LÊ HOÀI BẮC
NGUYỄN THANH NGHỊ

Kỹ năng lập trình

NHÀ XUẤT BẢN
KHOA HỌC VÀ KỸ THUẬT

LÊ HOÀI BẮC – NGUYỄN THANH NGHỊ

KỸ NĂNG LẬP TRÌNH

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
HÀ NỘI

$$\frac{\text{60-6T7.3}}{\text{KHKT-05}}\ \text{6-287-04}$$

LỜI NÓI ĐẦU

Dã bao giờ bạn phải...

tốn thời gian cài đặt một thuật toán sai chưa?

sử dụng cấu trúc dữ liệu quá phức tạp?

kiểm chứng chương trình nhưng lại không nhận ra được một lỗi dễ dàng trông thấy?

tốn cả một ngày trời để tìm ra một lỗi mà lẽ ra chỉ cần trong vòng 5 phút?

cần cài đặt chương trình thực thi nhanh hơn gấp 3 lần và dùng ít bộ nhớ hơn?

cố gắng để chuyển một chương trình chạy trên máy tính lớn sang máy tính cá nhân (hoặc ngược lại) chưa?

sửa chương trình của người khác?

viết lại một chương trình mà bạn không hiểu nó?

Những điều này thường xuyên xảy ra đối với lập trình viên. Nhưng giải quyết được những vấn đề này thường tỏ ra khó khăn bởi vì những chủ đề như kiểm chứng, gỡ rối, tính khả chuyển, tốc độ thực thi chương trình, những cách thức thiết kế khác nhau, và phong cách lập trình – những vấn đề thực tế liên quan đến lập trình thường không phải là trọng tâm của khoa học máy tính hoặc của những khóa học lập trình. Hầu hết các lập trình viên học được những điều này một cách ngẫu nhiên và tình cờ thông qua kinh nghiệm, và một số người chưa bao giờ học được những điều đó.

Trong một thế giới rộng lớn với nhiều quan hệ phức tạp, cùng với các công cụ, ngôn ngữ và các hệ thống thường xuyên thay đổi, và áp lực nặng nề để càng có nhiều thêm mọi thứ, người ta có thể không thấy được những nguyên lý cơ bản - đó là đơn giản, rõ ràng, tổng quát – tạo nên nền

tăng để xây dựng được những phần mềm tốt. Và mọi người cũng có thể bỏ qua những lợi ích như tự động hóa quá trình phát triển phần mềm do các công cụ và các ký hiệu mang lại.

Hướng tiếp cận trong quyển sách này là dựa trên những nguyên lý cơ bản và có liên hệ mật thiết với nhau này; chúng có thể được áp dụng trong mọi mức độ của quá trình xử lý, tính toán. Đó là ***tính đơn giản****: giúp chương trình ngắn gọn và dễ quản lý;* ***tính rõ ràng****: đảm bảo dễ hiểu cho cả người lẫn máy;* ***tính tổng quát****: chương trình có thể thực hiện được trong một phạm vi rộng lớn và vẫn thích ứng tốt khi có những tình huống mới xảy ra; và* ***tính tự động hóa****: giải phóng con người thoát khỏi những công việc bình thường thông qua máy tính. Bằng cách nhìn công việc lập trình từ nhiều ngôn ngữ lập trình khác nhau, từ nhiều thuật toán và cấu trúc dữ liệu thông qua việc cải tiến quá trình thiết kế, gỡ rối, kiểm chứng và tốc độ thực thi chương trình, chúng tôi muốn trình bày những quan niệm tổng quát mà không phụ thuộc vào bất kỳ ngôn ngữ lập trình, hệ điều hành và mô hình lập trình nào.*

Quyển sách này muốn trình bày các vấn đề thực tế, chia sẻ những kinh nghiệm, và đưa ra những giải pháp giúp lập trình viên ở mọi cấp độ làm việc hiệu quả và năng suất hơn. Nó được trình bày cho nhiều loại độc giả khác nhau. Nếu bạn là sinh viên đã tham dự một hay hai khóa học lập trình và mong muốn trở thành một lập trình viên tốt hơn, quyển sách sẽ giúp bạn mở rộng một vài chủ đề mà bạn không có điều kiện để học ở nhà trường. Nếu bạn viết chương trình cho một phần công việc của bạn nhưng cần sự hỗ trợ của các hoạt động khác ngoài việc lập trình thì quyển sách này giúp bạn viết chương trình hiệu quả hơn. Nếu bạn là một lập trình viên chuyên nghiệp mà không có đủ điều kiện tiếp cận những chủ đề này ở trường học hoặc muốn ôn lại kiến thức đã học, hoặc bạn là một nhà quản lý dự án phần mềm muốn định hướng nhân viên của bạn đi đúng hướng thì quyển sách này là một tài liệu có giá trị.

Hy vọng rằng những lời khuyên trong quyển sách này sẽ giúp bạn viết được những chương trình tốt hơn. Một yêu cầu duy nhất là bạn đã thực

hiện lập trình, tốt nhất là C, C++ hoặc Java. Dĩ nhiên là càng có nhiều kinh nghiệm thì càng dễ tiếp thu, bởi lẽ chỉ trong vòng 21 ngày, không thể trở thành chuyên gia từ một người mới học. Những lập trình viên trên Unix và Linux sẽ thấy một vài ví dụ quen thuộc hơn những người lập trình trên Windows và Macintosh, tuy nhiên lập trình viên trên bất kỳ môi trường nào cũng sẽ thấy nhiều điều làm cho cuộc sống trở nên dễ chịu hơn.

Quyển sách được trình bày thành chín chương, mỗi chương sẽ tập trung vào một lĩnh vực chính của công việc lập trình.

Chương 1 trình bày về phong cách lập trình. Phong cách lập trình tốt sẽ đóng vai trò quan trọng trong việc lập trình nên được chọn để trình bày trước. Các chương trình được viết tốt bao giờ cũng tốt hơn những chương trình viết dở bởi lẽ chúng có ít lỗi hơn, dễ gỡ rối và dễ chỉnh sửa hơn. Do vậy nghĩ đến phong cách lập trình trước tiên sẽ rất quan trọng. Chương này cũng giới thiệu một chủ đề quan trọng trong việc lập trình tốt: cách sử dụng đặc ngữ phù hợp với ngôn ngữ đang sử dụng.

Thuật toán và cấu trúc dữ liệu là chủ đề của Chương 2. Đây chính là phần cốt lõi trong các môn học về khoa học máy tính và cũng là một phần chính yếu trong các khóa học lập trình. Vì phần lớn độc giả đều quen thuộc với phần này nên chúng tôi sẽ trình bày cô đọng về thuật toán và cấu trúc dữ liệu được sử dụng hầu hết trong các chương trình. Những thuật toán và cấu trúc dữ liệu phức tạp hơn thường được phát triển dựa trên những nền móng đã có sẵn cho nên cần nắm vững những kiến thức cơ bản trước.

Chương 3 mô tả cách thiết kế và cài đặt của một chương trình nhỏ dùng để mô phỏng các vấn đề thuật toán và cấu trúc dữ liệu trong môi trường thực tế. Chương trình này được cài đặt trong bằng 5 ngôn ngữ, thực hiện so sánh các phiên bản đó để thấy được cách thức xử lý cấu trúc dữ liệu trong mỗi phiên bản và cách thức diễn đạt cũng như tốc độ thực thi chương trình trên những ngôn ngữ khác nhau sẽ khác nhau như thế nào.

Phương thức giao tiếp giữa người dùng, giữa chương trình và giữa các thành phần của chương trình là phần cơ bản trong lập trình. Nhiều

chương trình thành công được là nhờ vào phương thức giao tiếp được thiết kế và cài đặt tốt. Chương 4 sẽ trình bày quá trình phát triển một thư viện nhỏ dùng để phân tích một định dạng dữ liệu được sử dụng rộng rãi. Ví dụ này tuy nhỏ nhưng đã cho thấy nhiều vấn đề liên quan đến thiết kế phương thức giao tiếp: tính trừu tượng, che giấu thông tin, quản lý tài nguyên, và xử lý lỗi.

Mặc dù chúng ta cố gắng viết chương trình đúng ngay từ lần đầu tiên nhưng lỗi và kế đến là việc gỡ rối là những điều không thể tránh khỏi. Chương 5 sẽ đưa ra những chiến lược và chiến thuật để công việc gỡ rối được thực hiện có hệ thống và hiệu quả. Ở đây sẽ đề cập đến những dấu hiệu xảy ra của những lỗi thông dụng và tầm quan trọng của việc định vị những lỗi đó.

Kiểm chứng là nỗ lực đem lại sự đảm bảo hợp lý rằng chương trình hoạt động đúng và vẫn còn chạy đúng khi nó phát triển. Trọng tâm của Chương 6 là việc kiểm chứng thủ công hoặc tự động có hệ thống. Các phép kiểm chứng điều kiện biên sẽ giúp kiểm soát được những yếu điểm tiềm ẩn của chương trình. Tự động hóa và các khung kiểm chứng giúp mở rộng công việc kiểm chứng mà tốn ít công sức.

Máy tính ngày càng chạy nhanh, đồng thời trình biên dịch cũng ngày càng chạy tốt cho nên nhiều chương trình ngày càng chạy nhanh hơn rất nhiều so với trước đây. Tuy nhiên, cũng có những chương trình chạy rất chậm, hoặc sử dụng nhiều bộ nhớ, hoặc vừa chạy chậm vừa tốn bộ nhớ. Chương 7 trình bày một cách có trình tự hướng tiếp cận nhằm giải quyết vấn đề sử dụng tài nguyên có hiệu quả để chương trình vẫn còn chạy đúng và ổn định sau khi được cải tiến.

Chương 8 sẽ đề cập về tính khả chuyển. Các chương trình thành công là những chương trình vẫn có thể tồn tại đủ lâu khi có sự thay đổi về môi trường, hoặc là chúng phải được chuyển sang những hệ thống mới, phần cứng mới, hoặc một quốc gia khác. Mục đích của tính khả chuyển là làm giảm số lượng công việc của quá trình bảo trì chương trình xuống bằng cách giảm thiểu lượng thay đổi cần thiết để đáp ứng môi trường mới.

Công việc tính toán được thực hiện trên nhiều ngôn ngữ khác nhau, không chỉ trên các ngôn ngữ đa dụng mà còn trên những ngôn ngữ đặc thù, trên những lĩnh vực hẹp hơn. Chương 9 sẽ đưa ra một vài ví dụ về tầm quan trọng của các ký hiệu trong công việc tính toán và cũng cho thấy rằng chúng có thể được sử dụng để làm đơn giản hóa các chương trình, hướng dẫn cài đặt, và thậm chí còn giúp chúng ta viết ra những chương trình dùng để phát sinh ra các chương trình khác.

Nói đến vấn đề lập trình, chúng ta phải đưa ra nhiều đoạn mã nguồn. Hầu hết các ví dụ ở đây chủ yếu dùng để minh họa cho quyển sách này mặc dù một số ví dụ nhỏ được điều chỉnh từ một số nguồn khác. Đa số các chương trình được viết bằng C, với một vài chương trình được viết bằng C++ và Java, và một số ít được viết bằng ngôn ngữ script. Ở mức thấp nhất, C và C++ hầu như là như nhau và những chương trình C cũng chính là những chương trình C++ hợp lệ. C++ và Java lần lượt là những thế hệ sau của C, thừa hưởng cú pháp, tính hiệu quả cũng như tính diễn đạt khác, đồng thời cũng thêm vào các hệ thống kiểu dữ liệu cũng như các thư viện đồi dào.

Trong công việc hàng ngày, cả ba loại ngôn ngữ này thường được sử dụng cùng với những ngôn ngữ khác. Việc lựa chọn ngôn ngữ để cài đặt tùy thuộc vào bài toán: các hệ điều hành được viết tốt nhất bằng một ngôn ngữ hiệu quả và không bị giới hạn như C hoặc C++; mô tả chương trình nhanh thường thực hiện dễ dàng nhất trên một trình thông dịch lệnh hoặc ngôn ngữ script như Awk hoặc Perl; hướng đến giao diện người dùng thì Visual Basic và Tcl/Tk là những ứng cử viên nặng ký cùng với Java.

Ngoài ra còn có một vấn đề quan trọng nữa trong việc lựa chọn ngôn ngữ để cài đặt cho các ví dụ. Bởi vì không có ngôn ngữ nào giải quyết mọi bài toán đều tốt như nhau, cũng như không có một ngôn ngữ nào là tốt nhất để thể hiện mọi đề tài. Các ngôn ngữ bậc cao cũng ảnh hưởng đến các quyết định trong thiết kế chương trình. Nếu chúng ta sử dụng ngôn ngữ cấp thấp hơn thì phải xem xét đến các giải pháp khác nhau cho cùng một bài toán. Bằng cách đưa ra nhiều chi tiết, chúng ta có thể giải quyết những bài

toán này tốt hơn. Kinh nghiệm cho thấy rằng: thậm chí khi sử dụng các tiện ích do ngôn ngữ cấp cao cung cấp, chúng ta sẽ thấy rất quý giá nếu như biết được những tiện ích này có liên hệ ra sao với các vấn đề cấp thấp; nếu không có cách nhìn thấu đáo đó, chúng ta rất dễ mắc phải các vấn đề về tốc độ thực thi chương trình cũng như các hiệu ứng khó hiểu. Vì lẽ đó, cuốn sách này chúng tôi thường sử dụng ngôn ngữ C để trình bày các ví dụ, mặc dù trong thực tế rất có thể sẽ sử dụng ngôn ngữ khác.

Tuy nhiên, hầu hết các phần trong quyển sách này đều độc lập với bất kỳ một ngôn ngữ lập trình cụ thể nào. Việc chọn lựa cấu trúc dữ liệu cũng còn do ngôn ngữ ảnh hưởng; một số ngôn ngữ có thể hỗ trợ rất ít trong khi một số ngôn ngữ khác đưa ra nhiều sự lựa chọn khác nhau. Tuy vậy, cách tiếp cận trong việc chọn lựa sẽ như nhau. Chi tiết trong việc kiểm chứng và gỡ rối ở các ngôn ngữ khác nhau là khác nhau, nhưng chiến lược và chiến thuật trong các ngôn ngữ này lại hoàn toàn như nhau. Hầu hết các kỹ thuật dùng để cải tiến chương trình đều có thể áp dụng cho bất kỳ ngôn ngữ nào.

Cho dù bạn sử dụng bất kỳ ngôn ngữ nào để viết chương trình, nhiệm vụ của người lập trình là sử dụng tốt các công cụ sẵn có. Một người lập trình tốt vẫn có thể vượt qua những ngôn ngữ lập trình nghèo nàn cũng như các hệ điều hành còn non yếu, nhưng ngay cả môi trường lập trình tuyệt vời cũng không thể cứu vãn một người lập trình có khả năng còn khiêm tốn. Chúng tôi hy vọng rằng: cho dù kỹ năng cũng như kinh nghiệm hiện có của bạn như thế nào đi nữa, quyển sách này cũng sẽ giúp bạn lập trình tốt hơn.

MỤC LỤC

LỜI NÓI ĐẦU 3

Chương 1: PHONG CÁCH LẬP TRÌNH

1.1. Tên hàm hay tên biến 15
1.2. Biểu thức và phát biểu 20
1.3. Tính nhất quán và các đặc ngữ 27
1.4. Các hàm macro 38
1.5. Các số tối nghĩa 41
1.6. Chú thích 47
1.7. Vì sao phải lo lắng về phong cách lập trình? 55

Chương 2: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

2.1. Tìm kiếm 57
2.2. Sắp xếp 60
2.3. Thư viện hàm 64
2.4. Hàm Quicksort trong Java 69
2.5. Ký hiệu O 73
2.6. Các mảng mở rộng 75
2.7. Danh sách liên kết 80
2.8. Cây 90
2.9. Bảng băm 98
2.10. Tổng kết 103

Chương 3: THIẾT KẾ VÀ CÀI ĐẶT

3.1. Giải thuật chuỗi Markov 106
3.2. Chọn lựa cấu trúc dữ liệu 109
3.3. Xây dựng cấu trúc dữ liệu trong C 111
3.4. Phát sinh kết quả 117
3.5. Bản cài đặt trong Java 122
3.6. Bản cài đặt trong C++ 129

3.7. Awk và Perl 133
3.8. Tốc độ thực thi 137

Chương 4: GIAO TIẾP

4.1. Giá trị được phân cách bằng dấu phẩy – CSV (Comma-Separated Values) 141
4.2. Một thư viện mẫu 144
4.3. Thư viện dùng chung 151
4.4. Bản cài đặt bằng C++ 167
4.5. Nguyên tắc thiết kế phương thức giao tiếp 175
4.6. Quản lý tài nguyên 179
4.7. Thoát, thực hiện lại, dừng thất bại? 183
4.8. Giao diện người dùng 191

Chương 5: GỠ RỐI

5.1. Trình gỡ rối 195
5.2. Có đầu mối, phát hiện ra lỗi dễ dàng 197
5.3. Không có đầu mối, khó phát hiện ra lỗi 203
5.4. Phương sách cuối cùng 210
5.5. Các lỗi không có khả năng xuất hiện lại 214
5.6. Công cụ gỡ rối 217
5.7. Lỗi của người khác 222
5.8. Tổng kết 224

Chương 6: KIỂM CHỨNG

6.1. Kiểm tra trong khi viết mã nguồn 227
6.2. Kiểm chứng có hệ thống 235
6.3. Kiểm chứng tự động 241
6.4. Mô hình kiểm chứng 246
6.5. Kiểm chứng tự động với tập dữ liệu có giá trị lớn 250
6.6. Những mẹo trong việc kiểm chứng 255
6.7. Ai sẽ thực hiện kiểm chứng? 257
6.8. Kiểm chứng chương trình Markov 258
6.9. Tổng kết 262

Chương 7: TỐC ĐỘ THỰC THI

7.1. Hiện tượng nghẽn cổ chai 264
7.2. Kỹ thuật lập sơ đồ sử dụng thời gian 273
7.3. Chiến lược tăng tốc độ 280
7.4. Tinh chỉnh mã 285
7.5. Dùng hiệu quả không gian 291
7.6. Ước tính 293
7.7. Tổng kết 297

Chương 8: TÍNH KHẢ CHUYỂN

8.1. Ngôn ngữ 299
8.2. Tập tin tiêu đề (header) và thư viện 308
8.3. Tổ chức chương trình 310
8.4. Sự cô lập 316
8.5. Chuyển đổi dữ liệu 317
8.6. Nâng cấp và tính khả chuyển 317
8.7. Tổng kết 321

Chương 9: KÝ HIỆU

9.1. Định dạng dữ liệu 323
9.2. Các biểu thức có quy tắc 334
9.3. Một số công cụ lập trình 346
9.4. Các trình thông dịch, trình biên dịch và máy ảo 352
9.5. Các chương trình để viết các chương trình khác 363
9.6. Sử dụng các macro để phát sinh mã nguồn 369
9.7. Biên dịch trong khi thực thi 370

PHỤ LỤC 37 9

TÀI LIỆU THAM KHẢO 385

Chương 1

PHONG CÁCH LẬP TRÌNH

Đoạn mã nguồn sau đây trích từ một chương trình lớn được viết từ nhiều năm về trước:

```
if ((country == SING) || (country == BRNI) ||
      (country == POL) || (country == ITALY))
      {
      /*
* Nếu biến country là Singapore, Brunei hoặc Poland
* thì thời gian hiện hành là thời gian cần tìm
      */
            ...
```

Mã nguồn này được viết, định dạng và được chú thích một cách đầy đủ. Do đó, chương trình được tích hợp từ những đoạn mã nguồn trên chạy rất tốt; những lập trình viên tạo ra chương trình này rất tự hào về những gì họ đã làm. Nhưng đoạn trích này gây bối rối cho những người bình thường. Mối quan hệ giữa `Singapore`, `Brunei`, `Poland` và `Italy` ở đây là gì? Tại sao `Italy` lại không được đề cập trong phần chú thích. Vì việc chú thích và phần cài đặt khác nhau nên chắc chắn hoặc phần chú thích sai hoặc phần cài đặt sai. Cũng có thể là tất cả đều sai. Phần cài đặt là phần đã được thực thi và thử nghiệm nên nó có khả năng đúng hơn phần chú thích. Cũng có thể là phần chú thích đã không được cập nhật trong khi phần cài đặt đã được cập nhật. Phần chú thích trên không nói hết được về mối quan hệ giữa ba nước đã được đề cập. Nếu bạn là người phải bảo trì đoạn mã nguồn trên thì bạn phải cần biết nhiều thông tin hơn.

Những dòng trên minh họa những sai sót thường gặp trong lập trình: hầu như chương trình vẫn chạy tốt nhưng có một vài phần cần phải được cải tiến lại.

Những phần trong sách này nói về lập trình ứng dụng – làm sao để viết một chương trình thật sự. Mục đích của chúng tôi là giúp cho bạn viết được những chương trình cũng tốt như những ví dụ được trình bày trong cuốn sách này, đồng thời tránh được những lỗi sai và những yếu điểm trong chương trình. Sau đây, chúng ta sẽ bàn về cách viết chương trình sao cho tốt ngay từ khi bắt đầu và việc phát triển chương trình như thế nào.

Chúng ta sẽ bắt đầu thảo luận về phong cách lập trình. Mục đích của phong cách lập trình là làm cho chương trình dễ đọc hơn, và một phong cách tốt quyết định đến việc lập trình tốt. Chúng tôi muốn đề cập đến phong cách lập trình đầu tiên vì nó sẽ được áp dụng trong toàn bộ cuốn sách này.

Có nhiều cách để viết chương trình hơn là việc chỉ viết đúng cú pháp, sửa lỗi hay làm cho nó chạy nhanh hơn. Chương trình không chỉ được máy tính đọc mà còn có cả những người lập trình sẽ đọc nó. Một chương trình tốt là một chương trình dễ hiểu và dễ sửa đổi.

Những nguyên tắc của phong cách lập trình dựa trên những cảm nhận chung bằng kinh nghiệm, không dựa vào những quy luật hay chỉ dẫn nào khác. Chương trình phải được viết rõ ràng và đơn giản - logic, diễn đạt một cách tự nhiên, dễ hiểu, tên hàm hay tên biến phải có nghĩa, định dạng rõ ràng và phải có phần chú thích - và đặc biệt cần phải tránh những việc thử tài khéo léo hoặc dùng những cấu trúc không bình thường trong khi viết chương trình. Tính nhất quán là cần thiết vì khi đó người khác sẽ dễ dàng hiểu chương trình của bạn và bạn cũng dễ dàng hiểu chương trình của người khác, nếu tất cả mọi người đều có phong cách viết giống nhau. Do đó cần phải có những quy ước cục bộ giữa những người lập trình hay phải được một chương trình quản lý và tốt nhất là nên tuân theo một quy ước chung đã được sử dụng rộng rãi.

Các phần kế tiếp sẽ minh họa các phong cách lập trình thông qua

việc đưa ra những ví dụ nhỏ của những chương trình thật sự tốt và những chương trình không tốt để từ đó nêu ra sự khác biệt giữa hai phong cách khác nhau. Những chương trình ví dụ này không phải là tự tạo ra. Tất cả các chương trình viết không được tốt đều được trích ra từ những đoạn mã nguồn thật sự đã được viết bởi những lập trình viên bình thường làm việc dưới áp lực công việc khá lớn, trong một khoảng thời gian quá ngắn. Một trong số các chương trình này sẽ được làm gọn, súc tích hơn nhưng phải không được gây ra lỗi trình bày. Rồi chúng ta sẽ viết lại những đoạn chương trình viết không được tốt thành những đoạn chương trình tốt hơn và nêu ra cách thức cải tiến tốt hơn đó. Nhưng vì chúng là những đoạn chương trình thật sự nên chúng còn có thể có những lỗi tiềm ẩn khác nữa.

Chúng ta quy ước những đoạn chương trình viết chưa được tốt sẽ được thêm dấu chấm hỏi (?) ở lề trái như ví dụ sau:

```
?       #define ONE 1
?       #define TEN 10
?       #define TWENTY 20
```

Tại sao những định nghĩa trên viết chưa được tốt? Nếu ta cần một mảng lớn hơn 20 phần tử thì việc thay chỉnh sửa định nghĩa trên là cần thiết. Mỗi tên được định nghĩa nên giữ một vai trò nào đó và thay cho một giá trị cụ thể trong chương trình:

```
#define INPUT_MODE      1
#define INPUT_BUFSIZE   10
#define OUTPUT_BUFSIZE  20
```

1.1. Tên hàm hay tên biến

Tên của một hàm hay một biến nên được đặt như thế nào? Phải làm sao thể hiện được chúng thuộc đối tượng nào và mục đích của chúng là gì? Thông thường tên hàm hay tên biến phải thể hiện được thông tin, súc tích, dễ nhớ, và cả dễ đọc nữa (nếu có thể được). Nếu tầm vực của hàm hay biến càng rộng thì tên của chúng càng cần phải thể hiện nhiều thông tin hơn.

Dùng tên có tính gợi nhớ cho các biến toàn cục và tên ngắn cho các biến cục bộ

Các biến toàn cục, theo định nghĩa, có thể được dùng ở bất kỳ nơi nào trong chương trình, do vậy mà cần phải được đặt tên có độ dài và có tính gợi tả đủ để nhắc nhớ ý nghĩa của tên đó cho lập trình viên. Và cũng nên mô tả ngắn gọn mục đích của biến ở phần khai báo biến toàn cục:

```
int npending = 0; //chiều dài hiện hành của hàng đợi
```

Các hàm, lớp và cấu trúc toàn cục cũng nên được đặt tên đầy đủ ý nghĩa để nói lên được vai trò của chúng trong chương trình.

Ngược lại, các biến cục bộ chỉ cần một tên ngắn là đủ; bên trong một hàm, tên `n` cho một biến là đủ; `npoints` là quá tốt, còn tên `numberOfpoints` dài quá mức cần thiết.

Các biến cục bộ có tính quy ước có thể dùng tên rất ngắn. Ví dụ như việc dùng `i` và `j` cho các chỉ số vòng lặp, `p` và `q` cho các con trỏ, `s` và `t` cho chuỗi đã là quá thông dụng. Cho nên nếu dùng các tên dài sẽ không có ích, thậm chí còn không tốt. Hãy thử so sánh hai đoạn mã nguồn sau:

```
?       for (theElementIndex = 0; theElementIndex <
numberOfElement;
?       theElementIndex++)
?       elementArray[theElementIndex] = theElementIndex;
```

và

```
        for (i = 0; i < nelems; i++)
              elem[i] = i;
```

Các lập trình viên thường được khuyến khích dùng tên biến dài mà không chú ý tới ngữ cảnh của biến. Điều này là sai lầm: thực ra thì chính sự ngắn gọn thường làm cho chương trình sáng sủa.

Có nhiều quy ước đặt tên khác nhau và cũng có nhiều cách đặt tên khác nhau mang tính địa phương. Một vài quy ước phổ biến có: dùng tên bắt đầu hoặc kết thúc bằng chữ `p` cho các con trỏ; viết hoa chữ cái đầu cho các

biến toàn cục; và viết hoa tất cả các chữ cái cho các hằng số. Một số nơi còn dùng những quy tắc có tính bao quát hơn, chẳng hạn như dùng ký hiệu để mô tả loại biến cũng như các thông tin hữu ích về biến, ví dụ như tên `ptr` cho biến con trỏ đến một ký tự, tên `strTo` và `strFrom` chỉ các chuỗi dùng để ghi vào hay đọc ra. Về mặt chính tả, việc dùng `npending` hay `numPending` hay `num_pending` chỉ là vấn đề thói quen; các quy tắc chi tiết không quan trọng bằng sự nhất quán của quy ước.

Các quy ước đặt tên giúp cho mã nguồn chương trình dễ hiểu hơn, cả mã nguồn do chính bạn viết hay cũng như do người khác viết. Chúng cũng giúp chọn tên mới dễ dàng hơn khi đang viết chương trình. Chương trình càng dài thì tầm quan trọng của việc lựa chọn tên sao cho tốt, gợi nhớ và có hệ thống càng lớn.

`Namespace` trong C++ và các gói trong Java cung cấp cách thức quản lý tầm vực của tên và giúp giữ ý nghĩa của tên rõ ràng mà không cần các tên dài quá mức.

Tính nhất quán

Hãy dùng các tên nói lên sự liên hệ cho những gì có liên quan với nhau sao cho thấy được mối liên quan cũng như làm nổi bật sự khác nhau giữa chúng. Trong lớp Java dưới đây, tên của các hàm thành viên không chỉ rối rắm mà còn không nhất quán:

```
?     class UserQueue {
?     int nofItemsInQ, frontOftheQueue, queueCapaccity;
?     public int noOfUsersInQueue() {….}
?     }
```

Chữ `"queue"` xuất hiện dưới các dạng `Q`, `Queue` và `queue`. Nhưng do các hàng đợi chỉ có thể truy nhập được từ một biến kiểu `UserQueue` nên tên các hàm thành viên không cần đề cập đến chữ `"queue"` nữa; ngữ cảnh là đã đủ, do vậy

```
?     queue.queueCapacity
```

là thừa. Chương trình nên được viết lại như sau:

```
class UserQueue {
        int nitems, front, capacity;
        public int nusers () {...}
}
```

khi đó ta có các phát biểu ở dạng:

```
queue.capacity ++ ;
n = queue.nusers();
```

Dùng tên ở thể chủ động cho các hàm

Nên dùng các động từ ở thể chủ động, có thể kèm theo danh từ, để đặt tên hàm:

```
now = date.getTime();
putchar ('\n');
```

Các hàm trả về giá trị logic (`true` hay `false`) nên được đặt tên sao cho giá trị trả về không bị mơ hồ. Chẳng hạn như, đoạn mã nguồn kiểm tra một số thuộc hệ bát phân sau:

```
?       if (checkoctal (c)) ...
```

không chỉ ra được giá trị nào là `true` và giá trị nào là `false`, trong khi

```
if (isoctal (c)) ...
```

chỉ rõ rằng hàm trả về giá trị đúng nếu đối số thuộc hệ bát phân và trả về giá trị sai trong trường hợp ngược lại.

Tính chính xác

Tên không giống như một tập nhãn, nó truyền tải thông tin đến người đọc. Một tên không rõ ràng có thể dẫn đến những lỗi bí hiểm. Chẳng hạn như, macro `isoctal` đã được viết và lưu hành với lỗi như sau trong nhiều năm:

```
?       #define isoctal (c)   ((c) >  '0' && (c) <  '8')
```

thay vì mã nguồn đúng như sau:

```
#define isoctal (c) ((c) >= '0' && (c) <= '7')
```

trong trường hợp trên, tên truyền tải nội dung chính xác nhưng sự thực thi không đúng; *một tên tốt lại rất dễ che giấu một lỗi thực thi nếu không cẩn trọng.*

Dưới đây là một ví dụ trong đó tên hàm và mã nguồn hoàn toàn trái ngược:

```
?     public boolean inTable (Object obj) {
?           int j = this.getIndex(obj);
?           return (j == nTable);
?     }
```

Hàm `getIndex` trả về một giá trị giữa `0` và `nTable-1` nếu tìm được đối tượng, và trả về `nTable` nếu không tìm được. Như vậy giá trị logic trả về bởi `inTable` trái ngược với ngụ ý của tên hàm. Khi còn đang viết mã nguồn thì điều này có thể không thành vấn đề, nhưng sau này khi sửa chữa chương trình chắc chắn tên đó sẽ gây nhầm lẫn.

Bài tập 1-1. Hãy nhận xét cách chọn tên và giá trị của đoạn mã nguồn dưới đây:

```
?     #define TRUE 0
?     #define FASLE 1
?     if ((ch = getchar ()) == EOF)
?           not_eof = FALSE;
```

Bài tập 1-2. Cải tiến hàm sau đây:

```
?     int smaller (char *s, char *t) {
?           if (strcmp(s, t) < 1)
?                 return 1;
?           else
?                 return 0;
?     }
```

Bài tập 1-3. Hãy kiểm tra đoạn mã nguồn sau:

```
?     if   ((falloc(SMRHSHSCRTCH,   S_TFEXT   ,   0644,
MAXRODDDHSH)) < 0)
?           ...
```

1.2. Biểu thức và phát biểu

Tương tự như việc chọn cách đặt tên để giúp người đọc chương trình hiểu rõ, việc viết các biểu thức hoặc các phát biểu cũng phải được thực hiện sao cho chúng có ý nghĩa càng tường minh càng tốt, mã nguồn càng sáng sủa càng tốt. Dùng các khoảng trắng trước và sau các toán tử để ám chỉ một nhóm: tổng quát hơn, dùng định dạng để giúp dễ đọc. Điều này tuy tầm thường song lại có giá trị rất lớn, cũng như việc giữ cho bàn làm việc gọn gàng để dễ tìm đồ dùng. Tuy nhiên khác với bàn làm việc, chương trình của bạn có thể được nhiều người khác xem xét.

