thể tự ý đặt cho hàm một cái tên bất kì, nhưng hàm `main` là một hàm đặc biệt, chương trình của bạn được thực thi bắt đầu từ hàm này. Điều này có nghĩa là mỗi chương trình muốn chạy được đều phải có hàm `main` ở đâu đó trong chương trình.

Hàm `main` sẽ gọi các hàm khác để thực thi công việc của chương trình, những hàm mà bạn tự viết hoặc là các hàm được viết sẵn trong các thư viện để cung cấp cho bạn. Dòng đầu tiên của chương trình trên,

```
#include <stdio.h>
```

nói với trình biên dịch là chương trình này có sử dụng đến thư viện vào/ra chuẩn (standard INPUT/ OUTPUT); dòng này xuất hiện hầu hết ở đầu các chương trình C. Thư viện này được diễn tả chi tiết ở **chapter 7** và phần **phụ lục B**.

Cách thức truyền dữ liệu giữa các hàm là thông qua việc gọi hàm mà hàm đó có cung cấp danh sách các giá trị, nó được gọi là các *đối*, tới hàm mà nó được gọi tới. Cặp dấu ngoặc đơn sau tên hàm dùng để bao danh sách các đối của hàm. Trong chương trình này,

```
#include <stdio.h>                    Bao gồm thông tin về thư viện vào/ra chuẩn
main()                                Khai báo hàm tên main không có đối
```

```
{                                          Các câu lệnh được viết trong cặp ngoặc nhọn
      printf("Hello, wolrd\n");            Gọi hàm printf có sẵn trong thư viện stdio.h để
                                                 in ra chuỗi kí tự trong cặp dấu " "
                                                 \n là kí tự chuyển xuống dòng mới
}
```

Hàm `main` được định nghĩa không có đối, có nghĩa là bên trong ngoặc đơn sau tên main không có gì, ta ghi `()`.

Các câu lệnh của hàm `main` cũng như các hàm khác nói chung được bao bởi cặp dấu ngoặc nhọn `{}`. Hàm `main` trên chỉ có một câu lệnh,

```
printf("Hello, world\n");
```

Để gọi một hàm ta chỉ cần viết tên hàm đó ra, miễn là nó đã được định nghĩa, tiếp đó là cặp dấu ngoặc đơn, bên trong là danh sách các đối truyền vào, với câu lệnh trên ta đã gọi hàm `printf` với đối là "`Hello, world\n`". `printf` là một hàm thư viện chuẩn có chức năng in ra màn hình dữ liệu có trong cặp dấu nháy kép, ở trường hợp này đó là một xâu kí tự.

Một dãy liên tiếp các kí tự ở trong cặp dấu nháy kép, như "`Hello, world\n`", được gọi là một xâu kí tự hay chính xác hơn là hằng xâu kí tự. Từ thời điểm này việc sử dụng các xâu kí tự thường sẽ là đối của hàm `printf` hay các hàm khác.

Cặp kí tự `\n` trong "`Hello, world\n`" trong ngôn ngữ C là một kí hiệu có chức năng chuyển xuống dòng mới, có nghĩa là sau khi in "`Hello, world`" thì con trỏ sẽ chuyển xuống dòng tiếp theo, thụt về đầu dòng bên trái để tiếp tục in dữ liệu ra (nếu có). Nếu như ta bỏ `\n` đi thì khi đó thấy rằng sẽ chỉ có một dòng duy nhất in xâu kí tự trên và con trỏ ở ngay sau kí tự '`d`', tức không có chuyển sang dòng mới. Bạn phải sử dụng `\n` để có thể xuống dòng tiếp theo nếu muốn trong đối của hàm `printf`, nếu bạn thử

```
printf("Hello, world
");
```

khi đó trình biên dịch sẽ báo lỗi.

Hàm `printf` không hỗ trợ tự động xuống dòng mới, vì vậy ta có thể ứng dụng điều này để xây dựng dữ liệu được in ra theo từng giai đoạn. Cụ thể sẽ được thể hiện thông qua chương trình sau, chương trình này cũng in ra xâu kí tự "`Hello, world`" giống hệt như chương trình trên

```
#include <stdio.h>
main()
{
      printf("Hello, ");
      printf("world");
```

```
        printf("\n");
    }
```

Chú ý rằng mặc dù `\n`, về hình thức, gồm hai kí tự nhưng trong ngôn ngữ C nó chỉ xuất hiện như một kí tự. Những kiểu kí tự như thế này cung cấp một cơ chế chung và mở rộng cho việc biểu diễn những kí tự ẩn. Một số các kiểu kí tự tương tự mà ngôn ngữ C cung cấp như `\t` để cách ra một tab, `\b` để lùi về một kí tự, `\r` trở về đầu dòng, `\"` in ra dấu nháy kép, `\\` in ra kí tự vạch chéo ngược. Có một danh sách đầy đủ ở **phần 2.3.**

**Bài tập 1-1**. Chạy chương trình "`Hello, world`" trên hệ điều hành của bạn. Thử bỏ lần lượt các phần của chương trình thì bạn sẽ nhận thông báo gì.

**Bài tập 1-2**. Tìm hiểu xem nếu thêm kí tự \c vào trong phần đối của hàm printf thì chương trình sẽ chạy như thế nào.

### 1.2 Biến và biểu thức phép toán

Chương trình tiếp theo sau đây sử dụng công thức $℃ = (\mathbf{5}/\mathbf{9})\,(℉ - \mathbf{32})$ để in ra bảng chuyển đổi từ ℉ sang ℃ như sau:

```
0     -17
20    -6
40    4
60    15
80    26
100   37
120   48
140   60
160   71
180   82
200   93
220   104
240   115
260   126
280   137
300   148
```

Chương trình này vẫn chỉ bao gồm định nghĩa một hàm duy nhất là hàm `main`. Nó dài hơn chương trình "`Hello, world`" ở phần trước, phức tạp hơn. Chương trình giới thiệu một số ý tưởng, bao gồm cả các chú thích, các khai báo, biến, các phép tính toán, vòng lặp và định dạng đầu ra (OUTPUT).

```
#include <stdio.h>
/* In bảng chuyển đổi giữa độ F và độ C
   với f = 0, 20, …, 300. */
main()
{
      int f, c;
      int lower, upper, step;
      lower = 0;                  /* Nhiệt độ thấp nhất */
      upper = 300;                /* Nhiệt độ cao nhất */
      step = 20;                  /* Bước nhảy */
      f = lower;
      while(f <= upper){
            c = 5 * (f - 32) / 9;
            printf("%d\t%d\n", f, c);
            f = f + step;
      }
}
```

Hai dòng

```
/* In bảng chuyển đổi giữa độ F và độ C
   với f = 0, 20, …, 300. */
```

ở chương trình trên là *chú thích*, có chức năng giải thích nhanh, tóm tắt chương trình này sẽ làm công việc gì. Mọi kí tự nằm giữa `/*` và `*/` đều được trình biên dịch bỏ qua; sử dụng những dòng chú thích này sẽ khiến cho chương trình của chúng ta dễ hiểu hơn. Những dòng chú thích này có thể để ở bất cứ chỗ trống nào trong chương trình.

Trong ngôn ngữ C, mọi biến đều phải được khai báo trước khi sử dụng chúng, thường thì nó sẽ ở ngay đầu các hàm, trước các dòng lệnh thực thi. Một *lời khai báo* sẽ cho chúng ta biết đặc tính của biến đó; nó bao gồm tên kiểu và danh sách các biến, như là

```
int f, c;
int lower, upper, step;
```

Tên kiểu `int` ở trên có nghĩa là các biến được liệt kê sau đó có kiểu là các *số nguyên*, trái lại với tên kiểu `float`, tức là kiểu *số thực*, các số này bao gồm một phần là phần nguyên, phần bên kia là phần thập phân. Khoảng giá trị của hai kiểu `int` (số nguyên) và `float` (số thực) phụ thuộc vào máy tính mà bạn đang sử dụng; thường thì đối với `int` 16-bit, tức nó chiếm 2 byte bộ nhớ, có giá trị khoảng từ -32768 tới +32767, điều này cũng giống với `int` 32-bit, tức 4 byte. Một biến kiểu `float` thường có giá trị 32-bit, 4 byte, với sáu chữ số phần thập phân và độ lớn ở khoảng từ $10^{-38}$ tới $10^{+38}$.

Ngoài ra thì ngôn ngữ C cũng cung cấp thêm một vài các kiểu dữ liệu cơ bản khác `int` và `float` như

| | |
|---|---|
| `char` | kiểu kí tự 1 byte |
| `short` | kiểu số nguyên ngắn |
| `long` | kiểu số nguyên lớn |
| `double` | kiểu số thực lớn |

Độ lớn của các kiểu dữ liệu này cũng phụ thuộc vào máy tính của bạn. Từ những kiểu dữ liệu này mà sau đó chúng ta sẽ có thêm các đối tượng dữ liệu khác như *mảng*, *cấu trúc*, *union*, *con trỏ*, hay là *các hàm trả về giá trị*.

Chương trình chuyển đổi nhiệt độ trên có phép toán bắt đầu bằng các *câu lệnh gán,*

```
lower = 0;
upper = 300;
step = 20;
f = lower;
```

có nghĩa gán cho các biến bằng các giá trị khởi đầu. Các câu lệnh riêng lẻ đều kết thúc bằng các dấu chấm phẩy.

Mỗi dòng của bảng chuyển đổi trên đều được tính toán và in ra như nhau, cho nên vì thế ta sẽ sử dụng vòng lặp, có nghĩa là lặp đi lặp lại công việc tính toán rồi in ra màn hình kết quả theo từng dòng; điều này là ứng dụng của vòng lặp `while`

```
while(f <= upper){
      …
}
```

Vòng lặp `while` hoạt động như sau: Điều kiện trong cặp dấu ngoặc đơn được kiểm tra. Nếu như nó đúng ( `f` nhỏ hơn hoặc bằng `upper`), các câu lệnh bên trong vòng lặp (ba câu lệnh nằm trọn trong cặp ngoặc nhọn) được thực thi. Sau khi thực thi hết, nó tiếp tục quay trở lại kiểm tra điều kiện và nếu nó vẫn đúng, nó tiếp tục thực thi lần lượt ba câu lệnh trong đó. Đến khi điều kiện được kiểm tra là sai (`f` lớn hơn `upper`) thì vòng lặp sẽ kết thúc, chương trình tiếp tục thực hiện câu lệnh tiếp theo ngay sau vòng lặp đó. Nhưng sau đó không có câu lệnh nào, khi đó chương trình kết thúc.

Thân của vòng lặp while có thể chỉ có một hay nhiều câu lệnh được bao trong cặp dấu ngoặc nhọn, hoặc nếu như chỉ cần thực hiện một câu lệnh thì có thể sẽ không cần cặp dấu ngoặc nhọn, ví dụ như

```
while( i < j)
      i = 2 + i;
```

Trong cả hai trường hợp, chúng tôi luôn thụt đầu dòng các câu lệnh được điều khiển bởi vòng lặp `while` một tab khoảng trống, như vậy sẽ cho bạn thấy rõ hơn các câu lệnh nào nằm trong vòng lặp. Việc trình bày như vậy sẽ giúp chương trình sáng sủa và logic hơn. Dù rằng các trình biên dịch ngôn ngữ C chẳng hề quan tâm chương trình của bạn trông như thế nào, nhưng sẽ khiến cho người đọc cảm thấy rối và khó chịu nếu không trình bày rõ ràng như vậy. Chúng tôi khuyên bạn nên viết mỗi câu lệnh ở trên một dòng, và sử dụng các khoảng trống (tab/space) để trình bày rõ các câu lệnh thuộc nhóm công việc nào. Phong cách lập trình cũng là một kĩ thuật lập trình tất yếu mà bạn nên có, trong cuốn sách này chúng tôi đã chọn một trong một vài phong cách soạn thảo chương trình phổ biến, hãy tự chọn ra cho mình một cách mà khiến bạn cảm thấy hài lòng nhất.

Hầu hết công việc đều hoàn thành trong thân của vòng lặp `while`. Giá trị ℃ được tính toán và gán cho biến `c` bởi câu lệnh

```
c = 5 * (f - 32) / 9;
```

Lí do không tính `5/9` luôn theo công thức đã cho ở ban đầu mà nhân `5` với `(f - 32)` trước rồi mới chia cho `9` vì ở trong ngôn ngữ C, cũng giống với các ngôn ngữ khác, hai số nguyên chia cho nhau sẽ ra một số nguyên, tức là sẽ chỉ lấy phần nguyên, còn phần thập phân sẽ bị cắt bỏ. Trong khi đó `5` và `9` đều là hai số nguyên, khi lấy `5/9` thì kết quả sẽ là `0` và điều đó có nghĩa giá trị ℃ sẽ luôn có kết quả bằng `0`.

Ở ví dụ này cũng cho ta thấy thêm được cách hoạt động của hàm `printf`. `printf` là một hàm xuất dữ liệu chuẩn, chúng ta sẽ tìm hiểu chi tiết ở **chapter 7**. Đối đầu tiên của nó là một chuỗi kí tự cần in ra màn hình, trong đó có kí hiệu mới là `%`, đây được gọi là *đặc tả* có chức năng biểu thị các đối được thay thế lần lượt sau đó của hàm `printf`, và kiểu mà nó được in ra. Trong trường hợp này, `%d` là đặc tả biểu diễn cho đối là số nguyên, vì thế câu lệnh

```
printf("%d\t%d\n", f, c);
```

sẽ in ra màn hình lần lượt hai giá trị của biến `f` và biến `c` và giữa chúng có khoảng trống tab (`\t`). `\n` cho biết con trỏ sẽ xuống dòng tiếp theo để tiếp tục in theo vòng lặp.

Mỗi cấu trúc đặc tả `%` ở trong đối đầu tiên của hàm `printf` nó được ứng lần lượt với đối thứ hai, thứ ba,...; Nó cần phải hợp lí ứng với kiểu của đối sau đó, nếu không bạn sẽ nhận kết quả sai.

Cần phải hiểu một điều, rằng `printf` không nằm trong cấu trúc của ngôn ngữ C; chẳng hề có nguồn vào/ra nào tự định nghĩa trong ngôn ngữ C cả. `printf` chỉ là một trong các hàm hữu dụng có trong thư viện chuẩn mà nó thường truy cập vào các chương trình theo ngôn ngữ C. Cách hoạt động của hàm `printf` được định nghĩa theo chuẩn của ANSI (Viện Tiêu chuẩn Quốc gia Hoa Kỳ), tuy nhiên, những đặc tính của nó đều giống với các trình biên dịch và thư viện mà nó theo đúng chuẩn được đề ra.

Để tập trung, chúng tôi sẽ không nói nhiều đến nguồn vào/ra cho đến **chapter 7**. Cụ thể hơn, chúng tôi sẽ hoãn lại việc trình bày các định dạng của đầu vào (INPUT). Nếu bạn thực sự cần phải đưa dữ liệu đầu vào là các con số, hãy đọc trước **phần 7.4** về hàm `scanf`. Hàm `scanf` cũng giống như `printf`, chỉ có điều nó đọc đầu vào chứ không phải in đầu ra ra màn hình như `printf`.

Có một vài vấn đề nhỏ với chương trình chuyển đổi nhiệt độ ở trên. Vấn đề đơn giản nhất là dữ liệu đầu ra in lên màn hình chưa được "dễ thương" cho lắm bởi vì các số không được tinh chỉnh đúng. Điều này rất dễ sửa; nếu chúng ta đưa vào mỗi đặc tả `%d` một con số chỉ độ rộng thì các số khi in ra sẽ được đẹp hơn. Để dễ hiểu, thì câu lệnh `printf` sẽ thay như sau

```
printf("%3d %6d\n", f, c);
```

câu lệnh này sẽ in ra lần lượt các giá trị của `f` với độ rộng ứng với ba kí tự, tiếp đó là khoảng trắng và in tiếp giá trị của `c` với độ rộng ứng với sáu kí tự, sau đó chương trình sẽ in ra bảng như sau:

```
  0      -17
 20       -6
 40        4
 60       15
 80       26
100       37
…
```

Vấn đề lớn hơn ở đây là chúng ta đã sử dụng phép toán trên số nguyên, cho nên giá trị của ℃ sau khi tính toán sẽ không dược chính xác; ví dụ 0℉ thực chất là khoảng -17.8℃, chứ không phải -17**.** Để có thể tính toán chính xác hơn, chúng ta phải sự dụng phép toán trên số thực. Điều này sẽ cần phải thay đổi một chút trong chương trình trên. Đây là một phiên bản thứ hai:

```
#include <stdio.h>
/* In bảng chuyển đổi giữa độ F và độ C
   với f = 0, 20, …, 300; phiên bản số thực */
main()
{
      float f, c;
      int lower, upper, step;
      lower = 0;                  /* Nhiệt độ thấp nhất */
      upper = 300;                /* Nhiệt độ cao nhất */
      step = 20;                  /* Bước nhảy */
      f = lower;
      while(f <= upper){
            c = (5.0 / 9.0) * (f - 32.0);
            printf("%3.0f %6.1f\n", f, c);
```

```
            f = f + step;
        }
    }
```

Chương trình này cũng giống với chương trình trước đó, chỉ khác ở chỗ biến `f` và `c` bây giờ là kiểu `float` (số thực), và phép tính cũng giống với công thức tính ở đầu bài hơn. Chúng ta không thể dùng 5/9 ở chương trình cũ vì như thế nó sẽ bị triệt tiêu mất phần thập phân, kết quả sẽ ra 0 là sai. Còn bây giờ đối với kiểu số thực, khi lấy 5.0/9.0 như vậy sẽ ra kết quả là một số thực, có phần thập phân, đúng kết quả hơn, chính xác hơn.

Nếu như một phép toán có hai toán hạng là các số nguyên, thì phép toán sẽ trở thành phép toán giữa các số nguyên, tức nó sẽ cho kết quả là một số nguyên. Nhưng nếu có một toán hạng là một số thực thì nó sẽ chuyển thành phép toán giữa hai số thực (số nguyên kia sẽ chuyển thành số thực), tức kết quả sẽ là môt số thực. Cụ thể hơn với chương trình trên, với phép toán (`f-32)`, thì 32 sẽ chuyển thành số thực, tức là 32.0 rồi thực hiện phép toán với hai số thực. Tuy nhiên việc viết tường minh hằng số thực nên có thêm phần thập phân đằng sau phần nguyên dù rằng phần thập phân chỉ là số 0, như vậy sẽ khiến cho người đọc bớt bị hack não hơn.

Những quy tắc cho điểm này sẽ được trình bày chi tiết hơn ở **chapter 2**. Từ bây giờ, chỉ cần chú ý rằng câu lệnh gán

```
f = lower;
```

và điều kiện kiểm tra

```
while(f <= upper)
```

cũng sẽ hoạt động như thế, số nguyên sẽ chuyển thành số thực và rồi thực hiện phép toán.