Canh chỉnh lề để thể hiện cấu trúc

Một lối canh chỉnh lề nhất quán là cách dễ nhất làm cho cấu trúc của chương trình trở nên tường minh. Ví dụ sau có định dạng không tốt:

```
?     for (n++; n < 100; field[n++] = '\0');
?     *i = '\0'; return('\0');
```

Cải tiến phần nào cách định dạng của đoạn mã nguồn trên:

```
?     for (n++; n<100; field[n++] = '\0')
?           ;
?     *i = '\0';
?     return '\n';
```

Tốt hơn nữa là nên đưa phần lệnh gán vào phần thân của vòng lặp và tách rời phần tăng giá trị biến vòng lặp, khi đó vòng lặp có hình thức quy ước hơn và do vậy dễ hiểu hơn:

```
for (n++; n<100; n++)
```

```
        field[n] = '\0';

    *i = '\0';

    return '\n';
```

Các biểu thức nên mang tính tự nhiên

Hãy viết biểu thức như cách bạn đọc. Các biểu thức điều kiện dạng phủ định luôn luôn khó hiểu:

```
?   if  (!(block_id  <  actblks)  ||  !(block_id  >=
unblocks))

?           ...
```

Mỗi phép kiểm tra trong điều kiện `if` được phát biểu ở dạng phủ định, dù rằng không cần thiết phải dùng cả hai phát biểu đều phủ định như thế. Đảo lại, biểu thức ta phát biểu ở dạng khẳng định như sau:

```
if ( (block_id < actblks) ||  (block_id >= unblocks))

            ...
```

Khi đó, mã nguồn có thể được đọc một cách tự nhiên.

Dùng dấu ngoặc để tránh tối nghĩa

Dấu ngoặc đánh dấu một nhóm và có thể được dùng để làm rõ nội dung ngay cả khi không cần dùng. Các dấu ngoặc phía trong ở ví dụ trên thực tế là không cần thiết, nhưng cả hai cặp dấu ngoặc đó đều không gây ảnh hưởng gì. Những lập trình viên có thể bỏ chúng đi, bởi vì các toán tử so sánh (`<  <=  ==  !=  >=  >`) có thứ tự ưu tiên cao hơn các toán tử logic (`&&`, `||`).

Dù vậy, khi dùng lẫn lộn các toán tử khác (không phải toán tử so sánh). tốt hơn nên dùng dấu ngoặc. C và các ngôn ngữ tương tự cho thấy một số vấn đề nguy hiểm về thứ tự ưu tiên, và do vậy rất dễ mắc lỗi. Vì các toán tử logic ràng buộc chặt chẽ hơn phép gán, dấu ngoặc là bắt buộc đối với hầu hết các biểu thức liên hệ chúng với nhau.

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

Các toán tử trên bit & và | có thứ tự ưu tiên thấp hơn các toán tử quan hệ như ==, nên cho dù có sự xuất hiện của chúng, biểu thức

```
?       if (x&MASK == BITS)
?               ...
```

thực sự có nghĩa là:

```
?       if (x & (MASK == BITS))
?               ...
```

mà rõ ràng đó không phải là ý định của lập trình viên. Do đó, biểu thức trên cần có dấu ngoặc:

```
if ((x&MASK) == BITS)
        ...
```

Ngay cả khi không cần thiết, dấu ngoặc cũng có thể có ích nếu khó thực hiện gom nhóm ngay từ lần đầu tiên. Đoạn mã nguồn sau đây không cần phải dùng dấu ngoặc:

```
?       leap_year = y % 4 == 0 && y % 100 != 0 || y % 400
== 0;
```

nhưng nếu có dấu ngoặc sẽ giúp chúng dễ hiểu hơn:

```
leap_year = ((y % 4 == 0) && (y % 100 != 0)) || (y% 400
== 0);
```

Ta cũng đã bỏ bớt một số chỗ trống: việc nhóm các toán hạng gồm các toán tử có thứ tự ưu tiên cao hơn giúp độc giả thấy ngay được cấu trúc của đoạn chương trình.

Phân tích những biểu thức phức tạp thành những biểu thức đơn giản hơn.

C, C++ và Java có cú pháp và số lượng toán tử phong phú, và rất dễ

bị cuốn vào việc thu gọn chương trình bằng cách nhồi nhét tất cả vào trong chỉ một phát biểu. Loại biểu thức giống như biểu thức sau đây rất ngắn gọn song đã đưa quá nhiều phép toán vào chỉ trong một phát biểu:

```
?       *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

Đoạn mã nguồn trên sẽ dễ hiểu hơn nếu chia nhỏ ra thành nhiều đoạn:

```
if (2*k < n-m)
        *xp = c[k+1];
else
        *xp = d[k--];
*x += *xp;
```

Tính sáng sủa

Năng lực sáng tạo vô tận của các lập trình viên đôi khi được dùng để viết mã nguồn ngắn nhất có thể nhất, hoặc để tìm các cách thông minh hơn để đạt được mục đích. Tuy nhiên, đôi khi những kỹ năng này được áp dụng sai chỗ, vì mục tiêu là viết mã nguồn sao cho sáng sủa chứ không phải là viết các mã nguồn để thể hiện tài khéo léo.

Hãy xem thử phép tính phức tạp sau đây thực hiện việc gì?

```
? subkey = subkey >> (bitoff  - ((bitoff >> 3) << 3));
```

Biểu thức trong cùng thực hiện dịch `bitoff` sang phải 3 bit. Kết quả được dịch lại sang trái 3 bit, như vậy đã thay thế 3 bit được dịch bằng số không. Sau đó, lấy giá trị bitoff ban đầu trừ cho kết quả vừa có được. Kết quả thu được là 3 bit cuối của *bitoff*. Ba bit này được dùng để dịch *subkey* sang phải.

Như vậy biểu thức ban đầu tương đương với:

```
subkey = subkey >> (bitoff & 0x7);
```

Phải mất thời gian để giải đáp được biểu thức ban đầu thực hiện công việc gì; còn biểu thức sau sáng sủa hơn và ngắn gọn hơn. Những lập

trình viên có kinh nghiệm thậm chí còn làm gọn hơn bằng cách dùng một phép gán:

```
subkey >>= bitoff & 0x7;
```

Một số cấu trúc hay bị lạm dụng. Toán tử `?:` có thể dẫn đến những mã nguồn bí hiểm:

```
?       child = (!LC&&!RC) ? 0: (!LC?RC:LC);
```

Hầu như không thể hiểu được đoạn mã nguồn trên thực hiện việc gì nếu không lần theo tất cả các nhánh của biểu thức. Đoạn mã nguồn sau dây dài hơn nhưng dễ theo dõi hơn vì đã làm rõ các nhánh:

```
if (LC ==  0  && RC == 0)
        child = 0;
else if ((LC == 0)
        child = RC;
else
        child = LC;
```

Toán tử `?:` rất tốt cho các biểu thức ngắn trong đó thay thế 4 dòng `if-else` bằng một, ví dụ như trong đoạn mã nguồn sau:

```
max = (a>b) ? a:b;
```

hay như trong

```
printf ("The list has %d item %s \n", n, n !=1 ? "":
"s");
```

nhưng không phải là một sự thay thế chung cho các biểu thức điều kiện.

Sự sáng sủa không đồng nghĩa với sự ngắn gọn. Thông thường mã nguồn sáng sủa hơn cũng sẽ ngắn gọn hơn, như trong ví dụ về phép dịch bit ở trên, nhưng cũng có khi dài hơn, như trong các biểu thức điều kiện viết lại ở dạng `if-else`. *Tiêu chuẩn đúng đắn cần quan tâm là sự dễ hiểu.*

Cẩn thận với hiệu ứng lề

Các toán tử như `++` có hiệu ứng lề: bên cạnh việc trả về một giá trị, chúng còn sửa giá trị của biến. Hiệu ứng lề có thể rất thuận tiện, nhưng cũng có thể gây rắc rối vì hai hành động truy xuất giá trị và cập nhật biến có thể không xảy ra đồng thời. Trong C và C++, thứ tự thực hiện của hiệu ứng lề không được định nghĩa, do vậy phép gán phức tạp sau đây có thể cho kết quả sai:

```
?       str[i++] = str[i++] = ' ';
```

Ý định của lập trình viên là lưu hai khoảng trắng vào các vị trí tiếp theo trong chuỗi `str`. Nhưng tùy theo khi nào `i` được cập nhật, một vị trí trong `str` có thể bị bỏ qua và `i` có thể cuối cùng chỉ tăng một đơn vị. Hãy ngắt phát biểu trên thành hai:

```
str[i++] = ' ';

str[i++] = ' ';
```

Dù chỉ tăng có một lần, phép gán sau có thể cho nhiều kết quả khác nhau:

```
?       array[i++] = i;
```

Nếu giá trị ban đầu của `i` là 3, phần tử mảng có thể được gán giá trị 3 hay 4.

Không chỉ việc tăng hay việc giảm có hiệu ứng lề. Các lệnh về nhập xuất cũng là một nguồn gây các hành động ngầm. Ví dụ sau đọc hai số có liên quan bằng thủ tục nhập chuẩn:

```
?       scanf ("%d %d", &yr, &profit[yr]);
```

Kết quả sau khi thực thi ví dụ trên không như mong muốn vì một phần của biểu thức sửa đổi `yr` và một phần khác sử dụng biến đó. Giá trị của `profit[yr]` có thể không bao giờ đúng trừ khi giá trị mới của `yr` bằng giá trị cũ. Có thể cho rằng đáp số phụ thuộc vào thứ tự tính các đối số, nhưng vấn đề thực sự ở chỗ tất cả các đối số của hàm `scanf` được tính trước khi thủ tục được gọi, do vậy `&profit[yr]` sẽ luôn luôn được tính với giá trị cũ của `yr`. Loại vấn đề như vậy có thể xảy ra với hầu như bất kỳ ngôn ngữ nào.

Cách sửa chữa thông thường là ngắt biểu thức ra:

```
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

Hãy chú ý hiệu ứng lề trong mọi biểu thức.

Bài tập 1-4. Cải tiến các đoạn mã nguồn sau:

```
?     if (!(c = 'y' || c == 'Y'))
?           return;
?     length = (length < BUFSIZE) ? length : BUFSIZE;
?     flag = flag ? 0:1;
?     quote = (* line == '"') ? 1:0 ;
?     if (val & 1)
?           bit = 1;
?     else
?           bit = 0;
```

Bài tập 1-5. Đoạn mã nguồn sau sai ở những vị trí nào?

```
?     int read(int *ip) {
?           scanf("%d", ip);
?           return *ip;
?     }
?     ...
?     insert (&graph[vert], read(&val), read(&cnt));
```

Bài tập 1-6. Hãy liệt kê mọi kết quả đoạn mã nguồn sau có thể thu được với các thứ tự tính toán khác nhau:

```
?     n = 1;
?     printf( "%d %d\n", n++, n++);
```

Hãy thử trên càng nhiều trình biên dịch càng tốt, để xem điều gì xảy ra trong thực tế.

1.3. Tính nhất quán và các đặc ngữ

Tính nhất quán thường giúp ta xây dựng được các chương trình tốt. Nếu định dạng thay đổi một cách lộn xộn, hay nếu vòng lặp khi thì chạy theo kiểu tăng, khi thì chạy theo kiểu giảm, hay nếu sao chép chuỗi bằng hàm `strcpy` ở chỗ này và dùng vòng lặp `for loop` ở chỗ khác, những sự thay đổi đó sẽ làm cho khó thấy được chương trình thực hiện công việc gì. Nhưng nếu một phép tính được thực hiện theo cùng một cách ở mọi nơi,

Dùng kiểu canh chỉnh lề và dấu ngoặc móc một cách nhất quán

Kiểu canh chỉnh lề thể hiện cấu trúc, nhưng kiểu canh chỉnh lề nào là tốt nhất? Có nên đặt dấu ngoặc móc cùng một dòng với phát biểu `if` hay nên đặt trên dòng mới? Các lập trình viên luôn tranh cãi về cách sắp xếp chương trình, nhưng một phong cách nhất định không quan trọng bằng việc sử dụng sao cho nhất quán.

Có nên dùng dấu ngoặc móc dù không thật cần thiết hay không? Như dấu ngoặc đơn, dấu ngoặc móc có thể giúp tránh tối nghĩa và đôi khi làm cho mã nguồn sáng sủa hơn. Vì mục đích nhất quán, nhiều lập trình viên kinh nghiệm luôn luôn đặt dấu ngoặc móc bao quanh một vòng lặp hay thân một phát biểu `if`, nhưng nếu phần thân đó chỉ gồm một phát biểu thì lại không cần thiết, do vậy thường bị bỏ đi. Nếu chính bạn cũng có thói quen bỏ đi, hãy nhớ rằng bạn không bỏ đi trong những lúc cần dùng để giải quyết các phát biểu `else` rất tối nghĩa được minh họa ở đoạn trích sau:

```
?       if (month == FEB)  {
?           if (year%4 == 0)
?               if (day >29)
?                   legal = FALSE;
?           else
?               if (day < 29)
?                   legal = FALSE;
?       }
```

Sự canh chỉnh lề của lập trình viên đã đi sai đường. vì phần `else` thực sự gắn với dòng

```
?       if (day > 29)
```

và như vậy mã nguồn là sai. Do đó, khi có một `if` theo ngay sau một `if` khác, thì luôn luôn phải dùng ngoặc móc:

```
?       if (month == FEB) {
?               if (year%4  == 0) {
?                       if (day >29)
?                               legal= FASLE;
?               } else {
?                       if (day >28)
?                               legal = FASLE;
?               }
?       }
```

Các công cụ lập trình có thể giúp hạn chế loại lỗi này.

Ngay khi lỗi đã được sửa, đoạn mã nguồn trên vẫn rất khó theo dõi. Đoạn mã nguồn trên sẽ dễ hơn nếu ta dùng một biến để lưu số ngày của tháng hai:

```
?       if (month == FEB) {
?               int nday;
?
?               nday = 28;
?               if (year%4 == 0)
?                       nday = 29;
?               if (day > nday)
?                       legal = FASLE;
?       }
```

Tuy nhiên, mã nguồn trên vẫn còn sai, vì năm 2000 là năm nhuận, trong khi năm 1900 và năm 2100 lại không phải là năm nhuận. Do đó, cần phải sửa lại điều kiện năm nhuận.

Nhân đây xin nói thêm, nếu đang làm việc trên một chương trình không phải do mình viết, hãy giữ lại phong cách của chương trình đó. Khi sửa đổi chương trình, đừng dùng phong cách của riêng bạn dù bạn ưa thích hơn. Sự nhất quán của chương trình quan trọng hơn phong cách cá nhân, vì sẽ giúp dễ dàng cho người khác theo dõi chương trình.

Dùng đặc ngữ để đạt được tính nhất quán

Tương tự như ngôn ngữ tự nhiên, ngôn ngữ lập trình có các đặc ngữ, tức là các cách quy ước mà lập trình viên có kinh nghiệm sẽ dùng để viết các đoạn mã nguồn thông thường sao cho phù hợp với ngôn ngữ đang viết. Một phần trọng tâm khi học bất cứ ngôn ngữ nào là tập làm quen với các đặc ngữ của ngôn ngữ đó.

Một trong những đặc ngữ thông thường nhất là hình thức của một vòng lặp. Hãy xem xét mã nguồn của C, C++, Java cho việc duyệt qua *n* phần tử của một mảng, để khởi tạo mảng chẳng hạn. Một số người có thể viết mã nguồn như sau:

```
?     i =0;
?     while (i <- n-1)
?           array [i++] = 1.0;
```

hoặc:

```
?     for (i = 0; i<n; )
?           array [i++] = 1.0;
```

hoặc thậm chí viết như sau:

```
?     for (i = n ; --i >= 0;)
?           array[i] = 1.0;
```

Tất cả đều đúng, nhưng dạng phù hợp với đặc tính của các ngôn ngữ trên là:

```
for (i = 0; i<n ; i++)
    array [i] = 1.0;
```

Đây không phải là một sự chọn lựa tùy ý. Đoạn mã nguồn như trên duyệt qua từng phần tử của mảng đánh số từ `0` đến `n-1`; đặt toàn bộ quyền điều khiển vòng lặp vào `for`, chạy theo thứ tự tăng dần, và dùng toán tử `++` rất có đặc ngữ để cập nhật biến vòng lặp; và trả lại cho biến chỉ số một giá trị biết trước ngay sau phần tử cuối của mảng. Một cách ví von, nếu các ngôn ngữ này là tiếng mẹ đẻ của bạn, bạn sẽ có thể nhận biết không cần phải học và viết đúng không cần đến một giây suy nghĩ.

Trong C++ hay Java, một biến thể thông thường sẽ bao gồm cả phần khai báo biến vòng lặp:

```
for (int i 0; i<n, i ++)
    array[i] = 1.0;
```

Sau đây là vòng lặp chuẩn để quét qua một danh sách trong C:

```
for (p = list; p != NULL; p    p-> next )
    ...
```

Một lần nữa, toàn bộ điều khiển của vòng lặp đều nằm trong vòng lặp `for`. Đối với các vòng lặp không xác định số lần lặp, ta hay dùng:

```
for ( ;  ;)
    ...
```

nhưng

```
while (1)
    ...
```

cũng rất thường gặp. Đừng viết khác những dạng trên.

Cách canh chỉnh lề cũng nên canh chỉnh sao cho phù hợp với đặc tính của ngôn ngữ. Kiểu trình bày từ trên xuống như sau đây làm giảm tính dễ đọc; trông giống 3 phát biểu hơn là một vòng lặp:

```
?       for (
?               ap = arr;
?               ap < arr + 128;
?               *ap++ = 0
?               )
?       ;
?               ;
?       }
```

Một vòng lặp chuẩn sẽ dễ đọc hơn:

```
for ( ap= arr; ap < arr+128; ap++)
        *ap = 0;
```

Những hình thức trình bày lộn xộn ngổn ngang cũng làm cho mã nguồn trải dài ra trên nhiều trang và điều này làm giảm tính dễ đọc của chương trình.

Một loại đặc ngữ thông dụng khác là gom một tác vụ vào trong điều kiện lặp, như trong đoạn mã nguồn sau:

```
while  (( c = getchar ()) != EOF)
        putchar(c);
```

Phát biểu `do-while` ít dùng hơn nhiều so với phát biểu `for` và `while`, vì nó luôn thực hiện ít nhất một lần, sau đó mới kiểm tra điều kiện thay vì cần phải thực hiện trước. Trong nhiều trường hợp, đó là một kiểu hành động rất dễ gây lỗi, như trong đoạn chương trình viết lại sau đây của vòng lặp `getchar`:

```
?       do {
?               c = getchar ();
?               putchar (c);
?       } while  (c != EOF);
```

Đoạn chương trình trên xuất ra một ký tự sai vì sự kiểm tra điều kiện

xảy ra sau khi gọi hàm `putchar`. Vòng lặp `do-while` chỉ đúng khi thân vòng lặp luôn phải được thực hiện ít nhất là một lần; ta sẽ xét một số ví dụ ở phần sau.

Một thuận lợi của việc dùng nhất quán các đặc ngữ là nó giúp bạn chú ý đến các vòng lặp không viết theo chuẩn, mà chính các nơi này thường xảy ra lỗi:

```
?       int i, *iArray, nmemb;
?
?       iArray  = malloc(nmemb * sizeof(int));
?       for (i = 0; i<= nmemb; i++)
?               iArray[i] = i;
```

Vùng nhớ được cấp phát cho `nmemb` phần tử, từ `iArray[0]` đến `iArray[nmemb-1]`, nhưng do sự kiểm tra điều kiện của vòng lặp là `<=`, nên vòng lặp chạy quá khỏi vị trí cuối của mảng và ghi đè lên giá trị đang có ở vị trí tiếp theo trong bộ nhớ. Điều không may là các lỗi kiểu này thường không phát hiện được cho đến khi này sinh các lỗi khác mà nguyên nhân chính là việc ghi đè lên vùng nhớ.

C và C++ cũng có các đặc ngữ dùng cho việc cấp phát vùng nhớ cho chuỗi và thao tác trên chuỗi. Khi đó, các mã nguồn không dùng các đặc ngữ này thường bị lỗi:

```
?       char *p, buf[256];
?       gets(buf);
?       p = malloc(strlen(buf));
?       strcpy(p, buf);
```

Không bao giờ nên dùng hàm `gets`, vì không có cách nào giới hạn lượng dữ liệu sẽ được nhập vào. Điều này dẫn tới các vấn đề về an toàn dữ liệu sẽ được đề cập trong Chương 6, ở đó ta sẽ chỉ ra rằng hàm `fgets` luôn luôn tốt hơn. Nhưng vẫn còn một vấn đề khác là hàm `strlen` không tính giá trị kết thúc chuỗi `'\0'`, trong khi hàm `strcpy` lại thực hiện sao chép cả giá trị kết thúc chuỗi. Do vậy chỗ được cấp phát không đủ, và hàm `strcpy`

ghi lấn qua vị trí cuối của khoảng được cấp phát. Trong trường hợp này, cách viết thích hợp thường được dùng là:

```
p = malloc(strlen(buf) + 1);
strcpy(p, buf);
```

hoặc trong C++:

```
p = new char(strlen(buf) + 1);
strcpy(p, buf);
```

Chú ý là phải có phần '+1'.

Java không bị lỗi này, vì chuỗi không được mô tả như là một mảng kết thúc bằng ký tự NULL. Các phần tử của mảng luôn được kiểm tra, do vậy không thể truy xuất ra bên ngoài ranh giới của một mảng trong Java.

Hầu hết các môi trường C và C++ cung cấp một hàm thư viện, hàm `strdup`, tạo một bản sao của chuỗi dùng hàm `malloc` và `strcpy`, giúp dễ tránh loại lỗi này. Thật không may, hàm `strdup` không sẵn có trong bản ANSI C chuẩn.

Cũng lưu ý thêm, cả đoạn mã nguồn gốc ban đầu cũng như đoạn mã nguồn đã sửa chữa đều không kiểm tra giá trị trả về bởi hàm `malloc`. Ta bỏ qua vấn đề này để tập trung vào điểm chính, nhưng trong một chương trình thực tế, giá trị trả về bởi các hàm `malloc`, `realloc`, `strdup`, hay bất cứ thủ tục cấp phát nào cũng đều cần phải được kiểm tra.

Dùng else-if cho các rẽ nhánh nhiều hướng.

Rẽ nhánh nhiều hướng thường được viết ở dưới dạng mắt xích `if ... else if ... else` như sau:

```
if (condition)
        statement1
else if (condition2)
        statement2
```

```
...
else if (condition_n)
        statement_n
else
        default-statement
```

Điều kiện `condition` được đọc từ trên xuống dưới: đầu tiên nếu điều kiện `condition` được thỏa, phát biểu `statement` theo sau được thực hiện, và phần còn lại sẽ bị bỏ qua. Phần phát biểu `statement` có thể chỉ có một phát biểu hoặc có một nhóm phát biểu đặt trong ngoặc móc. Phần `else` cuối cùng dành cho trường hợp mặc định, khi không xảy ra trường hợp nào trong số các lựa chọn trước đó. Phần `else` cuối cùng có thể bỏ đi nếu không có hành động nào cần thực hiện trong trường hợp mặc định đó, hoặc có thể đưa ra một thông báo lỗi giúp bắt những điều kiện "không thể xảy ra".

Hãy canh chỉnh lề tất cả các phần `else` từ trên xuống hơn là sắp hàng mỗi `else` với `if` tương ứng. Canh chỉnh lề từ trên xuống dưới như thế nhấn mạnh rằng các phép kiểm tra được thực hiện lần lượt đồng thời giúp giữ chúng không lọt ra khỏi lề bên phải.

Một chuỗi liên tiếp các phát biểu `if` được lồng vào nhau thường là dấu hiệu viết mã nguồn vụng về, nếu không chứa lỗi:

```
?     if (argc == 3)
?           if ((fin = fopen(argv[1], "r")) != NULL)
?                 if ((fout = fopen(argvp[2], "w")) !=
NULL) {
?                       while (( c = getc(fin)) != EOF)
?                             putc(c, fout);
?                 fclose (fin); fclose(fout);
?                 } else
?                 printf("Không thể mở output file
```

```
%s\n", argv[2]);
      ?           else
      ?                 printf("Không  thể  mở  input  file
%s\n", argv[1]);
      ?     else
      ?           printf  ("Dùng  inputfile  outputfile  thông
thường\n");
```

Thứ tự các phát biểu `if` trên buộc ta phải nhớ những phép kiểm tra nào đã được thực hiện, để tại một điểm xác định ta có thể lấy ra chúng cho đến khi ta xác định được hành động tương ứng. Nhưng vì nhiều nhất chỉ có một hành động được thực hiện, ta thực sự nên dùng một `else if`. Thay đổi thứ tự các rẽ nhánh sẽ giúp viết một phiên bản mới sáng sủa hơn, đồng thời cũng giúp tránh được các lỗi tiềm ẩn trong những đoạn mã nguồn vụng về:

```
if (argv !=3)
   printf ("Dùng inputfile outputfile thông thường\n");
else if ((fin = fopen (argv [1], "r")) == NULL)
      printf ("Không thể mở input file %s\n", argv[1]);
else if ((fout = fopen(agrv[2], "w")) == NULL) {
            printf  ("Không  thể  mở  output  file  %s\n",
argv[2]);
            fclose(fin);
      } else {
            while ((c = getc(fin)) != EOF)
                  putc(c, fout);
            fclose(fin);
            fclose(fout);
      }
```

Ta dò theo phép kiểm tra cho đến khi phép đầu tiên có giá trị đúng, thực hiện hành động tương ứng, và tiếp tục cho đến khi gặp `else` cuối cùng. Quy tắc ở đây là theo dõi mỗi rẽ nhánh càng kỹ càng tốt thông qua hành động đi kèm theo nhánh đó. Hay nói cách khác, mỗi khi làm một phép kiểm tra hãy thực hiện một hành động tương ứng.

Những ý định dùng lại một phần mã nguồn thường làm chương trình bị rối:

```
?     switch(c) {
?           case '-': sign = -1; /*cho trôi qua, không
dùng lệnh break */
?           case '+': c = getchar();
?           case '.': break;
?           default: if (!isditgit(c))
?                 return 0;
?     }
```

Đoạn chương trình trên khá rắc rối vì không dùng lệnh `break` trong phát biểu `switch` để tránh phải lặp lại một dòng mã nguồn. Điều này cũng không phù hợp với đặc tính của phát biểu `switch`; các phát biểu `case` hầu như luôn luôn kết thúc bằng `break`, chỉ có rất ít ngoại lệ phải được chú thích đầy đủ. Một hình thức và cấu trúc thích hợp hơn sẽ dễ đọc hơn, dù rằng hơi dài hơn:

```
?   switch (c){
?   case '-':
?   sign = -1;/* cho trôi qua, không dùng lệnh break */
?   case '+':
?           c = getchar();
?           break;
?   case '.':
?           break;
```

```
?       default:
?               if (!isdigit(c))
?                       return 0;
?               break;
?       }
```

Sự gia tăng kích thước giúp làm tăng tính sáng sủa. Tuy nhiên, đối với một cấu trúc không bình thường như vậy một chuỗi các phát biểu `else-if` thậm chí còn sáng sủa hơn:

```
if (c == '-') {
        sign = -1;
        c = getchar();
} else if (c == '+' ) {
        c = getchar ();
} else if (c !== '.' && !isdigit(c)) {
        return 0;
}
```

Các dấu ngoặc móc bao các khối chỉ gồm có một dòng giúp làm nổi bật cấu trúc song song.

Cách sử dụng liệt kê (cho trôi qua, không dùng lệnh `break`) có thể chấp nhận được khi một số phát biểu `case` có mã nguồn hoàn toàn giống nhau; được viết như sau:

```
case '0':
case '1':
case '2':
        ...
break;
```

và không cần thêm chú thích nào cả.

Bài tập 1-7. Viết lại các đoạn chương trình C/C++ sau cho sáng sủa hơn:

```
?     if (isatty(stdin));
?     else if (isatty(stdout));
?           else if (istty(stderr);
?                 else return(0);
?     if (reval != SUCCESS)
?     {
?           return (retval);
?     }
?     /* Tất cả đều thực hiện */
?     return SUCCESS;
?     for (k = 0; k++ <5; x += dx)
?           scanf("%lf", &dx);
```

Bài tập 1-8. Xác định lỗi trong đoạn chương trình Java sau đây và sửa lại bằng cách dùng một vòng lặp sao cho phù hợp với đặc trưng của ngôn ngữ Java:

```
?     int count = 0;
?     while (count < total)  {
?           count ++;
?           if (this.getName(count)  ==
nametable.userName()){
?                 return (true);
?           }
?     }
```

1.4. Các hàm macro

Có một xu hướng trong các lập trình viên C là viết macro thay vì viết hàm cho các phép tính rất ngắn thường dùng; việc nhập xuất như hàm

`getchar` và kiểm tra ký tự như `isdigit` là các ví dụ được công nhận chính thức. Lý do của điều này là tốc độ thực thi: một macro tránh được những phí tổn của một lời gọi hàm. Vấn đề này không gây tranh cãi vào thời kỳ mà C mới ra đời, lúc đó máy chạy chậm và tốn nhiều chi phí cho các lời gọi hàm; nhưng nay thì điều đó không còn thích hợp nữa. Với các máy và các trình biên dịch hiện đại, các macro đôi khi gây ra nhiều điều bất lợi hơn những ích lợi mà chúng mang lại.

Tránh các hàm macro

Trong C++, sự có mặt của các hàm `inline` làm cho các hàm macro trở nên không còn cần thiết; còn trong Java, không có macro. Trong C, các hàm macro gây ra rắc rối nhiều hơn là giải quyết rắc rối.