Với đặc tả `%3.0f` được viết trong đối của hàm `printf` chỉ ra rằng một số thực (ở đây là biến `f`) sẽ được in ra với độ rộng là ba chữ số với không có phần thập phân (sau dấu chấm động của đặc tả là 0). Tương tự đặc tả `%6.1f` biểu diễn số thực (ở đây là biến `c`) với độ rộng là 6 kí tự cùng với một chữ số ở phần thập phân (sau dấu chấm động của đặc tả là 1). Dữ liệu đầu ra sẽ được in thành bảng sau:

```
  0   -17.8
 20    -6.7
 40     4.4
 …
```

Độ rộng và độ chính xác của đặc tả có thể bỏ qua nếu bạn không cần phải biểu diễn một cách chính xác: `%6f` chỉ ra rằng số thực đó có độ rộng ít nhất sáu kí tự, vì không có độ chính xác nên nó sẽ mặc định in ra 6 chữ số ở phần thập phân; `%.2f` thì biểu diễn số thực với hai chữ số ở phần thập phân, còn độ rộng thì không bắt buộc, nó sẽ in ra đúng số lượng chữ số ở phần nguyên mà đối đó có; và `%f` thì bỏ qua các thông số về độ rộng và độ chính xác, nó sẽ mặc định in ra số thực với 6 chữ số ở phần thập phân.

| | |
|---|---|
| `%d` | in ra số nguyên với đúng số kí tự ứng với số chữ số |

| | |
|---|---|
| `%6d` | in ra số nguyên với độ rộng là 6 kí tự |
| `%f` | in ra số thực mặc định (6 chữ số ở phần thập phân) |
| `%6f` | in ra số thực mặc định với độ rộng là 6 kí tự |
| `%6.2f` | in ra số thực với độ rộng là 6 kí tự, cùng với 2 chữ số ở phần thập phân |

Ngoài những đặc tả đó, hàm `printf` còn phân biệt với các đặc tả khác như `%o` là hệ bát phân, `%x` là hệ thập lục phân, `%c` là kí tự, `%s` là xâu kí tự, và `%%` thì sẽ in ra kí tự `%`.

**Bài tập 1-3**. Thử sửa đổi chương trình chuyển đổi nhiệt độ ở trên để in ra phần tiêu đề cho chương trình ra màn hình.

**Bài tập 1-4**. Viết chương trình chuyển đổi nhiệt độ từ °C sang °F.

### 1.3. Câu lệnh lặp for

Có rất nhiều cách để viết một chương trình cho một bài toán hay công việc nào đó. Bây giờ chúng ta thử "biến hóa" một chút cho chương trình chuyển đổi nhiệt độ ở phần trước.

```
#include <stdio.h>
/* in bảng chuyển độ F sang độ C */
main()
{
      int f;
      for(f = 0; f <= 300; f = f + 20)
            printf("%3d %6.1f\n", f, (5.0/9.0)*(f-32));
}
```

Chương trình này vẫn sẽ cho ra kết quả giống với chương trình trước, nhưng nó trông hoàn toàn khác. Phần lớn thay đổi ở đây là chương trình đã bỏ hầu hết các biến; chỉ có mỗi biến `f` là giữ lại và cho nó thuộc kiểu `int`. Giá trị nhiệt độ nhỏ nhất, lớn nhất và bước nhảy xuất hiện dưới dạng hằng số ở trong câu lệnh `for`, đây là một cấu trúc mới, còn giá trị của biến `c` (biến lưu giá trị của °C ) bây giờ sẽ là phần đối thứ ba của hàm `printf` thay vì ta phải sử dụng một câu lệnh gán riêng biệt như trước.

Sự thay đổi cuối cùng ở đây là một nguyên tắc chung – ở bất cứ phạm vi nào nơi mà nó được cho phép sử dụng giá trị của các loại biến, bạn có thể sử dụng một biểu thức phức tạp để biểu diễn kiểu giá trị của biến đó. Ở đối thứ ba của hàm `printf` phải là một kiểu số thực để ứng với đặc tả `%6.1f`, như ta thấy biểu thức ở đó sẽ cho kết quả là một giá trị số thực, và giá trị đó được biểu diễn như trong đối đầu tiên của hàm `printf`.

Câu lệnh `for` là một *vòng lặp*, một dạng tổng quát của vòng lặp `while`. Nếu bạn so sánh câu lệnh lặp `for` với `while` trước đó, cách hoạt động của nó sẽ rõ ràng hơn cho bạn. Trong cặp dấu ngoặc đơn có tổng cộng ba phần, ngăn cách nhau bởi dấu chấm phẩy. Phần đầu tiên là phần khởi tạo,

```
f = 0
```

phần này được thực hiện duy nhất một lần, trước khi bắt đầu vòng lặp. Phần tiếp đó là điều kiện để điều khiển vòng lặp:

```
f <= 300
```

Điều kiện này được kiểm chứng: nếu nó đúng, phần thân của vòng lặp (ở đây chỉ có hàm `printf`) được thực thi. Sau đó ở phần cuối cùng sẽ là phần tăng lên, có thể hiểu là bước nhảy

```
f = f + 20
```

được thực thi, và sau đó quay lại phần điều kiện để kiểm tra tiếp tục. Vòng lặp sẽ kết thúc nếu như điều kiện có kết quả sai. Cũng giống với câu lệnh lặp `while`, ở phần thân của nó có thể chỉ có một câu lệnh, hoặc là một nhóm các câu lệnh được bao trọn bởi cặp dấu ngoặc nhọn. Phần khởi gán, điều kiện, bước nhảy có thể là bất kì biểu thức, phép toán nào.

Sự lựa chọn giữa `for` và `while` là tùy ý, phụ thuộc vào tùy bài toán, tùy công việc cái nào sẽ khiến cho chương trình trở nên rõ ràng hơn. Câu lệnh lặp `for` thường thích hợp cho các vòng lặp có sự khởi tạo, bước nhảy là những câu lệnh riêng và chúng có quan hệ với nhau, như thế sẽ rõ ràng hơn khi dùng câu lệnh lặp `while` khi mà nó gộp tất cả các câu lệnh ở một chỗ để thực hiện.

**Bài tập 1-5**. Thiết lập một chương trình chuyển đổi nhiệt độ như ví dụ trong bài viết, nhưng lần này hãy chuyển ngược lại từ 300 về 0.

### 1.4 Biểu tượng hằng

Một phần nữa trước khi chúng ta bỏ cái bài toán chuyển đổi nhiệt độ mãi mãi để chuyển sang các vấn đề khác. Đúng là một kiểu thực hành không hay nếu ta cứ mải mê với "các con số kì diệu" như 300 và 20 ở trong chương trình; Nó cho biết một ít thông tin tới người mà có thể sẽ đọc chương trình sau đó, và nó khó để thay đổi một cách có hệ thống. Một cách để giải quyết với đống con số kì diệu này là bằng việc thay nó bằng một cái tên có ý nghĩa. Với lệnh `#define` sẽ định nghĩa một tên tượng trưng hay biểu tượng hằng cho một dãy các kí tự cụ thể:

```
#define tên chuỗi-kí-tự-thay-thế
```

Sau đó, mọi thứ kí tự ở phần `tên` (không nằm trong cặp dấu nháy kép và cũng không trùng với các tên hoặc từ được định nghĩa trước đó của chương trình) sẽ thay thế cho các kí tự ở phần `chuỗi-kí-tự-thay-thế`. Phần `tên` cũng giống như nguyên tắc của một tên biến: gồm các kí tự chữ cái và kí tự số và phải bắt đầu bằng một kí tự chữ cái. `chuỗi-kí-tự-thay-thế` có thể là bất cứ dãy kí tự nào; nó không giới hạn.

```
#include <stdio.h>

#define LOWER 0       /* nhiệt độ thấp nhất */
```

```
#define UPPER 300     /* nhiệt độ cao nhất */
#define STEP 20       /* bươc nhảy */
/* in bảng chuyển đổi độ F sang độ C */
main()
{
      int f;
      for(f = LOWER; f <= UPPER; f = f + STEP)
            printf("%3d %6.1f", f, (5.0/9.0)*(f-32));
}
```

`LOWER`, `UPPER` và `STEP` là các *biểu tượng hằng*, không phải là biến, nên nó sẽ không cần khai báo. Các biểu tượng hằng thường được viết bằng các kí tự in hoa để phân biệt với các tên biến. Chú ý rằng không có dấu chấm phẩy ở cuối câu lệnh `#define`.

### 1.5 Kí tự vào/ra

Chúng sẽ xem xét về hệ thống các chương trình có mối quan hệ trong việc xử lý dữ liệu kí tự. Bạn sẽ thấy rằng nhiều chương trình chỉ là phiên bản mở rộng của những bài mẫu mà chúng ta sẽ đề cập ở đây.

Khuôn mẫu của đầu vào (INPUT) và đầu ra (OUTPUT) được hỗ trợ bởi thư viện chuẩn rất đơn giản. Việc nhập kí tự vào hay xuất kí tự ra, không màng tới việc nó bắt đầu từ đâu hoặc nó xuất đi đâu, các kí tự này đều theo một luồng. Một *luồng kí tự* là một dãy liên tiếp các kí tự chia thành nhiều dòng; mỗi dòng bao gồm không có gì hay nhiều kí tự nối tiếp theo một kí tự chuyển dòng (`\n`). Thư viện chuẩn có trách nhiệm cho việc khiến những kí tự vào hay ra theo y như khuôn mẫu này; lập tình viên sử dụng thư viện này không cần phải lo lắng có bao nhiêu dòng có thể biểu diễn được bên ngoài chương trình.

Thư viện chuẩn cung cấp một vài các hàm để đọc và viết một kí tự một lần, `getchar` và `putchar` là hai hàm đơn giản nhất. Mỗi lần nó được gọi tới, hàm `getchar` đọc kí tự tiếp theo từ luồng kí tự và trả về giá trị của nó. Đó là sau khi

```
c = getchar()
```

thì biến `c` sẽ chứa kí tự tiếp theo của dữ liệu đầu vào. Các kí tự thường được gõ từ bàn phím; còn đầu vào từ các *FILE* sẽ được đề cập ở **chapter 7**.

Hàm `putchar` thì in ra một kí tự mỗi khi nó được gọi đến:

```
putchar(c)
```

Câu lệnh có nghĩa sẽ in ra một kí tự ứng với giá trị số nguyên của biến `c` theo bảng mã *ASCII*, thường thì sẽ in lên màn hình. Việc gọi hàm `putchar` hay `printf` có thể được xen lẫn nhau; dữ liệu đầu ra sẽ xuất hiện ứng với việc gọi hàm nào được thực hiện.

#### 1.5.1 Sao chép dữ liệu

Với hàm getchar và putchar, bạn đã có thể viết được một số chương trình tiện ích mà không cần phải hiểu rõ về đầu vào hay đầu ra (INPUT/OUTPUT). Ví dụ đơn giản nhất là một chương trình sao chép kí tự đầu vào (nhập vào) và xuất kí tự đó ra đầu ra (in ra) theo mẫu ý tưởng:

*Đọc một kí tự*

`while(`*kí tự không phải là mã kết thúc file*`)`

*Đưa kí tự vừa đọc ra màn hình*

*Đọc một kí tự*

Để chuyển nó thành chương trình C như sau

```
#include <stdio.h>
/* sao chép một kí tự vào rồi in kí tự đó ra; phiên bản 1 */
main()
{
      int c;
      c = getchar();
      while(c != EOF){
            putchar(c);
            c = getchar();
      }
}
```

Toán tử quan hệ `!=` có nghĩa là "không bằng".

Kí tự nào xuất hiện trên màn hình hay là từ bàn phím, hay bất cứ gì khác, được lưu vào trong như một mẫu bit. Kiểu `char` đặc biệt chỉ dành cho việc lưu trữ dữ liệu kí tự, nhưng cũng có thể sử dụng dữ liệu số nguyên. Chúng tôi đã cố tính sử dụng kiểu int nhưng có lí do quan trọng.

Vấn đề ở đây là việc phân biệt khi nào kết thúc của dữ liệu đầu vào từ dữ liệu hợp lệ. Cách giải quyết là hàm `getchar` trả về một giá trị "đặc biệt" khi mà không còn dữ liệu đầu vào nữa, một giá trị mà nó không thể nhầm lẫn được với các kí tự thực sự. Kí tự đó được gọi là *kí tự kết thúc file*, kí hiệu `EOF` trong "end of file". Chúng ta phải khai báo biến `c` với kiểu đủ lớn để lưu trữ các giá trị mà `getchar` trả về. Chúng ta không thể dùng kiểu `char` khi mà biến `c` phải đủ lớn để lưu trữ `EOF` thêm vào ngoài các kí tự cho phép. Vì thế chúng ta đã dùng kiểu `int`.

`EOF` là một kiểu số nguyên được định nghĩa trong thư viện chuẩn `<stdio.h>`, nhưng giá trị số cụ thể của `EOF` sẽ chẳng có vấn dề gì miễn là nó không giống với giá trị của các kí tự kiểu `char` khác. Bằng việc sử dụng biểu tượng hằng, chúng ta chắc chắn rằng sẽ không có gì trong chương trình phụ thuộc vào giá trị số cụ thể của `EOF` này.

Chương trình sao chép ở trên sẽ được viết chính xác hơn từ các lập trình viên có kinh nghiệm. Trong ngôn ngữ C, bất cứ câu lệnh gán nào, như là

```
c = getchar()
```

là một phép toán (phép biểu diễn) và nó có giá trị, giá trị đó nằm ở vế bên trái của câu lệnh gán. Điều này có nghĩa việc gán có thể trờ thành một phần của một phép phép toán (phép biểu diễn) lớn hơn. Nếu như câu lệnh gán của một kí tự cho biến `c` được đặt trong phần kiểm tra điều kiện của vòng lặp `while`, thì chương trình sẽ được viết như sau:

```
#include <stdio.h>

/* sao chép một kí tự vào rồi in kí tự đó ra; phiên bản 2 */

main()

{

      int c;

      while((c = getchar()) != EOF)

            putchar(c);

}
```

Vòng lặp `while` nhận một kí tự, gán cho biến `c`, rồi bắt đầu kiểm tra điều kiện kí tự đó có phải là `EOF` hay không. Nếu không, thân của vòng lặp `while` sẽ được thực thi, in kí tự đó ra màn hình. Sau đó nó tiếp tục lặp lại từ đầu. Khi mà nó đã tới điểm kết thúc của đầu vào, vòng lặp `while` sẽ kết thúc, chương trình cũng kết thúc.

Phiên bản này tập trung vào dữ liệu đầu vào – chỉ có liên quan tới một lần dùng hàm `getchar` – và giúp cho chương trình co lại đơn giản hơn. Chương trinh bây giờ tối ưu hơn rất nhiều, và một khi bạn hoàn thiện được mình trong kĩ năng lập trình, việc đọc chương trình sẽ dễ dàng hơn. Bạn sẽ gặp phải phong cách lập trình này thường xuyên. (Khi lập trình rất dễ tạo ra những chương trình khó đọc, khó hiểu, tuy nhiên chúng tôi sẽ cố hạn chế cái xu hướng hack não này để bạn dễ hiểu hơn)

Cặp ngoặc đơn bao quanh phần gán bên trong phần điều kiện rất cần thiết. Độ ưu tiên của toán tử `!=` lớn hơn `=`, có nghĩa là nếu không có cặp ngoặc đơn này thì việc kiểm tra `!=` sẽ thực hiện trước việc gán `=`. Vì thế câu lệnh

```
c = getchar() != EOF
```

Sẽ giống với câu lệnh

```
c = (getchar() != EOF)
```

Điều này sẽ khiến cho việc gán của biến `c` gặp sự không không mong muốn, nó có thể là giá trị 0 hoặc 1 tùy thuộc vào việc gọi hàm `getchar` đã đọc dữ liệu `EOF`(tức là đã kết thúc đầu vào) hay chưa. (Chi tiết hơn sẽ ở **chapter 2**)

**Bài tập 1-6**. Thử kiểm chứng xem giá trị sau khi `getchar() != EOF` là 0 hay 1.

**Bài tập 1-7**. Viết chương trình in ra giá trị của `EOF`.

### 1.5.2 Đếm kí tự

Chương trình sau đây sẽ có chức năng đếm các kí tự; nó cũng tương tự với chương trình sao chép ở phần trước.

```
#include <stdio.h>
/* đếm kí tự đầu vào; phiên bản 1 */
main()
{
      long nc;
      nc =0;
      while(getchar() != EOF)
            ++nc;
      printf("%ld\n", nc);
}
```

Câu lệnh

```
++nc;
```

giới thiệu một toán tử mới, ++, có nghĩa là *tăng thêm 1*. Bạn có thể thay vì viết `nc = nc + 1` nhưng `++nc` sẽ ngắn gọn và thường sẽ hiệu quả hơn. Có một toán tử khác tương tự đó là `--` *giảm xuống 1*. Hai toán tử `++` và `--` có thể là *tiền tố* (`++nc`) hoặc *hậu tố* (`nc++`); nhưng hai kiểu này thì lại cho kết quả khác nhau, sẽ trình bày chi tiết hơn ở **chapter 2**, nhưng `++nc` và `nc++` đều tăng biến `nc` thêm 1. Từ bây giờ chúng ta sẽ tạm dùng kiểu tiền tố.

Chương trình đếm kí tự này tích trữ số lượng đếm vào một biến kiểu `long` thay vì kiểu `int`. Kiểu số nguyên `long` có ít nhất 32 bit. Mặc dù trên nhiều máy tính, `int` và `long` có cùng độ lớn, một số khác thì một biến `int` có 16 bit, với giá trị lớn nhất là `32767`, và nó sẽ nhận tương đối ít dữ liệu truyền vào và làm "tràn bộ nhớ" của biến đếm kiểu `int`. Đặc tả `%ld` cho hàm `printf` biết rằng nó sẽ ứng với một đối là kiểu số nguyên dài (`long`).

Có thể "đối phó" với những con số còn lớn hơn nữa bằng việc sử dụng kiểu số thực `double`. Chúng ta cũng sẽ sử dụng vòng lặp `for` thay vì `while`, để minh họa một cách khác để viết vòng lặp.

```
#include <stdio.h>
/* đếm kí tự đầu vào; phiên bản 2 */
main()
{
```

```
        double nc;
        for(nc=0; getchar() != EOF; ++nc)
              ;
        printf("%.0f\n", nc);
}
```

Hàm printf sử dụng đặc tả `%f` cho cả kiểu `float` và `double`; `%.0f` chỉ ra in số thực không có phần thập phân, vì sau dấu chấm động là số 0.

Phần thân của vòng lặp trống trơn, bởi vì công việc đã được thực hiện ở phần điều kiện và phần tăng (bước nhảy). Nhưng về các nguyên tắc ngữ pháp của ngôn ngữ C yêu cầu rằng vòng lặp `for` phải có phần thân. Dấu chấm phẩy "cô độc" kia, được gọi là *câu lệnh null*, đã làm hài lòng yêu cầu đó. Chúng ta để nó ở một ròng riêng để cho dễ nhận thấy.

Trước khi chúng ta rời bỏ chương trình đếm kí tự này, nhận xét rằng nếu dữ liệu đầu vào không chứa kí tự nào, điều kiện của vòng lặp `while` và `for` sẽ sai ngay lúc gọi hàm `getchar`, chương trình sẽ chẳng làm gì tiếp nữa. Điều này là quan trọng. Một trong những thứ hay về vòng lặp `while` và `for` là nó kiểm tra điều kiện ở ngay đầu vòng lặp, trước khi truy cập vào phần thân. Nếu như không có gì để làm, thì cũng chẳng có việc gì để hoàn thành, ngay cả khi điều đó có nghĩa là bỏ qua phần thân của vòng lặp. Các chương trình nên hoạt động một cách thông minh khi ta không đưa dữ liệu đầu vào. Vòng lặp `while` và `for` giúp ta đảm bảo rằng các chương trình làm các công việc hợp lí cùng với các điều kiện được đề ra.

### 1.5.3 Đếm dòng

Chương trình tiếp theo có chức năng đếm số lượng dòng từ dữ liệu đầu vào. Nhưng đã nêu ở trên, thư viện chuẩn đảm bảo rằng một luồng kí tự vào xuất hiện thành các dòng liên tiếp, mỗi dòng kết thúc bằng một kí hiệu (một kí tự) sang dòng mới (`\n`). Vì vậy, việc đếm các dòng chỉ là việc đếm các kí tự sang dòng mới này:

```
#include <stdio.h>
/* đếm các dòng dữ liệu vào */
main()
{
      int c, nl;
      nl = 0;
      while((c=getchar()) != EOF)
            if(c == '\n')
                  ++nl;
      printf("%d\n", nl);
}
```

Trong thân của vòng lặp `while` bây giờ bao gồm một câu lệnh `if`, nó điều khiển câu lệnh `++nl` sau đó. Câu lệnh `if` này kiểm tra điều kiện bên trong cặp ngoặc đơn, và nếu điều kiện đúng, câu lệnh theo nó sẽ được thực thi (hoặc một nhóm các câu lệnh được bao trong cặp ngoặc nhọn). Chúng tôi đã một lần nữa thụt đầu dòng câu lệnh sau `if` để biểu diễn rằng nó được điều khiển bởi `if`.

Hai kí tự bằng == là một toán tử so sánh "bằng". Sử dụng kí hiệu này là để phân biệt với toán tử gán = được dùng trong các câu lệnh gán. Đặc biệt chú ý đến điều này vì những người mới học ngôn ngữ C đôi lúc vẫn sẽ nhầm lẫn giữ = và ==. Dù nó là điều kiện để so sánh, nhưng kết quả vẫn là hợp lệ nên chương trình sẽ không báo lỗi, chúng ta sẽ rõ hơn ở **chapter 2**.

Một kí tự được viết trong cặp dấu nháy đơn đại diện cho một giá trị số nguyên bằng với giá trị số của kí tự đó trong trong bộ kí tự mặc định của máy tính (thường là theo bảng mã ASCII). Kí tự này được gọi là *hằng kí tự*, mặc dù đây chỉ là một cách khác để viết một số nguyên nhỏ. Ví dụ `'A'` là một hằng kí tự; trong bảng mã ASCII thì giá trị của nó là `65`, đại diện cho kí tự `A`. Đương nhiên dùng `'A'` sẽ "thích hơn" là dùng giá trị 65: điều này là hiển nhiên, và nó là kí tự độc lập với một bộ kí tự đặc biệt.

Các kiểu kí hiệu được sử dụng trong hằng xâu kí tự (xâu kí tự được bao bởi cặp nháy kép) cũng hợp lệ như một hằng kí tự, vì thế `'\n'`, kí hiệu chuyển sang dòng mới, tuy gồm hai kí tự nhưng nó vẫn được máy tính xem như là một kí tự dạng đặc biệt (đã nói ở phần trước), và nó có giá trị là 10 trong bảng mã ASCII. Bạn nên cẩn thận chú ý một lần nữa rằng `'\n'` chỉ là một kí tự trong ngôn ngữ C (từ nay sẽ chỉ nói là một kí tự), và nó biểu diễn cho giá trị của một số nguyên; ở khía cạnh khác, `"\n"` lại là một hằng xâu kí tự (bao bởi cặp nháy kép) nhưng chỉ có một kí tự duy nhất. Bài về *xâu kí tự và các kí tự* sẽ được tìm hiểu sâu hơn ở **chapter 2**.

**Bài tập 1-8**. Viết một chương trình đếm các kí tự trống, tab và sang dòng mới.

**Bài tập 1-9**. Viết chương trình sao chép dữ liệu vào và xuất ra màn hình, thay thế các các xâu kí tự trống thành duy nhất một kí tự trống.

**Bài tập 1-10**. Viết chương trình sao chép dữ liệu vào và xuất ra màn hình, thay thế các khoảng tab thành kí tự `\t`, backspace thành `\b` và dấu vạch chéo ngược thành `\\`. Điều này sẽ làm rõ các khoảng tab và backspace hơn.

### 1.5.4 Đếm từ

Đây là bài toán thứ tư trong các bài toán hữu dụng đếm dòng, từ, các kí tự, với một cách định nghĩa đơn giản thì một từ sẽ bao gồm một dãy các kí tự mà trong nó không có khoảng trống, khoảng tab hay xuống dòng mới. Sau đây là một phiên bản chương trình đơn giản.

```
#include <stdio.h>

#define IN 1     /* bên trong một từ */

#define OUT 0    /* bên ngoài một từ */

/* đếm dòng, từ, và các kí tự đầu vào */

main()

{
```

```
        int c, nl, nw, nc, state;
        state = OUT;
        nl = nw = nc = 0;
        while((c = getchar()) != EOF){
              ++nc;
              if(c == '\n')
                    ++nl;
              if(c == ' ' || c == '\n' || c == '\t')
                    state = OUT;
              else if(state == OUT){
                    state = IN;
                    ++nw;
              }
        }
        printf(" %d %d %d\n", nl, nw, nc);
}
```

Mỗi khi chương trình gặp kí tự đầu tiên của một từ, nó sẽ đếm thêm một từ. Biến `state` ghi lại trạng thái là chương trình có đang xử lí một từ hay không; ban đầu nó "không ở bên trong từ nào", có nghĩa gán cho biến đó giá trị của hằng `OUT`. Chúng ta sử dụng biểu tượng hằng `IN` và `OUT` cho giá trị lần lượt 1 và 0 bởi vì chúng sẽ khiến cho chương trình đọc dễ dàng hơn. Trong một chương trình nhỏ như thế này, nó tạo nên những chênh lệch nhỏ không đáng kể, nhưng ở các chương trình lớn hơn, để có thể trình bày nó một cách rõ ràng thì buộc mình phải khiêm tốn, cố gắng viết được các chương trình như thế này ngay từ khi bắt đầu. Bạn sẽ thấy rằng rất dễ tạo nên những thay đổi sâu hơn trong các chương trình, nơi mà các biểu tượng hằng có thể thay thế cho các con số.

Dòng

```
nl = nw = nc = 0;
```

gán cho ba biến trên có giá trị 0. Đây không phải là một trường hợp đặc biệt, nhưng có thực tế quan trọng rằng một phép gán là một biểu thức giá trị và các phép gán sẽ kết hợp từ phải sang trái. Điều này có nghĩa giống với cách viết

```
nl = (nw = (nc = 0));
```

Toán từ `||` có nghĩa "hoặc", vì thế dòng

```
if(c == ' ' || c == '\n' || c == '\t')
```

chỉ ra "nếu `c` là một khoảng trống hoặc `c` là kí tự chuyển dòng hoặc `c` là khoảng tab". Nó tương ứng với toán tự `&&` có nghĩa "và"; nó có độ ưu tiên cao hơn `||`. Các biểu thức kết hợp với `&&` hoặc `||` mặc định đánh giá từ phải sang trái, và nó đảm bảo rằng việc đánh giá sẽ kết thúc ngay khi biết được nó đúng hay sai. Nếu `c` là một khoảng trống, việc kiểm tra sẽ không cần thiết đối với kí tự chuyển dòng hay khoản tab, tức nó sẽ không kiểm tra nữa. Cũng không có gì đặc biệt cụ thể ở đây, nhưng nó có ý nghĩa trong nhiều trường hợp phức tạp hơn, chúng ta sẽ sớm gặp nó.

Ví dụ trên cũng cho ta thấy một điểm mới nữa là ở `else`, nó định ra rõ hành động thay thế nếu như điều kiện trong câu lệnh `if` sai. Khái quát hơn

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

Một và chỉ một trong hai câu lệnh kết hợp với lệnh `if-else` sẽ được thực thi. Nếu như biểu thức điều kiện đúng, *câu lệnh 1* thực thi; nếu không, *câu lệnh 2* sẽ thực thi. Ở phần câu lệnh có thể là một hay nhiều câu lệnh được bao bởi cặp ngoặc nhọn. Trong chương trình đếm từ, sau `else` điều khiển hai câu lệnh nếu như `if` có kết quả điều kiện là sai.

**Bài tập 1-11**. Bạn sẽ thử kiểm tra chương trình đếm từ này như thế nào ? Những kiểu dữ liệu nhập vào như thế nào sẽ có thể gây ra lỗi hoặc kết quả sai nếu có ?

**Bài tập 1-12**. Viết chương trình in ra mỗi dòng chỉ một từ.

### 1.6 Mảng

Bây giờ chúng ta sẽ viết một chương trình đếm số lần xuất hiện của mỗi con số, mỗi khoảng trắng (space, tab, xuống dòng), và các kiểu kí tự khác. Điều này cho phép chúng ta minh họa một số khía cạnh của ngôn ngữ C trong một chương trình.

Có 12 loại dữ liệu vào (các chữ số từ 0 đến 9, khoảng trống và các kí tự khác), vì thế sử dụng một mảng sẽ rất tiện lợi để lưu trữ số lượng xuất hiện các con số hơn là phải khai báo cả chục biến. Sau đây là một phiên bản của chương trình:

```
#include <stdio.h>
/* đếm các chữ số, khoảng trống và các kí tự khác */
main()
{
        int c, i, nwhite, nother;
        int ndigit[10];
        nwhite = nother = 0;
        for(i = 0; i < 10; ++i)
```

```
            ndigit[i] = 0;
        while((c = getchar()) != EOF)
            if(c >= '0' && c <= '9')
                ++ndigit[c-'0'];
            else if(c == ' ' || c == '\n' || c == '\t')
                ++nwhite;
            else
                ++nother;
        printf("digits = ");
        for(i = 0; i < 10; ++i)
            printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n", nwhite, nother);
    }
```

Đầu ra của chương trình sẽ có dạng như thế này nếu ta nhập "123 abc"

```
digits = 0 1 1 1 0 0 0 0 0 0, white space = 1, other = 3
```

Có một dạng khai báo mới là

```
int ndigit[10];
```

cho ta biết rằng `ndigit` là một mảng gồm có 10 phần tử số nguyên. Chỉ số của mảng luôn luôn bắt đầu từ 0 trong ngôn ngữ C, vì thế các phần tử sẽ là `ndigit[0]`, `ndigit[1]`,…, `ndigit[9]`. Điều này được chỉ rõ ở trong vòng lặp `while` khi khởi tạo và in mảng ra màn hình.

Phần chỉ số có thể là bất cứ cách biểu diễn số nguyên nào, có thể bao gồm các biến như biến `i` và các hằng số nguyên.

Chương trình cụ thể này dựa vào tính chất của các kí tự đại diện cho các chữ số. Ví dụ ở phần kiểm tra điều kiện

```
if(c >= '0' && c <= '9')…
```

định rõ xem biến `c` có phải là một chữ số hay không. Nếu đúng, giá trị số của chữ số đó sẽ là

```
c – '0'
```

Điều này chỉ hoạt động nếu `'0'`, `'1'`, …, `'9'` là các giá trị gia tăng liên tiếp. May mắn rằng điều này đúng với mọi bộ kí tự.

Rõ ràng, kiểu `char` chỉ là các số nguyên nhỏ, vì thế các biến kiểu này và các hằng đều giống hệt với các biến kiểu `int` trong các biểu thức phép toán. Điều này là tự nhiên và tiện lợi; ví dụ, `c – '0'` là một

kiểu biểu diễn số nguyên với giá trị từ 0 đến 9 ứng với kí tự từ `'0'` tới `'9'` được lưu trữ trong ngôn ngữ C, như vậy nó có thể biểu diễn chỉ số phần tử cho mảng `ndigit`.

Việc quyết định xem kí tự đó là một chữ số, khoảng trống hay các kí tự khác được thực hiện thông qua

```
if(c >= '0' && c <= '9')
        ++ndigit[c-'0'];
    else if(c == ' ' || c == '\n' || c == '\t')
        ++nwhite;
    else
        ++nother;
```

Kiểu mẫu

`if` (*điều kiện 1*)

*câu lệnh 1*;

`else if` (*điều kiện 2*)

*câu lệnh 2*;

…

…

`else`

*Câu lệnh n*;

diễn ra thường xuyên trong nhiều chương trình là cách để diễn tả nhiều quyết định thực hiện. Các *điều kiện* được kiểm tra ngay đầu tiên cho tới khi một *điều kiện* được thỏa mãn; điều này tương ứng phần *câu lệnh* đó sẽ được thực thi, và tất cả cấu trúc `if` đó sẽ kết thúc ngay. (ở phần câu lệnh có thể sẽ có nhiều câu lệnh được bao bởi cặp ngoặc đơn) Nếu như không có *điều kiện* nào thỏa mãn, phần *câu lệnh* ở phần `else` cuối cùng sẽ được thực thi nếu có. Nếu như phần `else` cuối cùng và phần *câu lệnh* đó không có, thì sẽ chẳng có công việc nào được thực hiện. Có thể có nhiều nhóm

`else if` (*điều kiện*)

*câu lệnh*;

giữa `if` đầu tiên và `else` cuối cùng.

Theo khía cạnh về phong cách lập trình, thì nên định dạng cấu trúc điều kiện này như chúng tôi đã trình bày ở trên. Như vậy sẽ cho ta một cách nhìn khái quát hơn về các điều kiện cũng như quyết định làm công việc nào theo cấu trúc trên.

Câu lệnh `switch`, sẽ được trình bày chi tiết ở **chapter 3**, cung cấp một cách khác để viết các quyết định (điều kiện) thành nhiều nhánh, mà cụ thể hợp lí khi điều kiện đó là một số nguyên hay các kí tự biểu diễn khác. Để thấy rõ hơn thì chúng tôi sẽ trình bày chương trình này theo phiên bản sử dụng `switch` ở **phần 3.4**.

**Bài tập 1-13**. Hãy thử viết chương trình in ra biểu đồ chiều dài của các từ nhập vào. Sẽ dễ dàng nếu như biểu đồ đó được in các thanh như `'-'` hoặc `'='` theo chiều ngang; theo chiều dọc thì có vẻ sẽ thách thức hơn một chút.

**Bài tập 1-14**. Viết chương trình in ra biểu đồ tần số của các kí tự khác nhau của dữ liệu nhập vào.

### 1.7 Hàm

Trong ngôn ngữ C, một hàm cũng giống như một chương tình con tượng tự như trong ngôn ngữ Fortran hay Pascal. Hàm cung cấp một cách cách tiện lợi để gói gọn một số công việc mà nó có thể dùng sau đó, mà không cần phải lo lắng gì về hệ thống xử lí của nó. Với những hàm được thiết kế hợp lí, đúng đắn, thì ta có thể bỏ qua "*một công việc được hoàn thành như thế nào*"; chỉ cần biết được "*cái gì đã hoàn thành*" là đủ. Ngôn ngữ C tạo ra cách sử dụng một hàm rất dễ, tiện lợi và có hiệu quả; bạn sẽ hay gặp một hàm ngắn được định nghĩa và được gọi một lần, bởi vì nó sẽ làm sáng sủa hơn một số mảng code của chương trình.

Cho tới giờ chúng ta đã dùng các hàm như `printf`, `getchar` và `putchar` mà ta đã được cung cấp; bây giờ sẽ là thời điểm để chúng ta tạo ra một số hàm cho riêng mình. Vì ngôn ngữ C không có toán tử lũy thừa như `**` của ngôn ngữ Fortran, chúng tôi sẽ minh họa các phần của hàm được định nghĩa bằng việc viết một hàm `power(m,n)` để tính lũy thừa của biến `m` với số mũ dương `n`. Ví dụ giá trị kết quả của `power(2,5)` sẽ là 32. Đây không phải là một cách tính lũy thừa thường theo thực tế, khi mà nó chỉ tính với lũy thừa dương của các số nguyên nhỏ, nhưng nó đủ để làm ví dụ minh họa.

Dưới đây sẽ là hàm `power` và một chương trình chính để thực hành, vì vậy bạn sẽ thấy toàn bộ cấu trúc của nó.

```
#include <stdio.h>
int power(int m, int n);
/* thử hàm tính lũy thừa */
main()
{
      int i;
      for(i = 0; i < 10; ++i)
            printf("%d %d %d\n", i, power(2,i), power(-3,i));
      return 0;
}
/* tính lũy thừa của base với số mũ n; n >= 0 */
```

```
int power(int base, int n)
{
      int i, p;
      p = 1;
      for(i = 1; i <= n; ++i)
            p = p * base;
      return p;
}
```

Định nghĩa hàm có dạng như sau:

*kiểu-trả-về tên-hàm* (*khai báo các tham số nếu có*)

{

*Các-khai-báo-biến*

*Các-câu-lệnh*

}

Các định nghĩa hàm có thể xuất hiện tùy theo nhu cầu, và trong một tệp chính hoặc có thể nhiều tệp khác, mặc dù không có hàm nào có thể tách ra giữa các tệp với nhau. Nếu như chương trình chính bao gồm nhiều tệp, bạn phải liệt kê hết để rồi biên dịch và tải nó, nhưng đó là vấn đề của hệ điều hành, không phải là đặc tính của ngôn ngữ. Từ thời điểm này, chúng tôi sẽ đảm bảo rằng các hàm sẽ nằm chung một tệp, vì thế mọi thứ bạn đã học về việc chạy chương trình C vẫn hoạt động được.

Hàm `power` được gọi hai lần bởi hàm `main`, trong câu lệnh

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Mỗi lần gọi đều truyền cho hàm `power` hai đối số, mỗi lần đó đều trả về một giá trị số nguyên đã định định dạng và in ra ở đối đầu tiên của hàm `printf`. Trong một cách biểu diễn, `power(2,i)` chỉ là một số nguyên giống như 2 và `i`. (Không phải hàm nào cũng trả về một giá trị số nguyên; chúng ta sẽ tìm hiểu thêm ở **chapter 4**).

Dòng đầu tiên của hàm `power`,

```
int power(int m, int n);
```

thông báo về các kiểu tham số và tên, và kiểu trả về của hàm. Các tên tham số được sử dụng bởi hàm `power` chỉ được dùng với hàm `power`, và nó không hiện hữu trên các hàm khác: các thói quen khác có thể sử dụng tên giống nhau mà không gây xung đột gì. Điều này đúng đối với biến `i` và `p`: biến `i` trong hàm `power` không hề giống với biến `i` trong `main`.

Chúng ta sẽ thường dùng *tham số*, là biến được đặt tên trong cặp dấu ngoặc đơn được liệt kê khi định nghĩa hàm, và sẽ lấy giá trị của *đối số* để sử dụng khi gọi hàm đó. Các thuật ngữ *đối số hình thức* và *đối số thực tế* đôi khi được sử dụng với cùng đặc thù.

Giá trị mà hàm `power` tính toán được trả về cho hàm `main` bởi câu lệnh `return`. Khái quát hơn với `return`:

`return` *biểu-thức*;

Một hàm không cần trả về một giá trị; một câu lệnh return không có biểu thức điều khiển, không có giá trị hữu dụng, được trả về cho đối tượng gọi nó, nó cũng kết thúc ngay khi gặp ngoặc nhọn phải. Và việc gọi hàm có thể bỏ qua giá trị trả về của một hàm.

Bạn cần phải chú ý rằng có một câu lệnh `return` ở cuối hàm `main`. Vì `main` cũng là một hàm như những hàm khác, nó có thể trả về một giá trị cho đối tượng gọi nó, tác động đến môi trường nơi mà chương trình được thực thi. Thường thì việc trả về giá trị 0 có nghĩa là trạng thái kết thúc bình thường; giá trị khác không là dấu hiệu không bình thường hoặc các điều kiện kết thúc sai. Vì để cho đơn giản hơn, chúng tôi đã "lờ đi" câu lệnh `return` trong hàm `main` cho tới thời điểm này, nhưng chúng tôi sẽ thêm chúng từ bây giờ, coi như đó là một lời nhắc nhở rằng các chương trình nên trả về trạng thái cho môi trường lập trình của nó.