Một trong số rắc rối nghiêm trọng nhất do các hàm macro gây nên là một tham số xuất hiện hơn một lần trong phần định nghĩa, có thể được tính hơn một lần; nếu đối số trong lời gọi hàm chứa một biểu thức có hiệu ứng lề, kết quả sẽ là một lỗi rất khó nhận thấy. Đoạn mã nguồn sau đây dự định thực hiện một trong số các phép kiểm tra ký tự từ thư viện `<ctype.h>`:

```
?       #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

Chú ý rằng tham số `c` xuất hiện hai lần trong thân của macro. Nếu `isupper` được gọi trong một ngữ cảnh như sau:

```
?       while (isupper(c = getchar()))
?           ...
```

khi đó mỗi lần một ký tự nhập lớn hơn hay bằng `A`, nó sẽ được bỏ qua và đọc tiếp ký tự khác để so sánh với `Z` (do cách thay thế nguyên văn của macro). Ngôn ngữ C chuẩn được viết cẩn thận để cho phép hàm `isupper` và các hàm tương tự đóng vai trò macro, nhưng chỉ khi chúng được bảo đảm thao tác trên đối số một lần duy nhất để cách cài đặt này không bị phá vỡ.

Tốt hơn hết là nên dùng các hàm của thư viện `ctype` thay vì tự mình cài đặt, và sẽ an toàn hơn nếu không lồng các hàm như `getchar` có hiệu ứng lề vào nhau. Viết lại phép kiểm tra trên bằng cách dùng hai biểu thức thay vì một biểu thức sẽ làm chương trình sáng sủa hơn và cũng giúp nhận

biết vị trí kết thúc tập tin một cách tường minh:

```
while((c = getchar ()) != EOF && isupper(c))
    ...
```

Đôi khi sự định nghĩa phức tạp gây vấn đề về tốc độ thực thi hơn là một lỗi tường minh. Hãy xét ví dụ sau:

```
?   #define   ROUND   TO_INT(x)   ((int)   ((x)
-(((x)>0)?0.5:-0.5)))
?       ...
?   size =  ROUND _TO_INT(sqrt(dx*dx + dy*dy);
```

Đoạn chương trình trên thực hiện phép tính lấy căn bậc hai một số gấp đôi số lần cần thiết. Ngay cả khi được cho các đối số đơn giản, một biểu thức phức tạp như phần thân của macro `ROUND_TO_INT` sẽ được dịch thành nhiều chỉ thị, mà lẽ ra nên được đặt trong chỉ một hàm để gọi ra khi cần. Việc thực thi một macro ở mọi vị trí xuất hiện của nó sẽ làm cho chương trình đã dịch trở nên lớn ra. (Các hàm `inline` của C++ cũng có trở ngại này).

Đóng ngoặc phần thân của macro và các đối số

Nếu bạn vẫn muốn dùng macro, hãy cẩn thận. Macro làm việc theo cách thay thế nguyên văn: các tham số trong định nghĩa được thay thế bằng các đối số của lời gọi và biểu thức của macro thay thế nguyên văn lời gọi macro. Đây là điều khác biệt khó chịu so với hàm. Biểu thức

```
1 / square(x)
```

sẽ cho kết quả đúng nếu `square` là một hàm, nhưng nếu đó là một macro như sau:

```
?   #define square(x)  (x) * (x)
```

biểu thức sẽ được khai triển sai như sau:

```
1 / (x) * (x)
```

Do đó, macro phải được viết lại ở dạng

```
#define square(x) ((x)*(x))
```

Tất cả các dấu ngoặc đơn trên đều cần thiết. Và việc dùng dấu ngoặc cho đúng bên trong macro cũng vẫn còn chưa giải quyết được vấn đề tính toán nhiều lần. Nếu một thao tác thông dụng trong chương trình, ta hãy dùng hàm cho thao tác đó.

Trong C++, các hàm `inline` tránh được vấn đề về cú pháp trong khi vẫn có thể cho được ưu thế về tốc độ thực thi như macro. Chúng thích hợp cho các hàm ngắn dùng để gán hoặc lấy một giá trị đơn lẻ.

Bài tập 1-9. Xác định lỗi trong định nghĩa macro sau:

```
?   #define ISDIGIT(c) ((c>= '0') && (c <= '9')) ? 1: 0
```

1.5. Các số tối nghĩa

Số tối nghĩa là các hằng số, kích thước mảng, vị trí ký tự, hệ số chuyển đổi và những giá trị số khác xuất hiện trong chương trình.

Đặt tên cho các số tối nghĩa

Nguyên tắc chung là tất cả giá trị số trừ 0 và 1 đều có thể xem là tối nghĩa và nên được đặt tên cho nó. Một con số trong mã nguồn của chương trình sẽ không thể cho biết gì về tầm quan trọng cũng như xuất xứ của nó, làm cho chương trình trở nên khó hiểu và khó sửa đổi. Đoạn trích sau được trích ra từ một chương trình xuất ra một biểu đồ về tần số của ký tự trên một màn hình văn bản 24×80 thật sự khó hiểu một cách không cần thiết vì sự xuất hiện của một loại các số tối nghĩa:

```
?       fac = lim / 20;    /* Hệ số tỷ lệ */
?       if (fac <1)
?             fac =1;
?                          /* tạo biểu đồ */
?       for (i = 0, col = 0; i< 27; i++, j-+) {
?             col += 3;
?             k = 21 - (let[i] / fac);
```

```
            star = (let[i] == 0) ? ' ' : '-';
        for (j = k; j < 22; j++)
            draw(j, col, star);
    }
    draw (23, z , ' ');  /* Đánh nhãn cho trục x */
    for (i = 'A'; i<= 'Z'; i++);
        printf(" c ", i);
```

Đoạn mã nguồn trên chứa các số như: 20, 21, 22, 23, 27. Chúng có mối liên hệ rõ ràng gì hay không? Thực tế, chỉ có 3 con số có giá trị quan trọng trong chương trình: 24 là số dòng của màn hình; 80 là số cột, và 26 là số chữ trong bảng chữ cái. Nhưng không có một con số nào trong 3 số trên xuất hiện trong đoạn mã nguồn. điều này càng làm cho những giá trị số kia trở nên kỳ bí.

Bằng cách đặt tên cho các số chính trong phép tính, ta có thể làm cho đoạn mã nguồn trở nên dễ đọc hơn. Ta thấy, ví dụ như, số 3 là kết quả của (80-1)/26. Thêm vào một số định nghĩa đơn giản, ta có kết quả sau:

```
enum {
        MINROW    = 1,              /* top edge */
        MINCOL    = 1,              /* left edge  */
        MAXROW    = 24,             /* bottom edge (<=) */
        MAXCOL    = 80,             /* right edge (<= ) */
        LABELROW  = 1,              /* position of labels */
        NLET      = 26,             /* size of alphabet */
        HEIGHT    = MAXROW - 4      /* height of bars */
        WIDTH     = (MAXCOL - 1)/NLET/* width of bars */
};

        ...
        fac = (lim - HEIGHT -1) /HEIGHT;/* Hệ số tỷ lệ */
```

```
            if (fac < 1)
                  fac = 1;
            for (i=0; i<NLET; i++) {        /* tạo biểu đồ */
                  if (let[i] == 0)
                        continue ;
                  for (j = HEIGHT - let[i]/fac; j< HEIGHT;
j++)
                        draw(j+1   ,   LABSLROW,   (i+1)*WIDTH,
'*');
            }
            draw(MAXROW-1, MINCOL -1, '    ');    /* đánh nhãn
cho trục x */
            for (i = 'A' ; i<= 'Z'; i++)
                  printf("%c", i);
```

Bây giờ những gì vòng lặp chính thực hiện đã trở nên rõ ràng hơn: đó là một vòng lặp rất thích hợp chạy từ 0 đến *NLET-1*, cho biết rằng vòng lặp chạy trên các phần tử của dữ liệu. Cũng vậy, lời gọi đến hàm *draw* cũng dễ hiểu hơn vì các từ như *MAXROW* và *MINCOL* nhắc ta nhớ thứ tự các đối số. Điều quan trọng nhất là đoạn mã nguồn trên đã có thể chạy được với kích thước màn hình hoặc kiểu dữ liệu khác. Các giá trị số đã không còn kỳ bí nữa, và cả chương trình cũng vậy.

Định nghĩa các con số dưới dạng hằng số, không dùng macro

Các lập trình viên C thường dùng theo truyền thống *#defined* để quản lý các giá trị của số tối nghĩa. Tuy bộ xử lý trước của C là một công cụ mạnh nhưng lại kém thông minh, và dùng macro là một cách lập trình nguy hiểm vì macro thay đổi cấu trúc của chương trình. Hãy dùng ngôn ngữ đúng dắn để thực hiện công việc. Trong C và C++, các hằng số nguyên có thể được định nghĩa bằng phát biểu *enum*, như ta thấy trong ví dụ trước. Hằng số mọi loại có thể được định nghĩa bằng *const* trong C++:

```
const int MAXROW = 24, MAXCOL = 80;
```

hoặc bằng *final* trong Java:

```
static final int MAXROW = 24,  MAXCOL = 80;
```

C cũng có các giá trị *const* nhưng không dùng được làm phần tử biên của mảng, do đó phát biểu *enum* vẫn là phương pháp được chọn trong C.

Dùng các hằng số dạng ký tự, không dùng dạng số nguyên

Các hàm trong thư viện *<ctype.h>* hoặc các hàm tương đương nên được dùng để kiểm tra tính chất các ký tự. Một phép kiểm tra như sau:

```
?     if (c>= 65 && c <= 90)
?     ...
```

hoàn toàn phụ thuộc vào một bảng mã ký tự đặc thù nào đó. Tốt hơn nên dùng:

```
?     if (c>= 'A' && c<= 'Z')
?     ...
```

nhưng có thể lại không có hiệu quả mong muốn nếu các ký tự không được sắp thứ tự liên tục trong bảng mã hoặc nếu bảng chữ cái còn chứa những ký tự khác. Cách tốt nhất là dùng thư viện trong C hay C++ như sau:

```
if ( isuppper (c))
...
```

còn trong Java dùng:

```
if (Character.isUpperCase(c))
...
```

Một vấn đề có liên quan là số *0* rất thường xuất hiện trong nhiều chương trình, nhiều ngữ cảnh. Trình biên dịch sẽ chuyển đổi sang kiểu thích hợp, song nếu kiểu của mỗi giá trị 0 được rõ ràng sẽ có thể giúp người đọc dễ hiểu vai trò của chúng. Ví dụ, dùng *(void*)0* hay *NULL* cho một con trỏ rỗng trong C, và dùng *'\0'* thay vì *0* cho byte rỗng ở cuối một chuỗi. Nói

cách khác, đừng viết:

```
?       str = 0;
?       name[i] = 0;
?       x = 0;
```

mà hãy viết:

```
str = NULL;
name[i] = '\0';
x   0.0;
```

Chúng tôi thích dùng những hằng số tường minh khác nhau, và dùng số 0 để chỉ giá trị số nguyên, vì mỗi hằng số như vậy biểu thị từng công dụng của giá trị 0 và vì vậy cung cấp được thêm chút ít thông tin về số *0* đó. Tuy nhiên, trong C++, ký hiệu được chấp nhận cho một con trỏ rỗng lại là `0` hơn là `NULL`. Còn Java giải quyết vấn đề này tốt hơn cả bằng cách định nghĩa từ khóa `null` cho một tham chiếu đối tượng không trỏ đến một cái gì cả.

Dùng chính ngôn ngữ đó để tính toán kích thước của đối tượng

Đừng dùng một giá trị kích thước tường minh cho bất kỳ kiểu dữ liệu nào; ví dụ, hãy dùng `sizeof(int)` thay vì giá trị 2 hay 4. Cũng lý do tương tự, `sizeof(array[0])` tốt hơn `sizeof(int)` vì cần phải thay đổi chương trình ít hơn khi kiểu mảng thay đổi.

Hàm `sizeof` đôi khi là một cách thuận tiện giúp tránh phải dùng thêm tên cho giá trị kích thước mảng. Ví dụ, nếu ta viết:

```
char buf [1024];
fgets(buf, sizeof(buf), stdin );
```

thì kích thước của bộ đệm vẫn còn là một số tối nghĩa, nhưng chỉ xuất hiện có một lần trong phần khai báo. Có thể không cần thiết phải đặt tên cho kích thước của một mảng cục bộ, nhưng rất cần thiết viết mã nguồn sao cho không phải sửa đổi lại nếu kích thước hay kiểu mảng thay đổi.

Các mảng của Java có trường `length` trả ra giá trị số phần tử:

```
char buf [] = new char[1024];

for (int i=0; i< buf .length; i++)

...
```

Không có trường `.length` tương đương trong C và C++, nhưng đối với một mảng (không áp dụng cho con trỏ) có khai báo tường minh, macro sau tính số phần tử của mảng:

```
#define     NELEMS(array)     (sizeof(array)
sizeeof(array[0]))

double  dbuf[100];

for (i = 0; i < NELEMS(dbuf); i++)

...
```

Kích thước của mảng được xác định ở một nơi duy nhất; phần mã nguồn còn lại sẽ không phải thay đổi gì nếu kích thước mảng thay đổi. Không có vấn đề gì xảy ra với phép tính nhiều lần cho đối số của macro ở đây, vì không có một hiệu ứng lề nào, và thực tế thì phép tính được thực hiện trong khi chương trình được biên dịch. Đây là cách dùng macro thích hợp vì macro này thực hiện một công việc mà một hàm không thực hiện được: tính kích thước của một mảng ngay từ phần khai báo mảng đó.

Bài tập 1-10. Hãy viết lại các định nghĩa sau đây để giảm thiểu nguy cơ mắc lỗi

```
?     #define FT2METER        0.3048
?     #define METER2FT        3.28084
?     #define MI2FT           5280.0
?     #define MI2KM           1.609344
?     #define SQMI2SQKM       2.589988
```

1.6. Chú thích

Chú thích dành cho việc giúp người đọc chương trình. Chú thích giúp người đọc không phải bằng cách nhắc những điều mã nguồn đã nêu, hay bằng cách phủ nhận mã nguồn, cũng như không phải bằng cách làm rối trí người đọc với cách trình bày phức tạp mà còn mắc lỗi. Chú thích tốt nhất trợ giúp cho sự thấu hiểu một chương trình bằng cách chỉ ra một cách ngắn gọn những chi tiết đáng chú ý hoặc bằng cách cung cấp một cái nhìn chi tiết hơn về tiến trình của chương trình.

Đừng cố gắng chú thích những điều hiển nhiên

Chú thích không nên trình bày lại những thông tin tự nó đã rõ. Dưới đây là một số trường hợp chú thích không có giá trị:

```
?       /*
?       * default
?       */
?       default:
?               break;
?
?       /* return SUCCESS */
?       return SUCCESS;
?       zerocount++;        /* Tăng biến zerocounter */
?
?       /* Khởi tạo "total" bằng "number received" */
?       node -> total   node->number_received;
```

Tất cả các chú thích trên đây đều nên xóa đi vì chúng chỉ là một mớ lộn xộn.

Chú thích nên bổ sung những gì không trực tiếp thấy ngay từ mã nguồn, hay nên gom lại vào một chỗ thông tin nằm rải rác nhiều nơi. Khi một điều gì khó nhận thấy xảy ra, một chú thích có thể giúp làm sáng tỏ,

nhưng nếu hành động đã hiển hiện rõ ràng, nhắc lại thành lời văn là vô nghĩa:

```
?     while ((c = getchar()) != EOF && isspace(c))
?           ;                    /* bỏ qua ký tự trống */
?     if (c == EOF)              /* kết thúc file */
?           type    endoffile;
?     else if (c == '(')         /* là ngoặc đơn trái */
?           type = leftparent;
?     else if (c == ')')         /* là ngoặc đơn phải */
?           type = rightparent;
?     if (c == ';')              /* là dấu chấm phẩy */
?           type = semicolon;
?     else if (is_op(c))         /* là thao tác */
?           type = operator;
?     else if (isdigit(c))       /* là số */
?           ...
```

Những chú thích trên cũng nên xóa đi, vì tên được chọn cẩn thận đã tự truyền tải thông tin.

Chú thích cho hàm và dữ liệu toàn cục

Chú thích rõ ràng là có thể có ích. Ta chú thích cho các hàm, các biến toàn cục, các định nghĩa hằng, các trường trong các cấu trúc và lớp, và tất cả những gì mà một lời tóm tắt ngắn gọn có thể giúp cho việc nắm bắt chương trình.

Các biến toàn cục có xu hướng làm thành từng cụm nằm rải rác trong chương trình; một chú thích sẽ đóng vai trò một lời nhắc để tham khảo khi cần. Sau đây là một ví dụ trích ở Chương 3 của cuốn sách này:

```
struct State { /* tiếp đầu ngữ+danh sách tiếp vĩ ngữ */
      char *pref[NPREF];       /* từ tiếp đầu ngữ */
```

```
    Suffix *suf;      /* danh sách tiếp vĩ ngữ */
    State  *next ;  /* con trỏ next trong bảng băm */
};
```

Lời chú thích cho một hàm sẽ giúp cho người đọc hiểu được từng giai đoạn thực hiện của chương trình. Nếu mã nguồn không quá dài hoặc có tính kỹ thuật cao, chỉ một dòng chú thích đã đủ:

```
// Hàm random: trả về một số nguyên trong đoạn [0..r-1]
    int random(int r)
    {
        return (int)(Math.floor(Math.random()*r));
    }
```

Đôi khi mã nguồn khó thực sự, có thể vì thuật toán phức tạp hoặc vì cấu trúc dữ liệu rắc rối, khó hiểu. Trong trường hợp đó, một lời chú thích giúp cho người đọc hiểu được bản chất của vấn đề. Cũng rất có giá trị khi gợi ý lý do vì sao đã thực hiện một rẽ nhánh. Chú thích dưới đây giới thiệu một sự thi hành rất hiệu quả một khai triển hàm cos rời rạc dùng trong một bộ giải mã hình ảnh dạng JPEG.

```
/*
*  idct: Scaled integer implementation of
*  Inverse    two dimensional 8x8 Discrete Cosine
Transform,
* Chen-Wang  algorithm (IEEE ASSP-32, pp 803-816, Aug
1984)
*
* 32 - bit integer arithmetic (8 - bit coefficients)
* 11 multiplies, 29 adds per DCT
*
```

```
* Coefficients extended to 12 bits for IEEE 1180 - 1990
compliance
*/
      static void idct(int b[8*8])
      {
            ...
      }
```

Chú thích hữu ích này cho biết ra tài liệu tham khảo, mô tả ngắn gọn dữ liệu sử dụng, trình bày ngắn gọn khả năng thực thi của thuật toán, và cho biết bằng cách nào và tại sao thuật toán gốc được sửa chữa.

Đừng chú thích cho những đoạn mã nguồn dở mà hãy viết lại nó

Hãy chú thích những gì bất thường hay có khả năng gây nhầm lẫn, nhưng khi chú thích trở nên lấn át mã nguồn chương trình thì cần xem xét sửa chữa mã nguồn. Ví dụ sau đây dùng một chú thích dài, rối rắm và dùng một phát biểu điều kiện biên dịch cho phát biểu `printf` để giải thích cho chỉ một phát biểu:

```
?     /* Nếu result bằng 0 một sự tương ứng được tìm
thấy vì thế giá trị
?     trả về là true (khác 0). Ngược lại, result khác 0
thì giá trị trả
?     về là false (bằng 0). */
?     #ifdef  DEBUG
?     printf("*** trả về !result    %d\n", !result);
?     fflush(stdout);
?     #endif
?
?     return(!result);
```

Dạng phủ định rất khó hiểu và nên tránh dùng. Một phần của rắc rối

là ở tên biến `result` khó hiểu. Một tên có tính gợi nhớ hơn, như `matchfound`, sẽ làm cho chú thích trở nên không cần thiết, cũng không cần luôn phát biểu `printf`.

```
#ifdef  DEBUG

printf("** tra về giá trị tương ứng matchfound
%d\n",matchfound);

fflush(stdout);

#endif

return(matchfound);
```

Đừng phủ nhận mã nguồn

Hầu hết các chú thích đều khớp với mã nguồn lúc ban đầu, nhưng về sau khi lỗi được sửa chữa và chương trình được phát triển, các chú thích thường vẫn không thay đổi so với trước, kết quả là các chú thích không khớp với mã nguồn nữa. Đây là lời giải thích hợp lý cho sự không nhất quán trong ví dụ mở đầu chương này.

Cho dù nguồn gốc của sự thiếu nhất quán là gì đi nữa, một chú thích mâu thuẫn với mã nguồn chương trình sẽ gây nhầm lẫn, và rồi phải tốn một khoảng thời gian dài để sửa lỗi một cách vô ích vì nhầm tưởng một chú thích sai là đúng. Khi thay đổi mã nguồn, hãy luôn chắc chắn rằng chú thích vẫn luôn được chính xác.

Chú thích không phải chỉ nên khớp với mã nguồn mà còn nên trực tiếp giải thích cho mã nguồn. Chú thích trong ví dụ sau là đúng - nhằm giải thích cho mục đích của hai dòng kế tiếp - nhưng có vẻ như mâu thuẫn với mã nguồn: vì chú thích giải thích về ký tự xuống dòng còn mã nguồn lại nói về khoảng trắng:

```
?     time (&now);
?     strcpy(date, ctime(&now));
?     /* bỏ ký tự xuống dòng đã được copy từ ctime */
?     i    0;
```

```
?       while (date[i] >= ' ') i ++;
?       date[i] = 0;
```

Một cách cải tiến là viết lại mã nguồn cho phù hợp với đặc trưng của ngôn ngữ hơn:

```
?       time(&now);
?       strcpy(date, ctime(&now));
?       /* bỏ ký tự xuống dòng đã được copy từ ctime */
?       for (i = 0; date[i] != '\n'; i++)
?             ;
?       date[i] = '\0';
```

Mã nguồn và chú thích bây giờ đã khớp với nhau, nhưng vẫn có thể cải tiến thêm cả hai bằng cách viết cho trực tiếp hơn. Vấn đề là phải xóa ký tự xuống dòng mà `ctime` đưa vào ở cuối chuỗi nó trả về. Chú thích cần giải thích điều đó, và mã nguồn cũng cần thực hiện như thế:

```
time(&now);
strcpy(date, ctime(&now));
/* ctime()đưa ký tự xuống hàng vào cuối chuỗi; xóa nó */
date[strlen(date)-1] = '\0';
```

Biểu thức sau cùng trên phù hợp với đặc tính của C dùng để xóa ký tự cuối cùng trong một chuỗi. Mã nguồn bây giờ đã trở nên ngắn gọn, phù hợp với đặc tính của ngôn ngữ, sáng sủa, và chú thích đã trực tiếp giải thích được lý do vì sao đưa vào chú thích như vậy.

Rõ ràng, không gây nhầm lẫn

Chú thích được coi là để giúp cho người đọc ở những chỗ khó chứ không phải để gây thêm khó khăn cho họ. Ví dụ sau đây theo đúng nguyên tắc đưa chú thích cho các hàm và giải thích các đặc tính bất thường; nhưng ngược lại, hàm được dùng là hàm `strcmp` và đặc tính bất thường nằm ngoài công việc cần thực hiện, cài đặt theo giao tiếp chuẩn thông thường:

```
?     int stromp(char *s1, char *s2 )
?     /* hàm so sánh chuỗi trả về  -1 nếu s1 */
?     /* trước s2 theo thứ tự từ điển, 0 nếu bằng */
?     /* và trả về 1 nếu s1 sau s2  */
?     {
?           while (*s1  ==  *s2 ) {
?                 if (*s1  == '\0') return(0);
?                 s1++;
?                 s2-+;
?           }
?           if (*s1 > *s2) return(1);
?           return(-1);
?     }
```

Khi chỉ một vài từ không đủ để giải thích mã nguồn đang thực hiện, điều đó thường là dấu hiệu cho biết rằng nên viết lại đoạn mã nguồn đó. Ở đây, mã nguồn có thể cải tiến được, song vấn đề thực sự là ở chú thích, nó dài cũng xấp xỉ như phần thân chương trình và cũng gây nhầm lẫn, vì chữ `trước` và `sau` trong phần chú thích trên chưa đúng. Ở đây ta đã quá thổi phồng để nói lên rằng thủ tục này rất khó hiểu, nhưng vì nó cài đặt một hàm chuẩn, lời chú thích có thể giúp ích theo cách tóm tắt các hành động và cho biết các định nghĩa bắt đầu ở đâu; đó là tất cả những gì cần thiết:

```
      /* Hàm stromp; trả về < 0 nếu s1<s2 , > 0 nếu s1
>s2, 0 nếu chúng bằng nhau */
      /*    ANSI C, phần 4.11.4.2 */
      int stromp(const char *s1, const char *s2)
      {
            ...
      }
```

Các sinh viên thường được dạy rằng cần chú thích mọi thứ. Những lập trình viên chuyên nghiệp thường được yêu cầu phải chú thích tất cả mã nguồn của họ. Nhưng mục đích của chú thích có thể không giữ được nếu cứ tuân theo những nguyên tắc đó một cách mù quáng. Chú thích là phương tiện để giúp cho người đọc hiểu những phần chương trình không dễ hiểu nếu chỉ có mã nguồn chương trình. Hãy viết mã nguồn càng dễ hiểu càng tốt; làm được điều này tốt chừng nào thì càng cần ít chú thích chừng đó. Mã nguồn viết tốt cần ít chú thích hơn là mã nguồn viết dở.

Bài tập 1-11. Hãy bình luận những lời chú thích sau:

```
?     void dict :: insert(string& w)
?     // trả về 1 nếu w có trong từ điển, ngược lại trả
về 0
?     if (n > MAX  || n%2 > 0) // kiểm tra trường hợp
số chẵn
?     // viết một thông báo
?     // thêm vào line_number mỗi khi mỗi dòng được
viết
?     void write_message()
?     {
?           // tăng biến line_nuber
?           line_number = line_number  + 1;
?           fpintf (fout, "%d %s\n%d  %s\n%d %s\n",
?                 line_number, HEADER,
?                 line_number -1, BODY,
?                 line number +2, TRAILER);
?           // tăng biến line nuber
?           line_number = line_number + 2;
?     }
```

1.7. Vì sao phải lo lắng về phong cách lập trình?

Trong chương này ta đã nói về những điều quan tâm chủ yếu trong phong cách lập trình: tên có tính gợi nhớ, sự sáng sủa trong các biểu thức, dòng điều khiển rõ ràng; mã nguồn và chú thích dễ hiểu, và tầm quan trọng của việc dùng một cách nhất quán các cách viết được quy ước cũng như các cách viết phù hợp với đặc tính của ngôn ngữ trong việc đạt được những điều trên. Không thể chối cãi rằng những điều này là vô ích.

Nhưng vì sao lại phải lo lắng về phong cách? Có ai quan tâm đến hình thức bên ngoài của chương trình một khi nó chạy tốt? Không phải là sẽ mất nhiều thời gian hơn để làm cho chương trình trông đẹp mắt hơn sao? Không phải là các quy tắc dù sao cũng đều là tùy ý sao?

Câu trả lời là mã nguồn chương trình được viết tốt sẽ dễ đọc và dễ hiểu hơn, và hầu như chắc chắn là ít lỗi hơn, và có khả năng có kích thước nhỏ hơn so với mã nguồn được chắp vá cẩu thả và không hề trau chuốt. Trong tình huống cấp bách cần hoàn thành chương trình cho kịp với một thời hạn nào đó, thường vấn đề phong cách dễ bị gác sang một bên để giải quyết sau. Quyết định này có thể sẽ phải trả giá đắt. Một số ví dụ trong chương này đã cho thấy có thể mắc sai lầm như thế nào nếu không chú trọng đến phong cách đúng mức. Mã nguồn tùy tiện cũng là mã nguồn dở – không chỉ nguy hiểm và khó đọc mà còn có thể gây nên nhiều hậu quả nghiêm trọng khác.

Quan điểm cơ bản là phong cách tốt phải là vấn đề thói quen. Nếu ngay từ ban đầu khi viết mã nguồn bạn đã nghĩ đến vấn đề phong cách, và nếu bạn dành thời gian để duyệt lại và cải tiến, bạn sẽ tập cho mình một thói quen tốt. Một khi đã trở nên quen, tiềm thức của bạn sẽ trợ giúp cho bạn trong nhiều chi tiết vụn vặt, và ngay cả khi bạn phải viết mã nguồn trong trạng thái chịu áp lực công việc cao, mã nguồn của bạn cũng sẽ tốt hơn.

Chương 2

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Nghiên cứu về thuật toán và cấu trúc dữ liệu là nền tảng của ngành khoa học máy tính, nó bao gồm nhiều kỹ thuật tinh tế và phân tích toán học phức tạp. Một thuật toán hay một cấu trúc dữ liệu tốt sẽ giúp giải quyết một bài toán chỉ trong vài giây, ngược lại sẽ phải mất nhiều năm mới giải được.

Trong các lĩnh vực chuyên biệt như đồ họa, cơ sở dữ liệu, phân tích cú pháp, phân tích số học, và mô phỏng, khả năng giải các bài toán này phụ thuộc rất nhiều vào thuật toán và cấu trúc dữ liệu có tốt hay không. Nếu bạn cài đặt các chương trình mà mình chưa thực hiện bao giờ, thì bạn phải tìm được những vấn đề liên quan đã được giải quyết rồi. Đừng lãng phí thời gian cho những vấn đề phức tạp mà đã được làm tốt rồi.

Mọi chương trình đều phụ thuộc vào thuật toán và cấu trúc dữ liệu, nhưng có một số lại cần đến sự sáng tạo ra các phương pháp mới hoàn toàn. Ngay cả đối với một chương trình khá phức tạp như trình biên dịch hay trình duyệt Web, đều dùng hầu hết các cấu trúc dữ liệu là mảng, danh sách liên kết, cây, và bảng băm. Một chương trình đòi hỏi sự tối ưu nó sẽ phải dựa trên những cái đơn giản. Do đó, đối với các lập trình viên, nhiệm vụ là phải biết cách áp dụng những thuật toán và cấu trúc dữ liệu nào sẵn có phù hợp với bài toán của mình.

Tóm lại, chỉ có một vài thuật toán cơ bản được dùng trong hầu hết các chương trình tìm kiếm và sắp xếp cơ bản, thậm chí còn thường được đưa vào thư viện dùng chung. Tương tự, mọi cấu trúc dữ liệu cũng dẫn xuất từ các cấu trúc dữ liệu cơ bản. Vì vậy, nội dung trong chương này cũng sẽ quen thuộc với hầu hết các lập trình viên. Chúng tôi đã đưa ra vài phiên bản sẵn có để thảo luận và bạn có thể cài đặt cụ thể nếu cần thiết, nhưng bạn nên

tìm hiểu những gì ngôn ngữ lập trình và các thư viện đã cung cấp trước khi cài đặt.

2.1. Tìm kiếm

Cấu trúc mảng là lựa chọn tốt nhất để lưu trữ dữ liệu dạng bảng tĩnh. Việc khởi tạo các mảng lúc biên dịch sẽ rất dễ và ít tốn kém. (Trong Java, quá trình khởi tạo diễn ra tại thời điểm thực thi chương trình, nhưng điều này không quan trọng lắm trong việc cài đặt chi tiết trừ khi kích thước mảng khá lớn). Ví dụ chương trình phát hiện ra các từ được dùng khá nhiều trong một đoạn văn rườm rà, ta có thể viết như sau:

```
char *flab[] = {
        "actually",
        "just",
        "quite",
        "ready",
        NULL
};
```

Thủ tục tìm kiếm phải cần biết có bao nhiêu phần tử có trong mảng. Ta có thể đưa kích thước của mảng làm tham số đầu vào; hoặc cách ta dùng ở đây là đánh dấu cuối mảng bằng NULL:

```
/* Hàm lookup : tìm kiếm tuần tự các hàm trong mảng */
int lookup(char *word , char *array[])
{
        int i;
        for (i= 0; array[i] != NULL; i++)
                if (strcmp(word, array[i]) == 0)
                        return i ;
        return  -1;
}
```

Trong C và C++, ta có thể khai báo mảng kiểu chuỗi như sau: `char *array[]` hay `char **array`. Mặc dù 2 cách này là tương đương, nhưng cách thứ nhất cho ta thấy cách sử dụng tham số rõ ràng hơn.

Thuật toán tìm kiếm trên được gọi là *tìm kiếm tuần tự* vì nó sẽ tìm lần lượt từng phần tử để tìm ra phần tử mong muốn. Khi số lượng các phần tử tương đối nhỏ, phương pháp tìm kiếm tuần tự thực hiện đủ nhanh. Các thư viện chuẩn cũng hỗ trợ các hàm tìm kiếm tuần tự cho các kiểu dữ liệu khác nhau chẳng hạn như hàm `strchr` và `strstr`: tìm ra ký tự hay chuỗi con đầu tiên trong một chuỗi C hay C++, trong Java lớp `String` cũng có hàm tương tự `indexOf`, và thuật toán tìm kiếm tổng quát trong C++ được ứng dụng với mọi kiểu dữ liệu. Do đó, nếu một hàm nào đó có thể dùng được với kiểu dữ liệu của bạn, thì bạn có thể dùng hàm đó.

Phương pháp tìm kiếm tuần tự rất dễ, tuy nhiên số phép thực hiện tỷ lệ với số lượng dữ liệu được tìm kiếm; nếu gấp đôi số lượng phần tử thì thời gian tìm kiếm cũng sẽ nhân đôi trong trường hợp phần tử cần tìm không xuất hiện trong mảng. Do thời gian thực hiện tuyến tính với kích thước dữ liệu nên phương pháp này còn được gọi là *tìm kiếm tuyến tính.*

Sau đây là đoạn chương trình minh họa kiểu dữ liệu mảng được trích từ chương trình phân tích từ loại HTML, trong đó định nghĩa các từ dựa trên hơn 100 ký tự đơn:

```
typedef struct Nameval  Nameval ;
struct Nameval {
      char  *name;
      int   value;
};
/* Biến name là chuỗi, chẳng hạn như AElig là chữ ghép
giữa A và E. */
/* Biến value là các giá trị được mã hóa ra dưới dạng
Unicode (Unicode/ISO10646). */
```

```
Nameval htmlchars[] = {
        "AElig" ,    0x00c6,
        "Aacute",    0x00c1,
        "Acirc",     0x00c2,
        /* ... */
        "Zeta",      0x03b6,
};
```

Đối với mảng có kích thước lớn như trên, việc sử dụng *tìm kiếm nhị phân* sẽ hiệu quả hơn. Thuật toán tìm kiếm nhị phân là một phiên bản ban đầu của phương pháp tra cứu tự điển. Kiểm tra phần tử ở giữa, nếu giá trị này lớn hơn giá trị mà chúng ta cần tìm thì chúng ta tiếp tục tìm trong nữa phần đầu, ngược lại ta tìm trong nữa phần sau. Lặp lại tìm kiếm cho đến khi nào tìm thấy được giá trị cần tìm hay xác định được nó không có.

Đối với phương pháp tìm kiếm nhị phân, bảng dữ liệu phải là bảng dữ liệu đã được sắp xếp, và chúng ta cũng phải biết trước được độ dài của bảng. Macro NELEMS trong Chương 1 có thể hỗ trợ điều này:

```
        printf ("The HTML table has %d words\n",
NELEMS(htlmchars));
```

Một hàm tìm kiếm nhị phân cho dạng bảng dữ liệu này có thể trình bày như sau:

```
        /* Hàm lookup: thực hiện tìm kiếm nhị phân và trả về
index */
        int lookup(char *name , Nameval tab[] , int ntab)
        {
                int low, high, mid, cmp ;
                low  = 0;
                high = ntab - 1;
                while ( low <= high ) {
```

```
            mid = ( low + high ) / 2;
            cmp = strcmp(name, tab[mid].name);
            if (cmp < 0)
                  high = mid -1;
            else if (cmp > 0)
                  low = mid + 1;
            else  /* tìm thấy */
                  return mid;
      }
   return - 1; /* không tồn tại trong bảng dữ liệu */
}
```

Khi đó, lời gọi hàm:

half = lookup ("frac12", htmlchars, NELEMS (htmlchars));

sẽ thực hiện việc tìm kiếm chỉ số mà chuỗi `"frac12"` trong mảng chuỗi ký tự `htmlchars`

Tìm kiếm nhị phân sẽ giảm bớt một nửa dữ liệu trong mỗi bước. Do đó, số bước sẽ tương đương với số lần chúng ta có thể chia n cho 2 (khoảng $\log_2 n$). Nếu ta có 1000 phần tử, tìm kiếm tuyến tính sẽ mất đến 1000 bước, trong khi đó theo cách tìm kiếm nhị phân chỉ mất khoảng 10 bước. Nếu có 1 triệu phần tử, tìm kiếm tuyến tính mất 1 triệu bước còn tìm kiếm nhị phân mất khoảng 20 bước. Càng nhiều phần tử, phương pháp tìm kiếm nhị phân càng hiệu quả hơn. Dựa trên kích thước dữ liệu nhập (sẽ thay đổi tuỳ theo cài đặt), tìm kiếm nhị phân nhanh hơn tìm kiếm tuyến tính rất nhiều.

2.2. Sắp xếp

Tìm kiếm nhị phân chỉ thực hiện khi các phần tử đã được sắp xếp. Nếu việc tìm kiếm được dự định thực hiện nhiều lần trên một tập dữ liệu, thì nên sắp xếp dữ liệu này trước và sau đó dùng giải thuật tìm kiếm nhị phân để thực hiện việc tìm kiếm. Nếu tập dữ liệu được biết trước, thì ta có thể sắp

xếp chúng trước trong khi ta viết chương trình và đưa nó vào trong phần khởi tạo. Nếu như tập dữ liệu chưa được biết trước thì nó sẽ được sắp trong khi chạy chương trình.

Một trong những thuật toán sắp xếp tốt nhất là quicksort đã được C.A.R. Hoare đưa ra vào năm 1960. Quicksort là một ví dụ rất tốt về cách làm giảm bớt các tính toán phụ. Nó thực hiện bằng cách chia một mảng nhiều phần tử thành 2 phần: một phần nhỏ và một phần lớn:

```
Lấy 1 phần tử p trong mảng (phần tử làm mốc)
Phân chia các phần tử còn lại thành 2 phần
      "phần nhỏ" gồm các phần tử có giá trị nhỏ hơn p,
và
      "phần lớn" gồm các phần tử có giá trị lớn hơn p.
Lặp lại sắp xếp tương tự trong mỗi phần.
```

Khi quá trình này kết thúc thì mảng đã được sắp theo thứ tự. Thuật toán Quicksort chạy nhanh là vì một khi 1 phần tử được biết nhỏ hơn phần tử làm mốc, ta không cần phải so sánh nó với bất kỳ phần tử nào trong nhóm lớn: tương tự như vậy, các phần tử nhóm lớn không cần so sánh với các phần tử của nhóm nhỏ. Điều này làm nó nhanh hơn các thuật toán sắp xếp đơn giản khác như: thuật toán chèn (insertion sort), thuật toán nổi bọt (bubble sort) là so sánh từng phần tử trực tiếp với mọi phần tử khác.

Thuật toán Quicksort khá thực tế và hiệu quả; nó cũng được nghiên cứu mở rộng theo nhiều hướng khác nhau. Phiên bản mà chúng tôi cung cấp cho các bạn ở đây chỉ là cách cài đặt đơn giản nhất và dĩ nhiên là không thể nhanh nhất.

Hàm quicksort sau sắp xếp 1 mảng các số nguyên:

```
/* Hàm quicksort: sắp xếp mảng v[0]..v[n-1] thành mảng
có thứ tự tăng */
void quicksort (int v[], int n )
{
```

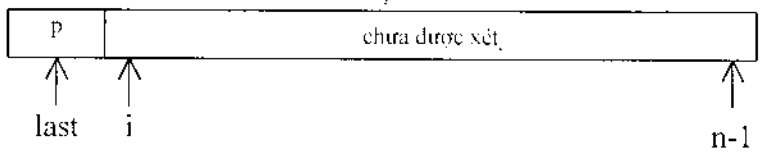
```
        int i, last;
        if (n <= 1)
              return ;
        swap(v, 0, rand()%n);    /* đổi chỗ phần tử làm
                                    mốc với phần tử v[0] */
        last = 0;
        for (i = 1; i < n; i++) /* phân vùng */
              if (v[i] < v[0])
                    swap(v, ++last, i );
        swap(v, 0, last); /* phục hồi giá trị làm mốc */
        quicksort(v, last);/* gọi đệ quy cho phần đầu */
        quicksort(v+last+1, n-last-1);/*gọi đệ quy cho
phần sau*/
  }
```

Thao tác `swap` (đổi chỗ giữa 2 phần tử trong mảng) xuất hiện 3 lần trong trong thuật toán `quicksort`, do đó cách tốt nhất là tách thành hàm riêng như sau:

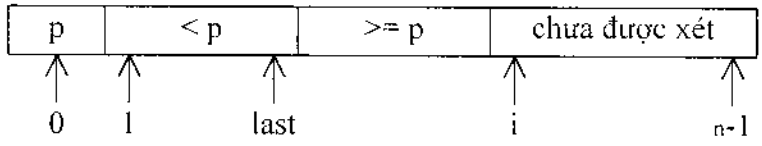
```
/* Hàm swap : đổi chỗ v[i] và v[j] */
void swap (int v[], int i, int j)
{
      int temp;
      temp = v[i];
      v[i] = v[j];
      v[j] = temp;
}
```

Việc chia vùng sẽ chọn một phần tử ngẫu nhiên làm mốc, tạm thời đổi chỗ nó với phần tử đầu của mảng, sau đó duyệt hết các phần tử còn lại, chuyển những phần tử nhỏ hơn phần tử làm mốc lên đầu (tại vị trí `last`) và

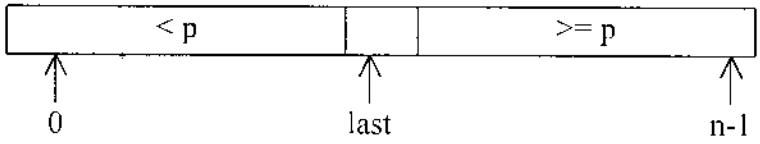
các phần tử lớn hơn về cuối (tại vị trí `i`). Lúc bắt đầu tiến trình, ngay sau khi phần tử làm mốc đổi chỗ với phần tử đầu gán `last = 0` và từ phần tử thứ `i = 1` đến phần tử thứ `n-1` của mảng chưa được xét:



Sau vài bước lặp thì từ phần tử thứ 1 đến phần tử `last` đều nhỏ hơn phần tử làm mốc, từ phần tử `last+1` đến `i-1` đều lớn hơn hoặc bằng phần tử làm mốc, và phần tử `i` đến `n-1` là chưa được xét. Cho đến khi `v[i] >= v[0]`, thuật toán sẽ đổi chỗ `v[i]` với chính nó; điều này lãng phí 1 số bước nhưng không gây ảnh hưởng nhiều:



Sau khi mọi phần tử đã được phân vùng, phần tử 0 sẽ được chuyển đổi với phần tử `last` để đặt phần tử mốc vào vị trí cuối của phần đầu (phần nhỏ); điều này duy trì thứ tự sắp xếp trong phần đầu. Khi đó mảng sẽ có dạng:



Quá trình tương tự được áp dụng cho mảng con trái và mảng con phải; khi kết thúc, toàn bộ mảng sẽ được sắp xếp.

Thuật toán `quicksort` nhanh như thế nào? Trong trường hợp tốt nhất.

- Lần thứ 1, chia các phần tử thành 2 nhóm mỗi nhóm n/2 phần tử.
- Lần phân vùng thứ 2, chia 2 nhóm n/2 phần tử thành 4 nhóm mỗi nhóm n/4 phần tử.
- ...

Điều này tiếp tục cho đến khoảng cấp $\log_2 n$, do đó tổng số bước thực hiện trong trường hợp tốt nhất xấp xỉ $n+2\times n/2 + 4\times n/4 + 8\times n/8 + ...$ ($\log_2 n$ số hạng), bằng $n\log_2 n$. Trong trường hợp trung bình do dựa trên logarit cơ số 2 nên ta nói độ phức tạp của thuật toán quicksort xấp xỉ *nlogn*.

Việc cài đặt `quicksort` rất rõ ràng nhưng cũng có một điểm yếu. Nếu mỗi lần chọn phần tử mốc chia các phần tử thành 2 nhóm xấp xỉ nhau thì phân tích của chúng ta là chính xác, nhưng nếu việc phân chia không xấp xỉ diễn ra thường xuyên thì độ phức tạp của thuật toán sẽ tăng lên gần bằng n^2. Bảng cài đặt của chúng ta dùng phần tử ngẫu nhiên làm phần tử mốc sẽ làm giảm bớt khả năng mà các dữ liệu bất thường gây ra quá nhiều việc chia vùng chênh lệch. Nhưng nếu tất cả các giá trị nhập đều bằng nhau thì cài đặt của chúng ta chỉ chia ra được 1 phần tử trong mỗi lần và do đó độ phức tạp của thuật toán sẽ xấp xỉ n^2.

Các thuật toán phụ thuộc khá nhiều vào dữ liệu nhập. Các dữ liệu nhập xấu hoặc bất lợi có thể gây ra kết quả không tốt cho thuật toán, làm nó chạy khá chậm hay phải dùng rất nhiều bộ nhớ. Trong trường hợp của thuật toán quicksort, mặc dù việc cài đặt đơn giản trên đôi khi chạy chậm, nhưng nếu cải tiến hơn sẽ giảm được khả năng xấu đó xuống gần bằng không.

2.3. Thư viện hàm

Các thư viện chuẩn cho C và C++ gồm các hàm sắp xếp mà chúng

chạy tốt đối với các dữ liệu nhập bất lợi và được điều chỉnh chạy nhanh nhất có thể.

Một số thư viện viết sẵn các hàm sắp xếp cho bất kỳ kiểu dữ liệu nào, khi đó ta cần phải biết cách giao tiếp với chúng. Trong C, xét hàm thư viện `qsort`, ta cần cung cấp hàm so sánh cho hàm `qsort` này, nó sẽ được hàm `qsort` gọi đến bất cứ khi nào cần so sánh 2 giá trị. Vì các giá trị có thể thuộc bất cứ loại nào, hàm so sánh có 2 con trỏ `void*` trỏ đến các thành phần dữ liệu sẽ được so sánh. Hàm ép kiểu các con trỏ thành một kiểu thích hợp, trích các giá trị dữ liệu, so sánh chúng và trả về kết quả (âm, bằng không hoặc dương tùy thuộc vào giá trị phần bé, bằng hay lớn hơn giá trị kia).

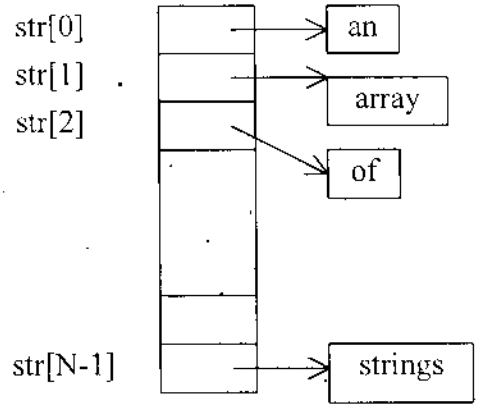
Sau đây là một cài đặt cho việc sắp xếp một mảng các chuỗi, trường hợp này thường gặp. Chúng ta định nghĩa hàm `scmp` để ép kiểu các đối số và gọi hàm `strcmp` để so sánh:

```
/*Hàm scmp: so sánh chuỗi *p1 và *p2 */
int scmp(const void *p1, const void *p2)
{
      char *v1, *v2;
      v1 = *(char **)p1;
      v2 = *(char **)p2 ;
      return strcmp(v1, v2);
}
```

Chúng ta có thể viết hàm này bằng một dòng, nhưng dùng các biến tạm làm cho chương trình dễ đọc hơn.

Chúng ta không thể sử dụng hàm `strcmp` trực tiếp như một hàm so sánh được bởi vì *qsort* truyền địa chỉ của mỗi phần tử trong mảng, `&str[i]` (thuộc kiểu `char**`), không phải `str[i]` (kiểu `char*`) như được mô tả trong hình sau:

mảng N con trỏ:



Để sắp các phần tử từ `str[0]` đến `str[N-1]` của mảng các chuỗi, `qsort` phải được gọi với đối số là một mảng, chiều dài của mảng và kích thước của các phần tử được sắp xếp và một hàm so sánh như sau:

```
char *str[N];
qsort(str, N, sizeof(str[0]), scmp);
```

Sau đây là một hàm tương tự `icmp` để so sánh các số nguyên:

```
/* Hàm icmp: so sánh 2 số *p1 và *p2 */
int icmp(const void *p1, const void *p2)
{
      int v1, v2;
      v1 = *(int *)p1;
      v2 = *(int *)p2;
      if (v1 < v2)
            return -1;
      else if (v1 == v2)
            return 0;
      else
```

```
        return 1;
}
```

Ta có thể viết:

```
?       return v1-v2;
```

Nhưng nếu `v2` là số dương lớn, và `v1` là số âm lớn, hay ngược lại, thì kết quả là sẽ bị tràn số và trả về kết quả sai. Việc so sánh trực tiếp tuy dài hơn nhưng an toàn hơn.

Một lần nữa, lời gọi hàm đến `qsort` đòi hỏi một mảng, kích thước của mảng, kích thước các phần tử được sắp và hàm so sánh như sau:

```
int arr[N];
qsort(arr, N, sizeof(arr[0]), icmp);
```

ANSI C cũng định nghĩa một hàm tìm nhị phân, hàm `bsearch` cũng giống như hàm `qsort` đòi hỏi một con trỏ đến hàm so sánh (thông thường tương tự như hàm được dùng trong `qsort`); nó trả về một con trỏ đến phần tử tìm được hoặc `NULL` nếu không tìm thấy. Đây là hàm tra cứu HTML của chúng tôi, được viết lại từ hàm `bsearch`:

```
/* Hàm lookup: dùng hàm hàm bsearch để tìm tên
trong một tab, trả về một index */
int lookup(char *name, Nameval tab[], int ntab)
{
        Nameval key, *np;
        key.name = name;
        key.value = 0 ; /* không dùng đến */
        np = (Nameval *) bsearch (&key, tab, ntab,
                        sizeof(tab[0]), nvcmp);
        if (np==NULL)
                return -1;
        else
```

```
        return np->tab;
}
```

Cũng như hàm `qsort`, hàm so sánh nhận địa chỉ của các phần tử được so sánh vì thế khóa phải có cùng kiểu; trong ví dụ này, chúng ta cần cấu trúc `Nameval` giả để truyền vào hàm so sánh. Hàm so sánh là hàm `nvcmp` sẽ so sánh 2 phần tử `Nameval` bằng cách gọi `strcmp` trên các thành phần chuỗi của chúng mà không quan tâm đến giá trị của chúng:

```
/* Hàm nvcmp: so sánh 2 tên của Nameval */
int nvcmp(const void *va, const void *vb)
{
        const Nameval *a, *b;
        a = (Nameval *) va;
        b = (Nameval *) vb;
        return strcmp(a->name, b->name);
}
```

Điều này tương tự với hàm `scmp` nhưng cũng khác vì các chuỗi được lưu trữ như là các phần của một cấu trúc.

Sự thiếu đầy dủ trong việc cung cấp khóa đồng nghĩa với việc hàm `bsearch` ít hiệu quả bằng hàm `qsort`. Một hàm sắp xếp tổng quát tốt gồm 1 hay 2 trang dòng lệnh, trong khi hàm tìm kiếm nhị phân không lâu hơn nhiều so với thời gian để giao tiếp với hàm `bsearch`. Tuy nhiên, đó là một ý kiến hay để sử dụng hàm `bsearch` thay vì viết riêng cho mình. Sau nhiều năm, thật ngạc nhiên tìm kiếm nhị phân đã được chứng minh là rất khó cho các lập trình viên làm đúng.

Thư viện C++ chuẩn có một giải thuật tổng quát là sắp xếp, nó có độ phức tạp là O(nlogn). Chương trình sẽ dễ dàng hơn bởi vì nó không cần phải ép kiểu dữ liệu hay kích thước của phần tử, và nó không đòi hỏi một hàm so sánh bên ngoài.

```
int arr[N];
```

```
sort(arr, arr+N);
```

Thư viện C++ cũng có các hàm tìm kiếm nhị phân tổng quát, với các lợi điểm đáng chú ý.

Bài tập 2-1. Hàm `quicksort` thể hiện đệ quy một cách tự nhiên nhất. Viết lại hàm này bằng phương pháp lặp và so sánh 2 phiên bản. (Nó cho thấy thật khó để cài đặt `quicksort` bằng cách lặp và khi cài đặt bằng phương pháp đệ quy thì nó thật gọn).

2.4. Hàm Quicksort trong Java

Với Java thì khác hẳn. Các phiên bản đầu tiên không có hàm sắp xếp chuẩn, vì thế chúng ta phải tự viết chúng. Trong những phiên bản gần đây, nó có cung cấp một hàm sắp xếp, tuy nhiên hàm này hoạt động trên những lớp có cài đặt *phương thức* `Comparable`, vì thế chúng ta chỉ việc gọi thực hiện hàm thư viện sắp sếp khi có yêu cầu. Nhưng vì các kỹ thuật này hữu ích trong một số tình huống khác, trong phần này chúng ta sẽ nghiên cứu chi tiết việc cài đặt hàm `quicksort` trong Java.

Thật dễ dàng để điều chỉnh hàm `quicksort` cho các kiểu mà ta cần sắp xếp, nhưng nó dễ truyền đạt hơn khi viết một hàm sắp xếp tổng quát mà nó có thể được gọi cho bất kỳ loại đối tượng nào hơn là chỉ các kiểu của hàm `qsort`.

Một sự khác nhau lớn so với C hay C++ là trong Java chúng ta không thể truyền một hàm so sánh đến một hàm khác được; không có các con trỏ hàm. Vì vậy chúng ta tạo ra một *phương thức* mà nội dung duy nhất của nó là một hàm so sánh hai đối tượng. Đối với mỗi kiểu dữ liệu được sắp xếp, chúng ta tạo ra một lớp với một hàm thành viên mà có cài đặt *phương thức* cho loại dữ liệu này. Chúng ta truyền một thể hiện của lớp đó đến hàm sắp xếp, mà sau đó nó sẽ dùng hàm so sánh bên trong lớp để so sánh các phần tử.

Chúng ta bắt đầu bằng việc định nghĩa một *phương thức* mang tên `Cmp` mà nó khai báo một thành phần là một hàm so sánh `cmp` sẽ so sánh hai

đối tượng:

```
interface Cmp {
        int cmp(Object x, Object y);
}
```

Sau đó chúng ta có thể viết các hàm so sánh cài đặt *phương thức* này; chẳng hạn, lớp này định nghĩa một hàm so sánh *các số nguyên*:

```
// Hàm Icmp: so sánh các số nguyên
class Icmp implements Cmp {
        public int cmp(Object o1, Object o2)
        {
                int i1 = ((Integer) o1).intValue();
                int i2 = ((Integer) o2).intValue();
                if (i1 < i2)
                        return -1;
                else if (i1 == i2)
                        return 0;
                else
                        return 1;
        }
}
```

và hàm sau so sánh các chuỗi:

```
// Hàm Scmp: so sánh các chuỗi
class Scmp implements Cmp {
        public int cmp(Object o1, Object o2)
        {
                String s1 = (String) o1;
                String s2 = (String) o2;
```

```
            return s1.compareTo(s2);
        }
    }
```

Chúng ta có thể sắp xếp chỉ những kiểu dữ liệu kế thừa từ đối tượng với cơ chế này; nó không thể được áp dụng vào các kiểu cơ bản như kiểu `int` hay `double`. Đây là lý do tại sao chúng ta sắp xếp các kiểu `Interqers` mà không sắp xếp kiểu `ints`.

Với các thành phần trên, bây giờ chúng ta có thể chuyển một hàm `quicksort` trong C thành Java, hàm so sánh từ đối tượng `Cmp` được truyền vào như là một đối số. Sự thay đổi quan trọng nhất là việc sử dụng các chỉ mục `left` và `right`, vì Java không có các con trỏ chỉ đến mảng.

```
    //Quicksort.sort: quicksort v[left]..v[right]
    static void sort(Object[] v, int left, int right,
Cmp cmp)
    {
        int i, last;
        if (left >= right)// không cần làm gì cả
            return;
        swap(v, left, rand(left, right));//chuyển
phần tử làm mốc
        last = left ;                          //  đến
v[left]
        for (i=left+1; i <= right; i++)   //  phân
vùng
            if (cmp.cmp(v[i], v[left] ) < 0)
                swap( v, ++last, i );
        swap(v, left, last);    // đổi chỗ v[left]
và v[last]
        sort(v, left, last-1, cmp); // gọi đệ quy
```

```
cho phần đầu
            sort(v, last+1, right, cmp);// gọi đệ quy
cho phần sau
      }
```

`Quicksort.sort` dùng hàm `cmp` để so sánh một cặp đối tượng và gọi hàm `swap` để hoán vị chúng.

```
      // Hàm Quicksort.swap:  đổi chỗ v[i] và v[j]
      static void swap(Object[] v, int i, int j)
      {
            Object  temp;
            temp = v[i];
            v[i] = v[j];
            v[j] = temp;
      }
```

Số phát sinh ngẫu nhiên được thực hiện bởi một hàm phát sinh ngẫu nhiên số nguyên trong khoảng từ `left` đến `right`:

```
      static Random rgen = new Random();
      // Hàm Quicksort.rand: trả về một số nguyên ngẫu
nhiên trong khoảng [left, right]
      static int rand(int left , int right )
      {
            return                  left
Math.abs(rgen.nextInt())%(right-left+1);
      }
```

Chúng ta tính giá trị tuyệt đối sử dụng hàm `Math.abs`, bởi vì hàm phát sinh số ngẫu nhiên trong Java trả về cả số nguyên âm lẫn dương.

Hàm `sort`, `swap` *và* `rand`, và đối tượng phát sinh `rgen` là các hàm thành viên của lớp `Quicksort`.

Cuối cùng, để gọi `Quicksort.sort` sắp xếp mảng kiểu `string`, chúng ta có thể viết như sau:

```
String[] sarr = new String[n];
// Khởi tạo n chuỗi của mảng chuỗi sarr...
Quicksort.sort(sarr, 0, sarr.length-1, new
Scmp());
```

Lời gọi `sort` với đối tượng so sánh chuỗi được tạo ra cho trường hợp này.

Bài tập 2-2. Hàm `quicksort` được viết bằng Java ở trên chuyển đổi một lượng lớn các kiểu như là các thành phần được ép kiểu từ kiểu nguyên thủy (giống như kiểu `Integer`) thành kiểu `Object` và ngược lại. Hãy thực nghiệm với một phiên bản của `Quicksort.sort` để sắp xếp một kiểu cụ thể nào đó nhằm ước lượng chi phí thể hiện phải trả là gì khi chuyển đổi kiểu dữ liệu.

2.5. Ký hiệu O

Ta mô tả lượng công việc được thực hiện bởi một giải thuật nào đó theo n, số lượng các phần tử nhập vào. Việc tìm kiếm trên tập dữ liệu chưa sắp xếp có thể tốn thời gian tỉ lệ với n; nếu chúng ta sử dụng tìm kiếm nhị phân trên dữ liệu đã được sắp, thời gian sẽ là $\log n$. Thời gian sắp xếp có thể là n^2 hoặc $n\log n$.

Chúng ta cần một cách để làm cho các ước lượng này chính xác hơn, trong khi đó các chi tiết khác như tốc độ CPU và chất lượng của trình biên dịch (và người lập trình) còn trừu tượng. Chúng ta muốn so sánh thời gian chạy và yêu cầu về không gian của các giải thuật độc lập với ngôn ngữ lập trình, trình biên dịch, kiến trúc máy tính, tốc độ bộ xử lý, hệ thống và các yếu tố phức tạp khác.

Có một ký hiệu chuẩn cho ý tưởng này gọi là "ký hiệu O". Tham số cơ bản của nó là n, quy mô của bài toán và *độ phức tạp* hay thời gian chạy

được thể hiện như là một hàm theo n. Ký tự "O" là viết tắt của từ *order*, như trong tìm kiếm nhị phân là $O(logn)$; nó tốn $logn$ bước để tìm trong một mảng gồm n phần tử. Ký hiệu $O(f(n))$ nghĩa là một khi n đủ lớn thì thời gian chạy tỉ lệ gần như tương ứng với hàm $f(n)$, chẳng hạn $O(n^2)$ hay $O(n\log n)$. Các phỏng đoán gần đúng như thế này là rất có giá trị cho việc phân tích trên lý thuyết và rất hữu ích cho việc so sánh giữa các giải thuật, nhưng cụ thể có thể khác nhau trong thực tế. Chẳng hạn một giải thuật có độ phức tạp là $O(n^2)$ có thể chạy nhanh hơn giải thuật có độ phức tạp là $O(n\log n)$ với các giá trị n nhỏ, nhưng với n đủ lớn, giải thuật có độ phức tạp là $O(n\log n)$ sẽ chạy nhanh hơn.

Trong một số tình huống, chúng ta cần phải phân biệt giữa trường hợp xấu nhất và trường hợp thể hiện được sự mong đợi. Việc này khó thực hiện, vì nó phụ thuộc vào dữ liệu nhập. Thông thường ta tính chính xác trường hợp xấu nhất cho dù nó có thể bị nhầm trong một số. Thời gian chạy xấu nhất của hàm `quicksort` là $O(n^2)$ nhưng thời gian mong đợi là $O(n\log n)$. Bằng cách chọn phần tử mốc một cách cẩn thận trong mỗi lần gọi hàm quicksort, chúng ta có thể giảm từ $O(n^2)$ xuống bằng không; trong thực tế, một hàm `quicksort` được cài đặt tốt thường chạy trong khoảng thời gian $O(n\log n)$.

Sau đây là một số ký hiệu và tên gọi của các độ phức tạp thường gặp:

Ký hiệu	**Tên**	**Ví dụ**
O (1)	Hằng số	Truy cập một phần tử của mảng
O ($\log n$)	Logarit	Giải thuật tìm kiếm nhị phân
O (n)	Tuyến tính	So sánh chuỗi
O ($n\log n$)	$n\log n$	Giải thuật quicksort
O (n^2)	Lũy thừa 2	Các phương pháp sắp xếp thông thường

$O(n^3)$	Lũy thừa 3	Giải thuật nhân ma trận
$O(2^n)$	Hàm mũ 2^n	Phân vùng

Truy cập một phần tử trong mảng có thời gian cố định là $O(1)$. Một giải thuật loại bỏ một nửa dữ liệu nhập sau mỗi lần, giống như giải thuật tìm kiếm nhị phân, sẽ có độ phức tạp là $O(\log n)$. So sánh 2 chuỗi n ký tự với `strcmp` là $O(n)$. Giải thuật nhân ma trận truyền thống có độ phức tạp $O(n^3)$, vì mỗi phần tử của dữ liệu xuất là kết quả của tích n cặp và cộng chúng lại, và có n^2 phần tử trong mỗi ma trận.

Các giải thuật có thời gian hàm mũ thường là kết quả của việc lượng giá tất cả các khả năng: có 2^n tập con của một tập n phần tử, vì vậy một giải thuật đòi hỏi xem xét tất cả các tập con sẽ là hàm mũ hay $O(2^n)$. Các giải thuật có thời gian hàm mũ thông thường rất chậm trừ khi n rất nhỏ, vì khi thêm một phần tử vào bài toán sẽ tăng thời gian chạy lên gấp đôi. Không may có nhiều bài toán như thế, chẳng hạn "bài toán người du lịch" khá nổi tiếng, bài toán này cho đến nay người ta chỉ biết đến những giải thuật với thời gian lũy thừa. Khi đó như trong trường hợp này, các giải thuật xấp xỉ với kết quả chính xác thường được dùng thay thế.

Bài tập 2-3. Cho biết dữ liệu nhập nào là trường hợp xấu nhất của giải thuật quicksort? Cố gắng tìm ra một số trường hợp mà nó làm cho thư viện của bạn chạy chậm.

Bài tập 2-4. Thiết kế và cài đặt giải thuật sắp xếp một mảng n số nguyên càng chậm càng tốt. Bạn phải cài đặt một cách bình thường: giải thuật phải chạy và kết thúc đúng đắn, và không được dùng các thủ thuật vòng lặp làm tốn thời gian một cách vô lý. Tính độ phức tạp của giải thuật của bạn theo n?

2.6. Các mảng mở rộng

Các mảng được dùng trong các phần trước đây là tĩnh, với kích thước và nội dung cố định tại thời điểm biên dịch. Nếu bảng ký tự HTML được thay đổi tại thời điểm chạy, một bảng băm sẽ thích hợp hơn cho cấu

trúc dữ liệu. Mở rộng một mảng được sắp xếp bằng cách chèn n phần tử, mỗi lần một phần tử là phép toán có độ phức tạp $O(n^2)$ mà điều này nên tránh khi n lớn.

Thông thường mặc dù chúng ta cần theo dõi một biến nhưng một vài thứ khác và các mảng có thể vẫn là một lựa chọn. Để giảm chi phí cấp phát bộ nhớ, mảng phải được giảm kích thước và để sạch sẽ mảng phải được tập trung lại với các thông tin cần thiết để duy trì nó. Trong C++, hay Java, điều này sẽ được thực hiện từ các thư viện chuẩn: trong C, chúng ta có thể đạt được kết quả tương tự với một `struct`.

Đoạn chương trình sau định nghĩa một mảng các phần tử `Nameval` có thể mở rộng được: các phần tử mới được thêm vào cuối mảng, mảng sẽ được mở rộng khi cần. Bất kỳ một thành phần nào cũng có thể được truy cập thông qua chỉ số của nó trong khoảng thời gian cố định. Điều này tương tự với các lớp `vector` trong thư viện Java và C++.

```
typedef struct Nameval Nameval;
struct Nameval {
        char   *name;
        int    value;
};
struct NVtab {
        int    nval; /* Số phần tử hiện hành */
        int    max; /* Số phần tử tối đa */
        Nameval *nameval /* Mảng các cặp Nameval */
} nvtab;
enum { NVINIT = 1, NVGROW = 2};
/* Hàm addname: thêm một phần tử vào nvtab */
int addname(Nameval newname)
{
```

```
Nameval  *nvp;
if (nvtab.nameval ==  NULL) {/*Lần gọi đầu tiên */
      nvtab.nameval =
      (Nameval*)malloc(NVINIT * sizeof(Nameval));
      if (nvtab.nameval== NULL)
            return -1;
      nvtab.max = NVINIT;
      nvtab.nval   0;
      } else if (nvtab.nval >  nvtab.max) {
      nvp = (Nameval *) realloc(nvtab.nameval,
      (NVGROW*nvtab.max)*sizeof(Nameval));
      if (nvp == NULL)
            return  1;
      nvtab.max  *= NVGROW;
      nvtab.nameval = nvp;
      }
      nvtab.nameval[nvtab.nval] = newname;
      return nvtab.nval++;
}
```

Hàm `addname` trả về chỉ số của một phần tử vừa được thêm vào, hay -1 nếu có lỗi.

Lời gọi hàm `realloc` nhằm khởi tạo bộ nhớ và tăng kích thước mảng, duy trì các phần tử hiện tại và trả về một con trỏ đến nó hoặc `NULL` nếu không đủ bộ nhớ. Gấp đôi kích thước mảng trong mỗi lần khởi tạo lại bộ nhớ để duy trì chi phí sao chép mỗi phần tử là cố định; nếu mảng tăng lên 1 phần tử sau mỗi lần gọi, độ phức tạp sẽ là $O(n^2)$. Vì địa chỉ của mảng có thể thay đổi khi nó được khởi tạo lại, phần còn lại của chương trình phải tham chiếu đến các phần tử của mảng bằng các chỉ số, không phải các con

trỏ. Chú ý rằng đoạn mã nguồn trên không tương đương với:

```
?       nvtab.nameval                   (Nameval        *)
realloc(nvtab.nameval,
?               (NVGROW*nvtab.max)*sizeof (Nameval));
```

trong tình huống này, nếu việc khởi tạo lại bộ nhớ không thành công, mảng ban đầu sẽ bị mất.

Chúng ta sẽ bắt đầu với một số khởi tạo rất nhỏ (`NVINIT 1`) cho kích thước mảng. Điều này ép chương trình tăng kích thước mảng ngay lập tức và do đó đảm bảo rằng phần này của chương trình được thi hành. Kích thước ban đầu có thể được tăng lên một khi chương trình bắt đầu chạy cho dù chi phí bắt đầu nhỏ không đáng kể.

Giá trị trả về của hàm `realloc` không cần được ép kiểu thành kiểu dữ liệu cuối cùng bởi vì C hỗ trợ kiểu `void` một cách tự động. Nhưng C++ thì không: chúng ta cần phải ép kiểu. Chúng ta có thể tranh luận về vấn đề có an toàn hơn hay không để ép kiểu hoặc là không ép kiểu (việc ép kiểu có thể che giấu các lỗi thật sự). Chúng tôi chọn việc ép kiểu bởi vì nó tạo chương trình đúng cho cả C và C++; cái giá phải trả là ít được kiểm tra lỗi hơn từ trình biên dịch C, nhưng nó chỉ là một việc nhỏ bởi việc kiểm tra thêm được thực hiện khi ta sử dụng cả 2 trình biên dịch.

Xóa một tên có thể bị lỗi bởi vì chúng ta phải quyết định làm cách nào lấp khoảng trống trong mảng. Nếu thứ tự của các phần tử không là vấn đề, ta sẽ lấy phần tử cuối cùng lấp vào chỗ trống. Tuy nhiên, nếu thứ tự cần giữ đúng, chúng ta phải di chuyển các phần tử nằm sau phần tử trống lên trước một vị trí:

```
/* Hàm delname: xóa nameval tương ứng đầu tiên ra khỏi nvtab
*/
            int delname(char *name)
            {
                  int i;
```

```
for (i = 0; i <nvtab.nval; i++)
if (strcmp(nvtab.nameval[i].name, name) ==0) {
        memmove(nvtab.nameval+i,
        nvtab.nameval + i + 1,
        (nvtab.nval-(i+1))*sizeof(Nameval));
              nvtab.nval--;
              return 1;
        }
        return 0;
}
```

Lời gọi hàm memmove sẽ di chuyển các phần tử lên trước một vị trí; memmove là hàm thư viện chuẩn cho việc chép các khối dữ liệu có kích thước tùy ý trong bộ nhớ.

ANSI C chuẩn định nghĩa 2 hàm: hàm memcpy thực thi nhanh nhưng có thể viết đè trên bộ nhớ nếu nguồn và đích chồng chéo lên nhau, và hàm memmove tuy chậm hơn nhưng luôn luôn đúng. Các lập trình viên đã chọn sự chính xác hơn là tốc độ, do vậy nên chỉ có một hàm thôi. Giả sử chúng ta luôn sử dụng hàm memmove.

Ta có thể thay thế hàm memmove bằng vòng lặp sau:

```
int j;
for (j = i; j < nvtab.nval-1; j++)
        nvtab.nameval[j] = nvtab.nameval[j+1];
```

Chúng ta thích sử dụng hàm memmove bởi vì nó tránh việc dễ gặp lỗi chép các phần tử theo thứ tự sai. Nếu chúng ta chèn thay vì xóa, vòng lặp cần phải chạy ngược lại để tránh việc viết đè lên các phần tử. Bằng cách gọi hàm memmove chúng ta không cần phải suy nghĩ về nó mỗi lần gặp phải.

Một cách thay thế để dịch chuyển các phần tử của một mảng là đánh dấu xóa các phần tử như là không sử dụng nữa. Khi đó thêm một phần tử

mới: trước tiên tìm kiếm chỗ được đánh dấu xóa và chỉ tăng mảng nếu không tìm thấy chỗ đánh dấu xóa. Trong ví dụ này, một phần tử có thể được đánh dấu thành không sử dụng bằng cách gán trường `name` thành `NULL`.

Dùng mảng là cách đơn giản nhất để nhóm dữ liệu hiệu quả và tiện lợi; do đó hầu hết các ngôn ngữ lập trình đều cung cấp các mảng chỉ mục và thể hiện chuỗi là một mảng các ký tự. Các mảng dễ sử dụng và tốn khoảng $O(1)$ khi truy cập đến bất kỳ phần tử nào, làm việc tốt với tìm kiếm nhị phân và `quicksort` và cần ít không gian trước. Đối với các tập dữ liệu có kích thước cố định, nó được xác lập tại thời điểm biên dịch hay đối với các tập dữ liệu nhỏ được đảm bảo, dùng mảng là thích hợp nhất. Nhưng duy trì một tập thay đổi của các giá trị trong mảng có thể rất phức tạp, vì vậy nếu số lượng phần tử không biết trước và có thể lớn, chúng ta cần sử dụng một cấu trúc dữ liệu khác.

Bài tập 2-5. Trong đoạn chương trình trên, hàm `delname` không gọi hàm `realloc` để giải phóng bộ nhớ bằng cách xóa. Điều này có đáng giá không? Làm thế nào chúng ta quyết định có làm như thế hay không?

Bài tập 2-6. Cài đặt các thay đổi cần thiết cho hàm `addname` và `delname` để xóa các phần tử bằng cách đánh dấu các phần tử xóa như là không sử dụng nữa. Phần còn lại của chương trình sẽ bị tách rời như thế nào nếu có sự thay đổi này?

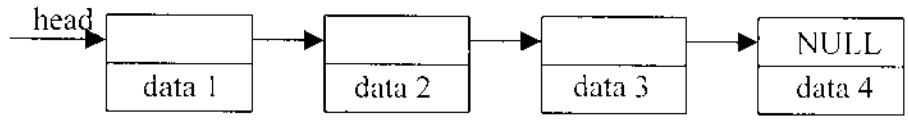
2.7. Danh sách liên kết

Tiếp theo cấu trúc dữ liệu mảng, danh sách liên kết là cấu trúc dữ liệu chung hầu hết điển hình trong các chương trình. Nhiều ngôn ngữ lập trình có cung cấp vài kiểu danh sách chẳng hạn như ngôn ngữ Lisp – nhưng trong C thì ta phải tự xây dựng. Trong C++ và Java các danh sách được cài đặt trong các thư viện, nhưng chúng ta cần phải biết dùng nó thế nào và khi nào thì dùng nó. Trong phần này chúng ta sẽ thảo luận các danh sách liên kết trong C nhưng vẫn có thể áp dụng rộng rãi hơn.

Một danh sách liên kết đơn là một tập các phần tử, mỗi phần tử như vậy có phần dữ liệu và một con trỏ trỏ đến phần tử kế tiếp. Đầu của danh

sách là một con trỏ head, nó trỏ đến phần tử đầu tiên và cuối của danh sách được đánh dấu bởi một con trỏ NULL.

Sau đây là phần minh họa của một danh sách có bốn phần tử:



Có một vài sự khác nhau cơ bản giữa mảng và danh sách. Đầu tiên, mảng có kích thước cố định, còn kích thước của một danh sách thì bao gồm kích thước nó cần để giữ các nội dung và các con trỏ. Thứ hai, danh sách có thể được sắp xếp bằng việc hoán đổi con trỏ của các phần tử, việc sắp xếp này nhanh hơn là di chuyển các khối cần thiết trong mảng. Cuối cùng, khi các phần tử được chèn vào hay xóa ra khỏi danh sách thì phần còn lại không cần phải di chuyển; nếu chúng ta lưu trữ các con trỏ đến các phần tử trong một vài cấu trúc dữ liệu khác, chúng sẽ không làm mất tính đúng đắn bởi sự thay đổi trong danh sách.

Nếu một tập hợp mà các phần tử thay đổi một cách thường xuyên, đặc biệt là nếu số lượng các phần tử không đoán trước được thì danh sách là một lựa chọn để lưu giữ chúng tốt nhất; còn ngược lại mảng sẽ tốt hơn cho dữ liệu tĩnh tương đối.

Sau đây là một số thao tác cơ bản của một danh sách liên kết: thêm một phần tử mới vào đầu hay cuối danh sách, tìm kiếm một phần tử nào đó, thêm một phần tử mới vào trước hay sau một phần tử nào đó, và xóa một phần tử nào đó. Tính đơn giản của danh sách là dễ dàng thêm vào nó những thao tác thích hợp.

Mỗi phần tử của danh sách liên kết trong C thường được bắt đầu bằng một kiểu dữ liệu, và một con trỏ trỏ tới phần tử kế tiếp:

```
typedef struct Nameval Nameval;
struct Nameval {
        char            *name;
```

```
    int         value;
    Nameval     *next;
};
```

Khởi động một danh sách không rỗng tại thời điểm biên dịch là một việc khó khăn, vì thế, không giống như mảng, danh sách được tạo thành một cách linh động. Đầu tiên, chúng ta cần tạo ra một phần tử. Thông thường thì dùng một hàm để thực hiện điều này, ở đây chúng ta dùng hàm `newitem`:

```
/* Hàm newitem: khởi tạo một phần tử mới từ tên và giá
trị đưa vào */
Nameval * newitem(char *name, int value)
{
    Nameval *newp;
    newp = (Nameval*)emalloc(sizeof(Nameval));
    newp->name = name;
    newp->value = value;
    newp->next = NULL;
    return newp;
}
```

Hàm `emalloc` là hàm mà ta sẽ dùng trong suốt cuốn sách này; nó gọi hàm `malloc`, và nếu sự cấp phát hỏng, nó sẽ thông báo lỗi và thoát chương trình. Đoạn mã minh họa sẽ được trình bày trong Chương 4, còn bây giờ cứ giả sử như hàm `emalloc` cấp phát bộ nhớ lúc nào cũng đủ và không bao giờ gặp lỗi.

Cách đơn giản và nhanh nhất để tạo một danh sách liên kết là thêm phần tử mới vào đầu danh sách:

```
/* Hàm addfront: thêm newp vào trước listp trong danh
sách */
Nameval *addfront(Nameval *listp, Nameval *newp)
```

```
{
        newp->next = listp;
        return newp;
}
```

Khi một danh sách bị thay đổi thì phần tử đầu tiên có thể bị thay đổi, điều này xảy ra khi hàm `addfront` được gọi. Các hàm cập nhật phải trả về một con trỏ đến phần tử mới, giá trị con trỏ trả về được lưu trữ trong một biến quản lý của danh sách. Hàm `addfront` và các hàm khác trong nhóm này tất cả đều trả về một con trỏ đến phần tử đầu tiên; sau đây là phần minh họa cho điều này:

```
nvlist = addfront( nvlist, newitem("smiley",0x263A ));
```

Cách viết này chạy tốt dù là danh sách tồn tại là rỗng, và nó dễ dàng kết hợp các hàm trong các biểu thức. Nhưng nó dường như tự nhiên hơn là sự truyền con trỏ đến nơi giữ con trỏ đầu của một danh sách.

Thêm một phần tử vào cuối một danh sách là thao tác có độ phức tạp *O(n)*, bởi vì chúng ta cần phải từ đầu danh sách đến cuối danh sách:

```
/* Hàm addend: thêm newp vào cuối của danh sách
listp */
Nameval *addend(Nameval *listp, Nameval *newp)
{
        Nameval * p;

        if (listpp == NULL)
                return newp;
        for (p = listp; p->next != NULL; p = p->next)
                ;
```

```
        p->next = newp;
        return listp;
    }
```

Nếu chúng ta muốn hàm `addend` chỉ có mức độ phức tạp là *O(1)* thì chúng ta có thể giữ một con trỏ riêng trỏ đến cuối danh sách. Mặt hạn chế của cách tiếp cận này, ngoài việc lo lắng bảo trì con trỏ cuối danh sách ra, thì một danh sách không còn được đại diện bởi một biến trỏ đơn nữa. Chúng ta sẽ cài minh họa với một ví dụ đơn giản sau.

Để tìm cho một phần tử với một tên nào đó, theo sau các con trỏ `next`:

```
/* Hàm lookup: tìm tuần tự một phần tu với một tên được
nhập vào trong listp */
    Nameval *lookup(Nameval *listp, char *name)
    {
        for ( ; listp != NULL; listp = listp->next
)
            if (strcmp(name,listp->name) == 0)
                return listp;
        return NULL; /* không tìm thấy*/
    }
```

Điều này có độ phức tạp *O(n)* và không có cách nào để cải tiến nhanh hơn trong trường hợp tổng quát. Ngay cả danh sách được sắp xếp, chúng ta cần duyệt danh sách để lấy phần tử cần tìm. Tìm kiếm nhị phân không được áp dụng trong danh sách liên kết.

Để in ra các phần tử của một danh sách, chúng ta có thể viết một hàm duyệt danh sách và in mỗi phần tử; để tính chiều dài của danh sách, ta có thể viết một hàm để duyệt danh sách và tăng biến đếm lên. Hàm `apply` trong khi duyệt danh sách thực hiện lời gọi một hàm khác cho mỗi phần tử của danh sách. Chúng ta có thể làm cho hàm `apply` linh hoạt hơn bằng việc

cung cấp nó với một đối số được truyền cho mỗi khi nó gọi hàm. Vì thế hàm `apply` có ba đối số: con trỏ đến đầu danh sách, một hàm được áp dụng cho mỗi phần tử của danh sách, và một đối số cho hàm đó:

```
/* Hàm apply: gọi thực thi hàm fn cho mỗi phần tử trong
danh sách listp */
      void apply(Nameval *listp,
            void (*fn) (Nameval*, void*), void *arg)
      {
            for ( ; listp != NULL; listp = listp->next)
                  (*fn)(listp,arg); /* lời gọi hàm */
      }
```

Đối số thứ hai của hàm `apply` là một con trỏ đến hàm mà hàm này có hai đối số và trả về có kiểu `void`. Cách viết chuẩn nhưng cú pháp rắc rối.

```
void (*fn)(Nameval*, void*)
```

khai báo `fn` là một con trỏ đến một hàm có giá trị trả về kiểu `void`, tức là, một biến giữ địa chỉ của một hàm trả về kiểu `void`. Hàm có hai đối số `Nameval*` là một phần tử của danh sách, `void*` là một con trỏ tổng quát trỏ đến một đối số cho hàm đó.

Để dùng hàm `apply`, ví dụ in ra các phần tử của một danh sách, chúng ta có thể viết một hàm nhận đối số là định dạng chuỗi ký tự:

```
/* Hàm printnv: in tên và giá trị dùng định dạng trong
arg */
      void printnv(Nameval *p, void *arg)
      {
            char *fmt;
            fmt = (char *) arg;
            printf(fmt, p->name, p->value);
      }
```

lời gọi hàm sẽ như sau:

```
apply(nvlist, printnv, " s: %x\n");
```

Để đếm các phần tử, chúng ta định nghĩa một hàm mà đối số của nó là một con trỏ đến một số nguyên sẽ được tăng lên:

```
/* Hàm inccounter: tăng biến đếm *arg */
void inccounter(Nameval *p, void *arg)
{
        int *ip;
        /* con trỏ p không được dùng */
        ip = (int *)arg;
        (*ip)++;
}
```

và gọi nó được như sau:

```
int n;
n = 0;
apply(nvlist, inccounter, &n);
printf("%d phần tử trong danh sách nvlist\n",n);
```

Không phải mọi thao tác đều đúng với kiểu áp dụng hàm `apply` này. Lấy ví dụ khi hủy danh sách, chúng ta cần phải dùng một cách cẩn thận hơn:

```
/* Hàm freeall: giải phóng tất cả các phần tử của listp
*/
        void freeall(Nameval* listp)
        {
                Nameval *next;
                for( ; listp != NULL; listp = next) {
                        next = listp->next;
```

```
        /* giả sử trường name được giải phóng ở đâu đó */
                free(listp);
            }
        }
```

Vùng nhớ không thể được dùng sau khi nó đã bị giải phóng, vì thế chúng ta cần lưu con trỏ `listp->next` vào một biến cục bộ `next`, trước khi giải phóng vùng nhớ được trỏ đến bởi `listp`. Đoạn mã sau đây sai, không thể giải phóng toàn bộ danh sách.

```
?       for( ; listp != NULL; listp = listp->next)
?           free(listp);
```

giá trị của `listp->next` bị có thể bị ghi đè bởi hàm `free` và đoạn mã trên sẽ sai.

Chú ý rằng hàm `freeall` không giải phóng vùng nhớ do `listp->name` trỏ đến. Giả sử rằng trường `name` của mỗi `Nameval` sẽ được giải phóng chỗ nào đó, hay chưa bao giờ cấp phát. Chắc chắn rằng các phần tử được cấp phát và giải phóng theo quy định các đối số giữa hàm `newitem` và hàm `freeall` một cách nhất quán; phải đảm bảo rằng những vùng nhớ nào cần được giải phóng là đã được giải phóng và những vùng nhớ không được giải phóng thì không nên giải phóng. Nếu các điều trên không được tuân thủ lỗi xảy ra là chuyện bình thường.

Trong các ngôn ngữ khác, gồm có Java, bộ dọn dẹp dữ liệu sẽ giải quyết giùm bạn vấn đề này. Chúng ta sẽ trở lại vấn đề này trong Chương 4 trong chủ đề quản lý tài nguyên.

Việc xóa một một phần tử ra khỏi danh sách là một công việc phức tạp hơn là thêm vào một phần tử mới :

```
/* Hàm delitem: xóa phần tử có trường là name đầu tiên
có trong danh sách */
    Nameval *delitem(Nameval *listp, char *name)
    {
```

```
        Nameval *p, *prev;
        prev = NULL;
        for (p = listp; p != NULL; p = p->next) {
                if (strcmp(name, p->name) == 0) {
                        if (prev == NULL)
                                listp = p->next;
                        else
                                prev->next = p->next;
                        free(p);
                        return listp;
                }
                prev = p;
        }
    eprintf("%s không có trong danh sách", name);
        return NULL;
    }
```

Cũng như hàm `freeall`, hàm `delitem` không giải phóng vùng nhớ do trường `name` trỏ đến.

Hàm `eprintf` hiển thị thông báo lỗi và thoát khỏi chương trình quá vụng về. Việc xử lý các lỗi có thể gặp khó khăn và cần thảo luận nhiều hơn nữa trong Chương 4, và sẽ cũng trình bày luôn phần cài đặt của hàm `eprintf` trong Chương 4.

Hầu hết các ứng dụng lớn của bạn cũng giống như các chương trình thông thường về cấu trúc danh sách và các thao tác cơ bản trên danh sách. Nhưng có nhiều thay đổi trong các thư viện của các ngôn ngữ, gồm có C++ Standard Template Library, hỗ trợ danh sách liên kết kép, mỗi phần tử có hai con trỏ, một trỏ đến phần tử sau đó và một trỏ đến phần tử trước nó. Danh sách liên kết đôi có yêu cầu cao hơn, nhưng thao tác tìm kiếm phần tử

cuối cùng và xóa phần tử hiện hành là các thao tác có độ phức tạp *O(1)*. Một vài con trỏ được cấp phát thêm cho các mục đích riêng: vì chúng cần được dùng nhiều hơn các phần tử khác trong danh sách tại cùng một thời điểm.

Danh sách không chỉ thích hợp cho những thao tác chèn và xóa ở giữa mà còn là một cấu trúc tốt cho việc quản lý dữ liệu không thứ tự có kích thước dao động bất thường, đặc biệt khi truy xuất hướng đến việc vào sau ra trước (Last In First Out - LIFO) như là ngăn xếp. Hiệu quả dùng bộ nhớ của danh sách tốt hơn dùng mảng khi phải dùng nhiều ngăn xếp mà kích thước của nó luôn phải tăng hay giảm một cách độc lập nhau. Chúng vẫn chạy tốt khi thông tin thực sự được sắp xếp. Nếu bạn cần kết hợp việc cập nhật thường xuyên với truy xuất ngẫu nhiên thì một cách tiếp cận hay hơn đó là dùng cấu trúc dữ liệu phi tuyến, như là cây hay bảng băm.

Bài tập 2-7. Cài đặt vài thao tác khác trên danh sách như: tạo bản sao, trộn, phân chia, chèn vào trước và sau một phần tử nào đó. Hai thao tác chèn trên khác nhau như thế nào?

Bài tập 2-8. Viết hai phiên bản đệ quy và lặp cho việc đảo ngược một danh sách. Không tạo lại một danh sách mới mà nên dùng lại các phần tử đang tồn tại trong danh sách.

Bài tập 2-9. Hãy viết một danh sách liên kết trên C, mỗi phần tử của danh danh sách giữ một biến `void*`, và con trỏ next. Biến `void*` trỏ đến phần chứa dữ liệu. Làm tương tự như vậy cho C++ thông qua việc định nghĩa một `template` và cho Java thông qua việc định nghĩa một lớp, mà lớp này giữ danh sách kiểu `Object`. Các điểm mạnh và yếu của các ngôn ngữ khác nhau để thực hiện công việc này là gì?

Bài tập 2-10. Hãy tạo ra một bộ dữ liệu thử nghiệm để kiểm tra các hàm trên danh sách liên kết mà bạn cho là đúng. Chương 6 sẽ thảo luận các chiến lược để thực hiện việc kiểm tra.

2.8. Cây

Cây là cấu trúc dữ liệu có thứ bậc, lưu trữ một tập các phần tử, mỗi phần tử có một giá trị và nó có thể trỏ đến một hay nhiều phần tử khác, và nó được trỏ bởi một phần tử khác. Ngoại trừ gốc của cây thì không có phần tử nào trỏ đến nó.

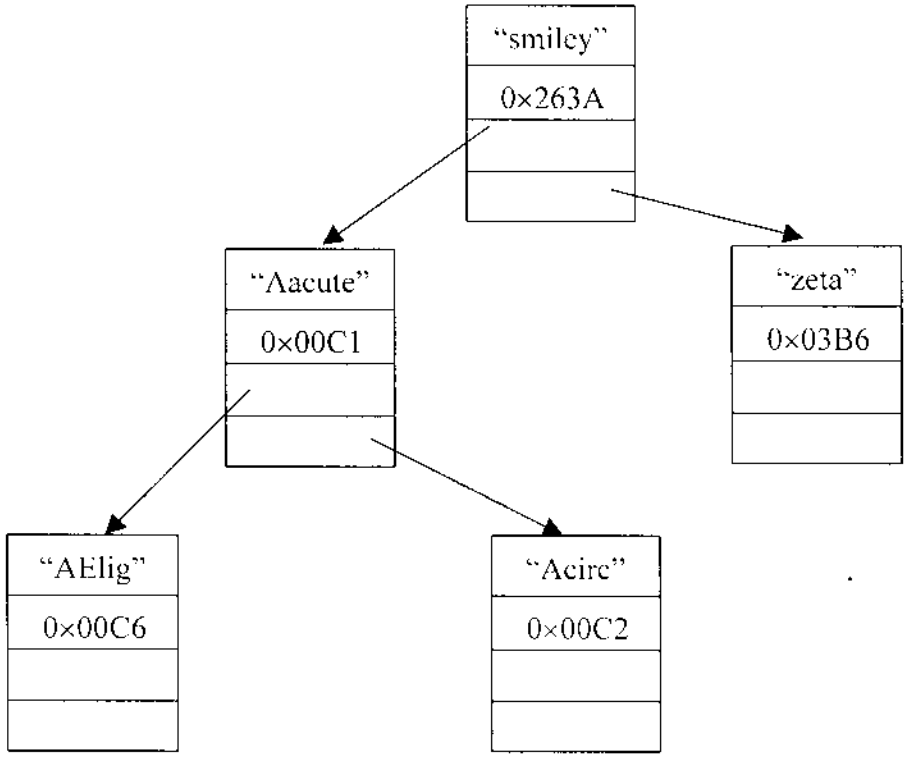
Cây có nhiều kiểu cấu trúc phức tạp chẳng hạn như cây phân tích cú pháp của một câu hay một chương trình, hay cây gia phả nó diễn tả mối quan hệ giữa người này với người khác trong gia đình. Chúng ta sẽ minh họa các nguyên lý bằng cây tìm kiếm nhị phân, (Binary Search Trees - BST) nó có hai liên kết tại mỗi nút. BST là bản cài đặt dễ nhất và minh họa được các tính chất cơ bản của cây. Một nút trong BST có một giá trị và hai con trỏ, con trỏ trái và con trỏ phải, trỏ đến cây con của nó. Các con trỏ đến cây con nó có thể là `NULL` nếu nút đó có ít hơn hai cây con. Trong BST, giá trị tại các nút của cây sẽ định nghĩa nên cây đó: tất cả các giá trị tại mỗi nút của cây con trái đều nhỏ hơn giá trị nút cha của nó, và tất cả các giá trị tại mỗi nút của cây con phải đều lớn hơn giá trị nút cha của nó. Nhờ tính chất này mà ta có thể tìm kiếm một giá trị nào đó trên cây tìm kiếm nhị phân hay xác định rằng giá trị đó nó không tồn tại trong cây một cách nhanh chóng.

Một phiên bản cấu trúc cây cho `Nameval` không phức tạp như sau:

```
typedef struct Nameval Nameval;
struct Nameval {
        char        *name;
        int         value;
        Nameval     *left;      /* nho hon */
        Nameval     *right;     /* lon hon */
};
```

Các chú thích nhỏ hơn và lớn hơn ở trên dựa vào tính liên kết: cây con trái lưu giữ giá trị nhỏ hơn và cây con phải lưu giữ giá trị lớn hơn tại mỗi nút.

Bằng một ví dụ cụ thể, hình này cho thấy một tập con của một bảng tên các đặc tính được lưu giữ bằng cây tìm kiếm nhị phân của các `Nameval`, được sắp thứ tự theo các giá trị ký tự ASCII của trường tên:



Với nhiều con trỏ đến các phần tử khác nhau trong mỗi nút của cây nên các thao tác trong cây chỉ tốn thời gian là *O(logn)*, chứ không như nhiều thao tác mất thời gian là *O(n)* trong danh sách hay trong mảng. Nhiều con trỏ tại mỗi nút làm giảm thiểu thời gian độ phức tạp của các thao tác bởi giảm số lượng các nút cần phải duyệt qua để tìm một phần tử nào đó. Cây tìm kiếm nhị phân (chúng ta sẽ gọi ngắn gọn là cây trong phần này nhưng vẫn được hiểu là cây tìm kiếm nhị phân) được xây dựng theo kiểu cây đệ quy, rẽ nhánh trái hay rẽ sang nhánh phải tương ứng, cho đến khi chúng ta tìm được chỗ đặt đúng liên kết với nút mới. Nút mới phải là một đối tượng có giá trị ban đầu hợp lệ của kiểu `Nameval` gồm có trường `name`, trường `value`, và hai con trỏ được gán bằng `NULL`. Nút mới được thêm vào là một nút lá, nghĩa là nó chưa có cây con.

```
/* Hàm insert: chèn newp vào treep, trả về con trỏ
treep */
Nameval *insert(Nameval *treep, Nameval *newp)
{
        int cmp;
        if (treep == NULL)
                return newp;
        cmp = strcmp(newp->name, treep->name);
        if (cmp == 0)
           weprintf("insert: duplicate entry %s ignored",
                        newp->name);
        else if (cmp < 0)
                  treep->left = insert(treep->left, newp);
                else
                  treep->right = insert(treep->right, newp);
                return treep;
        }
```

Từ đầu tới giờ ta chưa bàn gì về việc hai phần tử trùng nhau trên cây. Phiên bản này của hàm `insert` giải thích về chèn các phần tử giống nhau vào cây (khi biến `cmp == 0`). Hàm `insert` của danh sách không cần giải thích về việc trùng nhau vì nó phải thực hiện việc tìm kiếm trên danh sách nên làm cho phép chèn có độ phức tạp về thời gian là *O(n)* hơn là *O(1)*. Với cây, tuy nhiên, việc kiểm tra về cơ bản là không cần và các tính chất của cấu trúc dữ liệu không được định nghĩa rõ ràng nếu nó có sự trùng lặp. Trong các ứng dụng khác, mặc dù nó có thể nhất thiết phải chấp nhận trùng, hay nó có thể là lý do để hoàn toàn lờ đi chúng.

Hàm `weprintf` là một biến thể của `eprintf`: nó in ra câu thông báo lỗi và tiền tố `'w'` ở đầu là cảnh báo, nhưng nó sẽ không giống `eprintf` vì nó không kết thúc chương trình.

Một cây mà các đường đi từ nút gốc đến nút lá của xấp xỉ bằng nhau thì được gọi là cây cân bằng. Cây cân bằng thuận lợi trong việc tìm kiếm một phần tử vì thao tác tìm kiếm có độ phức tạp là *O(logn)*, vì giống như trong tìm kiếm nhị phân số lượng các phần tử giảm đi một nửa sau mỗi bước.

Nếu các mục được chèn vào cây một cách tự nhiên thì có lẽ cây không cân bằng; thực tế là cây có thể bị suy biến luôn. Nếu các phần tử được chèn vào dường như đã có thứ tự thì đoạn mã chèn của cây sẽ luôn chèn vào một nhánh nào đó, lúc đó cây thành một danh sách liên kết, lúc này gặp phải tất cả các vấn đề về tốc độ thực thi của danh sách. Nếu các phần tử được chèn vào theo thứ tự ngẫu nhiên thì không giống những điều đang xảy ra ở trên và cây cũng ít nhiều cân bằng.

Để cài đặt một cây mà đảm bảo được sự cân bằng thì quá phức tạp: đây là lý do vì sao có nhiều loại cây. Ở đây chúng ta không giải quyết vấn đề cân bằng mà giả sử rằng các dữ liệu nhập vào là hoàn toàn ngẫu nhiên và giữ cây cân bằng vừa đủ.

Đoạn mã của hàm `lookup` cũng tương tự như của hàm `insert`:

```
/* Hàm lookup: tìm kiếm một nút có trường là name trong
cây treep */
      Nameval *lookup(Nameval *treep, char *name)
      {
            int cmp;
            if (treep == NULL)
                  return NULL;
            cmp = strcmp(name, treep->name);
            if (cmp == 0)
                  return treep;
            else if (cmp < 0)
                  return lookup(treep->left, name);
```

```
        else
                return lookup(treep->right, name);
    }
```

Có hai vấn đề phải chú ý về hàm `insert` và hàm `lookup`. Đầu tiên, chúng trông giống rõ rệt giải thuật tìm kiếm nhị phân ở đầu chương. Điều này không tình cờ, bởi vì chúng dùng quan niệm của tìm kiếm nhị phân: chia để trị là nguyên nhân của việc làm tăng tốc độ thực thi của chương trình thành *logarit*.

Thứ hai, các hàm này đều là hàm đệ quy. Nếu viết lại chúng bằng các thuật toán lặp, chúng sẽ tương tự giải thuật tìm kiếm nhị phân. Thực tế phiên bản lặp của hàm `lookup` có thể được xây dựng bằng cách hiệu chỉnh lại phiên bản đệ quy. Nếu phần tử không tìm thấy thì hành động sau cùng của hàm `lookup` là trả về kết quả của một lời gọi đến chính nó, trường hợp này được gọi là đệ quy sau. Điều này có thể chuyển sang lặp bởi sự điều chỉnh lại các thông số và bắt đầu lại thủ tục. Hầu hết các phương pháp trực tiếp là dùng một lệnh nhảy, nhưng dùng một vòng lặp `while` thì rõ ràng hơn:

```
/* Hàm nrlookup: tìm kiếm một nút có trường là name
trong cây treep không dùng đệ quy */
    Nameval *nrlookup(Nameval *treep, char *name)
    {
        int cmp;
        while (treep != NULL) {
            cmp = strcmp(name,treep->name);
            if (cmp == 0)
                return treep;
            else if (cmp < 0 )
                treep = treep->left;
            else
```

```
            treep = treep->right;
        }
        return NULL;
    }
```

Mỗi lần chúng ta có thể duyệt khắp cây, các thao tác chung khác xảy đến một cách tự nhiên. Chúng ta có thể dùng một vài kỹ thuật đã thực hiện trong việc quản lý danh sách, bằng cách viết một thủ tục duyệt cây tổng quát và thực hiện lời gọi hàm tại mỗi nút. Một vấn đề nảy sinh là: ta sẽ thực hiện thao tác trên mỗi phần tử khi nào và ta sẽ xử lý phần còn lại của cây khi nào? Câu trả lời sẽ phụ thuộc vào những gì mà cây đang diễn tả: nếu nó đang lưu giữ dữ liệu có thứ tự, như là một cây tìm kiếm nhị phân, chúng ta sẽ duyệt nửa bên trái trước khi qua bên phải của cây. Đôi khi cấu trúc cây phản ánh vài bản chất thứ tự của dữ liệu, như cây gia phả, và thứ tự mà chúng ta duyệt qua các nút lá sẽ phụ thuộc vào mối quan hệ mà cây diễn tả.

Duyệt theo thứ tự giữa (*in-order*) sẽ thực hiện thao tác mong muốn tại nút hiện hành sau khi đã duyệt qua cây con trái và trước khi duyệt cây con phải:

```
/* Hàm applyinorder: gọi thực hiện hàm fn theo thứ tự
giữa */
    void applyinorder (Nameval* treep,
        void (*fn) (Nameval*, void*), void *arg)
    {
        if(treep == NULL)
            return;
        applyinorder(treep->left, fn, arg);
        (*fn)(treep, arg);
        applyinorder(treep->right, fn, arg);
    }
```

Trình tự này được dùng khi các nút đã được xử lý theo một thứ tự đã sắp xếp, ví dụ in tất cả chúng theo thứ tự, ta sẽ gọi thực hiện như sau:

```
applyinorder(treep, printnv, "%s : %x\n");
```

Dùng cây nhị phân tìm kiếm có thể đề nghị một cách sắp xếp như sau: chèn các mục vào cây, cấp phát một mảng có đúng kích thước tất cả các mục. Sau đó dùng cách duyệt cây theo thứ tự giữa để lưu trữ chúng vào trong mảng theo tuần tự các mục nhận được. Mảng lúc này đã được điền các mục đã sắp xếp.

Duyệt theo thứ tự sau (*post-order*) sẽ thực hiện thao tác mong muốn tại nút hiện hành sau khi đã duyệt qua cây con trái và cây con phải của nó:

```
/* Hàm applyinorder: gọi thực hiện hàm fn theo thứ tự
sau */
      void applypostorder(Nameval *treep,
      void (*fn)(Nameval*,void*), void *arg)
      {
            if( treep == NULL )
                  return;
            applypostorder(treep->left, fn, arg);
            applypostorder(treep->right, fn, arg);
            (*fn)(treep, arg);
      }
```

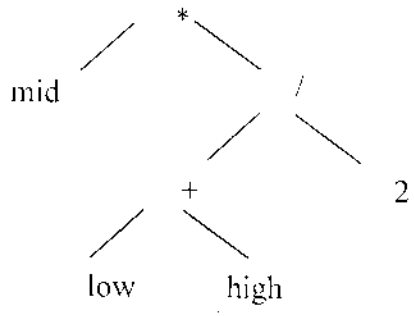
Duyệt theo thứ tự sau được dùng khi các thao tác của nút hiện hành phụ thuộc vào các cây con bên dưới nó. Ví dụ việc tính chiều cao của cây (lấy max chiều cao của mỗi cây con và cộng thêm mới), bố trí cây trong bản vẽ đồ thị (cấp phát không gian dựa trên từng trang cho mỗi cây con và kết hợp chúng theo không gian của nút này), và tính tổng không gian lưu trữ.

Sự lựa chọn thứ ba là duyệt cây theo thứ tự trước (*pre-order*). nó ít khi dùng vì thế chúng ta sẽ bỏ qua nó.

Thực tế, cây tìm kiếm nhị phân ít được dùng, dù rằng B-Tree (cây có rất nhiều nhánh), được dùng để bảo quản thông tin trong bộ nhớ thứ cấp. Thông thường ta dùng một cây để diễn tả cấu trúc của một lệnh hay một biểu thức. Ví dụ, câu lệnh:

```
mid = (low + high) / 2;
```

có thể được diễn tả bằng cây phân tích được mô tả như hình bên dưới. Để ước lượng cây, thực hiện một thao tác duyệt cây theo thứ tự sau và thực hiện thao tác xấp xỉ tại mỗi nút.



Chúng ta sẽ xem xét cây phân tích kỹ lưỡng hơn nữa trong Chương 9.

Bài tập 2-11. So sánh hiệu năng của hàm `lookup` và hàm `nrlookup`. So sánh phí tổn sự đệ quy với sự lặp như thế nào?

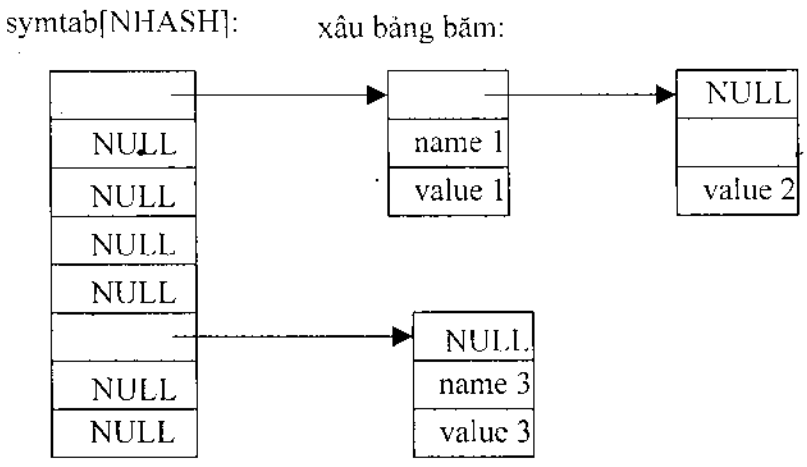
Bài tập 2-12. Dùng cách duyệt theo thứ tự giữa để tạo một thủ tục sắp xếp. Nó có độ phức tạp thời gian là gì? Dưới những điều kiện gì nó có thể có tác động xấu? So sánh hiệu năng cho `quicksort` của chúng ta với phiên bản có trong thư viện như thế nào?

Bài tập 2-13. Hãy tạo một tập dữ liệu thử nghiệm cho việc xác minh rằng các thủ tục cho cây là đúng.

2.9. Bảng băm

Bảng băm là một trong những phát minh lớn của nền khoa học máy tính. Nó kết hợp mảng, danh sách liên kết và các nguyên lý toán học để tạo ra một cấu trúc hiệu quả cho việc lưu trữ và truy xuất dữ liệu động. Một ứng dụng tiêu biểu là bảng ký hiệu, nó được kết hợp vài trị số (phần dữ liệu) với mỗi phần tử trong tập động các chuỗi ký tự (phần khóa). Một trình biên dịch mà bạn ưa thích hầu như chắc chắn dùng bảng băm để quản lý thông tin của mỗi biến trong chương trình. Một trình duyệt web có thể tận dụng bảng băm để giữ các vết của các trang thường được dùng, và kết nối với Internet hầu như chắc chắn dùng bảng băm để lưu trữ các tên miền được dùng thường xuyên và các địa chỉ IP của chúng.

Kỹ thuật truyền các khóa qua hàm băm sinh ra trị số băm mà sẽ được phân bổ đều qua miền số nguyên kích thước vừa phải. Giá trị băm được dùng là chỉ số bảng nơi thông tin được lưu trữ. Java có cung cấp một phương thức giao tiếp chuẩn cho bảng băm. Trong C và C++, dạng thông thường là kết hợp mỗi giá trị băm (hay mỗi "ô") thành một danh sách các mục mà nó cùng chia sẻ một giá trị băm, hình sau minh họa bảng băm:



Thực tế thì hàm băm được định nghĩa trước và có kích thước thích hợp với mảng được cấp phát, thông thường tại lúc biên dịch (vì tại lúc biên dịch đã biết kích thước của mảng). Mỗi phần tử của mảng là một danh sách mà một loạt các mục cùng nhau chia sẻ một giá trị băm. Một cách khác, bảng băm của *n* mục là mảng của các danh sách mà danh sách có chiều dài trung bình là *n* chia cho kích thước của mảng. Truy xuất một mục là một thao tác có độ phức tạp *O(1)* với điều kiện là chúng ta chọn lựa kỹ càng một hàm băm tốt và danh sách không tăng chiều dài.

Bởi vì bảng băm là một mảng của các danh sách, kiểu phần tử là giống nhau như cho một danh sách:

```
typedef struct Nameval Nameval;
struct Nameval {
        char *name;
        int value;
        Nameval *next;
};

Nameval *symtab[NHASH]; /* bảng ký hiệu */
```

Các kỹ thuật về danh sách chúng ta đã thảo luận trong phần 2.7 có thể được dùng để duy trì một loạt các băm riêng lẻ. Một khi bạn có một hàm băm tốt thì bảng băm hoạt động rất tốt: chỉ cần lựa chọn ô có giá trị băm và đi dọc theo danh sách để tìm những mục thỏa mãn. Đoạn mã sau cho hàm `lookup/insert` bảng băm. Nếu phần tử được tìm thấy thì nó sẽ được trả về. Ngược lại, nếu không tìm thấy và biến cờ `create` bằng `true` thì hàm `lookup` sẽ thêm một phần tử vào bảng. Cần nhắc lại là điều này không tạo ra một bản sao của trường `name`, giả sử rằng việc gọi hàm sẽ tạo ra một bản sao an toàn để thay thế.