Phần khai báo

```
int power(int m, int n);
```

ngay trước hàm `main` chỉ ra rằng `power` đó là một hàm cần hai đối số nguyên truyền vào và trả về một giá trị kiểu `int`. Phần khai báo này, được gọi là *nguyên mẫu hàm*, phải đúng với phần định nghĩa và các cách dùng của hàm `power`. Nó sẽ có có lỗi nếu như phần định nghĩa của hàm hay các cách sử dụng của nó không đúng với nguyên mẫu của hàm.

Phần tên của tham số không cần phải giống nhau. Thực vậy, tên của tham số là tùy chọn trong một nguyên mẫu của hàm, cho nên phần nguyên hàm chúng ta có thể viết

```
int power(int, int);
```

Việc chọn các tên tốt sẽ là một sự cung cấp tài liệu tốt, vì thế chúng ta sẽ thường sử dụng nó.

Chú ý: Điều thay đổi lớn nhất giữa ngôn ngữ C theo chuẩn ANSI và các phiên bản trước đó là việc định nghĩa và khai báo các hàm như thế nào. Ở phiên bản đầu tiên của ngôn ngữ C, thì việc định nghĩa hàm `power` sẽ được viết như sau:

```
/* tính lũy thừa của base với số mũ n; n >= 0 */
/* phong cách viết ở phiên bản cũ */
power(base, n)
int base, int n;
{
      int i, p;
```

```
        p = 1;
        for(i = 1; i <= n; ++i)
            p = p * base;
        return p;
}
```

Các tên của tham số được đặt trong cặp ngoặc đơn, và kiểu của chúng lại được khai báo sau đó, ngay trước dấu ngoặc nhọn trái; nếu các tham số không được khai báo kiểu thì nó sẽ mặc định là kiểu int. (phần thân của hàm này vẫn giống như trước đó)

Phần nguyên hàm của hàm `power` ngay đầu chương trình có thể có dạng như sau:

```
int power();
```

Không có tham số nào, vì thế trình biên dịch sẽ không có thể dễ dàng kiểm tra rằng hàm `power` đã được gọi một cách chính xác. Thực vậy, bởi vì theo mặc định hàm `power` sẽ trả về một giá trị kiểu int, toàn bộ phần khai báo có thể sẽ bị "lờ đi".

Cú pháp mới của các nguyên mẫu hàm khiến nó trở nên dễ dàng hơn cho trình biên dịch để kiểm tra lỗi trong số lượng các đối hay kiểu của các đối đó. Việc khai báo và định nghĩa kiểu cũ vẫn hoạt động ở ngôn ngữ C chuẩn ANSI, nhưng chúng tôi khuyên rằng bạn nên dùng nguyên mẫu mới này thay vì kiểu cũ khi mà bạn có trình biên dịch vẫn hỗ trợ nó.

**Bài tập 1-15**. Viết lại chương trình chuyển đổi nhiệt độ ở phần 1.2 với việc sử dụng hàm cho công việc chuyển đổi.

### 1.8 Đối số - Truyền theo giá trị

Một diện mạo của hàm trong ngôn ngữ C có thể không quen thuộc với các lập trình viên đã từng sử dụng một số ngôn ngữ khác, cụ thể là Fortran. Trong ngôn ngữ C, mọi đối của hàm đều truyền "theo giá trị". Điều này có nghĩa, hàm được gọi tới được truyền vào giá trị của các đối (là các biến gốc) vào trong các biến tạm thời chứ không phải là chính các biến gốc đó. Điều này dẫn tới một số đặc tính khác biệt đối với mọi ngôn ngữ khác như Fortran hay Pascal, vì ở các ngôn ngữ đó, việc gọi các hàm (trong các ngôn ngữ Pascal hay Fortran thì đây được gọi là các chương trình con) sẽ truyền các đối chính là các biến gốc đó vào, tức là làm việc với chính biến gốc đó, chứ không phải là một biến tạm thời lưu giá trị của các đối được truyền hàm vào như ngôn ngữ C.

Sự khác biệt chính trong ngôn ngữ C là hàm được gọi không thể thay đổi một biến trong việc gọi hàm này; nó chỉ có thể thay đổi biến cục bộ, tức là biến tạm thời lưu giá trị của đối được truyền vào.

Truyền giá trị được hiểu vui như một tài sản chả ai có trách nhiệm pháp lí với nó, tức giá trị của đối thích được truyền đi đâu thì truyền. Nó dẫn tới nhiều chương trình chặt chẽ với một vài biến bên ngoài, bởi vì các tham số có thể được dùng như những biến cục bộ được khởi tạo một cách tiện lợi trong thủ tục được gọi tới. Ví dụ, đây là một phiên bản khác của hàm `power` cho ta thấy được cách hoạt động của đặc tính này.

```
/* tính lũy thừa của base với số mũ n; n >= 0; phiên bản 2 */
```

```
int power(int base, int n)
{
	int p;
	for(p = 1; n > 0; --n)
		p = p * base;
	return p;
}
```

Tham số `n` được dùng như một biến tạm thời, và được đếm giảm xuống (ở vòng lặp `for` chạy ngược lại theo biến `n`) cho tới khi nó bằng 0; sẽ chẳng cần đến biến `i` nữa. Bất cứ điều gì xảy ra cho biến `n` bên trong hàm `power` chẳng có ảnh hưởng gì đến đối số mà hàm `power` được gọi tới cả.

Khi cần thiết, có thể sắp xếp cho một hàm thay đổi giá trị của một biến trong thủ tục gọi tới. Đối tượng gọi phải cung cấp *địa chỉ* của biến đó để được thiết lập thay đổi (nói chính xác là một *con trỏ* trỏ tới biến đó), và hàm được gọi đó phải khai báo tham số là một con trỏ và truy cập vào biến đó một cách trực tiếp. Chúng ta sẽ bàn về con trỏ ở **chapter 5**.

Các điều đã nói ở trên có sự khác biệt đối với các mảng. Khi một tên của một mảng được sử dụng như một đối, giá trị truyền cho hàm chính là địa chỉ của phần tử đầu tiên của mảng - Ở đây không có sự sao chép các phần tử của mảng. Bằng việc sử dụng các chỉ số mảng, hàm này có thể truy cập và thay đổi bất cứ phần tử nào của mảng. Điều này sẽ trình bày ở phần tiếp theo sau đây.

### 1.9 Mảng kí tự

Kiểu thông dụng nhất về mảng trong ngôn ngữ C là mảng các kí tự. Để minh họa cách sử dụng của các mảng kí tự và các hàm thao tác với chúng, chúng ta sẽ viết một chương trình để đọc một bộ các dòng văn bản và in ra dòng dài nhất. Dạng của nó khá đơn giản:

```
while (có dòng nào đó)
	if (nếu nó dài hơn dòng dài nhất trước đó)
		lưu nó lại
		lưu độ dài của nó
in dòng dài nhất ra
```

Khung sườn này chỉ rõ cho ta thấy chương trình chia ra các phần một cách tự nhiên. Một phần nhận dòng mới, phần khác kiểm tra nó, phần khác nữa lưu nó và còn lại sẽ điều khiển tiến trình này.

Vì mọi thứ phân chia khá rõ ràng và đẹp mắt, nên nó sẽ giúp ta sẽ viết chương trình theo hướng đó dễ dàng. Do đó, đầu tiên chúng ta viết hàm `getline` riêng biệt để nhận lấy dòng tiếp theo của dữ liệu đầu vào. Chúng ta sẽ cố xây dựng hàm này vẫn sẽ phù hợp với các bối cảnh bài bài toán khác. Ít nhất hàm `getline` phải trả về một giá trị báo biệu có thể là kết thúc tệp (`EOF`); một cách thiết kế hợp lí hơn sẽ trả về độ dài của dòng đó, hoặc 0 nếu gặp phải EOF. Giá trị 0 sẽ rất phù hợp với dấu hiệu `EOF`, bởi vì chẳng

có dòng nào hợp lệ với độ dài bằng 0 cả. Mỗi dòng văn bản đều có ít nhất một kí tự; kể cả một dòng chỉ có kí tự chuyển dòng mới (\n) cũng vẫn có độ dài là 1.

Khi ta tìm một dòng mà nó dài hơn dòng dài nhất trước đó, thì nó phải được lưu ở một nơi nào đó. Điều này gợi ý đến một hàm thứ hai, tên là copy, để sao chép dòng mới này tới một nơi khác "an toàn".

Cuối cùng, chúng ta cần một chương trình chính để điều khiển các hàm getline và copy. Sau đây là kết quả.

```
#include <stdio.h>
#define MAXLINE 1000  /* độ dài lớn nhất có thể nhập vào */
int getline(char line[], int maxline);
void copy(char to[], char from[]);
/* in dòng dài nhất */
main()
{
      int len;                  /* độ dài dòng hiện tại */
      int max;                  /* độ dài dòng lớn nhất */
      char line[MAXLINE];       /* dòng hiện tại */
      char longest[MAXLINE];    /* dòng dài nhất lưu ở đây */
      max = 0;
      while((len = getline(line, MAXLINE)) > 0)
            if(len > max){
                  max = len;
                  copy(longest, line);
            }
      if(max > 0)           /* kiểm tra nếu có dòng nào lớn nhất */
            printf("%s", longest);
      return 0;
}
/* getline: đọc một dòng vào cho s, trả về độ dài của nó */
int getline(char s[], int lim)
{
      int c, i;
```

```
        for(i = 0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
                s[i] = c;
        if(c == '\n'){
                s[i] = c;
                ++i;
        }
        s[i] = '\0';
        return i;
}
/* copy: sao chém từ 'from' vào 'to'; giả định nó đủ lớn */
void copy(char to[], char from[])
{
        int i;
        i=0;
        while((to[i] = from[i]) != '\0')
                ++i;
}
```

Hai hàm `getline` và `copy` được khai báo ở ngay đầu chương trình, chúng ta giả định nó được chứa trong cùng một file.

Hàm `main` và `getline` giao tiếp thông qua một cặp đối và một giá trị trả về. Ở hàm `getline`, các đối được khai báo bởi dòng

```
int getline(char s[], int lim)
```

xác định rõ rằng đối đầu tiên, `s`, là một mảng, và đối thứ hai, `lim`, là một số nguyên. Mục đích của việc cung cấp kích thước của một mảng trong phần khai báo (ở đây là `lim`) là để thiết lập lưu trữ ở một bên. Độ dài của mảng `s` (bảo bởi cặp ngoặc vuông) không cần thiết trong hàm `getline` khi mà kích thước của nó được thiết lập trong hàm `main`. Hàm `getline` dùng `return` để gửi một giá trị về cho đối tượng gọi, cũng như hàm `power` đã thực hiện. Dòng này cũng thông báo rằng `getline` trả về kiểu `int`; vì `int` là một kiểu trả về mặc định, nên nó có thể được bỏ qua.

Một số hàm trả về một giá trị có ích; một số khác, như hàm `copy`, chỉ được dùng cho công việc nào đó và không trả về giá trị nào. Kiểu trả về của hàm `copy` là `void`, có nghĩa là không có giá trị gì được trả về.

Hàm `getline` đặt một kí tự `'\0'` (kí tự *null*, có giá trị là 0) ở cuối mảng, nó được tạo ra để chỉ điểm cuối cùng của một dãy các kí tự (kết thúc chuỗi kí tự). Phát minh này cũng được sử dụng bởi ngôn ngữ C: khi một xâu kí tự hằng như

```
"hello\n"
```

xuất hiện trong chương trình C, nó được lưu trữ như một mảng các kí tự chứa các kí tự của xâu kí tự đó và kết thúc với một kí tự `'\0'` để chỉ điểm kết thúc.

| h | e | l | l | o | \n | \0 |
|---|---|---|---|---|---|---|

Đặc tả `%s` biểu diễn trong hàm `printf` ứng với đối là một xâu kí tự. Hàm `copy` cũng dựa vào thực tế rằng dữ liệu vào của đối đó (`from`) được kết thúc bằng kí tự `'\0'`, và nó sao chép kí tự này vào vào trong đối ở đầu ra (`to`). (Mọi điều này có nghĩa rằng `'\0'` không là một phần trong mẫu văn bản)

Có một điều đáng được nhắc đến trong việc truyền dữ liệu rằng ngay cả một chương trình đơn giản nhỏ như thế này cũng gặp phải một số vấn đề về thiết kế. Ví dụ, hàm `main` sẽ làm điều gì khi nó gặp một dòng lớn hơn nhiều so với giới hạn của nó ? hàm `getline` hoạt động một cách an toàn, khi đó nó sẽ ngưng việc nhận vào khi mảng đã đầy, ngay cả khi không có kí tự chuyển dòng mới nào (`\n`) được thấy. Bằng việc kiểm tra độ dài và kí tự cuối cùng được trả về, hàm `main` có thể xác định khi dòng đó quá dài hay không, và rồi đối phó như nó mong muốn. Ngắn gọn hơn, chúng ta đã bỏ qua vấn đề này.

Chẳng có cách nào để cho `getline` biết trước được độ dài của dữ liệu vào có thể như thế nào, vì thế nó sẽ kiểm tra cho tới khi tới giới hạn. Mặt khác, `copy` đã biết sẵn xâu kí tự lớn như thế nào, vì thế chúng ta đã không thêm vào việc kiểm tra lỗi cho nó.

**Bài tập 1-16**.

Bài tập 1-17.

Bài tập 1-18.

Bài tập 1-.19

1.10 Biến ngoài và phạm vi

Các biến bên trong hàm main, như line, longest,…, là các biến cục bộ trong hàm main. Bởi vì chúng được khai báo ở bên trong hàm main, nên không có hàm nào khác được truy cập trực tiếp tới chúng. Điều này cũng đúng với các biến ở trong các hàm khác; ví dụ, biến I trong hàm getline không giống với biến I trong hàm copy. Mỗi biến cục bộ ở trong mỗi hàm chỉ tồn tại khi hàm đó được gọi đến, và biến mất ngay khai hàm đó kết thúc. Điều này lí giải gì sao các biến được gọi là các biến tự động, theo như thuật ngữ ở các ngôn ngữ khác. Chúng tôi sẽ sử dụng thuật ngữ tự động từ nay trở về sau đối với các biến cục bộ như thế này này. (Chapter 4 sẽ bàn về lớp lưu trữ static, nơi mà các biến cục bộ vẫn giữ lại giá trị giữa các lần gọi)

Bởi vì các biến tự động "tới" và "đi" tùy thuộc vào sự hiện diện của hàm, chúng không giữ lại giá trị của mình từ lần gọi này sang lần gọi khác, và phải được thiết lập rõ ràng trên mỗi hàm. Nếu nó không được thiết lập, nó sẽ chứa "rác" (một giá trị bất kì).

Một sự thay thế cho các biến tự động, điều này có thể định nghĩa các biến *toàn cục* (biến ngoài) cho tất cả các hàm, khi đó, các biến như vậy có thể được truy cập bằng tên bởi các hàm. Bởi vì các biến ngoài có thể truy cập được một cách toàn cục, chúng có thể được sử dụng thay cho các danh sách đối để giao tiếp dữ liệu giữa các hàm. Hơn nữa, bởi vì biến toàn cụ có thể tồn tại mãi, hơn là "tới" rồi "đi" như việc gọi và kết thúc của các hàm (các biến cục bộ trong các hàm), nên chúng sẽ giữ giá trị của mình ngay cả khi các hàm thiết lập cho nó kết thúc.

Một biến toàn cục phải được định nghĩa, chính xác là một lần, bên ngoài các hàm; điều này thiết lập vùng lưu trữ cho nó ở một bên. Biến này cần phải được khai báo ở mỗi hàm hàm nếu muốn truy cập tới nó; bao gồm cả kiểu của nó. Phần khai báo sẽ cần một câu lệnh extern rõ ràng. Để bàn về điều này cụ thể, chúng tôi sẽ viết lại chương trình dòng dài nhất ở phần trước với line, longest và max là các biến toàn cục. Điều này sẽ yêu cầu việc chỉnh sửa các lần gọi, khai báo, và thân của cả ba hàm.

```
#include <stdio.h>
#define MAXLINE 1000        /* độ dài lớn nhất có thể nhập vào */
int max;                    /* độ dài dòng lớn nhất */
char line[MAXLINE];         /* dòng hiện tại */
char longest[MAXLINE];      /* dòng dài nhất lưu ở đây */

int getline(void);
void copy(void);
/* in dòng dài nhất; phiên bản đặc biệt */
main()
{
      int len;                    /* độ dài dòng hiện tại */
      extern int max;
      extern char longest[];
      max = 0;
      while((len = getline()) > 0)
            if(len > max){
                  max = len;
                  copy();
            }
      if(max > 0)           /* kiểm tra nếu có dòng nào lớn nhất */
            printf("%s", longest);
```

```
        return 0;
}
/* getline: phiên bản đặc biệt */
int getline(void)
{
        int c, i;
        extern char line[];
        for(i = 0; i < MAXLINE-1
                && (c=getchar())!=EOF && c!='\n'; ++i)
                line[i] = c;
        if(c == '\n'){
                line[i] = c;
                ++i;
        }
        line[i] = '\0';
        return i;
}
/* copy: phiên bản đặc biệt */
void copy(void)
{
        int i;
        extern char line[], longest[];
        i=0;
        while((longest[i] = line[i]) != '\0')
                ++i;
}
```

Các biến toàn cục trong main, getline, và copy được định nghĩa bằng các câu lệnh đầu tiên trong ví dụ trên, chỉ rõ kiểu dữ liệu và tạo ra vùng lưu trữ được phân bổ cho chúng. Theo cú pháp, các định nghĩa bên ngoài cũng giống với các định nghĩa đối với các biến cục bộ, nhưng nó được thực hiện bên ngoài các hàm. Trước khi hàm có thể sử dụng được các biến ngoài này, tên của các bên này phải được biết đến bởi hàm này. Một cách để làm điều này là viết từ khóa extern ở phần khai báo trong hàm này; phần khai báo cũng giống như mọi lần chỉ có điều thêm từ khóa extern ở ngay đầu.

Trong những hoàn cảnh nhất định, phần khai báo extern có thể được bỏ qua. Nếu như phần định nghĩa của một biến ngoài diễn ra bên trong một tệp chính cùng với các hàm, thì không cần phải có phần khai báo extern trong mỗi hàm đó. Các phần khai báo extern trong hàm main, getline và copy như vậy là dư. Thực tế, việc thực hành thông thường sẽ đặt các định nghĩa biến ngoài ở đầu tệp chính, và bỏ qua các phần khai báo extern.

Nếu như chương trình được viết trên nhiều tệp, một biến ngoài được định nghĩa trong tệp 1 và biến đó được dùng với tệp 2, tệp 3 thì việc khai báo extern là rất cần thiết cho tệp 2 và tệp 3 để sử dụng. Bạn có thể thực hành điều này thường qua việc chia chương trình thành nhiều tệp riêng lẻ, rồi gọi các tệp đó vào trong một tệp bằng cách sử dụng *chỉ thị tiền xử lý* #include ở đầu mỗi tệp. Ví dụ ta thường hay sử dụng các hàm có sẵn trong các thư viên chuẩn như #include <stdio.h>. Vấn đề này sẽ được bàn rõ hơn ở chapter 4, và các thư viện chuẩn này ở chapter 7 và phụ lục B.

Vì phiên bản đặc biệt này của getline và copy không có đối, chúng ta có thể suy đoán ngay nguyên mẫu của nó sẽ có dạng

```
Int getline();

Void copy();
```

Nhưng điều này chỉ phù hợp với phiên bản cũ của ngôn ngữ C, và nó sẽ bỏ qua việc kiểm tra các đối của hàm; từ khóa void phải được thêm vào để chỉ rõ rằng danh sách đối rỗng . Chúng ta sẽ bàn về vấn đề này sâu hơn ở chapter 4.

Bạn nên chú ý rằng chúng ta đang sử dụng các từ “định nghĩa” và “khai báo” một cách cẩn thận khi chúng ta nói về các biến toàn cục ở trong phần này. “Định nghĩa” quy vào nơi mà biến được tạo ra hay khởi tạo nơi lưu trữ; “khai báo” quy vào các nơi mà

Thêm nữa, có một xu hướng để khiến mọi thứ trong tầm của một biến extern

# CHAPTER 2: **KIỂU DỮ LIỆU, TOÁN TỬ VÀ BIỂU THỨC**

Biến và hằng là các đối tượng dữ liệu cơ bản được thao tác trong một chương trình. Phần khai báo liệt kê các biến được sử dụng, xác định kiểu dữ liệu và có thể gán thêm các giá trị khởi đầu cho chúng. Một biểu thức bao gồm các biến, hằng và các toán tử sẽ tạo thành một giá trị mới cho biểu thức đó. Kiểu dữ liệu của một đối tượng định rõ kích thước dữ liệu của đối tượng này có thể nhận được và các toán tử nào có thể thực hiện với nó. Những điều trên là các chủ đề quan trọng của chapter này.

Chuẩn ANSI đã tạo ra một số thay đổi nhỏ và thêm vào một số bổ sung cho các kiểu dữ liệu cơ bản và biểu thức. Bây giờ có thêm kiểu dữ liệu `signed` và `unsigned` cho số nguyên, các kí hiệu cho các hằng số không dấu và các kí tự hằng của hệ thập lục phân. Các phép tính số thập phân có thể được thực hiện ở độ chính xác đơn; ngoài ra còn có thêm một kiểu số thực `long double` dành cho việc mở rộng độ chính xác kép. Các hằng xâu kí tự có thể được ghép nối trong lúc biên dịch. Kiểu dữ liệu liệt kê bây giờ cũng là một phần của ngôn ngữ C, chính thức hóa một tính năng lâu dài trong chương trình. Các đối tượng có thể được khai báo với `const`, để khiến chúng không thể bị thay đổi trong suốt quá trình biên dịch. Ngoài ra còn có các nguyên tắc chuyển đổi tự động trong các kiểu dữ liệu số sẽ được trình bày trong chapter này.

## 2.1 Tên biến

Mặc dù chúng tôi đã không nói ở **chapter 1**, nhưng có một số hạn chế trong việc đặt tên biến cũng như biểu tưởng hằng. Các tên biến được tạo bởi các kí tự chữ cái và chữ số; kí tự đầu tiên phải là một kí tự chữ cái. Dấu gạch dưới "_" cũng được coi là một kí tự chữ; điều này đôi khi tiện lợi để cải thiện cho việc đọc một tên biến dài. Không được bắt đầu tên biến bằng dấu gạch dưới, vì một số thủ tục trong thư viện thường dùng với những cái tên kiểu như vậy. Chữ hoa và chữ thường cũng được phân biệt, vì thế `x` và `X` là hai tên hoàn toàn khác nhau. Theo truyền thống thì khi thực hành ngôn ngữ C, ta dùng chữ thường đối với các biến và chữ hoa đối với các biểu tượng hằng.

Tên của các biến trong ngôn ngữ C có thể lên tới 31 kí tự. Đối với tên hàm hay biến toàn cục, con số này có thể sẽ ít hơn 31, bởi vì tên bên ngoài có thể được sử dụng bởi các hợp ngữ và các chương trình nạp khác, theo đó các những ngôn ngữ này không thể kiểm soát tên quá dài như vậy được. Đối với các tên bên ngoài chỉ được phép có 6 kí là đúng với tiêu chuẩn đặt ra. Các từ như `if`, `else`, `int`, `float`,…là các *từ khóa*: bạn không thể dùng chúng để đặt tên biến và chúng phải là các chữ cái thường.

Thật tốt nếu như chọn tên biến mà nó thể hiện được mục đích của biến đó, điều đó sẽ khiến cho việc đọc chương trình không bị rơi vào trạng thái "rối loạn tiêu hóa". Chúng ta nên theo phong cách đặt tên ngắn cho các biến cục bộ (như là `a`, `b`, `i`, `j`…), đặc biệt là chỉ số của các vòng lặp, tên dài hơn đối với các biến toàn cục (nên chỉ có 6 kí tự nếu có làm việc với các hợp ngữ như đã nói ở trên).

## 2.2 Kiểu dữ liệu và kích thước

Ngôn ngữ C chỉ có một vài kiểu dữ liệu cơ bản:

| | |
|---|---|
| `char` | chỉ có một byte bộ nhớ, dùng để lưu trữ một kí tự trong bộ kí tự cục bộ |
| `int` | số nguyên, đây là kiểu dữ liệu cơ bản trong máy tính |
| `float` | số thập phân với độ chính xác đơn |
| `double` | số thập phân với độ chính xác kép |

Thêm vào đó có một số kiểu khác có thể đi theo với các kiểu cơ bản này. Như `short` và `long` có thể đi theo kiểu `int` để tạo ra các số nguyên có cú pháp như sau:

```
short int sh;
long int counter;
```

Chúng ta có thể bỏ đi từ `int` trong khai báo trên, vì `int` được mặc định nếu không có nó.

Điều này có nghĩa rằng `short` và `long` chắc hẳn phải cung cấp kích thước bộ nhớ khác cho số nguyên trong thực hành; `int` thường sẽ có kích thước cơ bản trong một máy tính cụ thể. `short` thường là 16 bit, tức chiếm 2 byte bộ nhớ, `long` thì 32 bit, 4 byte bộ nhớ, còn `int` thì có thể là 16 hoặc 32 bit, 2 byte hoặc 4 byte. Kích thước như thế nào thì tùy thuộc vào mỗi trình biên dịch sẽ chọn cho bản thân nó, nhưng cả các số kiểu `short` và `int` chắc chắn sẽ có ít nhất 16 bit, 2 byte, kiểu `long` ít nhất sẽ là 32 bit, 4 byte, `short` chắc chắn sẽ nhỏ hơn `int`, và `int` thì chắc chắn sẽ nhỏ hơn `long`. Ví dụ thực tế nếu các bạn biện dịch bằng trình biên bịch Borlarnd C thì kiểu `int` sẽ có 2 byte bộ nhớ, còn ở các trình biên dịch cấp cao hơn như Visual studio sẽ có giá trị bộ nhớ là 4 byte.

Ngoài ra còn có thêm `signed` hoặc `unsigned` cũng có thể đi với kiểu `char` hoặc kiểu `int`. Các số kiểu `unsigned` luôn là các số dương hoặc bằng 0, và nó tuân thủ theo quy tắc của toán học về modulo $2^n$, với `n` là số bit trong kiểu đó. Ví dụ, nếu kiểu char có 8 bit, tức 1 byte, các biến kiểu `unsigned char` sẽ có giá trị từ 0 tới 255 ($2^8$ là 256, vì tính thêm cả 0 nên sẽ có khoảng từ 0 tới $2^8 - 1$), trong khi mà biến kiểu `signed char` có giá trị từ -128 tới 127 (`signed` chỉ kiểu các giá trị có dấu, tức là âm, và vì có khoảng âm, thế nên khoảng của `signed char` sẽ lấy $2^8/2$ cho hai khoảng âm, dương và tính thêm số 0). `signed` cũng như `int`, đây là mặc định của ngôn ngữ C, tức là bạn có thể không có nó khi khai báo biến. Có một điểm cần lưu ý là dù mặc định của `char` là `signed` hay `unsigned` thì nó phụ thuộc vào máy tính đó, nhưng các kí tự in ra màn hình luôn có giá trị dương.

Kiểu `long double` là kiểu dành cho các số thập phân có độ chính xác kép. Cũng như kiểu các số nguyên, kích thước bộ nhớ của các đối tượng này phụ thuộc vào cách thiết lập của nó được định nghĩa như thế nào; `float`, `double` hay `long double` có thể đại diện cho một, hai hoặc ba kích thước khác biệt.

Các thư viên chuẩn `<limits.h>` và `<float.h>` chứa biểu tượng hằng của những kích thước này, cùng với những đặc tính của máy tính và trình biên dịch. Chúng được đề cập ở **phụ lục B**.

**Bài tập 2-1**. Viết chương trình xác định kích thước giá trị của các biến `char`, `short`, `int` và `long`, cả `signed` và `unsigned` bằng việc in các giá trị thích hợp trong các thư viện chuẩn có đề cập ở trên hoặc bằng cách tính toán trực tiếp. Nâng cao hơn một chút: với bài toán đó nhưng hãy thử với các kiểu số thập phân.

## 2.3 Hằng

Một hằng số nguyên như `1234` có kiểu `int`. Một hằng kiểu `long` được viết thêm vào ở cuối kí tự `l` hoặc `L`, ví dụ như `123456789L`; một số nguyên quá lớn mà kiểu `int` không chứa được sẽ tự động

chuyển sang kiểu `long`. Các hằng số không dấu được viết ở cuối thêm một kí tự `u` hoặc `U` cho kiểu `unsigned int`, còn với `unsigned long` thì tương tự là hai kí tự `ul` hoặc `UL`.

Các hằng số thập phân thì có chứa thêm phần thập phân (như `123.4`) hoặc là cách viết số mũ khoa học (như `1e-2`) hoặc là cả hai; mặc định kiểu của chúng sẽ là `double`. Nếu có kí tự cuối như `f` hoặc `F` thì nó sẽ là hằng số thập phân kiểu `float`; `l` hoặc `L` thì tương tự là kiểu `long double`.

Giá trị cả một số nguyên có thể xác định theo *hệ bát phân* hoặc *thập lục phân* thay vì hệ thập phân. Nếu cho đầu một số nguyên kí tự chữ số 0 thì nó sẽ có dạng là *hệ bát phân*; còn nếu ở đầu các kí tự `0x` hoặc `0X` thì sẽ là *hệ thập lúc phân*. Ví dụ, giá trị hệ thập phân 31 có thể viết thành 037 cho hệ bát phân và `0x1f` hoặc `0X1F` cho hệ thập lục phân. Hằng số bát phân và thập lục phân có thể thêm kí tự `L` hoặc `U` ở cuối để thành kiểu lần lượt là `long` và `unsigned`: tương tự `0XFUL` là kiểu `unsigned long` với giá trị theo hệ thập phân là 15.

Một hằng kí tự là một số nguyên, để biểu diễn rõ kí tự nào thay vì giá trị số nguyên của nó thì ta ghi kí tự đó vào cặp dấu nháy đơn, ví dụ `'x'`. Giá trị của hằng kí tự là giá trị số của kí tự đó trong mặc định của bộ kí tự trong máy tính (đã nói ở phần trước). Ví dụ, theo bảng mã ASCII thì kí tự hằng của `'0'` có giá trị là 48, nó khác với giá trị 0 nếu không có dấu nháy kép. Nếu ta viết `'0'` thay vì con số 48, thì như vậy sẽ dễ đọc hơn bởi vì chương trình coi chúng là một. Các hằng kí tự tham gia vào các phép toán biểu thức cũng như các số nguyên khác, mặc dù chúng chỉ tường dùng nhất là khi so sánh với các kí tự khác.

Một số kí tự nhất đinh có thể được biểu diễn như một hằng kí tự hoặc hằng xâu kí tự như \n (xuống dòng); trông nó có vẻ là gồm hai kí tự nhưng máy tính chỉ xem nó là một kí tự. Thêm nữa, các kí tự hằng kiểu này có thể được xác định giá trị số của nó thông qua

```
'\ooo'
```

Trong đó `ooo` là ba chữ số của hệ bát phân (0 đến 7) hoặc

```
'\xhh'
```

Trong đó `hh` là một hoặc nhều các chữ số  trong hệ thập lục phân (0 đến 9, `a` đến `f`, `A` đến `F`) . Vì thế chúng ta có thể viết

```
#define VTAB '\013'   /* kí tự tab dọc trong ASCII */
#define BELL '\007'   /* kí tự chuông trong ASCII */
#define VTAB '\xb'    /* kí tự tab dọc trong ASCII */
#define BELL '\x7'    /* kí tự chuông trong ASCII */
```

Bảng đầy đủ của các hằng kí tự đặc biệt

| | | | |
|---|---|---|---|
| `\a` | kí tự chuông báo động | `\\` | kí tự \ |
| `\b` | backspace | `\?` | Kí tự ? |
| `\f` | sang trang | `\'` | kí tự ‘ |
| `\n` | xuống dòng mới | `\"` | kí tự “ |

| | | | |
|---|---|---|---|
| \r | về đầu dòng | \ooo | kí tự biểu diễn theo hệ bát phân |
| \v | tab dọc | \xhh | kí tự biểu diễn theo hệ thập lục phân |

Hằng kí tự '\0' biểu diễn kí tự *null* với giá trị là 0. '\0' thường được viết thay cho 0 để nhấn mạnh tính chất của kí tự trong một vài cách biểu diễn hoặc biểu thức, nhưng vẫn chỉ có giá trị là 0.

Một *biểu thức hằng* là một biểu thức chỉ có thành phần là các hằng. Nó thường xuất hiện ở những nơi có các phép tính với các số là các hằng, như là

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

hoặc

```
#define LEAP 1        /* trong năm nhuận */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Một hằng xâu kí tự là một dãy các kí tự hoặc có thể không có gì được bao bởi cặp nháy kép, như

```
"I am a string"
```

hoặc

```
""    /* một hằng xâu kí tự trống */
```

Cặp dấu nháy kép không được tính trong chuỗi kí tự này, nhưng phải cần để phân biệt với chuỗi. Nhưng nếu muốn nó được xuất hiện trong chuỗi thì ta sử dụng kí tự đặc biệt \". Các chuỗi hằng có thể nối tiếp nhau trong khi biên dịch:

```
"hello," " world"
```

cũng tương tự như

```
"hello, world"
```

Điều này tiện lợi cho việc tách các chuỗi dài trên nhiều dòng.

Thường thì các chuỗi hằng thực chất lại là các mảng kí tự. Mọi chuỗi kí tự đều có thêm một kí tự *null* '\0' ẩn ở cuối cùng, vì thế số lượng các kí tự mà ta thấy trong cặp ngoặc kép đó sẽ cần nhiều hơn 1 khi lưu trữ. Không có một giới hạn nào về số lượng các kí tự trong một chuỗi, nhưng chương trình phải quét một chuỗi để xác định độ dàng của nó. Hàm thư viên chuẩn strlen(s) trả về độ dài của chuỗi kí tự s, nhưng không bao gồm kí tự *null* '\0'. Sau đây là minh họa của hàm đó:

```
/* strlen: trả về độ dài của chuỗi s */
int strlen(char s[])
{
    int i;
    i = 0;
```

```
        while(s[i] != '\0')
            ++i;
        return i;
    }
```

Hàm `strlen` và các hàm về chuỗi khác được định nghĩa trong thư viện chuẩn `<string.h>`.

Cẩn thận khi phân biệt giữa một hằng kí tự và một chuỗi hằng gồm 1 kí tự: `'x'` không giống với `"x"`. `'x'` là một số nguyên, được dùng để biểu diễn giá trị số của kí tự `x` trong bộ kí tự của máy tính. Còn `"x"` là một mảng kí tự bao gồm kí tự `x` và kí tự `'\0'`.

Ngoài ra còn có một kiểu hằng nữa, đó là kiểu `enum` (kiểu liệt kê). Đây là kiểu liệt kê danh sách của các giá trị hằng số nguyên, ví dụ như

```
enum boolean {NO, YES};
```

Mặc định hằng đầu tiên có giá trị 0, sau đó là 1, và tiếp tục tăng lên như vậy, trừ khi ta chỉ rõ giá trị cho nó. Nếu tất cả các hằng không được xác định các giá trị cụ thể, các hằng đó sẽ tiếp tục tăng lên từ các hằng được xác định giá trị gần nhất, như ví dụ sau

```
enum escapes {BELL = '\a', BACKSPACE = '\b', TAB = '\t',
              NEWLINE = '\n'};
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL };
           /* khi đó FEB sẽ là 2, MAR là 3 và tiếp tục như vậy */
```

Các tên trong danh sách phải khác nhau nhưng các giá trị không cần phải khác nhau trong cùng một danh sách.

Kiểu `enum` này cùng cấp một cách tiện lợi để tạo danh sách các hằng cùng các giá trị có liên kết với nhau, đây là một cách tạo hằng khác ngoài việc sử dụng `#define` nhưng tiện hơn khi các giá trị sẽ được tự động tạo ra cho bạn. Ta có thể khai báo biến với kiểu `enum` này, các trình biên dịch không cần phải kiểm tra rằng biến đó có chứa một giá trị hợp lệ cho kiểu liệt kê này không. Tuy nhiên, các biến kiểu liệt kê này cung cấp cơ hội cho việc kiểm tra và thường sẽ tốt hơn là `#define`. Thêm nữa, một debugger có thể in ra các giá trị của các biến kiểu liệt kê này theo biểu tượng hằng của chúng.

### 2.4 Khai báo biến

Mọi biến đều phải được khai báo trước khi sử dụng, mặc dù một số các khai báo nhất định có thể được tạo liên tục bởi thuộc tính. Một khai báo xác định một kiểu, và nó chứa danh sách của một hay nhiều biến theo kiểu đó, ví dụ như

```
int lower, upper, step;
char c, line[1000];
```

Các biến có thể được tách ra thành nhiều dòng khai báo khác nhau nếu muốn; các kiểu khai báo trên có thể tách ra như sau

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Cách khai báo này sẽ  khiến chương trình trống trải hơn, nhưng nó sẽ rất tiện nếu ta thêm vào mỗi khai báo một chú thích để diễn tả mục đích cho từng biến.

Một biến cũng có thể khởi tạo giá trị trong lúc khai báo. Nếu ta viết tiếp bên cạnh biến đó dấu = và một giá trị hay một biểu thức thì biến đó sẽ được khởi tạo giá trị ngay lúc đó

```
Char esc = '\\';
int i = 0;
int limit = MAXLINE + 1;
float eps = 1.0e-5;
```

Nếu biến đó không phải là một biến tự động, thì gần khởi tạo gán giá trị chỉ được thực hiện duy nhất 1 lần, thường thì trước khi thực thi, và giá trị đó phải là một giá trị hằng số hay biểu thức hằng. Một biến tự động được khởi tạo ngay khi bắt đầu một hàm hoặc một khối lệnh; việc khởi tạo này có thể là bất cứ biểu thức nào. Các biến toàn cục và biến tĩnh nếu không được khởi tạo thì sẽ có giá trị mặc định là 0. Còn đối với các biến tự động sẽ chứa "rác" (tức là một giá trị bất kì) khi không được khởi tạo rõ ràng.

Từ khóa `const` có thể đi theo với các khai báo của mọi biến để xác định giá trị của các biến này không thể thay đổi. Đối với mảng thì các phần sử sẽ không thể thay đổi được.