```
/* Hàm lookup: tìm tên trong symtab, với tùy chọn
create để chèn phần tử mới vào bảng băm*/
```

```
Nameval* lookup(char* name, int create , int value)
        {
        int h;
        Nameval *sym;
        h = hash(name);
        for (sym = symtab[h]; sym != NULL; sym = sym-
>next)
                if (strcmp(name, sym->name) == 0)
                        return sym;
                if (create) {
                        sym (Nameval*)
emalloc(sizeof(Nameval));
                        sym->name = name; /*giả su đã được
cấp phát rồi */
                        sym->value = value;
                        sym->next = symtab[h];
                        symtab[h] = sym;
                }
                return sym;
        }
```

Cách kết hợp này của hàm `lookup` và tùy chọn `insert` là phổ biến. Nếu không dùng cách này thì sẽ có một sự lặp lại; lần đầu phải viết:

```
if (lookup("name")  = NULL)
        additem(newitem("name", value));
```

và hàm băm được tính hai lần.

Mảng cần một kích thước lớn như thế nào? Một ý kiến thông thường là nó đủ lớn mà mỗi chuỗi băm sẽ có một vài phần tử, vì thế tác vụ tìm kiếm một phần tử trong hàm *lookup* sẽ là *O(1)*. Ví dụ, một trình biên dịch có thể

có kích thước mảng là một vài ngàn, vì một tập tin nguồn lớn có thể có đến vài ngàn dòng, và chúng ta đừng mong chờ hơn một định danh mới trên mỗi dòng mã.

Bây giờ chúng ta cần quyết định những thứ trong hàm băm, trị số băm nên tính toán. Một hàm băm cần được xác định và nên có tốc độ nhanh và phân bố dữ liệu đồng đều cho mảng từ đầu đến cuối. Một trong những thuật toán băm phổ biến nhất cho chuỗi ký tự làm trị số băm là cộng dồn trị số băm được nhân với một hệ số nhân rồi cộng với trị số mỗi byte của chuỗi đó, một kết quả nên được kết hợp toàn bộ các byte nhập vào. Theo kinh nghiệm trị số 31 và 37 đã được chứng minh là một sự lựa chọn tốt cho hệ số nhân trong hàm băm cho chuỗi các ký tự mã ASCII.

```
enum {MULTIPLIER = 31};
/* Hàm hash: tính trị số băm của chuỗi */
unsigned int hash(char* str)
{
        unsigned int h;
        unsigned char *p;
        h = 0;
        for (p = (unsigned char *)str; *p != '\0';
p++)
                h = MULTIPLIER * h + *p;
        return h % NHASH;
}
```

Trong tính toán dùng các ký tự không dấu bởi vì kiểu `char` là kiểu có dấu và ta muốn trị số băm là một số dương.

Hàm băm trả về là phép chia lấy dư với kích thước của mảng. Nếu hàm băm phân bố các trị số khóa đều, kích thước mảng chính xác sẽ không còn có ý nghĩa nữa. Khó mà chắc chắn rằng hàm băm là một hàm đáng tin cậy, ngay cả các hàm băm tốt nhất đi nữa cũng có vấn đề với một vài tập dữ

liệu nào đó. Vì vậy chọn kích thước mảng là một số nguyên tố là sự lựa chọn đúng đắn.

Kinh nghiệm cho thấy rằng các chuỗi có chiều dài thay đổi khó mà xây dựng một hàm băm tốt hơn các hàm băm trên, nhưng lại dễ dàng làm cho nó kém hơn. Phiên bản Java đầu tiên có hàm băm cho chuỗi mà sự băm có hiệu quả hơn nếu chuỗi có chiều dài lớn. Hàm băm giảm được thời gian bằng việc chỉ xem xét kỹ 8 hay 9 ký tự trong một chuỗi có chiều lớn hơn 16 ký tự. Mặc dù hàm băm nhanh hơn nhưng theo các tính chất thống kê thì nó xấu nên nó đã bị bỏ đi cho dù tốc độ thực thi có thế nào đi nữa. Bằng việc bỏ qua các đoạn trong chuỗi, chỉ tập trung ở những phần khác biệt. Các tên file bắt đầu với các định danh tiền tố dài như là tên thư mục và có thể chỉ khác nhau ở một vài ký tự cuối (.java khác .class). Các URL thường bắt đầu bằng `http://www.` và kết thúc bằng `.html`, vì vậy các chuỗi chỉ khác ở phần giữa. Một hàm băm thường chỉ xét kỹ ở những phần thường không thay đổi của tên, kết quả là một chuỗi các mục có cùng một trị số băm, vì vậy khi tìm kiếm sẽ chậm đi rất nhiều. Vấn đề này được giải quyết bằng cách thay thế một hàm băm tương đương mà chúng ta minh họa ở trên (với hệ số nhân là 37), xét mọi ký tự của chuỗi.

Một hàm băm là tốt cho một tập các đầu vào này (như tên các biến ngắn) nhưng có thể là xấu cho các tập đầu vào khác (như các URL), vì vậy các hàm băm nên được kiểm tra trên các tập dữ liệu nhập điển hình khác nhau. Nó băm các chuỗi ngắn tốt không? Các chuỗi dài? Các chuỗi có chiều dài?

Các chuỗi không hẳn là những thứ ta có có thể băm. Chúng ta có thể băm ba trục tọa độ của một phần tử trong mô phỏng vật lý, làm giảm lưu trữ bảng tuyến tính *(O(số phần tử))* thay vì một mảng 3 chiều *(O(xSize × ySize × zSize))*.

Các bảng băm rất tốt cho các bảng định danh, vì việc truy xuất đến một phần tử bất kỳ chỉ tốn *O(1)*. Chúng có một vài giới hạn. Nếu các hàm băm xấu hay kích thước bảng quá nhỏ thì các danh sách có thể tăng chiều dài ra (để xử lý dụng độ). Vì các danh sách không được sắp xếp nên độ phức

tạp là *O(n)*. Các phần tử không truy xuất trực tiếp trong sắp xếp thứ tự nhưng dễ dàng đếm chúng, cấp phát mảng, hay điền chúng với các con trỏ đến các phần tử. Mặc dù vậy, khi các tính chất được dùng như: thời gian tìm kiếm là không đổi, các tính chất chèn và xóa của bảng băm là không giống các kỹ thuật khác.

Bài tập 2-14. Hàm băm thật rất tốt cho các mục đích phổ dụng như băm các chuỗi. Tuy nhiên, các dữ liệu đặc biệt có thể là nguyên nhân gây ra trường hợp xấu. Hãy xây dựng một tập dữ liệu làm cho hàm băm trở nên yếu kém. Có dễ dàng tìm một tập tập dữ liệu xấu cho các giá trị khác nhau của NHASH không?

Bài tập 2-15. Hãy viết một hàm truy xuất các phần tử liên tiếp của bảng băm không được sắp xếp thứ tự.

Bài tập 2-16. Hãy thay đổi hàm lookup nếu chiều dài danh sách trung bình lớn hơn hệ số x, mảng tự động tăng bởi hệ số y và xây dựng lại bảng băm.

Bài tập 2-17. Hãy thiết kế một hàm băm để lưu trữ tọa độ các điểm hai chiều. Hàm bạn thích ứng thay đổi trong kiểu tọa độ dễ dàng như thế nào, ví dụ từ số nguyên sang số dấu chấm động hay từ hệ tọa độ Đềcác sang hệ tọa độ cực.

2.10. Tổng kết

Qua tất cả những gì trình bày ở trên, để có được một thuật toán phù hợp ta nên trải qua các bước sau:

- Đầu tiên, tiếp cận các thuật toán và cấu trúc dữ liệu có liên quan. Đoán xem chương trình sẽ xử lý dữ liệu ra sao. Nếu một số lượng dữ liệu vừa phải thì chọn một kỹ thuật đơn giản; nếu dữ liệu có thể lớn thì phải thiết kế sao cho nó không tăng theo tỷ lệ với các tập dữ liệu vào lớn.
- Sau đó, dùng thư viện hay các đặc trưng của ngôn ngữ nếu có thể. Trường hợp không có, hãy viết hoặc mượn một ví dụ ngắn, để dễ dàng hiểu sự thực hiện. Thử nó. Nếu sự đo lường chứng minh rằng nó quá

chậm, chỉ khi đó bạn mới nên nâng cấp các kỹ thuật cao cấp hơn.

Mặc dù có nhiều cấu trúc dữ liệu, nhưng để tăng tốc độ thực thi trong các trường hợp đặc biệt nào đó thì hầu hết các chương trình luôn dùng mảng, danh sách, cây, bảng băm. Mỗi loại cấu trúc trên đều có hỗ trợ một tập các thao tác cơ bản, thường dùng như: tạo một phần tử mới, tìm một phần tử, thêm một phần tử vào một vị trí nào đó, đôi khi có thể có thao tác xóa một phần tử và dùng một vài thao tác cho các tất cả các phần tử.

Mỗi thao tác có một thời gian tính toán, thông thường nó được xác định cụ thể trên từng kiểu dữ liệu trong các ứng dụng khác nhau.. Mảng hỗ trợ thời gian truy cập là không đổi cho bất kỳ phần tử nào, nhưng nó không thể tăng hay giảm một cách dễ dàng. Danh sách liên kết chỉ tốt cho việc thêm và xóa, nhưng mất *O(n)* thời gian cho việc truy xuất ngẫu nhiên các phần tử. Cây và bảng băm thì tốt hơn, nó cho phép truy xuất nhanh một phần tử nào đó cùng với sự tăng kích thước dễ dàng, miễn là vài tiêu chuẩn cân bằng được duy trì.

Có nhiều cấu trúc dữ liệu phức tạp khác cho các vấn đề chuyên dụng, nhưng tập các cấu trúc cơ bản vừa được trình bày trong chương này cũng đủ xây dựng được hầu hết các phần mềm.

Chương 3

THIẾT KẾ VÀ CÀI ĐẶT

Việc thiết kế cấu trúc dữ liệu là phần quyết định chính trong việc tạo ra một chương trình. Nếu các cấu trúc dữ liệu được đưa ra tốt thì các giải thuật đều hướng đến chúng và việc cài đặt trở nên tương đối dễ dàng.

Với quan điểm là làm cho đơn giản nhưng không bị sai. Trong chương trước chúng ta xem xét các cấu trúc dữ liệu cơ bản mà chúng là nền tảng của hầu hết các chương trình.Trong chương này ta sẽ tiếp tục kết hợp các cấu trúc đó trong thiết kế và cài đặt chương trình. Ta sẽ cho thấy các vấn đề ảnh hưởng đến cấu trúc dữ liệu như thế nào và mã nguồn theo cấu trúc dữ liệu đó được viết như thế nào.

Một khía cạnh của quan điểm này là việc chọn ngôn ngữ lập trình tương đối không quan trọng đối với thiết kế tổng thể. Ta sẽ thiết kế chương trình trừu tượng và sau đó viết nó bằng ngôn ngữ C, C++, Java, Awk và Perl. So sánh các bản cài đặt giữa các ngôn ngữ lập trình để xem chúng có thể giúp đỡ hay cản trở như thế nào và mức độ không quan trọng của chúng. Việc thiết kế chương trình có thể bị ảnh hưởng bởi ngôn ngữ lập trình nhưng nó không luôn bị chúng thống trị .

Giả sử, ta cần xây dựng một chương trình phát sinh ra một đoạn văn bản ngẫu nhiên bằng tiếng Anh sao cho nó có nghĩa. Nếu ta đưa ra các kí tự hay các từ một cách ngẫu nhiên thì kết quả sẽ trở nên vô nghĩa. Chẳng hạn như, việc chọn các ký tự và khoảng trắng ngẫu nhiên có thể xuất ra như sau:

```
xptmxgn xusaja afqnzgxl lhidlwcd rjdjuvpydrlwnjy
```

điều này không thật sự thuyết phục. Nếu ta ước lượng các ký tự thông qua tần xuất xuất hiện của chúng trong văn bản thì ta có thể được kết quả sau:

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

câu này cũng không tốt hơn nhiều. Các từ được chọn từ một từ điển một cách ngẫu nhiên cũng không tạo nên nhiều ý nghĩa hơn:

```
polydactyl equatorial splashily jowl verandah
circumscribible
```

để đạt kết quả tốt hơn chúng ta cần một mô hình thống kê với cấu trúc rõ ràng hơn, chẳng hạn tần xuất xuất hiện của cả cụm từ. Nhưng chúng ta có thể tìm thấy các thống kê này ở đâu?

Ta có thể lấy phần lớn vốn tiếng Anh và các nghiên cứu nó thật chi tiết. Thế nhưng có một cách tiếp cận dễ dàng và thú vị hơn đó là sử dụng bất cứ văn bản hiện có nào để cấu trúc nên mô hình thống kê của ngôn ngữ như đã được dùng trong văn bản đó, và từ đó phát sinh ra văn bản ngẫu nhiên có thống kê tương tự như văn bản nguyên thủy.

3.1. Giải thuật chuỗi Markov

Để giải quyết loại xử lý trên một cách thỏa đáng ta dùng giải thuật chuỗi *Markov*. Nếu chúng ta tưởng tượng rằng dữ liệu nhập vào như là một chuỗi các cụm từ chồng chéo lên nhau, giải thuật sẽ chia mỗi cụm từ này thành hai phần, một tiếp đầu ngữ gồm nhiều từ và một tiếp vĩ ngữ một từ theo sau tiếp đầu ngữ. Giải thuật chuỗi *Markov* xuất ra các cụm từ một cách ngẫu nhiên bằng cách chọn tiếp vĩ ngữ theo sau tiếp đầu ngữ dựa trên các thống kê của văn bản nguyên thủy. Các cụm từ gồm 3 từ cho kết quả tốt – một từ tiếp đầu ngữ hai từ được dùng để chọn ra một từ tiếp vĩ ngữ:

```
    ghép w1 và w2 thành một từ đầu tiên có hai từ cho văn bản
    in w1 và w2
lặp:
  chọn ngẫu nhiên w3 là tiếp đầu ngữ của w1 w2 trong văn bản
```

```
in w3
thay thế w1 và w2 bằng w2 và w3
thực hiện lại vòng lặp
```

Để minh họa, giả sử chúng ta muốn phát sinh một văn bản ngẫu nhiên dựa vào một vài câu dùng các tiếp đầu ngữ hai từ được viết như sau:

```
Show your flowcharts  and conceal your tables and
I will be mystified. Show your tables and your
flowcharts will be obvious (end)
```

Sau đây là một số cặp của các từ nhập vào và từ theo sau chúng:

Tiếp đầu ngữ nhập vào:	Tiếp vĩ ngữ theo sau:
Show your	flowcharts tables
your flowcharts	and will
flowcharts and	conceal
flowchart will	be
your tables	and and
will be	mystified, obvious.
be mystified	Show
be obvious	(end)

Giải thuật *Markov* xử lý văn bản này sẽ bắt đầu bằng việc xuất ra từ `Show your` và tiếp tục lấy ngẫu nhiên một trong hay từ hoặc `flowchart` hoặc `table`. Nếu nó chọn từ `flowchart` thì tiếp đầu ngữ hiện tại sẽ trở thành `your flowchart` và từ kế tiếp sẽ là `and` hoặc `will`. Nếu nó chọn từ `table` thì từ kế tiếp sẽ là `and`. Điều này tiếp tục cho đến khi nó phát sinh ra đủ các từ hoặc cho đến khi tiếp vĩ ngữ là dấu kết thúc.

Chương trình của chúng ta sẽ đọc một đoạn văn bản tiếng Anh và sử dụng giải thuật chuỗi *Markov* để phát sinh ra văn bản mới dựa trên tần suất xuất hiện của các cụm từ với độ dài cố định. Số lượng từ trong tiếp đầu ngữ, trong ví dụ của ta là hai, nó được xem như một tham số. Tạo tiếp đầu ngữ ngắn hơn có khuynh hướng phát sinh ra đoạn văn bản ít liền mạch hơn; tiếp đầu ngữ dài hơn có khuynh hướng phát sinh lại đúng nguyên văn văn bản nhập vào. Với văn bản tiếng Anh, dùng hai từ để chọn từ thứ ba là một sự kết hợp tốt; nó dường như tạo lại văn phong của dữ liệu nhập trong khi thêm vào văn phong bất thường riêng của nó.

Thế nào là một từ? Câu trả lời rõ ràng đó là một chuỗi các kí tự trong bảng chữ cái, nhưng nó có sự khác biệt giữa các từ có dấu câu dính vào vì vậy từ "words" và "words." là khác nhau. Điều này giúp cải thiện chất lượng của đoạn văn được phát sinh bằng cách đặt các chấm câu, và do đó (một cách gián tiếp) ngữ pháp sẽ ảnh hưởng đến việc chọn từ cho dù nó cũng cho phép đặt các dấu trích dẫn và dấu ngoặc đơn không tương xứng được chèn vào. Khi đó, ta sẽ định nghĩa một "từ" là phần ở giữa hai khoảng trắng, một định nghĩa không bị giới hạn trên các ngôn ngữ làm dữ liệu nhập và cho phép dấu câu dính vào các từ. Vì hầu hết các ngôn ngữ lập trình có các công cụ chia văn bản thành các từ cách nhau bởi khoảng trắng, điều này cũng dễ dàng cài đặt.

Theo cách tiếp cận này thì tất cả các từ, tất cả các cụm hai-từ và tất cả các cụm ba-từ xuất hiện trong dữ liệu xuất phải xuất hiện trong dữ liệu nhập, tuy nhiên chúng ta nên tổng hợp các cụm từ bốn-từ và dài hơn. Sau đây là một vài câu được phát sinh bởi chương trình mà chúng ta sẽ phát triển trong chương này, với dữ liệu nhập là văn bản được trích từ chương VII của tác phẩm *mặt trời vẫn mọc* của Ernest Hemingway:

```
As I started up the undershirt onto his chest black, and
big stomach muscles bulging under the light. "You see
them?" Below the line where his ribs stopped were two
raised white welts. "See on the forehead." "Oh, Brett, I
love you." "Let's not talk. Taling's all bilge. I'm going
```

```
away tomorrow." "Yes. Didn't I say so? I am." "Let's have
a drink, then."
```

Ở đây chúng ta thật may mắn là các dấu câu xuất hiện một cách chính xác; nhưng thật ra điều này không xảy ra cũng không sao.

3.2. Chọn lựa cấu trúc dữ liệu

Chúng ta định sẽ giải quyết bài toán với dữ liệu nhập lớn tới mức nào? Chương trình phải chạy với tốc độ nhanh ra sao? Nó dường như hợp lý khi cho chương trình chúng ta đọc vào toàn bộ cuốn sách vì vậy chúng ta cần được chuẩn bị đối với kích thước dữ liệu nhập là n lớn hơn hay bằng 100000 từ. Kết quả xuất ra sẽ là hàng trăm hay có thể hàng nghìn từ, và chương trình phải chạy trong vòng vài giây thay vì phải chạy trong vài phút. Với 100000 từ của văn bản nhập, n là khá lớn vì vậy các giải thuật không thể quá đơn giản nếu chúng ta muốn chương trình chạy nhanh.

Giải thuật *Markov* phải duyệt qua tất cả dữ liệu nhập trước khi bắt đầu phát sinh ra dữ liệu xuất, vì vậy nó phải lưu trữ toàn bộ dữ liệu nhập theo một dạng nào đó. Một khả năng có thể là đọc toàn bộ dữ liệu nhập và lưu chúng trong một chuỗi dài, nhưng chúng ta rõ ràng muốn dữ liệu nhập được cắt thành từng từ. Nếu ta lưu trữ nó như là một mảng các con trỏ trỏ tới các từ thì sự phát sinh dữ liệu xuất là đơn giản: để phát sinh một từ ta duyệt qua văn bản nhập để thấy được các từ tiếp vĩ ngữ có thể theo sau tiếp đầu ngữ nào, và sau đó chọn một cái ngẫu nhiên. Tuy nhiên, điều này có nghĩa là với mỗi một từ được phát sinh ra ta phải quét tất cả 100000 từ nhập vào; 1000 từ xuất ra có nghĩa là hàng trăm triệu phép so sánh chuỗi, điều này sẽ không nhanh được.

Một khả năng nữa là chỉ lưu trữ duy nhất các từ nhập vào cùng với một danh sách các vị trí chúng xuất hiện trong dữ liệu nhập để ta có thể xác định từ tiếp theo một cách nhanh chóng. Ta có thể dùng bảng băm như đã trình bày trong Chương 2, nhưng phiên bản đó không định địa chỉ một cách trực tiếp đối với các yêu cầu của giải thuật *Markov*, mà nó xác định trước

một cánh nhanh chóng tất cả các tiếp vĩ ngữ cho một tiếp đầu ngữ cho tương ứng.

Ta cần một cấu trúc dữ liệu để thể hiện một tiếp đầu ngữ và các tiếp vĩ ngữ tương ứng với nó một cách tốt hơn. Chương trình sẽ có 2 phần, phần nhập vào: xây dựng cấu trúc dữ liệu thể hiện các cụm từ; phần xuất ra: dùng cấu trúc dữ liệu đó để phát sinh dữ liệu xuất ngẫu nhiên. Trong cả 2 phần chúng ta cần tra cứu tiếp đầu ngữ (một cách nhanh chóng): trong phần nhập vào để cập nhật các tiếp vĩ ngữ của nó và trong phần xuất ra để chọn ngẫu nhiên các tiếp vĩ ngữ có thể. Điều này cần một bảng băm mà các khóa của chúng là các tiếp đầu ngữ và các giá trị của chúng là tập các tiếp vĩ ngữ cho các tiếp đầu ngữ tương ứng.

Đối với mục đích mô tả, ta sẽ dùng một tiếp đầu ngữ hai từ để mỗi từ xuất ra dựa trên một cặp từ liền trước nó. Số lượng các từ trong tiếp đầu ngữ không ảnh hưởng đến việc thiết kế và các chương trình nên xử lý bất cứ độ dài tiếp đầu ngữ nào, nhưng việc chọn ra một độ dài sẽ làm cho việc trình bày trở nên cụ thể. Tiếp dầu ngữ và tập tất cả các tiếp vĩ ngữ có thể của nó sẽ gọi là một *trạng thái* (`state`), nó là thuật ngữ chuẩn của các giải thuật *Markov*.

Cho trước một tiếp đầu ngữ, ta cần phải lưu tất cả các tiếp vĩ ngữ theo sau nó để ta có thể truy cập lại chúng sau này. Các tiếp vĩ ngữ là không có thứ tự và được thêm vào mỗi lần một từ. Ta không biết được chúng sẽ có bao nhiêu, vì thế ta cần một cấu trúc dữ liệu có thể mở rộng được một cách dễ dàng và hiệu quả, chẳng hạn danh sách liên kết hoặc mảng động.

Điều gì sẽ xảy ra nếu một cụm từ xuất hiện hơn một lần? Chẳng hạn `'might appear twice'` có thể xuất hiện hai lần nhưng `'might appear once'` chỉ xuất hiện một lần. Điều này có thể được thể hiện bằng cách đặt `'twice'` hai lần trong danh sách tiếp vĩ ngữ cho `'might appear'` hoặc đặt nó một lần cùng với biến đếm tương ứng được gán là 2. Ta thử nó với trường hợp có và không có biến đếm; trường hợp không có thì dễ hơn vì việc thêm một tiếp vĩ ngữ không đòi hỏi kiểm tra nó đã có hay chưa và các

thí nghiệm cho thấy rằng sự khác nhau trong thời gian chạy là không đáng kể.

Tóm lại, mỗi *trạng thái* gồm có một tiếp đầu ngữ và một danh sách các tiếp vĩ ngữ. Thông tin này được lưu trữ trong bảng băm với khóa là tiếp đầu ngữ. Mỗi tiếp đầu ngữ là tập hợp các từ có độ dài cố định. Nếu một tiếp vĩ ngữ xuất hiện nhiều hơn một lần đối với một tiếp đầu ngữ cho trước, mỗi lần xuất hiện sẽ được thêm vào.

Vấn đề cần giải quyết tiếp theo là cách thể hiện các từ. Cách dễ dàng là lưu chúng thành các chuỗi đơn. Vì hầu hết trong văn bản có nhiều từ xuất hiện nhiều lần, ta có thể tiết kiệm việc lưu trữ nếu ta duy trì một bảng băm thứ 2 cho các từ đơn thì mỗi từ được lưu trữ chỉ một lần duy nhất. Điều này sẽ tăng tốc quá trình băm các tiếp đầu ngữ vì chúng ta có thể so sánh các con trỏ thay vì so sánh các ký tự đơn; mỗi một chuỗi duy nhất có địa chỉ duy nhất. Ta sẽ để thiết kế đó như là một bài tập; các chuỗi được trình bày ở đây sẽ được lưu trữ độc lập.

3.3. Xây dựng cấu trúc dữ liệu trong C

Ta hãy bắt đầu bằng một cài đặt của C. Bước đầu tiên là định nghĩa các hằng số.