```
const double e = 2.7878682342;
const char msg[] = "warning: ";
```

Các đối là mảng trong định nghĩa hàm cũng có thể dùng từ khóa `const` để khai báo, điều này có nghĩa hàm sẽ không thể thay đổi dữ liệu của mảng được truyền vào:

```
int strlen(const char[]);
```

Kết quả sẽ phụ thuộc vào hệ thống xử lí nếu như có một sự cố gắng được thực hiện để thay đổi một biến `const`.

## 2.5 Toán tử số học

Các toán tử số học có trong ngôn ngữ C là +, −, *, / và phép lấy phần dư %. Chú ý một điều rằng các phép chia số nguyên sẽ có kết quả là một số nguyên, tức nó sẽ bỏ đi phần thập phân. Với biểu thức

```
x % y
```

sẽ có kết quả là phần dư của phép chia hai số x và y, và vì thế nếu x chia hết cho y thì khi đó kết quả sẽ là 0. Ví dụ, năm nhuận là năm có thể chia hết cho 4 nhưng không chia hết cho 100, ngoại trừ năm đó chia hết cho 400 thì nó cũng là một năm nhuận. Vì thế ta có câu lệnh

```
if((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
      printf("%d is a leap year\n", year);
else
      printf("%d is not a leap year\n", year);
```

Toán tử % không thể sử dụng với các kiểu float hay double. Việc lược bỏ phần thập phân trong phép toán hai số nguyên của toán tử / và dấu của kết quả với toán tử % thì tùy thuộc vào máy tính đối với các toán hạng âm, cũng như việc đó được thực hiện overflow hay underflow. (overflow có nghĩa là kết quả của một phép toán có giá trị cao hơn độ lớn mà bộ nhớ máy tính có thể chứa, underflow có nghĩa ngược lại tương tự)

Các toán tử hai ngôi như *, /, % có cùng độ ưu tiên và cao hơn +, − nhưng lại thấp hơn đối với toán tử một ngôi + và −. Biểu thức phép toán sẽ được tính từ trái sang phải nếu chúng có các toán tử có cùng độ ưu tiên.

Bảng 2-1 ở cuối chapter này sẽ tóm tắt độ ưu tiên và tính liên kết của tất cả các toán tử.

## 2.6 Toán tử quan hệ và logic

Các toán tử quan hệ

```
>     >=    <     <=
```

đều có cùng độ ưu tiên. Độ ưu tiên thấp hơn là hai toán tử so sánh bằng:

```
==    !=
```

Các toán tử quan hệ này đều có độ ưu tiên thấp hơn các toán tử số học, cho nên ví dụ biểu thức `i < lim-1` sẽ như `i < (lim-1)`.

Thú vị hơn hết là các toán tử && và ||. Các biểu thức liên kết với nhau bởi && hoặc || được đánh giá từ trái qua phải, phép đánh giá này sẽ kết thúc ngay khi kết quả được biết là đúng hay sai. Hầu hết các chương trình C đề dựa vào các đặc tính này. Ví dụ, dưới đây là một câu lệnh lặp từ hàm getline mà ta đã viết ở **chapter 1**:

```
for(i = 0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
      s[i] = c;
```

Việc kiểm tra xem mảng `s` có còn trống hay không trước khi đọc một kí tự là rất cần thiết, vì thế điều kiện `i < lim-1` cần phải được để trước. Thêm nữa, nếu như điều kiện này sai, chúng ta sẽ không tiếp tục đọc một kí tự khác.

Tương tự, sẽ không may nếu như biến `c` được kiểm tra với `EOF` trước khi hàm `getchar` được gọi; vì vậy việc việc và gán cần phải diễn ra trước khi biến `c` được kiểm tra.

Độ ưu tiền của `&&` lớn hơn của `||`, và cả hai đều nhỏ hơn độ ưu tiên của các toán tử quan hệ và toán tử so sánh bằng, vì thế biểu thức điều kiện

```
i < lim-1 && (c = getchar()) != '\n' && c!= EOF
```

sẽ chẳng cần thêm các cặp ngoặc đơn giữa các biểu thức. Nhưng khi độ ưu tiên của `!=` lớn hơn toán tử `=`, cho nên cặp ngoặc đơn cần phải được viết thêm như sau

```
(c = getchar()) != '\n'
```

để khi đó biến `c` được gán giá trị từ việc đọc kí tự vào của hàm `getchar`, rồi với kiểm tra với '`\n`'.

Giá trị số của biểu thức quan hệ hay biểu thức logic sẽ mặc định là 1 nếu như nó đúng, và có giá trị 0 nếu nó sai.

Toán tử phủ định `!`, là toán tử một ngôi có nghĩa "không", sẽ chuyển một toán hạng với giá trị khác 0 thành 0, và thành 1 nếu giá trị đó là 0. Một cách sử dụng thông thường của toán tử `!` ở cấu trúc như

```
if(!valid)
```

hơn là dùng

```
if(valid == 0)
```

Hai cách dùng này là như nhau, điều này khó để quyết định kiểu nào tốt hơn, tùy thuộc vào bạn cũng như những lập trình viên. Cấu trúc `!valid` ở trên đọc hiểu là ("nếu không `valid`"), nhưng nếu sử dụng ở các biểu thức phức tạp hơn thì sẽ rất khó để hiểu.

**Bài tập 2-2**. Viết một vòng lặp tương đương với vòng lặp for có nhắc tới ở trên nhưng không sử dụng `&&` hoặc `||`.

## 2.7 Các nguyên tắc chuyển đổi kiểu dữ liệu

Khi một toán tử có các toán hạng có kiểu khác nhau, chúng sẽ được chuyển đổi thành một kiểu dữ liệu bởi một số nguyên tắc. Nói chung, trong một phép toán, nó sẽ tự động chuyển một toán hạng có kiểu dữ liệu thấp sang kiểu dữ liệu cao hơn mà không hề mất dữ liệu gì của toán hạng đó, như chuyển một số nguyên sang số thập phân trong phép toán như `f + i`. Không cho phép những kiểu biểu diễn hay biểu thức không có nghĩa như việc sử dụng kiểu `float` cho các chỉ số (như chỉ số của mảng). Các biểu thức có thể làm mất đi dữ liệu, thông tin như việc gán kiểu số nguyên lớn cho kiểu số nguyên thấp hơn, hay là gán kiểu số thực cho kiểu số nguyên, điều này trình biên dịch có thể đưa ra lời cảnh báo (warning), nhưng chương trình vẫn có thể hoạt động.

Một biến kiểu `char` cũng chỉ là một số nguyên nhỏ, vì thế các biến kiểu này có thể tự do được dùng trong các biểu thức số. Điều này cho phép sự linh hoạt đáng kể trong một số kiểu thay đổi kí tự. Sự hoạt động của hàm `atoi` sau đây là một ví dụ điển hình, hàm này có chức năng chuyển đổi một chuỗi kí tự chữ số thành một số nguyên tương đương.

```
/* atoi: chuyển chuỗi s thành một số nguyên */
int atoi(char s[])
{
      int i, n;
      n = 0;
      for(i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
            n = 10 * n + (s[i] - '0');
      return n;
}
```

Như chúng ta đã bàn ở **chapter 1**, biểu thức

```
s[i] - '0'
```

cho kết quả là một con số ứng với kí tự được chứa trong `s[i]`, bởi vì giá trị của các kí tự `'0'`, `'1'`,… đều liền kề nhau và liên tiếp tăng dần.

Một ví dụ nữa của việc chuyển đổi từ kiểu `char` sang `int` là ở hàm `lower`, nó chuyển một kí tự chữ cái in hoa sang in thường theo bảng mã ASCII. Nếu như kí tự đó không phải là một kí tự in hoa, hàm `lower` sẽ trả về kí tự không đổi ban đầu.

```
/* lower: chuyển kí tự in hoa c sang thường; theo bảng mã ASCII */
int lower(int c)
{
      if(c >= 'A' && c <= 'Z')
            return c + 'a' - 'A';
      else
            return c;
}
```

Hàm này chỉ hoạt động theo bảng mã ASCII bởi vì các kí tự chữ cái này liên tiếp nhau theo bảng chữ cái Latinh, bảng chữ in thường liền trước bảng chữ in hoa, cho nên giá trị của chúng cũng sẽ liên tiếp tăng dần, hiểu logic một chút thì khoảng cách giữa các chữ in thường ứng với chữ cái đó in hoa sẽ là như nhau, ví dụ `'a' - 'A'` sẽ có cùng giá trị với `'b' - 'B'`, tương tự cho tới `Z`. Hàm này thì lại không hoạt động đúng với bộ kí tự EBCDIC.

Thư viện chuẩn <ctype.h>, được mô tả chi tiết ở **phụ lục B**, định nghĩa một nhóm các hàm cung cấp các phương thức thử nghiệm và chuyển đổi cho kí tự mà không phục thuộc vào bộ kí tự. Ví dụ, hàm `tolower(c)` trả về kí tự in thường của biến `c` nếu `c` là kí tự in hoa, vì thế ta có thể sử dụng hàm `tolower` này thay cho hàm `lower` đã trình bày ở trên. Ngoài ra biểu thức điều kiện

```
c >= '0' && c <= '9'
```

có thể được thay bằng

```
isdigit(c)
```

Chúng ta sẽ sử dụng các hàm trong thư viên <ctype.h> từ bây giờ.

Có một điểm khá là tinh tế về việc chuyển đổi các kí tự sang các số nguyên. Các ngôn ngữ không thể xác định rõ liệu các biến kiểu `char` có giá trị âm hay dương. Khi một biến kiểu `char` được chuyển đổi sang kiểu `int`, liệu nó có thể trả về là một số âm không nhỉ ? Câu trả lời có thể khác biệt giữa các máy tính, điều này phản ánh sự khác nhau về kiến trúc. Ở một số máy, một biến kiểu `char` nếu có bit tận cùng là 1 thì sẽ được chuyển đổi sang một số nguyên âm ("sign extension"). Một số khác, một biến kiểu `char` được chuyển đổi thành `int` bằng việc thêm vào các số 0 ở bit tận cùng bên trái, vì thế nó luôn có giá trị dương. (sign extension có nghĩa là phép toán tăng số lượng các bit của một số nhị phân trong khi vẫn giữ dấu và giá trị của số đó)

Trong ngôn ngữ C, nó đảm bảo rằng các kí tự trong bộ kí tự chuẩn của máy tính sẽ không bao giờ có giá trị âm, vì thế các kí tự này sẽ luôn có giá trị dương trong các biểu thức. Nhưng các mẫu bit được chứa trong các biến kí tự có thể xuất hiện âm ở trên một số máy tính, song lại là dương ở một số khác. Để linh động, hãy chỉ định rõ kiểu `signed` hoặc `unsigned` nếu không có dữ liệu kí tự nào được chứa trong biến kiểu `char`.

Các biểu thức quan hệ như `i > j` và các biểu thức logic được liên kết bởi `&&` và `||` được định nghĩa nhận giá trị 1 nếu đúng, 0 nếu sai. Vì thế phép gán

```
d = c >= '0' && c <= '9'
```

thiết lập giá trị 1 cho `d` nếu `c` là một chữ số, và giá trị 0 nếu không phải. Tuy nhiên, các hàm như `isdigit` có thể trả về bất kì giá trị khác 0 nào nếu đúng. Ở phần kiểm tra điều kiện của `if`, `while`, `for`,… "đúng" có nghĩa là "khác không", vì thế điều này chẳng có gì khác biệt.

Một toán tử hai ngôi như `+` hay `*` có hai toán hạng có kiểu khác nhau, kiểu nào "nhỏ hơn" sẽ được tự động chuyển thành kiểu "lớn hơn" trước khi biểu thức được thực thi. Kết quả sẽ có kiểu "lớn hơn". Các nguyên tắc chuyển đổi được trình bày tỉ mỉ ở **phần 6** của **phụ lục A**. Sau đây sẽ là một số nguyên tắc "không chính thức" cơ bản khá đầy đủ, nếu không tính đến kiểu `unsigned`:

Nếu một trong hai có kiểu `long double`, toán hạng còn lại sẽ chuyển thành kiểu `long double`.

Không thì nếu một trong hai có kiểu `double`, toán hạng còn lại sẽ chuyển thành kiểu `double`.

Không thì nếu một trong hai có kiểu `float`, toán hạng còn lại sẽ chuyển thành kiểu `float`.

Không thì chuyển `char` và `short` thành kiểu `int`.

Còn nếu một trong hai là kiểu `long`, toán hạng còn lại sẽ chuyển thành kiểu `long`.

Chú ý rằng các số kiểu `float` trong một biểu thức không tự động chuyển thành kiểu `double`; đây là một sự thay đổi từ phiên bản trước đó của ngôn ngữ C. Nói chung, các hàm toán học trong thư viện `<math.h>` đều sẽ dùng kiểu có độ chính xác kép (kiểu `double`). Thường thì khi sử dụng các mảng với kiểu số thực, ta sẽ hay dùng kiểu `float` hơn, bởi lí do duy nhất là nó sẽ tiết kiệm thời gian thực thi cũng như bộ nhớ chương trình, khi mà kiểu có độ chính xác kép (`double`, `long double`) sẽ ít được sử dụng hơn bởi nó cần rất nhiều tài nguyên bộ nhớ.

Các quy tắc chuyển đổi sẽ phức tạp hơn khi có toán hạng kiểu `unsigned` trong các biểu thức. Vấn đề ở đây là việc so sánh giữa các giá trị âm và dương sẽ phụ thuộc vào máy tính, bởi vì chúng tùy thuộc vào kích thước của nhiều kiểu số nguyên khác nhau. Ví dụ, giả sử kiểu `int` là 16 bit (2 byte) và `long` thì 32 bit (4 byte). Vậy thì `-1L < 1U`, bởi vì `1U`, đây là kiểu `int`, sẽ được chuyển thành kiểu `signed long`. Nhưng `-1L > 1UL`, bởi vì `-1L` sẽ được chuyển thành `unsigned long` và vì thế nó sẽ được coi là số dương lớn hơn.

Việc chuyển đổi cũng diễn ra ở các biểu thức gán; giá trị ở bên phải sẽ được chuyển thành kiểu của bên trái, đây cũng là kiểu của kết quả gán.

Một kí tự được chuyển thành một số nguyên, hoặc bằng "sign extension" hoặc không, đã được mô tả chi tiết ở trên.

Các kiểu số nguyên lớn hơn được chuyển đổi thành các kiểu nhỏ hơn hoặc thậm chí là kiểu `char` bằng việc hạ các bit bậc cao xuống. Như vậy trong

```
int i;
char c;
i = c;
c = i;
```

giá trị của `c` không đổi. Điều này vẫn đúng dù "sign extension" có liên quan ở đây hay không. Tuy nhiên, việc đảo ngược thứ tự gán có thể sẽ làm mất thông tin, dữ liệu.

Nếu biến `x` là kiểu `float` và `i` là kiểu `int`, thì khi `x = i` và `i = x` cả hai sẽ xảy ra sự chuyển đổi kiểu; từ `float` sang `int` sẽ làm mất phần thập phân. Khi kiểu `double` được chuyển thành `float`, giá trị được làm tròn hay là được lược bớt thì cái này là phụ thuộc vào hệ thống xử lí.

Khi đối của một lời gọi hàm là một biểu thức, việc chuyển đổi kiểu cũng diễn ra khi các đối được truyền cho các hàm. Khi không có nguyên mẫu hàm, kiểu `char` và `short` sẽ chuyển thành `int`, và `float` sẽ thành `double`. Điều này lí giải vì sao chúng tôi đã khai báo đối của hàm là kiểu `int` và `double` ngay cả khi hàm đó được gọi với kiểu `char` và `float`.

Cuối cùng, ta sẽ bàn đến một kiểu chuyển đổi nữa có thể "bắt ép" mọi biểu thức theo như kiểu mình muốn, đây như một toán tử một ngôi và được gọi là *ép kiểu* (cast). Ta có cấu trúc sau

(*tên-kiểu*) *biểu thức*

*biểu thức* sẽ được chuyển đổi thành kiểu ở phần *tên-kiểu* theo như các nguyên tắc đã nêu ở trên. Ý nghĩa chính xác của toán tử ép kiểu này là chỉ khi *biểu thức* được tính toán và xác định giá trị, khi đó toán tử ép

kiểu sẽ thực hiện với giá trị đó. Ví dụ, hàm `sqrt` nhận đối có kiểu là `double`, và sẽ trả về "rác" nếu như ta vô tình truyền vào một kiểu khác. (`sqrt` được định nghĩa trong thư viện `<math.h>`) Vì thế nếu `n` là một số nguyên, chúng ta có thể dùng

```
sqrt((double) n)
```

để chuyển đổi giá trị của `n` thành `double` trước khi truyền cho `sqrt`. Chú ý rằng toán tử ép kiểu tạo ra *giá trị* của `n` có kiểu lớn hơn, chứ `n` không bị thay đổi, nó vẫn là kiểu nguyên. Toán tử ép kiểu này có cùng độ ưu tiên với các toán tử một ngôi khác được tóm tắt ở bảng cuối chapter này.

Nếu các đối được khai báo bởi một nguyên mẫu hàm, khai báo này sẽ gây ra việc tự động ép kiểu cho bất cứ đối nào khi hàm được gọi. Như vậy, nếu cho hàm `sqrt` một nguyên mẫu như:

```
double sqrt(double);
```

thì lời gọi hàm

```
root2 = sqrt(2);
```

sẽ ép cho số nguyên 2 thành kiểu `double` có giá trị 2.0 mà không cần tới toán tử ép kiểu.

Thư viện chuẩn có bao gồm một "thiết bị" có chức năng tạo ra giả ngẫu nhiên các số nguyên cùng với một hàm bổ trợ cho nó trong việc tạo ra *seed* để giúp "thiết bị" này hoạt động; điều này sẽ minh họa cho một toán tử ép kiểu:

```
unsigned long int next = 1;

/* rand: trả về ngẫu nhiên số nguyên từ 0...32767 */

int rand(void)

{

      next = next * 1103515245 + 12345;

      return (unsigned int)(next/65536) % 32768;

}

/* srand: tạo ra seed */

void srand(unsigned int seed)

{

      next = seed;

}
```

**Bài tập 2-3**. Viết hàm `htoi(s)` có chức năng chuyển đổi một chuỗi các chữ số hệ thập lục phân (bao gồm cả 0x hoặc 0X) tương đương thành một số có giá trị kiểu nguyên.

## 2.8 Toán tử tăng và giảm

Ngôn ngữ C cung cấp hai toán tử đặc biệt giúp tăng hoặc giảm giá trị của biến. Toán tử ++ giúp tăng toán hạng đó thêm 1, trong khi toán tử −− sẽ giảm đi 1. Chúng ta đã sử dụng toán tử này như trong câu lệnh

```
if(c == '\n')
      ++nl;
```

Một điều nữa là cả hai toán tử ++ và −− có thể được sử dụng theo tiền tố (ở phía trước biến, như ++n) hoặc hậu tố (ở phía sau biến, như n++). Cả hai trường hợp này đều đều tăng giá trị n thêm 1. Nhưng cách biểu diễn ++n sẽ tăng biến n lên 1 *trước khi* giá trị của biến đó được sử dụng, trong khi n++ thì sẽ tăng biến n thêm 1 *sau khi* giá trị của nó được sử dụng. Điều này có nghĩa là ở một bối ảnh nào đó giá trị này được sử dụng thì ++n và n++ là khác nhau. Giả sử n có giá trị là 5, và sau đó

```
x = n++;
```

sẽ gán cho x giá trị là 5, nhưng

```
x = ++n;
```

thì sẽ gán cho x giá trị là 6. Ở cả hai trường hợp, biến n đều sẽ trở thành 6. Các toán tử này chỉ có thể sử dụng cho biến; nếu có biểu thức như (i+j)++ thì nó sẽ không hợp lệ.

Ở một bối cảnh khác khi không có việc sử dụng giá trị của biến nào đó, tức chỉ cần như

```
if(c == '\n')
      ++nl;
```

thì dù toán tử tăng/giảm được sử dụng ở hậu tố hay tiền tố thì nó cũng như nhau. Nhưng ở một số bài toán, những toán tử này được ứng dụng sử dụng một cách đặc biệt khôn khéo. Ví dụ, hãy thử xem xét hàm sqeeze(s,c) sau đây, nó có chức năng xóa hết các kí tự c trong chuỗi s.

```
/* sqeeze: xóa hết các kí tự c trong s */
void squeeze(char s[], int c)
{
      int i, j;
      for(i = j = 0; s[i] != '\0'; i++)
            if(s[j] != c)
                  s[j++] = s[i];
      s[j] = '\0';
}
```

Mỗi khi không có kí tự c nào xuất hiện, nó sẽ được sao chép vào vị trí j hiện tại, và chỉ khi đó thì biến j mới được tăng lên để chứa kí tự kế tiếp. Điều này cũng giống hệt với câu lệnh sau

```
if(s[j] != c){
      s[j] = s[i];
      j++;
}
```

Một ví dụ khác cũng tương tự như một cấu trúc câu lệnh ở trong hàm `getline` mà chúng ta đã viết ở **chapter 1**,

```
if(c == '\n'){
      s[i] = c;
      ++I;
}
```

chúng ta có thể thay thế bằng một cấu trúc câu lệnh khác hay hơn

```
if(c == '\n')
      s[i++] = c;
```

Ví dụ thứ ba sau đây chúng tôi sẽ viết lại hàm `strcat(s,t)`, hàm này có chức năng nối chuỗi `t` tiếp tục vào cuối của chuỗi `s`. `strcat` sẽ giả sử rằng `s` có đủ chỗ trống để chứa sự kết hợp này. Bởi đây là hàm do chúng ta viết lại, cho nên nó sẽ không trả lại giá trị gì; còn hàm `strcat` chuẩn trong thư viện `<string.h>` đã được định sẵn, nó sẽ trả về một con trỏ trỏ đến chuỗi kí tự đã được kết hợp.

```
/* strcat: nối chuỗi t vào cuối chuỗi s; s phải đủ lớn */
void strcat(char s[], char t[])
{
      int i, j;
      while(s[i] != '\0')         /* tìm vị trí cuối của s */
            i++;
      while ((s[i++] = t[j++]) != '\0')      /* sao chép t */
            ;
}
```

Mỗi kí tự sẽ được sao chép từ chuỗi `t` sang `s`, tiền tố `++` được dùng cho biến `i` và `j` đảm bảo rằng chúng là vị trí cho các kí tự tiếp theo được thông qua bởi vòng lặp.

**Bài tập 2-4**. Viết một phiên bản khác của hàm `sqeeze(s1,s2)` để xóa mỗi kí tự trong chuỗi `s1` giống với các kí tự trong chuỗi `s2`.

**Bài tập 2-5**. Viết hàm `any(s1,s2)`, có chức năng kiểm tra xem có bất kì kí tự nào trong chuỗi `s2` mà xuất hiện đầu tiên trong chuỗi `s1`, trả về vị trí đầu tiên của kí tự đó, hoặc là trả về -1 nếu `s1` không chứa bất kì kí tự nào trong `s2`. (Hàm thư viên chuẩn `strpbrk` cũng làm công việc tương tự này nhưng trả về con trỏ trỏ tới vị trí đó)

## 2.9 Toán tử thao tác bit

Ngôn ngữ C cung cấp sáu toán tử phục vụ cho các thao tác trên bit; các thao tác này chỉ có thể sử dụng cho các toán tử nguyên như char, short, int, long, có thể có dấu hoặc không.

| | |
|---|---|
| & | thao tác bit AND |
| \| | thao tác bit OR |
| ^ | thao tác bit XOR |
| << | dịch chuyển trái |
| >> | dịch chuyển phải |
| ~ | thao tác bit NOT (một ngôi) |

Toán tử AND, &, thường được dùng trong các thao tác được gọi là thao tác *mặt nạ*, dùng để tháo mặt nạ của các bộ bit (tức là đổi 1 hành 0); ví dụ

```
N = n & 0177;
```

Sẽ thiết lập hết bộ 7 bit bên phải của n thành 0.

Toán tử OR, |, được dùng để bật bộ bit (tức là đổi 0 thành 1):

```
X = x | SET_ON;
```

Câu lệnh này sẽ thiết lập cho các bit 0 thành bit 1, nếu là bit 1 thì không đổi.

Toán tử OR, ^, sẽ thiết lập là 1 nếu như hai bit có giá trị khác nhau, là 0 nếu nhưng chúng cùng giá trị.

Một điều cần phải phân biệt rõ giữa toán tử bit & và | với các toán tử logic && và || khi mà chúng cũng cho kết quả đánh giá là một giá trị chân lí (đúng hoặc sai). Ví dụ, nếu x là 1 và y là 2, thì khi đó x & y có kết quả là 0 trong khi x && y lại là 1.

Các toán tử dịch chuyển bit << và >> thực hiện dịch chuyển trái và phải cho toán hạng ở vế trái bằng một lượng các vị trí bit được cho ở bên vế phải, giá trị này phải dương. Cho nên x << 2 sẽ dịch chuyển giá trị của x sang trái hai vị trí bit, các bit trống sẽ thay bằng 0; điều này tương đương với việc nhân thêm 4. Việc dịch chuyển sang phải của một số không âm luôn thay các bit trống là các số 0. Còn việc dịch chuyển phải của các số âm sẽ thay bằng các bit dấu trong một số máy tính và bit 0 với một số khác.

Toán tử một ngôi ~ …; nó chuyển đổi các bit 1 thành 0 và ngược lại. Ví dụ,

```
X = x & ~077
```

Thiết lập các bit cuối của x thành 0. Chú ý rằng x & ~077

Xxx

Sau đây sẽ là một minh họa về một số toán tử thao tác bit, xem xét hàm getbits(x,p,n) trả về bit thứ n (dịch sang phải) của x, bắt đầu từ vị trí p. Giả sử vị trí bit 0 ở cuối cùng bên phải và n, p là các giá trị dương. Ví dụ, getbits(x,4,3)

Bài tập 2-6. Viết một hàm setbits(x,p,n,y) trả về

2.10 Toán tử gán và các biểu thức

Các biểu thức như

I = I +2

Có biến ở vế trái được lặp lại ở vế phải có thể được viết theo mẫu

I += 2

Toán tử += ở đây được gọi là toán tử gán.

Hầu hết các toán tử hai ngôi đều có toán tử gán tương ứng như mẫu op=, trong đó op là một trong các toán tử

+ - * / % << >> & ^ |

Nếu $expr_1$ và $expr_2$ là các biểu thức, thì

$expr_1$ op= $expr_2$

Sẽ tương đương với

$expr_1$ = ($expr_1$) op ($expr_2$)

Trừ khi $expr_1$ được tính toán 1 lần. Chú ý cặp ngoặc đơn xung quanh $expr_2$, lấy ví dụ:

X *= y+1

Có nghĩa

X = x * (y + 1)

Chứ không phải

X = x * y + 1

Thêm một ví dụ nữa, ta có hàm bitcount để đếm số lượng các bit 1 trong đối số nguyên.

/* bitcount: đếm các bit 1 trong x */

Int bitcount(unsigned x)

```
{
        Int b;
        For(b = 0; x != 0; x >>=1)
                If(x & 01)
                        B++;
        Return b;
}
```

Việc khai báo đối x kiểu unsigned là để đảm bảo rằng nếu nó được dịch chuyển sang phải, các bit trống sẽ được lấp đầy bằng các số 0, không phải là các bit dấu (bit 1), bất kể chương trình chạy trên máy tính nào.

Các toán tử gán

# CHAPTER 3: **ĐIỀU KHIỂN LUỒNG**

Các câu lệnh điều khiển luồng của một ngôn ngữ lập trình xác định rõ thứ công việc tính toàn nào được thực hiện. Chúng ta đã gặp hầu hết các cấu trúc điều khiển luồng thông dụng trong các ví dụ trước đây; bây giờ chúng ta sẽ hoàn tất "bộ điều khiển" này và tìm hiểu một cách kĩ càng, chính xác những cấu trúc mà ta đã bàn trước đó.

## 3.1 Câu lệnh và khối lệnh

Một biểu thức như `x = 0` hay `i++` hay `printf(…)` trở thành một *câu lệnh* khi nó kết thúc bởi dấu chấm phẩy như

```
x = 0;
i++;
printf(…);
```

Trong ngôn ngữ C, dấu chấm phẩy là dấu hiểu kết thúc của một câu lệnh, khác với công dụng ngăn cách ở trong ngôn ngữ Pascal.

Cặp ngoặc nhọn `{` và `}` được dùng để nhóm các câu lệnh chung với nhau trở thành một *khối lệnh*, điều này cũng tương đương nó như một câu lệnh lớn. Cặp ngoặc nhọn mà bao quanh các câu lệnh của một hàm là một ví dụ điển hình; một số ví dụ khác như cặp ngoặc nhọn bao quanh các câu lệnh sau `if`, `else`, `while` hoặc `for`. (Các biến có thể được khai báo bên trong bất cứ khối lệnh nào; chúng ta sẽ nói về điều này *ở* **chapter 4**) Chú ý là dù một khối lệnh cũng được coi như một câu lệnh lớn, nhưng không có dấu chấm phẩy ở ngoặc nhọn cuối của khối lệnh.

### 3.2 If-else

Câu lệnh `if-else` được dùng để biểu diễn các quyết định có điều kiện. Cú pháp như sau

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

phần `else` có thể có hoặc không. *Biểu thức* được tính toán; nếu nó đúng (khi đó nó sẽ có giá trị khác 0), *câu lệnh 1* được thực thi. Nếu nó sai (biểu thức có giá trị 0) và nếu có phần `else` thì *câu lệnh 2* sẽ thực thi thay thế.

Vì câu lệnh `if` đơn giản chỉ kiểm tra giá trị của một biểu thức, nên ta có thể viết tắt

```
if (biểu thức)
```

thay vì viết

```
if (biểu thức != 0)
```

Đôi khi điều này sẽ rõ ràng; một vài trường hợp khác có thể khó hiểu.

Bởi vì phần `else` của một câu lệnh `if-else` là tùy chọn, nên sẽ có một chút mơ hồ khi một `else` bị bỏ đi trong một cấu trúc liên tục các `if` lồng nhau. Ở trường hợp này thì phần `else` sẽ được kết hợp với `if` gần nhất trước đó. Ví dụ trong cấu trúc này,

```
if(n > 0)
        if(a > b)
                z = a;
        else
                z = b;
```

phần `else` sẽ đi cùng với `if` ở bên trong như chúng tôi đã thụt đầu dòng cho dễ nhận biết. Nếu đó không phải là điều bạn muốn, tức `else` đi với `if` bên ngoài, thì bạn cần phải dùng cặp ngoặc nhọn như sau:

```
if(n > 0){
        if(a > b)
                z = a;
}
else
        z = b;
```

Vấn đề này sẽ rất là nguyên hiểm đối với những tình huống như sau:

```
if(n >= 0)
      for(i = 0; i < n; i++)
            if(s[i] > 0){
                  printf(…);
                  return i;
            }
else        /* SAI */
      printf("error – n is negative\n");
```

Việc thụt đầu dòng như trên cho bạn thấy điều bạn muốn, nhưng trình biên dịch sẽ không hiểu như bạn và kết hợp phần `else` với phần `if` bên trong. Kiểu lỗi này có thể khó để nhận ra; nên sử dụng cặp ngoặc nhọn để phân rõ ý định của bạn với chương trình khi có các `if` lồng nhau.

Nhân tiện, hãy chú ý rằng có một dấu chấm phẩy sau `z = a` trong

```
if(n > 0)
      if(a > b)
            z = a;
      else
            z = b;
```

Điều này thể hiện mặt cấu trúc theo sau `if` là một câu lệnh, và biểu thức "`z = a;`" luôn kết thúc bằng dấu chấm phẩy.

### 3.3 Else-if

Cấu trúc

`if` (*biểu thức*)

*Câu lệnh*

`else if` (*biểu thức*)

*Câu lệnh*

`else if` (*biểu thức*)

*Câu lệnh*

`else if` (*biểu thức*)

*Câu lệnh*

`else`

*Câu lệnh*

thường hay xuất hiện nên nó đáng được dành ra một phần riêng biệt để bàn tới. Một cấu trúc liên tục các câu lệnh `if` này thường dùng để xây dựng nhiều quyết định có điều kiện. Các *biểu thức* được tính toán theo thứ tự; nếu bất cứ một *biểu thức* nào đúng, thì *câu lệnh* được kết hợp với nó sẽ được thực thi, và sau đó nó sẽ kết thúc chuỗi mắt xích này. Đương nhiên mỗi câu lệnh trên có thể là một câu lệnh đơn hay một khối lệnh bên trong cặp ngoặc nhọn.

Phần `else` cuối cùng có chức năng mặc định thực hiện *câu lệnh* sau đó nếu không có bất kì điều kiện nào được thỏa mãn. Đôi khi sẽ chẳng có hành động nào được mặc định diễn ra; trong trường hợp này thì

`else`

*Câu lệnh*

có thể được lược bỏ, hoặc nó có thể được dùng cho việc kiểm tra lỗi để bắt lấy một điều kiện "không có khả thi".

Để minh họa cho một quyết định ba chiều, dưới đây là một hàm tìm kiếm nhị phân quyết định một giá trị `x` cụ thể có xuất hiện trong mảng `v` đã được sắp xếp hay không. Các phần tử của `v` phải được sắp xếp thứ tự tăng. Hàm này trả về vị trí (một số trong khoảng từ 0 tới `n-1`) nếu như `x` có xuất hiện trong `v`, và -1 nếu không có.

Tìm kiếm nhị phân đầu tiên so sánh giá trị `x` nhập vào với phần tử giữa mảng `v`. Nếu `x` nhỏ hơn giá trị phần tử giữa, việc tìm kiếm sẽ tập trung vào nửa dưới của mảng, ngược lại nếu `x` lớn hơn. Ở một trong hai trường hợp đó, bước tiếp theo sẽ so sánh `x` với phần tử giữa của một nữa mảng đã chọn trước đó. Quá trình này sẽ chia khoảng cách của khu vực tìm kiếm thành hai phần cho tới khi giá trị đó được tìm thấy, hoặc khoảng đó trống.

```
/* binsearch: tìm x trong v[0] <= v[1] <=…<= v[n-1] */
 int binsearch(int x, int v[], int n)
{
      int low, high, mid;
      high = n-1;
      while(low <= high){
            mid = (low+high)/2;
            if(x < v[mid])
                  high = mid - 1;
            else if (x > v[mid])
                  low = mid + 1;
```

```
            else        /* nếu trùng */
                return mid;
        }
        return -1;      /*không trùng*/
    }
```

Quyết định cơ bản sẽ được thông qua việc kiểm tra `x` có nhỏ hơn, lớn hơn hay bằng với phần tử `v[mid]` hay không ở mỗi bước; điều này được diễn ra một cách tự nhiên trong `else-if`.

**Bài tập 3-1**. Hàm tìm kiếm nhị phần của chúng ta đã để hai điều kiện kiểm tra bên trong vòng lặp, trong khi chỉ cần một là đủ. Hãy viết một phiên bản khác với duy nhất một điều kiện kiểm tra bên trong thân vòng lặp và thử đo lường sự khác nhau trong việc chạy chương trình.

### 3.4 Switch

Câu lệnh `switch` là một câu lệnh có tính quyết định đa chiều được phân chia thành nhiều nhánh, nó căn cứ vào giá trị của  biểu thức để chọn một trong nhiều quyết định ứng với mỗi nhánh.

```
switch (biểu thức) {
        case nhãn:    câu lệnh
        case nhãn:    câu lệnh
        default:      câu lệnh
}
```

Mỗi `case` có *nhãn* có thể là một hằng số nguyên một hay biểu thức hằng số nguyên. Nếu một `case` có *nhãn* giá trị đúng với giá trị của *biểu thức* ở `switch`, các câu lệnh ứng với `case` đó được thực thi. Các *nhãn* của các `case` phải có giá trị khác nhau. Khi không có bất cứ *nhãn* giá trị nào của các `case` thỏa mãn với giá trị của *biểu thức* `switch`, thì các câu lệnh ở `default` sẽ được thực thi. `default` là tùy chọn; tức nếu như không có phần này, thì sẽ chẳng có viêc gì được thực hiện nếu không có bất cứ `case` nào có *nhãn* thỏa mãn. Các dòng `case` và `default` có thể trình bày theo bất cứ thứ tự nào, không nhất thiết phải theo như trên.

Ở **chapter 1** chúng ta đã viết một chương trình đếm số lần xuất hiện của các con số, khoảng trống, và các kí tự khác bằng việc sử dụng cấu trúc `if… else if… else`. Dưới đây là một chương trình như vậy nhưng sử dụng câu lệnh `switch`:

```
#include <stdio.h>
main()  /* đếm số lượng chữ số, khoảng trắng và các kí tự khác */
{
    int c, i, nwhite, nother, ndigit[10];
```