```
enum {
NPREF = 2,   /* số từ của tiếp đầu ngữ */
NHASH = 4093,/* kích thước của mảng bảng băm state */
MAXGEN = 10000 /* số từ tối đa có thể được phát sinh */
}
```

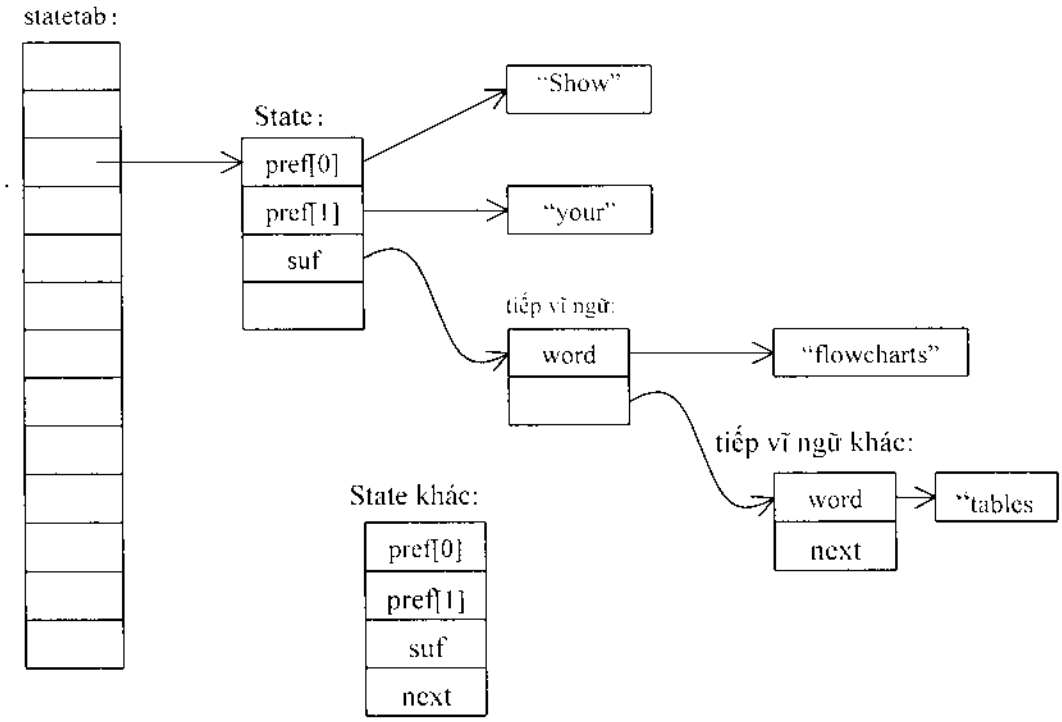
Khai báo này định nghĩa số từ (NPREF) của tiếp đầu ngữ, kích thước của mảng bảng băm (NHASH) và giới hạn trên của số lượng các từ phát sinh (MAXGEN). Nếu hằng số NPREF là một hằng số tại lúc biên dịch thay vì là biến khi chạy chương trình thì việc quản lý lưu trữ sẽ đơn giản hơn. Kích thước mảng được gán khá lớn vì chúng ta muốn dữ liệu nhập là các tài liệu rất lớn, có thể là cả cuốn sách. Ta chọn NHASH = 4093 để nếu dữ liệu vào có

10000 tiếp đầu ngữ khác nhau (các cặp từ) thì chuỗi trung bình sẽ rất ngắn (2 hay 3 tiếp đầu ngữ). Kích thước tiếp đầu ngữ càng lớn hơn thì chiều dài mong muốn của các chuỗi sẽ càng ngắn và nhờ đó sẽ tra cứu nhanh hơn. Chương trình này thật sự là một chương trình minh họa, vì thế tốc độ thực thi không phải là vấn đề quan trọng. Nhưng nếu ta tạo mảng quá nhỏ thì chương trình sẽ không xử lý dữ liệu nhập trong khoảng thời gian hợp lý; trái lại, nếu ta tạo mảng quá lớn thì sẽ không đủ bộ nhớ.

Tiếp đầu ngữ có thể được lưu như mảng các từ. Các phần tử của bảng băm sẽ được thể hiện là một kiểu dữ liệu `State`, gồm danh sách tiếp vĩ ngữ ứng với tiếp đầu ngữ:

```
typedef struct State State;
typedef struct Suffix Suffix;
struct State { /* tiếp đầu ngữ và danh sách tiếp vĩ ngữ
tương ứng */
      char  *pref[NPREF]; /* các từ làm tiếp đầu ngữ */
      Suffix *suf;       /* danh sách tiếp vĩ ngữ */
      State  *next;  /* con trỏ next trong bảng băm */
}
struct Suffix { // danh sách tiếp vĩ ngữ
    char     *word;      /* tiếp vĩ ngữ */
    Suffix *next;/* con trỏ next trong danh sách này */
}
State *statetab[NHASH] /* bảng băm của các state */
```

Một cách hình tượng, cấu trúc dữ liệu giống như sau:



Chúng ta cần một hàm băm cho các tiếp đầu ngữ, là mảng của các chuỗi. Khi đó, hàm băm sẽ được thực hiện qua tất cả các chuỗi trong mảng:

```
/* Hàm băm tính giá trị băm cho mang NPREF chuỗi */
unsigned int hash (char *s[NPREF])
{
        unsigned int h;
        unsigned char *p;
        int i;

        h = 0;
        for (i = 0; i < NPREF; i++)
```

```
                for (p = (unsigned char *) s[i]; *p !=
'\0'; p++)
                        h = MULTIPLIER * h + *p;
        return h % NHASH;
}
```

Một sự thay đổi tương tự cho hàm tra cứu để hoàn chỉnh việc cài đặt cho bảng băm như sau:

```
/* Hàm lookup: tìm kiếm tiếp đầu ngữ; hoặc tạo ra nó
khi được yêu cầu */
/* trả về một con trỏ nếu tìm thấy hoặc được tạo ra;
ngược lại, trả về NULL */
State* lookup(char *prefix[NPREF], int create)
{
int i, h;
State *sp;
h = hash(prefix);
for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
                if (strcmp(prefix[i]), sp->pref[i] != 0)
                        break
                if (i == NPREF)    /*tìm thấy */
                return sp;
        }
        if (create) {
        sp = (State *)emalloc(sizeof(State));
```

```
            for (i = 0; i < NPREF; i++)
                sp->pref[i] = prefix[i];
            sp->suf = NULL;
            sp->next = statetab[h];
            statetab[h] = sp;
        }
        return sp;
    }
```

Chú ý rằng hàm `lookup` không tạo ra bản sao của các chuỗi được đưa vào tìm kiếm khi nó tạo ra `State` mới; nó chỉ lưu các con trỏ trong `sp->pref[]`. Các lời gọi hàm tra cứu phải đảm bảo rằng dữ liệu sẽ không bị ghi đè lên sau này. Chẳng hạn như, nếu các chuỗi nằm trong vùng đệm nhập/xuất, một bản sao phải được tạo ra trước khi gọi hàm tra cứu; nếu không dữ liệu nhập tiếp theo có thể viết đè lên dữ liệu mà bảng băm trỏ tới. Các quyết định về thành phần nào sẽ giữ tài nguyên dùng chung của giao tiếp thường xuyên được phát sinh. Chúng ta sẽ khám phá chủ đề này trong chương kế tiếp.

Tiếp theo ta cần xây dựng một bảng băm khi tập tin được đọc vào:

```
    /*Hàm build: đọc thông tin nhập vào, xây dựng bảng tiếp
đầu ngữ */
    void build(char *prefix[NPREF], FILE *f)
    {
        char buf[100], fmt[10];
    /* tạo định dạng chuỗi; nếu dùng %s có thể bị tràn */
        sprintf(fmt, "%%%ds", sizeof(buf) - 1);
        while (fscanf(f, fmt, buf) != EOF)
            add(prefix, estrdup(buf));
    }
```

Một lời gọi bất thường đến hàm `sprintf` gây ra một vấn đề khó chịu tới hàm `fscanf`. nếu không bị tác động thì nó sẽ thực hiện tốt chức năng của nó. Một lời gọi hàm `fscanf` với định dạng `%s` sẽ đọc từ kế tiếp được phân cách bằng khoảng trắng từ tập tin vào vùng đệm, nhưng không giới hạn về kích thước: một từ dài có thể tràn vùng đệm nhập. tạo nên sự phá hủy. Nếu vùng đệm có độ dài là 100 bytes (độ dài này lớn hơn nhiều so với những gì chúng ta muốn thấy xuất hiện trong các văn bản bình thường), ta có thể dùng định dạng `%99s` (để dành một byte cho ký tự kết thúc chuỗi `'\0'`), điều này bảo cho hàm `fscanf` dừng sau khi đọc 99 bytes. Một từ dài sẽ được chia ra thành các chuỗi nhỏ, điều này không thích hợp nhưng an toàn. Chúng ta có thể khai báo như sau:

```
?       enum { BUFSIZE = 100 };
?       char    fmt[] = "%99s";  /* BUFSIZE-1 */
```

nhưng điều này đòi 2 hằng số cho một quyết định không có cơ sở – là kích thước của vùng đệm – và đưa ra yêu cầu để duy trì mối quan hệ của chúng. Vấn đề này có thể được giải quyết một lần và cho tất cả bằng cách tạo ra một định dạng chuỗi linh động bằng hàm `sprintf`, vì vậy đó chính là cách tiếp cận mà chúng ta dùng.

Hai đối số để xây dựng là mảng tiếp đầu ngữ chứa `NPREF` từ nhập vào và một con trỏ `FILE`. Hàm `add` cần truyền tiếp đầu ngữ và bản sao của từ nhập vào. Nó thêm một phần mới vào bảng băm và tăng tiếp đầu ngữ lên:

```
        /* Hàm add: thêm từ vào danh sách tiếp vĩ ngữ, cập nhật
tiếp đầu ngữ */
        void add(char *prefix[NPREF], char *suffix)
        {
                State *sp;
                sp = lookup(prefix, 1); /* tạo prefix mới nếu
không tìm thấy*/
                addsuffix(sp, suffix);
                /* dịch chuyển xuống trong bảng prefix */
```

```
        memmove(prefix, prefix + 1,
                (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = suffix;
    }
```

Lời gọi hàm `memmove` dùng để xóa dữ liệu từ một mảng. Nó dịch các phần tử từ `1` đến `NPREF-1` trong bảng tiếp đầu ngữ xuống các vị trí từ `0` đến `NPREF-2`, xóa tiếp đầu ngữ đầu tiên và tạo ra khoảng trống mới ở cuối.

Hàm `addsuffix` thêm một tiếp vĩ ngữ mới:

```
    /* Hàm addsuffix: thêm vào state, suffix không được
thay đổi*/
    void addsuffix(State *sp, char *suffix)
    {
        Suffix *suf;
        suf = (Suffix *) emalloc(sizeof(Suffix));
        suf->word = suffix;
        suf->next = sp->suf;
        sp->suf = suf;
    }
```

Chúng ta chia hành động cập nhật `State` vào hai hàm: hàm `add` thực hiện chức năng tổng quát trong việc thêm tiếp vĩ ngữ vào tiếp đầu ngữ, trong khi hàm `addsuffix` thực hiện hành động thêm một từ vào danh sách tiếp vĩ ngữ. Hàm `add` được dùng bởi hàm `build` nhưng hàm `addsuffix` được chỉ được gọi bên trong hàm `add`; nó là chi tiết cài đặt mà có thể thay đổi và dường như tốt hơn đưa nó thành hàm riêng, cho dù nó được gọi chỉ có một lần.

3.4. Phát sinh kết quả

Với cấu trúc dữ liệu đã xây dựng, bước kế tiếp là phát sinh ra dữ liệu xuất. Ý tưởng cơ bản như trước đây: cho một tiếp đầu ngữ, chọn ngẫu nhiên

một tiếp vĩ ngữ của nó, in ra, sau đó chuyển tới tiếp đầu ngữ kế. Đây là trạng thái cố định của quá trình xử lý; ta phải tìm ra cách thức để bắt đầu và kết thúc giải thuật. Việc bắt đầu giải thuật sẽ dễ dàng nếu chúng ta nhớ các từ của tiếp đầu ngữ đầu tiên và bắt đầu với chúng. Và nhờ thế việc ngừng giải thuật cũng dễ dàng. Ta cần một từ đánh dấu kết thúc để dừng giải thuật. Sau khi tất cả dữ liệu nhập xong, ta thêm một từ kết thúc, nó là một "từ" mà đảm bảo không xuất hiện trong bất kỳ dữ liệu nhập nào.

```
build(prefix, stdin);
add(prefix, NONWORD);
```

NONWORD có thể là từ sẽ không bao giờ xuất hiện trong dữ liệu nhập bình thường. Vì các từ nhập vào được cách nhau bởi khoảng trắng, do đó ký tự xuống dòng không thể nào là một từ:

```
char NONWORD[] = "\n"; /* không thể xuất hiện như một
từ thật sự */
```

Một điều quan tâm nữa là điều gì sẽ xảy ra nếu không đủ dữ liệu nhập để bắt đầu giải thuật? Có hai cách tiếp cận đối với loại vấn đề này, hoặc là thoát sớm nếu không đủ dữ liệu nhập, hoặc sắp xếp sao cho luôn có đủ dữ liệu và không quan tâm đến vấn đề kiểm tra nữa. Trong chương trình này, cách tiếp cận thứ hai là tốt hơn.

Chúng ta có thể khởi tạo việc xây dựng và phát sinh với một tiếp đầu ngữ tự cho nào đó, điều này đảm bảo luôn có đủ dữ liệu nhập cho chương trình. Để chuẩn bị dữ liệu cho các vòng lặp, ta khởi tạo mảng tiếp đầu ngữ gồm toàn các từ NONWORD. Điều này có lợi điểm là từ đầu tiên của tập tin nhập vào sẽ là tiếp vĩ ngữ đầu tiên của tiếp đầu ngữ giả, vì vậy vòng lặp phát sinh chỉ cần in các tiếp vĩ ngữ của nó tạo ra.

Trong trường hợp dữ liệu xuất ra quá dài không thể quản lý, ta có thể ngừng giải thuật sau khi một số từ được xuất ra hoặc khi nó gặp từ NONWORD là tiếp vĩ ngữ.

Thêm một vài từ NONWORD vào cuối dữ liệu thật sự làm đơn giản quá trình xử lý chính các vòng lặp của chương trình; đó là một ví dụ của kỹ thuật thêm các giá trị *lính canh* để đánh dấu các giới hạn biên.

Như khi làm luật, cố gắng giải quyết các điều bất thường, các ngoại lệ và các trường hợp đặc biệt trong dữ liệu. Rất khó để viết đúng mã nguồn vì thế luồng điều khiển cần phải càng đơn giản và càng đúng quy tắc càng tốt.

Hàm generate dùng giải thuật đã được mô tả ban đầu. Nó xuất ra một từ trên một dòng, chúng có thể được nhóm thành các dòng dài hơn bằng một chương trình xử lý văn bản. Đó là chương trình định dạng đơn giản gọi là fmt sẽ được trình bày trong Chương 9.

Với việc dùng các chuỗi NONWORD để khởi tạo và đánh dấu kết thúc đã tạo ra được sự bắt đầu và kết thúc một cách hợp lý trong hàm generate:

```
/* Hàm generate: phát sinh kết quả mỗi dòng một từ */
void generate (int nwords)
{
        State *sp;
        Suffix *suf;
        char *prefix[NPREF], *w;
        int i, nmatch;
        for (i = 0; i < NPREF; i++) /* khởi tại tiếp đầu
ngữ */
                prefix[i] = NONWORD;
        for (i = 0; i < nwords; i++) {
                sp = lookup(prefix, 0);
                nmatch = 0;
                for (suf = sp->suf; suf != NULL; suf = suf-
>next)
```

```
            if (rand() % ++nmatch == 0)
                w = suf->word;
        if (strcmp(w, NONWORD) == 0)
            break;
        print("%s\n", w);
        memmove(prefix, prefix+1,
            (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = w;
    }
}
```

Chú ý giải thuật chọn một phần tử ngẫu nhiên khi chúng ta không biết có bao nhiêu phần tử. Biến `nmatch` đếm số lượng các phần tử phù hợp khi duyệt qua danh sách. Biểu thức:

```
rand() % ++nmatch == 0;
```

tăng biến `nmatch` và sẽ đúng với xác suất `1/nmatch`. Do đó phần tử phù hợp đầu tiên được chọn với xác suất là 1, phần tử thứ hai sẽ thay nó với xác suất là `1/2`, phần tử thứ ba là `1/3` và v.v.... Tại bất kỳ thời điểm nào, mỗi một trong số `k` phần tử phù hợp được tìm thấy sẽ được chọn với xác suất `1/k`.

Lúc đầu ta gán tiếp đầu ngữ với giá trị khởi động để đảm bảo được đưa vào bảng băm. Các giá trị tiếp vĩ ngữ đầu tiên chúng ta tìm được sẽ là các từ đầu tiên của văn bản, vì chúng là duy nhất theo sau tiếp đầu ngữ ban đầu. Sau đó, các tiếp vĩ ngữ ngẫu nhiên sẽ được chọn. Vòng lặp gọi hàm `lookup` để tìm trong các phần của bảng băm cho tiếp đầu ngữ hiện tại, sau đó chọn tiếp vĩ ngữ ngẫu nhiên, in nó ra và chuyển sang tiếp đầu ngữ mới.

Nếu tiếp vĩ ngữ chúng ta chọn là `NONWORD` thì ta đã làm xong, vì ta đã chọn được `State` tương ứng với dữ liệu nhập. Nếu tiếp vĩ ngữ không phải là `NONWORD`, ta in ra, sau đó bỏ từ đầu tiên của tiếp đầu ngữ bằng cách gọi hàm `memmove`, chuyển tiếp vĩ ngữ thành từ cuối cùng của tiếp đầu ngữ và lặp lại.

Bây giờ chúng ta có thể viết chung tất cả chúng lại với nhau thành một hàm main để đọc dữ liệu nhập vào và phát sinh một số lượng từ được chỉ định trước:

```
/* Hàm main cho giải thuật chuỗi markow phát sinh văn
bản ngẫu nhiên */
int main(void)
{
        int i, nwords = MAXGEN;
        char *prefix[NPREF]; /* mảng prefix hiện hành */
        for (i = 0; i < NPREF; i++) /* khởi tạo prefix */
                prefix[i] = NONWORD;
        build(prefix, stdin);
        add(prefix, NONWORD);
        generate(nwords);
        return 0;
}
```

Đến đây đã hoàn tất việc cài đặt bằng C của chúng ta. Ta sẽ quay trở lại vào cuối chương để so sánh các chương trình được viết bằng các ngôn ngữ khác nhau. Các điểm mạnh của C là nó cho các nhà lập trình sự điều khiển hoàn toàn việc cài đặt và các chương trình được viết có khuynh hướng chạy nhanh. Tuy nhiên cái giá phải trả là các nhà lập trình C phải làm nhiều việc hơn, cấp phát và phục hồi bộ nhớ, tạo bảng băm và các danh sách liên kết và những việc tương tự. C là một công cụ sắc bén đối với một người có thể tạo nên một chương trình hiệu quả và đẹp hoặc nó cũng chính là một đống hỗn độn.

Bài tập 3-1. Giải thuật cho việc chọn một phần tử ngẫu nhiên từ một danh sách có chiều dài không biết trước phụ thuộc vào bộ phát sinh số ngẫu nhiên tốt. Hãy thiết kế và cài đặt các thực nghiệm để xác định phương pháp này làm việc như thế nào trong thực tế.

Bài tập 3-2. Nếu mỗi từ nhập vào được lưu trong một bảng băm thứ hai, văn bản chỉ lưu trữ một lần, điều này sẽ tiết kiệm không gian. Cách tổ chức này sẽ cho phép chúng ta so sánh các con trỏ thay vì là so sánh các chuỗi trong các bảng băm của các tiếp đầu ngữ, việc làm này sẽ giúp chạy nhanh hơn. Hãy cài đặt phiên bản này và tính sự thay đổi về tốc độ và tiêu hao bộ nhớ.

Bài tập 3-3. Hãy loại bỏ các câu có các từ lính canh `NONWORD` ở đầu và cuối của dữ liệu và thay đổi hàm `generate` để nó có thể bắt đầu và kết thúc hợp lý mà không cần chúng. Đảm bảo nó phát sinh ra dữ liệu chính xác đối với dữ liệu nhập là 0, 1, 2, 3 và 4 từ. So sánh cài đặt này với phiên bản sử dụng lính canh.

3.5. Bản cài đặt trong Java

Bản cài đặt thứ hai của giải thuật chuỗi *Markov* là bằng Java. Các ngôn ngữ hướng đối tượng như Java khuyến khích mọi người đặc biệt chú ý các giao tiếp giữa các `component` của chương trình mà chúng được đóng gói như là các thành phần dữ liệu độc lập được gọi là đối tượng (`object`) hoặc lớp (`class`) cùng với các hàm tương ứng gọi là các phương thức (`method`).

Java có thư viện phong phú hơn C bao gồm một tập các lớp `container` để nhóm các đối tượng hiện có theo nhiều cách khác nhau. Một ví dụ như kiểu dữ liệu `Vector` cho phép hoạt động như một mảng động có thể mở rộng và có thể lưu trữ bất kỳ loại đối tượng nào. Một ví dụ khác là lớp `Hashtable` của nó có thể lưu trữ và truy xuất dữ liệu của một kiểu dữ liệu dùng các đối tượng có kiểu dữ liệu khác như là khóa.

Trong ứng dụng của chúng ta, `Vector` của các chuỗi là sự lựa chọn tự nhiên để lưu các tiếp đầu ngữ và tiếp vĩ ngữ. Ta có thể dùng một `Hashtable` mà các khóa của chúng là các `Vector` tiếp đầu ngữ và các giá trị của chúng là các `Vector` tiếp vĩ ngữ. Thuật ngữ cho kiểu cấu trúc này được gọi là `map` từ tiếp đầu ngữ đến tiếp vĩ ngữ; trong Java, chúng ta không có kiểu `State` rõ ràng vì `Hashtable` ngầm định nối (`map`) tiếp đầu ngữ với tiếp vĩ ngữ. Thiết kế này khác với phiên bản viết bằng C, được cài đặt cấu trúc

State nó lưu cả tiếp đầu ngữ và danh sách tiếp vĩ ngữ, và được băm trên tiếp đầu ngữ để phục hồi hoàn toàn State.

Một Hashtable cung cấp một phương thức put để lưu trữ cặp giá trị khóa và phương thức get để trích giá trị của khóa:

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

Cài đặt của chúng ta có 3 lớp. Lớp đầu tiên là Prefix, lưu các từ của tiếp đầu ngữ:

```
class Prefix {
public Vector pref; // gồm NPREF từ liền quan với nhau
    ...
```

Lớp thứ hai là Chain, đọc dữ liệu nhập, xây dựng bảng băm và phát sinh dữ liệu xuất; sau đây là các biến của lớp này:

```
class Chain {
static final int NPREF = 2; // kích thước của prefix
    static final String NONWORD =  "\n";
        // "từ" không thể xuất hiện
    Hashtable statetab = new Hashtable();
        // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
        // khởi tạo tiếp đầu ngữ
    Random rand = new Random();
    ...
```

Lớp thứ ba là giao tiếp chung; nó bao gồm hàm main và các khởi tạo lớp Chain:

```
class Markov {
        static final int MAXGEN = 10000; // số từ được
tạo ra lớn nhất
        public static void main (String[] args) throws
IOException
        {
                Chain chain = new Chain();
                int nwords = MAXGEN;
                chain.build(System.in);
                chain.generate(nwords);
        }
}
```

Khi một thực thể của lớp `Chain` được tạo ra, nó sẽ lần lượt tạo ra một bảng băm và khởi tạo danh sách các tiếp đầu ngữ là `NONWORD`. Hàm `build` dùng các hàm của thư viện `StreamTokenizer` để phân tích dữ liệu nhập thành những từ riêng dựa vào ký tự khoảng trắng. Có ba lời gọi hàm trước khi vòng lặp thực hiện chuyển từ bộ phân tích cú pháp thành `State` đúng theo định nghĩa về "từ"của chúng ta.

```
// Hàm build: tạo bảng State từ dữ liệu nhập
void build(InputStream in) throws IOException
{
        StreamTokenizer st = new StreamTokenizer(in);
        st.resetSyntax();       // xóa các luật default
        st.wordChars(0, Character.MAX_VALUE);
                        // kích hoạt tất cả chuỗi */
        st.whitespaceChars(0, ' '); // chấp nhận khoảng
trắng
        while (st.nextToken() != st.TT_EOF)
```

```
                add(st.sval);
            add(NONWORD);
        }
```

Hàm `add` thêm vào `vector` các tiếp vĩ ngữ cho tiếp đầu ngữ hiện hành trong bảng băm; nếu không có tiếp vĩ ngữ nào (`vector` tiếp vĩ ngữ rỗng), hàm `add` sẽ tạo mới một `vector` và một tiếp đầu ngữ mới để lưu vào trong bảng băm. Ngược lại thì nó thêm từ mới vào `vector` tiếp vĩ ngữ và xét tới tiếp đầu ngữ tiếp theo bằng cách bỏ từ đầu tiên và tiếp tục thêm từ mới ở phần còn lại vào.

```
    // Hàm add: thêm từ vào danh sách tiếp vĩ ngữ, cập nhật
tiếp đầu ngữ
    void add(String word)
    {
        Vector suf = (Vector) statetab.get(prefix);
        if (suf == null) {
            suf = new Vector();
            statetab.put(new Prefix(prefix), suf);
        }
        suf.addElement(word);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(word);
    }
```

Chú ý rằng nếu `vector suf` bằng `null` thì hàm `add` sẽ thêm một tiếp đầu ngữ mới vào bảng băm, hay đúng hơn là chính tiếp đầu ngữ hiện hành. Bởi vì lớp bảng băm lưu các phần tử theo kiểu tham chiếu, và nếu ta không tạo ra một bản sao thì dữ liệu có thể bị ghi chồng vào bảng băm. Đây là vấn đề mà ta đã giải quyết trong chương trình C.

Hàm phát sinh thì tương tự như phiên bản trong C, nhưng có vẻ hay hơn một chút vì nó có thể chỉ mục một mảng các phần tử ngẫu nhiên một cách trực tiếp thay vì phải duyệt qua một danh sách.

```
// Hàm generate: phát sinh kết quả
void generate(int nwords)
{
        prefix = new Prefix(NPREF, NONWORD);
        for (int i =0; i < nwords; i++) {
                Vector s = (Vector) statetab.get(prefix);
                int r = Math.abs(rand.nextInt()) %
s.size();
                String suf = (String) s.elementAt(r);
                if (suf.equals(NONWORD))
                        break;
                System.out.println(suf);
                prefix.pref.removeElementAt(0);
                prefix.pref.addElement(suf);
        }
}
```

Hai `constructor` của lớp `Prefix` tạo mới các thực thể từ dữ liệu được cung cấp. `Constructor` đầu tiên là bản sao của tiếp đầu ngữ đã có sẵn, còn `contructor` thứ hai tạo ra một tiếp đầu ngữ từ n bản sao của một chuỗi; ta dùng nó để tạo ra `NPREF` bản sao `NONWORD` khi khởi tạo:

```
// Prefix constructor: tạo prefix mới từ prefix có sẵn
Prefix(Prefix p)
{
```

```
            pref = (Vector) p.pref.clone();
      }

      // Prefix constructor: tạo prefix mới từ n bản sao của
chuỗi str
      Prefix()
      {
            pref = new Vector();
            for (int i = 0; i < n; i++)
                  pref.addElement(str);
      }
```

Lớp `Prefix` cũng có hai phương thức `hashCode` và `equals`, và chúng chỉ được gọi trong phần cài đặt của bảng băm nhằm vào mục đích chỉ mục và tìm kiếm. Do đó cần có một lớp riêng chứa hai phương thức trên cho bảng băm và chúng giống như là hai phương thức của lớp `Prefix`.

Phương thức `hashCode` tạo ra một giá trị băm đơn bằng việc gọi kết hợp các phương thức `hashCode` của các phần tử trong `vector`:

```
      static final int MULTIPLIER = 31;
      // Phương thức hashCode: tạo giá trị băm từ tất cả các
tiếp đầu ngữ
      public int hashCode()
      {
            int h = 0;
            for (int i = 0; i < pref.size(); i++)
                  h = MULTIPLIER * h +
pref.elementAt(i).hashCode();
            return h;
      }
```

còn phương thức `equals` so sánh hai tiếp đầu ngữ có bằng nhau hay không:

```
    // Phương thức equals: so sánh hai tiếp đầu ngữ có bằng
nhau hay không
    public boolean equals(Object o)
    {
        Prefix p = (Prefix) o;
        for (int i = 0; i < pref.size(); i++)
            if
(!pref.elementAt(i).equals(p.pref.elementAt(i)))
                return false;
        return true;
    }
```

Bản cài đặt trong Java gọn hơn đáng kể so với phiên bản cài đặt trong C và chi tiết hơn; các `Vector` và bảng băm là các ví dụ tiêu biểu. Bên cạnh đó, việc quản lý bộ nhớ rất dễ vì các `vector` có thể mở rộng khi cần và bộ dọn dẹp bộ nhớ tự động thực hiện theo nguyên tắc là tự động khôi phục những vùng nhớ không dùng đến nữa. Nhưng để dùng lớp bảng băm, ta cần phải viết lại phương thức `hashCode` và `equals`, vì thế Java không tự động thực hiện mọi chi tiết.

Cách thức so sánh trong C và Java được mô tả và thực hiện trên cùng cấu trúc dữ liệu cơ bản, tuy nhiên các hàm trong Java có nhiều tính năng hơn. Chẳng hạn như, việc chuyển từ kiểu `Vector` sang kiểu mảng sẽ được thực hiện một cách dễ dàng. Còn trong C, cần phải hiểu rõ cách thức làm việc của tất cả các cấu trúc dữ liệu; chẳng hạn như việc cài đặt bảng băm trên mảng được lưu ở nhiều nơi khác nhau, hàm `lookup` phải biết cấu trúc `State` và tiếp vĩ ngữ, và tất cả đều biết được kích thước của mảng tiếp đầu ngữ.

Bài tập 3-4. Hãy viết lại giải thật *Markov* trong Java dùng cấu trúc dữ liệu mảng thay vì dùng `vector` cho tiếp đầu ngữ trong lớp `State`.

3.6. Bản cài đặt trong C++

Bản cài đặt thứ ba của chúng ta bằng ngôn ngữ C++. Vì ngôn ngữ C++ gần như bao gồm luôn ngôn ngữ C, nên nó cũng có thể dùng như C với một vài quy ước chuẩn, phiên bản nguyên thủy của giải thuật *Markov* trong C vẫn đúng trong C++. Tuy nhiên, một tính năng khác nữa của C++ là cho phép định nghĩa các lớp cho các đối tượng trong chương trình, nó khác với những gì chúng ta đã làm trong Java; điều này cho phép ta dấu đi các chi tiết cài đặt. Ta quyết định dùng thư viện mẫu chuẩn (STL: Standard Template Library) vì STL có các cơ chế đã được xây dựng sẵn nó đã tích hợp nhiều thứ mà chúng ta cần. Chuẩn ISO của ngôn ngữ C++ bao gồm STL như là một phần của sự tạo ra ngôn ngữ này.

STL cung cấp các `container` chẳng hạn như các `vector`, danh sách liên kết, tập hợp và một tập các giải thuật cơ bản như tìm kiếm, sắp xếp, chèn và xóa. Việc dùng các `template` trong C++, mỗi giải thuật của STL thao tác trên các `container` khác nhau, bao gồm cả các kiểu người dùng định nghĩa và các kiểu dữ liệu chuẩn. Các `container` ở đây được định nghĩa như các `template` của C++ mà đã được dùng cho kiểu dữ liệu cụ thể; chẳng hạn như `vector container` có thể được dùng để tạo ra một kiểu dữ liệu cụ thể như `vector<int>` hoặc `vector<string>`. Tất cả các thao tác trên `vector`, bao gồm các giải thuật chuẩn hỗ trợ sắp xếp, đều có thể được dùng trên các kiểu dữ liệu này.

Một `vector container` tương tự như `vector` trong Java, STL cung cấp một `deque container`. Một `deque` (được phát âm là "deck") là một `double-ended queue` nó tương ứng với những gì mà ta đã thực hiện với các tiếp đầu ngữ trong giải thuật *Markov*; nó giữ `NPREF` phần tử tiếp đầu ngữ, và cho phép lấy phần tử ra *(pop)* ở đầu và thêm mới một phần tử *(push)* vào cuối, cả hai thao tác trên tốn *O(1)*. Cấu trúc `STL deque` thì tổng quát hơn những gì mà chúng ta cần, vì nó cho phép `push` và `pop` ở cuối hàng đợi, nhưng việc đảm bảo tốc độ thực thi kiến cho nó luôn được chọn.

STL cũng cung cấp `map container`, dựa trên cây cân bằng, nó lưu các cặp giá trị khóa và thời gian truy xuất tới giá trị tương ứng với khóa là *O(logn)*. Cấu trúc `map` có thể không hiệu quả bằng bảng băm, nhưng nó cũng tốt vì khi dùng nó không cần phải viết thêm mã lệnh.

Ta cũng dùng các hàm so sánh đã được xây dựng sẵn, nó sẽ thực hiện việc so sánh chuỗi trong các chuỗi tiếp đầu ngữ.

Với các phần được mô tả trên, việc cài đặt sẽ trở nên rất dễ dàng. Sau đây là các khai báo:

```
typedef deque<string> Prefix;
map<Prefix, vector<string>> statetab; // prefix-
>suffixes
```

STL cung cấp một `template` cho cấu trúc `deque`; ký hiệu `deque<string>` khai báo nó là hàng đợi `deque` mà các phần tử có kiểu chuỗi. Vì kiểu dữ liệu trên được dùng một vài lần trong chương trình nên ta định nghĩa chúng thành `Prefix` bằng từ khóa `typedef`. Kiểu dữ liệu `map` lưu các tiếp đầu ngữ và các tiếp vĩ ngữ tương ứng, nhưng vì ta không đặt cho nó một tên riêng biệt nên cấu trúc `map` khai báo một biến `statetab` là sự kết hợp giữa các tiếp đầu ngữ và các `vector` chuỗi ký tự. Phiên bản cài đặt trong C++ cho thấy được tính tiện dụng của nó hơn C hoặc Java, vì ta không cần sự hỗ trợ của hàm băm và hàm `equals`.

Hàm `main` khởi tạo biến có kiểu dữ liệu `Prefix`, đọc dữ liệu vào (từ dòng nhập liệu chuẩn bằng việc gọi hàm `cin` trong thư viện `iostream` của C++), thêm vào cuối, và cũng giống như các phiên bản trước bước cuối cùng là phát sinh từ.

```
/* Hàm main của giải thuật chuỗi markov phát sinh văn
bản ngẫu nhiên trong C++ */
int main(void)
{
        int nwords = MAXGEN;
```

```
        Prefix prefix; // tiếp đầu ngữ hiện hành
        for (int i = 0; i < NPREF; i++)// khởi tạo prefix
                add(prefix, NONWORD);
        build(prefix, cin);
        add(prefix, NONWORD);
        generate(nwords);
        return 0;
}
```

Hàm `build` dùng hàm thư viện `iostream` để đọc dữ liệu vào mỗi lần một từ:

```
// Hàm build: đọc dữ liệu vào và tạo ra bang state
void build(Prefix& prefix, istream& in)
{
        string buf;
        while(in >> buf)
                add(prefix, buf);
}
```

Chuỗi `buf` sẽ tăng kích thước khi cần thiết để quản lý các từ có chiều dài tùy ý.

Hàm `add` cho thấy nhiều lợi điểm của việc dùng STL:

```
/* Hàm add: thêm từ vào danh sách các tiếp vĩ ngữ, cập
nhật lại tiếp đầu ngữ */
void add(Prefix& prefix, const string& s)
{
        if (prefix.size() == NPREF) {
                statetab[prefix].push_back(s);
                prefix.pop_front();
```

```
        }
        prefix.push_back(s);
    }
```

Container của kiểu dữ liệu map định nghĩa chồng toán tử lấy chỉ số (toán tử []) như là một hàm tìm kiếm. Khi đó biểu thức statetab[prefix] sẽ thực hiện tìm kiếm trong bảng statetab phần tử có khóa là prefix và trả về một tham chiếu tới phần tử mong muốn; một vector sẽ được tạo ra nếu không tồn tại khóa prefix. Hàm thành viên push_back của vector và deque cho phép thêm một chuỗi mới vào cuối vector hoặc deque; và hàm pop_front cho phép lấy chuỗi đầu tiên ra khỏi deque.

Việc phát sinh văn bản tương tự như các phiên bản trước:

```
// Hàm generate: phát sinh ra mỗi từ trên một dòng
void generate(int nwords)
{
    Prefix prefix;
    int i;
    for (i = 0; i < NPREF; i++) // khởi tạo prefix
        add(prefix, NONWORD);
    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();
        prefix.push_back(w);
    }
```

```
}
```

Tóm lại, mã nguồn trong phiên bản C++ cực kỳ trong sáng và đẹp mắt, cấu trúc dữ liệu trực quan và giải thuật rất dễ hiểu. Nhưng giá phải trả cho phiên bản này là chạy chậm hơn phiên bản được cài trong C, dù rằng nó không phải là phiên bản chạy chậm nhất. Chúng ta sẽ trở lại so sánh tốc độ thực thi của các phiên bản trong phần 3.8.

Bài tập 3-5. Điểm mạnh của STL là tính dễ tương thích với các cấu trúc dữ liệu khác nhau. Hãy thay đổi phiên bản C++ trên bằng cách dùng các cấu trúc dữ liệu khác để biểu diễn tiếp đầu ngữ, danh sách các tiếp vĩ ngữ và bảng `state`. Hãy so sánh tốc độ thực thi thay đổi như thế nào với các cấu trúc dữ liệu khác?

Bài tập 3-6. Hãy viết một phiên bản C++ chỉ dùng các lớp và kiểu dữ liệu chuỗi mà không dùng các hàm thư viện cao cấp. Hãy so sánh cách viết và tốc độ của nó với phiên bản dùng STL.

3.7. Awk và Perl

Để có một cái nhìn tổng quan về bài toán, phiên bản cuối cùng sẽ được cài đặt bằng hai ngôn ngữ `script` phổ biến là Awk và Perl. Chúng cung cấp việc quản lý chuỗi và mảng kết hợp cùng với các tính năng cần thiết khác cho ứng dụng này.

Mảng kết hợp là một cách biểu diễn tiện lợi của bảng băm; nó giống như một mảng nhưng các thành phần của nó có dạng chuỗi hay số tùy ý, hoặc có thể chỉ là một danh sách các dấu phẩy phân cách giữa chúng. Nó là một dạng ánh xạ từ một kiểu dữ liệu thành các kiểu dữ liệu khác. Trong Awk, tất cả các mảng là mảng liên kết; còn trong Perl thì có 2 quy ước về mảng: mảng chỉ mục (indexed array) là mảng mà các phần tử là số nguyên, mảng kết hợp được gọi là "băm", vì nó được cài đặt dưới dạng băm.

Bản cài đặt trong Awk và Perl được cài đặt với tiếp đầu ngữ hai từ.