```
        nwhite = nother = 0;
        for(i = 0; i < 10; i++)
              ndigit[i] = 0;
        while((c=getchar()) != EOF){
              switch(c){
              case '0': case '1': case '2': case '3': case '4':
              case '5': case '6': case '7': case '8': case '9':
                    ndigit[c-'0']++;
                    break;
              case ' ':
              case '\n':
              case '\t':
                    nwhite++;
                    break;
              default:
                    nother++;
                    break;
              }
        }
        printf("digits =");
        for(i = 0; i < 10; i++)
              printf(" %d", ndigit[i]);
        printf(", white space = %d, other = %d\n", nwhite, nother);
        return 0;
  }
```

Câu lệnh `break` sẽ lập tức ngưng việc thực thi và thoát ra khỏi `switch`. Bởi vì dù đã tìm thấy `case` có nhãn thỏa mãn giá trị biểu thức `switch`, sau khi các câu lệnh của `case` này thực hiện xong, nó sẽ tiếp tục kiểm tra tiếp các `case` tiếp theo, điều này là không cần thiết. `break` và `return` là hai câu lệnh thông dụng nhất để thoát khỏi một câu lệnh `switch`. Câu lệnh `break` cũng có thể được dùng để thoát ra khỏi các vòng lặp `while`, `for` và `do-while`, chúng ta sẽ bàn thêm trong chapter này.

Việc liên tục kiểm tra các `case` kể cả khi đã có một trường hợp thỏa điều kiện này có hai mặt. Ở mặt tốt, nó cho phép một số các `case` có thể được cùng gắn với một công việc, như việc kiểm tra các chữ số

ở chương trình trên là một ví dụ. Nhưng nó cũng có thể dựa vào mỗi câu lệnh `break` ở mỗi `case` để ngắn việc "chảy" liên tục xuống các `case` khác. Mặt khác, nó khiến cho chương trình không được "mạnh mẽ", ở những chương trình lớn hơn, việc không tối ưu sẽ có thể khiến cho việc chạy chương trình trở nên khó khăn hơn. Cho nên hãy cẩn trọng cho vấn đề này khi sử dụng `switch`.

Để có một hình thức tốt, hãy để `break` ở ngay sau `default` ngay cả khi dù nó không cần thiết. Một ngày nào đó nếu có một `case` nào khác được đặt ở cuối cùng, việc này có thể sẽ cứu rỗi linh hồn bạn lẫn chương trình.

**Bài tập 3-2**. Viết một hàm `escape(s,t)` có chức năng chuyển đổi các kí tự xuống dòng và tab hiện diện thành `\n` và `\t` từ chuỗi `t` sao chép sang chuỗi `s`, hãy dùng `switch`. Viết một hàm như thế theo hướng ngược lại, chuyển các kí tự `\n` và `\t` thành đúng chức năng của nó.

### 3.5 Vòng lặp while và for

Chúng ta đã gặp vòng lặp `while` và `for` rồi. Trong cấu trúc

`while` (*biểu thức*)

*câu lệnh*

*biểu thức* sẽ được tính toán. Nếu như nó khác không, câu lệnh sẽ được thực thi và biểu thức sẽ lại được tính toán tiếp tục. Vòng lặp này sẽ tiếp tục cho tới khi biểu thức có giá trị 0, tiếp đó các câu lệnh sau vòng lặp sẽ tiếp tục thực thi.

Câu lệnh `for`

`for` (*biểu thức 1*; *biểu thức 2*; *biểu thức 3*)

*câu lệnh*

cũng tương đương với

*biểu thức 1*;

`while` (*biểu thức 2*){

*câu lệnh*

*biểu thức 3*;

}

còn có thêm câu lệnh `continue` để sử dụng trong các vòng lặp này, sẽ được mô tả chi tiết ở **phần 3.7**.

Theo cấu trúc thì ba phần trong `for` đều là các *biểu thức*. Thông thường nhất, *biểu thức 1* và *3* là các phép gán hoặc lời gọi hàm và *biểu thức 2* sẽ là một biểu thức quan hệ. Ba phần này có thể không có, nhưng dấu chấm phẩy giữa các phần vẫn phải hiện diện. Nếu *biểu thức 1* và *3* không có, thì bây giờ vòng lặp `for` này sẽ giống với vòng lặp `while`. Nếu phần kiểm tra điều kiện, *biểu thức 2*, không có, thì điều kiện này sẽ luôn có giá trị đúng, ví dụ như

```
for( ; ; ){
        …
}
```

sẽ là một vòng lặp "vô hạn", nó vẫn có thể được phá vỡ ở trong thân theo nhiều cách thông qua các câu lệnh như `break` hoặc `return`.

Dùng `while` hay `for` thì phần lớn vẫn phụ thuộc vào tùy trường hợp cái nào sẽ thích hợp hơn. Ví dụ trong

```
while((c=getchar()) == ' ' || c == '\n' || c == '\t')
        ;
```

không có phần khởi tạo giá trị hay bước nhảy, vì thế `while` sẽ trình bày nó một cách tự nhiên, rõ ràng hơn là `for`.

Vòng lặp `for` sẽ thích hợp hơn khi có phần *khởi tạo giá trị* và *bước nhảy*, vì nó sẽ khiến cho chương trình sáng sủa hơn nhiều so với việc dùng `while`. Ví dụ điển hình như

```
for(i = 0; i < n; i++)
        …
```

giống với việc truy cập `n` phần tử đầu của một mảng nào đó, cũng tương tự với vòng lặp DO của ngôn ngữ Fortran hay `for` của Pascal. Cấu trúc vòng lặp for của ngôn ngữ C không mấy hoàn hảo, tuy nhiên, phần chỉ số và giới hạn của vòng lặp này có thể được thay đổi bên trong thân của nó, và chỉ số biến `i` vẫn giữ nguyên giá trị ngay cả khi vòng lặp này kết thúc bởi bất kì lí do nào. Các bộ phận trong vòng lặp này có thể là bất cứ biểu thức gì tùy ý hay bất cứ cách biểu diễn nào, cho nên nó không bị giới hạn bởi việc phải tuân thủ theo quy tắc cấp số cộng như hai ngôn ngữ kia. Dù sao thì vẫn là một phong cách lập trình tồi nếu như ta cho các biểu thức không liên quan gì với nhau ở phần khởi tạo và bước nhảy trong vòng lặp `for`. Điều đó sẽ khiến cho việc điều khiển vòng lặp trở nên khó khăn hơn, và mất tính liên kết giữa chúng.

Một ví dụ khác lớn hơn, đây là một phiên bản khác của hàm `atoi` có chức năng chuyển đổi một chuỗi thành một số nguyên tương ứng. Phiên bản này tổng quát hơn phiên bản ở **chapter 2**; nó sẽ bỏ qua các khoảng trống và nhận các dấu `+, −` nếu có. (Ở **chapter 4** có hàm `atof`, có chức năng giống với `atoi` nhưng chuyển chuỗi thành số thực)

Cấu trúc của chương trình này được mô tả như sau:

*bỏ qua các khoảng trống nếu có*

*nhận vào các dấu nếu có*

*nhận phần số nguyên rồi chuyển đổi nó*

Mỗi bước sẽ một phần riêng biệt. Toàn bộ quá trình sẽ kết thúc nếu gặp phải một kí tự đầu tiên khác với kí tự chữ số.

```
#include <ctype.h>
```

```
/* atoi: chuyển chuỗi s thành số nguyên; phiên bản 2 */
int atoi(char s[])
{
      int i, n, sign;
      for(i = 0; isspace(s[i]); i++)   /*bỏ qua khoảng trắng*/
            ;
      sign = (s[i] == '-') ? -1 : 1;
      if(s[i] == '+' || s[i] == '-')         /* bỏ qua dấu */
            i++;
      for(n = 0; isdigit(s[i]); i++)
            n = 10 * n + (s[i] - '0');
      return sign * n;
}
```

Thư viện chuẩn cũng cung cấp một hàm khác kĩ lưỡng hơn là `strtol` cho việc chuyển các chuỗi thành số nguyên lớn kiểu long; xem **phần 5** của **phụ lục B**.

Việc phần được điều khiển của vòng lặp được quy về một chỗ riêng là rất cần thiết khi mà có nhiều các vòng lặp được lồng nhau. Hàm tiếp theo sau đây là một giải thuật sắp xếp mảng của các số nguyên được phát minh vào năm 1959 bởi D. L. Shell. Ý tưởng cơ bản của thuật toán này là ở giai đoạn đầu, các phần tử cách xa nhau được so sánh, thay vì so sánh các phần tử liền kề nhau như ở các thuật toán đổi chỗ trực tiếp. Điều này có xu hướng loại bỏ lượng xáo trộn lớn một cách nhanh chóng, vì thế ở các giai đoạn sau sẽ làm ít công việc hơn. Khoảng cách giữa các phần tử được so sánh dần dần được giảm về 1, là thời điểm mà việc sắp xếp sẽ trở thành phương pháp đổi chỗ liền kề một cách hiệu quả.

```
/* shellsort: sắp xếp v[0]…v[n-1] theo thứ tự tăng dần */
void shellsort(int v[], int n)
{
      int gap, i, j, temp;
      for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                  for(j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap){
                        temp = v[j];
                        v[j] = v[j+gap];
                        v[j+gap] = temp;
```

```
                    }
    }
```

Có tất cả ba vòng lặp lồng nhau. Vòng lặp ngoài cùng điều khiển khoảng cách giữa hai phần tử được so sánh, co dần lại theo `n/2` cho tới khi nó bằng 0. Vòng lặp ở giữa duyệt khắp các phần tử. Vòng lặp trong cùng so sánh mỗi cặp phần tử mà nó được ngăn cách bởi `gap` và trao đổi nếu như nó không đúng theo thứ tự. Vì `gap` sau cùng sẽ được giảm về 1, tất cả các phần tử sau đó cũng sẽ được sắp xếp theo đúng thứ tự. Chú ý vòng lặp `for` ngoài cùng đã cho ta thấy được cái ý tổng quát mà ta đã bàn trước đó, nó điều khiển khá tốt các vòng lặp `for` bên trong trong khi cấu trúc của nó không theo quy tắc về cấp số cộng.

Một toán tử cuối cùng trong ngôn ngữ C là toán tử *comma* (dấu phẩy), thường sẽ được sử dụng nhiều trong câu lệnh `for`. Một cặp các biểu thức được ngăn cách bởi toán tử này được tính toán từ trái sang phải, kiểu dữ liệu và kết quả sẽ là kiểu và kết quả của toán hạng bên phải. Như vậy trong một câu lệnh `for`, nhiều biểu thức có thể được đặt trong các phần của `for`, ví dụ như việc truy cập hai phần tử song song. Điều này được minh hoạt trong hàm `reverse(s)` có chức năng đảo ngược chuỗi `s`.

```
#include <string.h>

/* reverse: đảo ngược chuỗi s */

void reverse(char s[])

{

      int c, i, j;

      for(i = 0, j = strlen(s)-1; i < j; i++, j--){

            c = s[i];

            s[i] = s[j];

            s[j] = c;

}
```

Các dấu phẩy ngăn cách các đối của hàm, các biến trong phần khai báo,… đều không phải toán tử *comma* và không đảm bảo rằng việc tính toán sẽ theo thứ tự từ trái sang phải.

Các toán tử *comma* nên được hạn chế sử dụng. Việc sử dụng tốt nhất là cho các cấu trúc có quan hệ với nhau một cách kiên cố, như trong vòng lặp `for` của hàm `reverse`, và trong các macro, nơi mà các bước tính toán có thể được biểu diễn trên cùng một biểu thức. Một biểu thức *comma* cũng có thể kết hợp cho việc trao đổi các phần tử trong `reverse`, khi mà việc trao đổi này có thể được viết như sau:

```
for(i = 0, j = strlen(s)-1; i < j; i++, j--)

      c = s[i], s[i] = s[j], s[j] = c;
```

**Bài tập 3-3**. Viết một hàm `expand(s1,s2)` có chức năng mở rộng cách viết tắt như `a-z` ở trong chuỗi `s1` tương đương thành một dãy chữ cái `abz…xyz` trong chuỗi `s2`. Cho phép cả chữ lẫn số, và hãy chuẩn bị cho các trường hợp như `a-b-c`, `a-z0-9` và `-a-z`.

### 3.6 Vòng lặp do-while

Như chúng ta đã bàn ở **chapter 1**, vòng lặp `while` và `for` kiểm tra điều kiện kết thúc ở trên đầu. Ngược lại với nó, vòng lặp thứ ba này sẽ kiểm tra điều kiện ở dưới cùng sau khi thực hiện các công việc ở thân; thân của vòng lặp này luôn luôn được thực hiện ít nhất một lần. Đó là vòng lặp `do-while`.

Cấu trúc của vòng lặp này là

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

*Câu lệnh* được thực thi và sau đó *biểu thức* mới được tính toán. Nếu nó đúng, câu lệnh sẽ được thực hiện một lần nữa và tiếp tục như vậy. Khi biểu thức có giá trị sai, vòng lặp sẽ kết thúc.

Kinh nghiệm cho thấy `do-while` thường được sử dụng ít hơn `while` và `for`. Dù sao, theo thời gian nó đã dần trở nên có giá trị hơn, như trong hàm `itoa` sau đây có chức năng chuyển đổi một số nguyên sang một chuỗi (ngược lại với `atoi`). Công việc sẽ phức tạp hơn một chút, bởi vì cách dễ nhất để lấy các chữ số đó ra từ một số nguyên là lấy ngược lại từ đuôi lên đầu. Vì thế chúng tôi đã chọn cách tạo ra chuỗi kí tự ngược của con số đó rồi sau đó dùng hàm `reserve` đảo ngược chuỗi đó lại.

```
/* itoa: chuyển số nguyên n thành các kí tự trong chuỗi s */
void itoa(int n, char s[])
{
      int i, sign;
      if((sign = n) < 0)          /* sign */
            n = -n;               /* chuyển n sang dương */
      i = 0;
      do{               /* tạo ra các chữ số ngược thứ tự */
            s[i++] = n %10 + '0'; /* lấy chữ số tiếp theo */
      }while((n/=10) > 0);        /* xóa chữ số đó đi */
      if(sign < 0);
            s[i++] = '-';
      s[i] = '\0';
      reserse(s);
}
```

Câu lệnh lặp `do-while` này là cần thiết, hoặc ít nhất là tiện lợi, như ở ví dụ trên, khi đó ít nhất một kí tự số sẽ được lưu vào trong mảng kí tự `s`, ngay cả khi nếu `n` là 0. Chúng tôi cũng đã sử dụng cặp dấu ngoặc nhọn bao quanh dòng lệnh duy nhất của vòng lặp `do-while`, mặc dù điều này là không cần thiết, nhưng nó sẽ rõ hơn đối với người mới bắt đầu.

**Bài tập 3-4**. Trong cách biểu diễn phần bù của 2, phiên bản về hàm itoa của chúng ta không thể đáp ứng được với số nguyên âm lớn nhất, khi đó n chỉ tới -2 xxx

**Bài tập 3-5**. Viết hàm itob(n,s,b) có chức năng chuyển đổi số nguyên n chuỗi kí tự biểu diễn theo hệ b rồi lưu vào chuỗi s. Cụ thể ví dụ nếu ta có itob(n,s,16) thì sẽ chuyển n thành chuỗi kí tự biểu diễn theo hệ thập lục phân rồi lưu vào chuỗi s.

**Bài tập 3-6**. Viết một phiên bản khác của hàm itoa nhận vào ba đối thay vì hai như trên. Đối thứ ba là

### 3.7 Break và continue

Đôi khi rất tiện lợi nếu như có thể thoát khỏi một vòng lặp ở giữa chừng khi cần thiết hơn là phải đợi đến khi điều kiện của vòng lặp đó sai. Câu lệnh `break` này sẽ cung cấp cho ta phương thức đó trong các vòng lặp như `for`, `while`, `do-while` cũng như trong `switch` mà ta đã gặp trong chapter này. Một câu lệnh `break` sẽ khiến cho vòng lặp gần nhất chứa nó hoặc `switch` kết thúc ngay lập tức.

Hàm tiếp theo sau đây, tên là `trim`, có chức năng xóa các khoảng trống, tab và kí tự chuyển dòng ở cuối một chuỗi, sử dụng câu lệnh `break` để thoát vòng lặp nếu như không có khoảng trống, tab hay kí tự chuyển dòng nào được thấy ở cuối chuỗi.

```
/* trim: xóa các khoảng trống, tab và kí tự chuyển dòng */
int trim(char s[])
{
      int n;
      for(n = strlen(s)-1; n >= 0; n--)
            if(s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
                  break;
      s[n+1] = '\0';
      return n;
}
```

Hàm `strlen` trả về chiều dài của một chuỗi. Vòng lặp `for` bắt đầu chạy từ phía cuối của chuỗi rồi quét ngược lại cho tới khi gặp được kí tự đầu tiên không phải là khoảng trống hay tab hay là kí tự chuyển dòng. Vòng lặp sẽ kết thúc khi gặp được một trong số chúng, hoặc là khi `n` giảm xuống âm (khi đó cả chuỗi đã được quét hết). Bạn nên kiểm tra lại rằng cách chạy này vẫn đúng ngay cả khi chuỗi không có gì hoặc chỉ chứa các kí tự trống.

Câu lệnh `continue` cũng có quan hệ với `break`, nhưng ít được sử dụng hơn; nó khiến cho vòng lặp `for`, `while` hoặc `do-while` gần nhất lặp lại từ đầu. Trong vòng lặp `while` và `do-while`, điều này có nghĩa phần điều kiện kiểm tra sẽ được thực thi ngay lập tức một lần nữa; trong vòng lặp `for`, việc điều khiển vòng lặp sẽ chuyển sang phần *bước nhảy* ngay lập tức. Câu lệnh `continue` chỉ được sử dụng trong các vòng lặp, không với `switch`. Nhưng nếu `switch` ở trong một vòng lặp thì câu lệnh `continue` vẫn hợp lệ nếu nó ở trong `switch`, khi đó nếu `continue` được thực thi, nó sẽ ảnh hưởng đến vòng lặp chưa `switch` đó.

Dưới đây là một ví dụ của một vòng lặp có `continue`, nó có chức năng truy cập vào các phần tử dương trong mảng `a`; các phần tử âm sẽ bị bỏ qua.

```
For(i = 0; i < n; i++){
      if(a[i]<0)       /*bỏ qua các phần tử âm*/
            continue;
      …
}
```

### 3.8 Câu lệnh goto và nhãn

Ngôn ngữ C cung cấp cấu trúc `goto` và các *nhãn* giúp bạn có thể "lạm dụng sự vô hạn" trong chương trình. Thực sự thì cấu trúc này chẳng bao giờ là cần thiết cả vì chúng ta có thể viết các chương trình một cách dễ dàng mà không cần đến nó. Vì vậy chúng tôi đã không dùng đến nó trong cuốn sách này.

Tuy nhiên, có một số tình huống mà nó vẫn có thể xuất hiện. Thường thì dùng để kết thúc một cấu trúc lồng phức tạp, như là thoát ra khỏi hai hay nhiều vòng lặp cùng một lúc. Câu lệnh `break` lại không thể làm được điều này khi nó chỉ thoát ra được vòng lặp gần nó nhất. Cho nên:

```
for(…)
      for(…){
            …
            if(rác)
                  goto error;
      }
…
error:
      dọn dẹp
```

Cách thức hoạt động của nó đơn giản y như tên của nó, như trong cách tổ chức trên, khi gặp lệnh `goto`, chương trình sẽ lập tức nhảy tới *nhãn* mà `goto` đã chỉ tới, ở đây là `error`, chương trình sẽ thoát hết các vòng lặp mà bắt đầu thực thi tiếp từ nhãn `error`.

Cách tổ chức như thế này sẽ tiện lợi nếu như không có phần code được viết để giải quyết các lỗi có thể xảy ra ở một số chỗ trong cấu trúc phức tạp đó.

Một nhãn cũng giống như một tên biến, và nó kết thúc bằng dấu hai chấm. Nó có thể được gắn với bất cứ câu lệnh nào trong cùng một hàm với `goto`. Một nhãn có thể ở bất cứ đâu trong một hàm. Đó là lí do vì sao nó có thể được "lạm dụng sự vô hạn" cho bất cứ công việc gì.

Một ví dụ khác, hãy xem xét vấn đề của việc xác định hai mảng `a` và `b` có cùng chung một phần tử hay không.

```
      for(i = 0; i < n; i++)
            for(j = 0; j < m; j++)
                  if(a[i] == b[j])
                        goto found;
      /* không thấy phần tử chung nào */
      …
found:
      /* tìm thấy */
      …
```

Hoặc nếu ta không cần dung cấu trúc `goto`, với một chút biến đổi, ta có thể hình thành code của bài toán này theo một ví dụ khác

```
      found =0;
      for(i = 0; i < n && !found; i++)
            for(j = 0; j < m && !found; j++)
                  if(a[i] == b[j])
                        found ==1;
      ff(found)
            /* tìm thấy */
            …
      else
            /* không thấy phần tử chung nào */
            …
```

Với một số ví dụ ngoại lệ như ở phần này, các code dựa vào cấu trúc `goto` thường sẽ rất khó hiểu và chỉnh sửa hơn code không chứa nó. Mặc dù chúng tôi không có ý kiến độc đoán về vấn đề này, nhưng cấu trúc này nên được sử dụng hạn chế, hoặc không bao giờ đụng đến thì hơn.