```
# markov.awk: giải thuật chuỗi markov với các tiếp đầu
ngữ gồm hai từ
```

```
    BEGIN { MAXGEN = 10000; NONWORD = "\n"; w1 = w2 =
NONWORD }
    {   for (i = 1; I <= NF; i++) { # đọc tất cả các từ
            statetab[w1,w2,++nsuffix[w1,w2]] = $i
            w1 = w2
            w2 = $i
        }
    }
    END {
        statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD # thêm
vào cuối
        w1 = w2 = NONWORD
        for (i = 0; i < MAXGEN; i++) {# phát sinh dữ liệu
            r = int(rand()*nsuffix[w1,w2]) + 1  #
nsuffix >=1
            p = statetab[w1, w2, r]
            if (p == NONWORD)
                exit
            print p
            w1 = w2
            w2 = p
        }
    }
```

Awk là một ngôn ngữ lập trình thực hiện theo mẫu (pattern): dữ liệu nhập được đọc mỗi lần một dòng, mỗi dòng tương ứng với một mẫu, một hành động tương ứng được thực hiện cho mỗi dòng. Có hai mẫu đặc biệt là BEGIN và END.

Một hành động là một khối các phát biểu đặt trong dấu ngoặc móc. Trong phiên bản Awk của chương trình *Markov*, khối `BEGIN` khởi tạo các tiếp đầu ngữ và một số biến khác.

Khối tiếp theo không có `mẫu` nào, vì vậy theo ngầm định sẽ được thực thi mỗi lần một dòng dữ liệu nhập. Awk tự động tách mỗi dòng của dữ liệu nhập thành các `trường (field)` khác nhau (mỗi `trường` được phân cách bằng khoảng trắng) được đặt tên từ `$1` đến `$NF`; biến `NF` là số lượng trường. Phát biểu:

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

tạo ra ánh xạ từ tiếp đầu ngữ đến các tiếp vĩ ngữ tương ứng. Mảng `nsuffix` đếm số các tiếp vĩ ngữ và phần tử `nsuffix[w1, w2]` đếm số lượng các tiếp vĩ ngữ được theo sau tiếp đầu ngữ nào đó. Bản thân các tiếp vĩ ngữ được lưu trong các phần tử của mảng `statetab[w1,w2,1]`, `statetab[w1,w2,2]`, v.v...

Khi khối `END` được thực thi thì toàn bộ dữ liệu nhập đã được đọc xong. Tại thời điểm đó, với mỗi tiếp đầu ngữ, có một phần tử của mảng `nsuffix` chứa kết quả đếm số tiếp vĩ ngữ, và sẽ có bấy nhiêu phần tử của bảng `statetab` chứa các tiếp vĩ ngữ.

Phiên bản viết bằng Perl cũng tương tự, nhưng dùng một `mảng dấu tên` thay cho một chỉ số thứ ba để theo dõi số các tiếp đầu ngữ; cũng đồng thời dùng phép gán bội để cập nhật tiếp đầu ngữ. Perl dùng các ký tự đặc biệt để chỉ loại biến: ký tự `$` để chỉ một biến đơn và `@` chỉ một mảng có chỉ số, còn dấu ngoặc vuông `[]` được dùng để đánh chỉ số cho mảng và dấu ngoặc móc `{}` dùng để đánh chỉ số cho việc băm.

```
# mawkov.pl: giai thuật chuỗi markov cho các tiếp đầu
ngữ gồm hai từ

$MAXGEN = 10000;

$NONWORD = "\n";

$w1 = $w2 = $NONWORD;    # giai đoạn khởi tạo
```

```
while (<>) {              # đọc từng dòng dữ liệu nhập
        foreach (split) {
                push(@{$statetab{$w1}{$w2}}, $_);
                ($w1, $w2) = ($w2, $_); # phép gán bội
        }
}
push(@statetab{$w1}{$w2}), $NONWORD);      # thêm vào
cuối
$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
        $suf = $statetab{$w1}{$w2};# tham chiếu đến mảng
        $r = int(rand @suf);       # @suf là số lượng phần
tử
        exit if (($t = $suf->[$r]) eq $NONWORD);
        print "$t\n";
        ($w1, $w2) = ($w2, $t);
}
```

Như trong chương trình trước, các ánh xạ từ tiếp đầu ngữ đến các tiếp vĩ ngữ của nó được lưu trong biến `statetab`. Trung tâm của chương trình là dòng:

```
push(@$statetab{$w1}{$w2}}, $_);
```

có tác dụng đưa tiếp một tiếp vĩ ngữ mới vào sau mảng dấu tên lưu trong `statetab{$w1}{$w2}`. Lúc mới khởi tạo, `$statetab{$w1}{$w2}` là một tham chiếu đến một mảng các tiếp vĩ ngữ, và `$suf->[$r]` trỏ đến tiếp vĩ ngữ thứ `r`.

Cả chương trình viết bằng Perl và Awk đều ngắn hơn so với ba phiên bản trước, nhưng lại khó sửa đổi để có thể đáp ứng với các tiếp đầu ngữ có độ dài không đúng bằng hai từ. Phần lõi của trình C++ STL (các

hàm add và generate) cũng có độ dài tương đương mà lại rõ ràng hơn. Tuy nhiên, các ngôn ngữ dạng script thường là một lựa chọn tốt cho các công việc lập trình với mục đích thực nghiệm, lập prototype, và thậm chí cho mục đích tạo ra sản phẩm nếu như thời gian thi hành không phải là vấn đề lớn.

Bài tập 3-7. Hãy sửa đổi các phiên bản Awk và Perl để có thể đáp ứng với các tiếp đầu ngữ có độ dài bất kỳ. Làm thực nghiệm để xác định xem tác dụng trên tốc độ thực thi của sự thay đổi này đạt được đến mức độ nào.

3.8. Tốc độ thực thi

Đã có một vài chương trình trên một số ngôn ngữ khác nhau để so sánh. Hãy đo thời gian chạy của các chương trình với dữ liệu nhập là quyển sách của Psalms trong bản Kinh thánh của Vua James, có 42685 từ (5238 từ riêng biệt và 22482 tiếp đầu ngữ). Văn bản này có tương đối đầy đủ các cụm từ lặp đi lặp lại (như "Blessed is the...") để có một danh sách tiếp vĩ ngữ chứa đến trên 400 phần tử, và có vài trăm chuỗi có đến hàng chục tiếp vĩ ngữ, do vậy nó là tập dữ liệu thử nghiệm tốt.

```
Blessed is the man of the net. Turn thee unto me,
and raise me up, that I may tell all my fears.
They looked unto him, he heard. My praise shall be
blessed. Wealth and riches shall be saved. Thou
hast dealt well with thy hid treasue: they are
cast into a standing water, the flint into a
standing water, and dry ground into watersprings.
```

Thời gian trong bảng dưới đây là số giây để phát sinh 10000 từ ở kết quả đầu ra; một máy tính là máy 250 MHz MIPS R10000 chạy Irix 6.4 và máy kia là 400 MHz Pentium II có 128 MB bộ nhớ chạy Windows NT. Thời gian thực thi hầu như hoàn toàn được xác định bởi kích thước dữ liệu vào; theo bảng so sánh thì việc phát sinh là rất nhanh. Bảng cũng cho biết kích thước xấp xỉ của chương trình tính bằng số dòng mã nguồn.

	250 MHz R10000	400 MHz Pentium II	Số dòng mã nguồn
C	0.36 giây	0.30 giây	150
Java	4.9	9.2	105
C++/STL/deque	2.6	11.2	70
C++/STL/danhsách	1.7	1.5	70
Awk	2.2	2.1	20
Perl	1.8	1.0	18

Các phiên bản C và C++ được biên dịch bằng các trình biên dịch đã được tối ưu hóa, trong khi đó bản Java chạy trên các trình biên dịch được kích hoạt khi chạy. Thời gian chạy của các bản C và C++ trên máy Irix là nhanh nhất trong cả ba trình biên dịch khác nhau. Phiên bản C của chương trình chạy nhanh nhất với mức độ khác biệt lớn; phiên bản Perl xếp thứ nhì. Tuy nhiên, thời gian liệt kê trong bảng trên chỉ là một nét phác họa dựa trên kinh nghiệm của chúng tôi trên một số trình biên dịch và thư viện cụ thể, cho nên bản thân bạn sẽ nhận thấy những kết quả rất khác biệt trên môi trường của bạn.

Có thể thấy ngay một số sai khác đối với bản `STL deque` trên Windows. Các thử nghiệm cho thấy `deque` đại diện cho tiếp đầu ngữ chiếm phần lớn thời gian chạy của chương trình dù không bao giờ chứa quá hai phần tử: ta có thể cho rằng chính cấu trúc dữ liệu `map` đã có ảnh hưởng lớn. Chuyển `deque` thành danh sách liên kết (danh sách liên kết đôi trong STL) vấn đề thời gian được cải thiện rất rõ rệt. Đây là một bằng chứng về sự hợp lý trên nền tảng của thiết kế STL rằng những thay đổi đó chỉ cần thực hiện bằng cách thay từ `deque` bằng từ `list` hoặc từ `map` bằng từ `hash` ở mỗi vị trí rồi biên dịch lại. Ta kết luận rằng STL, một `component` mới của C++, vẫn chưa được cài đặt hoàn thiện. Tốc độ thực thi giữa cách biên dịch STL và các cấu trúc dữ liệu khác là không thể dự đoán được. Điều tương tự cũng đúng cho Java, trong đó từng cách biên dịch cũng thay đổi nhanh chóng.

Thử nghiệm một chương trình có tính mỗi lần chạy một khác là một thách thức thú vị. Làm sao ta có thể biết chương trình làm việc như thế nào? Làm sao ta có thể chắc chắn rằng chương trình sẽ luôn chạy khi cần? Chương 6 về kiểm chứng có đưa ra một số đề nghị và mô tả cách kiểm chứng các chương trình *Markov* như thế nào.

Chương 4

GIAO TIẾP

Trong quá trình thiết kế, điều quan trọng là phải biết cân đối giữa các mục tiêu cần đạt và các ràng buộc. Thường sẽ có nhiều lựa chọn thiết kế khác nhau, nhưng không phải thiết kế nào cũng cho ra kết quả tốt, do đó vấn đề quan tâm là làm sao để chọn được một thiết kế tối ưu. Đối với các hệ thống nhỏ độc lập, việc quyết định một thiết kế chỉ ảnh hưởng trong phạm vi của hệ thống và một ít lập trình viên. Tuy nhiên, đối với một hệ thống lớn có giao tiếp với các hệ thống khác, thì việc quyết định chọn một thiết kế không dễ dàng chút nào, vì một số dịch vụ của hệ thống này có thể được chia se với các hệ thống khác, và như thế phạm vi ảnh hưởng của thiết kế không chỉ đối với hệ thống này mà còn đối với các hệ thống giao tiếp với nó.

Các vấn đề được xem xét trong một thiết kế là:

- Giao tiếp: hệ thống sẽ cung cấp các thao tác hoặc chức năng nào? Phương thức giao tiếp là kết quả của sự thống nhất các yêu cầu bài toán. Một phương thức giao tiếp tốt sẽ cung cấp các thao tác tiện dụng, dễ dùng và vừa đủ; tránh cung cấp quá nhiều chức năng đến nỗi gây khó chịu.
- Che giấu thông tin: thông tin nào thì cho phép nhìn thấy và thông tin nào thì không? Phương thức giao tiếp một mặt phải hỗ trợ các chức năng cho phép truy xuất dễ dàng tới các thành phần bên trong, một mặt che giấu các chi tiết thực hiện bên trong các chức năng này; điều này cho phép bạn thay đổi cách thức thực hiện của chức năng mà không làm ảnh hưởng đến người dùng.

- Quản lý tài nguyên: ai sẽ chịu trách nhiệm quản lý bộ nhớ và các tài nguyên có hạn khác? Vấn đề chính ở đây là: cấp phát, giải phóng vùng lưu trữ, và quản lý các tài nguyên chia sẻ.
- Xử lý lỗi: ai sẽ chịu trách nhiệm phát hiện lỗi, thông báo lỗi và cách thức thực hiện ra sao? Khi một lỗi được phát hiện thì các thao tác xử lý nào được kích hoạt để sửa chữa?

Trong Chương 2 đã trình bày các cấu trúc dữ liệu mà hầu hết các hệ thống thường dùng. Chương 3 trình bày cách thức kết hợp chúng lại thành một chương trình nhỏ. Ở đây ta sẽ tìm hiểu các giao tiếp giữa các thành phần từ các nguồn khác nhau. Chương này sẽ minh họa việc thiết kế phương thức giao tiếp bằng cách xây dựng một thư viện các hàm và các cấu trúc dữ liệu cho các thao tác thường gặp. Qua đó sẽ trình bày một số nguyên tắc thiết kế. Thông thường việc thiết kế được thực hiện theo rất nhiều cách thức khác nhau, tuy nhiên phần lớn các cách hầu như được thực hiện một cách tự phát, không theo một nguyên tắc thiết kế rõ ràng. Do đó nếu không tuân thủ những nguyên tắc thiết kế này thì kết quả thường là các thao tác của phương thức giao tiếp không tương thích; và đó chính là nguyên nhân dẫn đến các thất bại và cản trở thường thấy trong lập trình.

4.1. Giá trị được phân cách bằng dấu phẩy – CSV (Comma-Separated Values)

Giá trị được phân cách bằng dấu phẩy, hay CSV, là thuật ngữ được dùng một cách tự nhiên và phổ biến để biểu diễn bảng dữ liệu. Mỗi dòng của bảng dữ liệu là một dòng văn bản, trong đó các trường trên mỗi dòng được phân cách nhau bởi các dấu phẩy. Bảng dữ liệu ở cuối chương trước có thể được biểu diễn theo định dạng của CSV như sau:

```
,"250MHz","400MHz","Lines of"
,"R10000","Pentium II","source code"
C, 0.36 sec, 0.30 sec, 150
Java, 4.9, 9.2, 105
```

Định dạng này được đọc và ghi bởi những chương trình chẳng hạn như các chương trình bảng tính. Thỉnh thoảng định dạng này cũng được dùng trong các trang web phục vụ cho các dịch vụ như bảng báo giá. Một trang web thể hiện tỷ giá thường được thể hiện như sau:

Symbol	Last Trade		Change		Volume
LU	2:19 PM	86-1/4	+4-1/16	+4.94%	5,804,800
T	2:19 PM	60-11/16	-1-3/16	-1.92%	2,468,000
MSFT	2:24 PM	106-9/16	+1-3/8	+1.31%	11,474,900

Download với định dạng bảng tính

Sử dụng trình duyệt web để truy xuất đến những con số thì hiệu quả nhưng tốn nhiều thời gian. Rất phiền toái khi gọi trình duyệt, chờ, xem hàng loạt các quảng cáo thay phiên nhau trình diễn; nhập vào một danh dách các ô trống, chờ, và lại chờ, rồi lại xem hàng loạt các quảng cáo khác, và kết quả quan tâm chỉ là một ít số liệu. Dó là chưa kể đến việc yêu cầu trình duyệt xử lý các con số một cách chi tiết hơn, thì đòi hỏi phải có nhiều tương tác hơn nữa; chọn liên kết *"Download với định dạng bảng tính"* để lấy một tập tin chứa các thông tin theo dạng thức CSV, tương tự như sau (đã được sửa lại cho vừa trang):

```
"LU", 86.25, "11/4/1998","2:19PM",+4.0625,
                83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
                62.375, 62.625,60.4375,24680000
"MSFT",1065625, "11/4/1998","2:24PM",+1.1875
                105.8125,107.3125,105.5625,11474900
```

Trước một sự thể hiện không hiệu quả như thế, ta đưa ra một giải pháp là để cho máy tính thực hiện một phần công việc. Các trình duyệt cho

phép máy tính truy xuất dữ liệu trên máy chủ ở xa, và việc truy xuất dữ liệu sẽ thuận lợi hơn nếu ta hạn chế tương tác với trình duyệt. Để làm được điều này, chúng ta viết một thủ tục thuần xử lý văn bản bên dưới thao tác nhấn nút – Trình duyệt đọc thông tin bạn nhập vào và mã hoá thành dạng HTML, rồi gửi thông tin bạn nhập tới máy chủ và đọc thông tin dạng HTML từ máy chủ gửi về. Bằng cách sử dụng các công cụ và ngôn ngữ phù hợp, ta dễ dàng lấy được thông tin một cách tự động. Sau đây là một chương trình viết bằng ngôn ngữ Tcl dùng để truy xuất trang web có dạng như trên và lấy dữ liệu CSV có định dạng như trên, và đã lược bớt một số dòng `header` trước nó:

```
        # getquotes.tcl: lấy bảng báo giá của Lucent, AT&T,
Microsoft

        set so [socket quote.yahoo.com 80]  ;# kết nối đến
server

        set q "/d/quote.csv?s=LU+T+MSFT&f: sl1d1t1c1ohgv"

        puts $so "GET $q HTTP/1.0\r\n\n" ;# gửi yêu cầu đến
server

        flush $so

        puts [read $so] ;# đọc và in thông tin gửi về từ server
```

Chuỗi `f =...` nằm sau dấu & có vẻ khó hiểu, nó là chuỗi điều khiển phi văn bản, tương tự như đối số đầu tiên của hàm `printf`, nó xác định các giá trị được lấy về. Ở đây, chúng ta quy định *s* đại diện cho cột *symbol*, `l1` cho cột giá mới nhất (cột *last price*), `c1` cho cột thay đổi từ hôm qua (cột *change*), và v.v... Điều quan trọng ở đây không phải là chi tiết các cột như thế nào mà là khả năng tự động hoá: lấy thông tin mong muốn và chuyển nó sang dạng ta cần một cách tự động. Máy tính sẽ thực hiện công việc này cho chúng ta.

Thông thường chỉ mất vài giây để chạy chương trình `getqoutes`, nhanh hơn nhiều so với tương tác với trình duyệt. Khi có dữ liệu, ta có thể xử lý chi tiết hơn. Dữ liệu dạng CSV sẽ hữu dụng nếu có các thư viện dễ dùng cho phép chuyển đổi từ dạng CSV sang dạng ta cần, và ngược lại; điều

này có lẽ liên quan đến một số xử lý đơn giản chẳng hạn như các chuyển đổi số. Tuy nhiên, chúng ta không có trong tay một thư viện phổ biến viết sẵn cho CSV, do đó ta sẽ viết riêng một thư viện để sử dụng.

Trong các phần tiếp theo, ta sẽ xây dựng ba phiên bản của một thư viện dùng để đọc và chuyển dữ liệu dạng CSV sang dạng ta cần. Qua đó, chúng ta sẽ thảo luận các vấn đề phát sinh khi thiết kế một phần mềm để dùng chung với các phần mềm khác.

4.2. Một thư viện mẫu

Ta sẽ không thể có được bản thiết kế hoàn chỉnh cho một thư viện hoặc phương thức giao tiếp trong lần thiết kế đầu tiên. Như Fred Brooks đã từng viết: "hãy khởi đầu bằng việc đơn giản, bạn sẽ làm được việc lớn". Brooks đã làm như thế trên các hệ thống lớn, nhưng ý tưởng thì thích hợp cho bất cứ một thành phần quan trọng nào của phần mềm. Thông thường bạn sẽ có được bản thiết kế hoàn chỉnh khi bạn hiểu rõ các vấn đề của chương trình và sử dụng một phiên bản của nó.

Trên tinh thần này, chúng ta sẽ tiếp cận cấu trúc của một thư viện cho CSV bằng cách xây dựng một phiên bản, rồi phát triển lên các phiên bản tốt hơn. Phiên bản đầu tiên sẽ bỏ qua các trường hợp khó, tuy nhiên nó cũng chứng tỏ được sự hữu ích và cho ta kiến thức để hiểu được vấn đề.

Chúng ta bắt đầu với hàm `csvgetline`, hàm này có chức năng đọc một dòng của dữ liệu dạng CSV từ một tập tin và lưu vào bộ nhớ đệm; tiếp đến chia dòng này thành nhiều trường và loại bỏ các dấu nháy kép bao quanh mỗi trường, và lưu chúng vào một mảng, và trả về số lượng các trường. Sau đây là một phiên bản mẫu trong C; chúng ta đánh các dấu hỏi (?) để cho biết đây chỉ là một dạng mẫu:

```
?       char buf[2^ ;// vùng nhớ chứa dữ liệu nhập */
?       char *field[ ;j    /* các trường */
?
?       /* Hàm csvg tline: đọc và tính dữ liệu nhập thành
```

```
các trường,
      ?                          trả về số trường tách được */
      ?     /*  dữ  liệu  nhập:  "LU",  86.25,"11/4/1998",
"2:19PM", 14.0624 */
      ?     int csvgetline(FILE *fin)
      ?     {
      ?           int nfield;
      ?           char *p, *q;
      ?
      ?           if (fgets(buf, sizeof(buf), fin) == NULL)
      ?               return -1;
      ?           nfield = 0;
      ?           for  (q  =  buf;(p strtok(q,  ",\n\r"))  !=
NULL; q = NULL)
      ?                field[nfield++] = unquote(p);
      ?           return nfield;
      ?     }
```

Dòng ghi chú ở ngay trên hàm là một ví dụ của dạng dữ liệu nhập mà chương trình chấp nhận; các ghi chú như thế sẽ có ích cho lập trình viên trong việc phân tích dữ liệu nhập không theo một định dạng khác.

Định dạng CSV quá phức tạp để phân tích được bằng lệnh `scanf`, do đó ta dùng hàm `strtok` trong thư viện chuẩn của ngôn ngữ C. Mỗi lần gọi `strtok(p,s)` sẽ trả về con trỏ đến `token` đầu tiên trong `p` gồm các ký tự không có trong `s`; hàm `strtok` ngắt một `token` bằng cách thay thế ký tự tiếp theo của chuỗi ban đầu bằng một byte có giá trị `NULL`. Trong lần gọi đầu tiên, đối số đầu tiên của hàm `strtok` là chuỗi ban đầu; các lần gọi tiếp theo dùng giá trị `NULL` để chỉ ra lần quét tiếp theo nên bắt đầu ở vị trí nó ngừng lại trong lần gọi trước. Đây không phải là một giao tiếp tốt. Vì hàm `strtok`

lưu trữ một biến ở một nơi mà ta không kiểm soát được giữa các lần gọi, chỉ biết được duy nhất trình tự của các lần gọi được kích hoạt tại mỗi thời điểm; các lần gọi không liên quan với nhau được chèn vào sẽ gây cản trở nhau.

Hàm `unqoute` sau đây có chức năng loại bỏ dấu nháy kép ở đầu và cuối một chuỗi. Nó không xử lý các dấu nháy kép xuất hiện giữa chuỗi, bởi vì đây chỉ là một bản mẫu, không xét đến trường hợp tổng quát.

```
?     /* Hàm unquote: loại bỏ dấu nháy kép bao quanh
một chuỗi */
?     char *unquote(char *p)
?     {
?           if (p[0] == '"') {
?                 if (p[strlen(p)-1] == '"')
?                     p[strlen(p)-1] = '\0';
?                 p++;
?           }
?           return p;
?     }
```

Sau đây là một chương trình đơn giản dùng để kiểm tra sự thực hiện của hàm `csvgetline`:

```
?     int main(void)
?     {
?           int i; nf;
?
?           while ((nf = csvgetline(stdin)) != -1)
?                 for (i = 0; i < nf; i++)
```

```
?           printf("field[%d] = '%s'\n", i, field[i]);
?       return 0;
?       }
```

Lệnh `printf` xuất ra các trường trên các dòng riêng biệt và đóng mỗi trường trong dấu nháy đơn nhằm chỉ ra các lỗi do xử lý khoảng trắng không chính xác.

Bây giờ ta có thể chạy chương trình này với dữ liệu nhập là kết quả xuất ra của tập tin `getquotes.tcl`. Kết quả như sau:

% getquotes.tcl | csvtest

```
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7]   '85.0625'
field[8] = '2886600'
field[0] = 'T'
field[1] = '61.0625'
...
```

(Các dòng tiêu đề *HTTP* đã được xóa)

Bây giờ ta đã có được một bản mẫu dường như là chạy đúng trên dữ liệu có định dạng như trên. Có lẽ chúng ta nên thận trọng kiểm tra xem nó có còn đúng trên các bộ số liệu khác hay không; điều này là cần thiết đặc biệt là khi ta dự định cho những người khác sử dụng nó. Chúng ta tìm thấy một trang web khác mà ta có thể tải xuống và ghi vào một tập tin với các thông tin tương tự nhưng được định dạng khác: dùng các ký tự (\r) để phân cách các mẫu tin thay vì dùng xuống dòng và không dùng ký tự (\n) ở cuối tập tin. Ta đã hiệu chỉnh và định dạng để cho vừa với trang giấy:

```
"Ticket","Price","Chang","Open","Prev Close",
"Day High","Day Low","52 Week High",
"52 Week Low","Dividend","Yield","Volume",
"Average Volume","P/E"
      "LU",86.313,-
0.188,86.000,86.500,86.438,85.063,108.50,
          36.18,0.16,0.1,2946700,9675000,N/A
 "T",61.125,0.938,60.375,60.188,61.125,60.000,68.50,
          46.50,1.32,2.1,3061000,4777000,17.0
  "MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
          119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

Với dữ liệu nhập này, phiên bản mẫu của chúng ta gây ra lỗi nghiêm trọng.

Ta đã thiết kế một phiên bản mẫu và đã thử nghiệm nó trên một nguồn dữ liệu, và chúng ta cũng chỉ kiểm tra nó trên các dữ liệu của cùng một nguồn. Vì thế ta không ngạc nhiên khi trong lần thử đầu tiên trên một nguồn dữ liệu khác đã bị quá nhiều lỗi. Những dòng dữ liệu nhập dài, số lượng trường nhiều, và những dấu phân cách thừa hoặc thiếu đều gây ra lỗi.

Bản mẫu đơn giản này có thể dùng để phục vụ cho nhu cầu cá nhân hoặc để chứng minh cho tính khả thi của một hướng tiếp cận, chỉ thế thôi. Đã đến lúc chúng ta suy nghĩ lại cách thiết kế trước khi ta thực hiện cài đặt theo hướng khác.

Ta có thể có rất nhiều hướng lựa chọn cho một bản mẫu khác một cách hiển nhiên hay ngầm định. Sau đây là một số hướng chọn lựa, và không phải lúc nào nó cũng là cách tốt nhất cho một thư viện dùng chung. Mỗi hướng phát sinh một vấn đề cần quan tâm.

- Một phiên bản không xử lý những dòng dữ liệu nhập dài hoặc số trường nhiều. Nó có thể cho kết quả sai hoặc phát sinh lỗi bởi vì nó không kiểm tra giới hạn của dữ liệu nhập, nên rất khó có thể trả về dữ liệu có nghĩa trong các trường hợp có lỗi.
- Dữ liệu nhập được ngầm định chỉ chứa các ký tự xuống dòng ở cuối mỗi dòng.
- Các trường được phân cách bởi các dấu phẩy và các dấu nháy kép bao quanh nó được loại bỏ. Không chấp nhận các dấu phẩy hay dấu nháy kép là thành phần của trường.
- Dữ liệu nhập bị thay đổi; nó bị ghi đè lên trong quá trình tạo các trường.
- Không có dữ liệu nào được lưu giữa hai lần nhập liệu; nếu có thông tin gì cần ghi nhớ thì một bản sao sẽ được tạo ra.
- Truy xuất đến các trường thông qua một biến toàn cục, đó là tên mảng các trường được dùng bởi hàm `csvgetline` và những hàm gọi nó; không có sự kiểm soát các truy xuất tới nội dung của trường hay tới các con trỏ. Cũng không chặn các truy xuất ngoài giới hạn của mảng.
- Các biến toàn cục làm cho bản thiết kế không phù hợp trong môi trường đa tiến trình hay thậm chí cũng không phù hợp khi chỉ hoán đổi thứ tự giữa các lần gọi hàm.

- Lời gọi hàm phải mở và đóng các tập tin một cách hợp lệ; hàm `csvgetline` chỉ đọc từ các tập tin đã được mở.
- Việc nhập liệu và việc phân tích dữ liệu nhập không có sự liên hệ chặt chẽ; với mỗi lần gọi, hàm thực hiện đọc một dòng dữ liệu và cắt nó ra thành các trường, không quan tâm đến việc khác trong chương trình.
- Giá trị trả về là số lượng các trường trên mỗi dòng; mỗi dòng phải được cắt ra để tính giá trị này. Cũng không có cách nào phân biệt được lỗi do kết thúc tập tin và các lỗi khác.
- Không có cách nào để thay đổi bất kỳ một thuộc tính nào trong các thuộc tính này mà không làm ảnh hưởng đến mã nguồn.

Danh sách dài chưa đầy đủ này minh họa cho một số sự lựa chọn có thể có cho bản thiết kế. Mỗi cách lựa chọn sẽ có một cách cài đặt mã nguồn tương ứng. Điều đó có thể thực hiện được một cách nhanh chóng, cũng giống như việc phân tích một định dạng cố định từ một nguồn đã biết. Nhưng điều gì sẽ xảy ra nếu như định dạng thay đổi, hoặc trường có chứa dấu phẩy hoặc máy chủ tạo ra dòng dữ liệu dài hoặc nhiều trường?

Dường như vấn đề có thể được giải quyết một cách dễ dàng, vì bài toán thì nhỏ và chỉ cần một bản mẫu. Nhưng hãy tưởng tượng xem sau vài năm hoặc vài tháng ngồi viết phần mềm, mã nguồn trở thành một phần của một chương trình lớn hơn và các đặc tả cho chương trình thay đổi theo thời gian. Liệu hàm `csvgetline` có còn phù hợp nữa không? Nếu chương trình được dùng bởi những người khác thì sự chọn lựa nhanh chóng của bản thiết kế ban đầu có thể là nguyên nhân gây ra rắc rối sau một thời gian sử dụng. Đây là một đặc trưng lịch sử của các giao tiếp tồi. Nhiều phần mã nguồn được viết vội vàng và cẩu thả đã không tồn tại được trong một phần mềm được phổ biến rộng rãi; nó kết thúc nhanh chóng cũng như khi nó được tạo ra.

4.3. Thư viện dùng chung

Từ những điều học được từ bản mẫu trên, bây giờ chúng ta muốn xây dựng một thư viện có giá trị cho mục đích dùng chung. Yêu cầu rõ ràng nhất là chúng ta phải làm cho hàm `csvgetline` mạnh mẽ hơn; nó sẽ xử lý được các dòng nhập liệu dài hoặc số trường nhiều, và cũng phải cẩn thận hơn trong việc phân tích các trường.

Để tạo một phương thức giao tiếp cho những người khác có thể sử dụng, ta phải cân nhắc các vấn đề được liệt kê ở đầu của chương này: phương thức giao tiếp, che giấu thông tin, quản lý tài nguyên, và xử lý lỗi. Mối tương quan giữa chúng có tác động mạnh đến việc thiết kế. Việc phân chia các vấn đề này hơi có tính chủ quan, bởi vì chúng có liên hệ mật thiết với nhau.

Phương thức giao tiếp.

Chúng ta quyết định dựa trên ba thao tác cơ bản sau:

```
char *csvgetline(FILE *): đọc một dòng CSV mới.

char *csvfield(int n):    trả về trường thứ n của
dòng hiện hành

int csvnfield(void):      trả về số trường trong
dòng hiện hành
```

Hàm `csvgetline` nên trả về giá trị nào? Điều được mong đợi là trả về càng nhiều thông tin hữu ích càng tốt. Trong bản mẫu trên, giá trị trả về là số lượng các trường; tuy nhiên số trường luôn được tính thậm chí các trường không được sử dụng. Một giá trị khả thi khác là độ dài của dòng dữ liệu nhập, vì nó có ảnh hưởng việc giữ lại ký tự sang dòng mới ở cuối. Sau khi xem xét nhiều ứng dụng, ta quyết định hàm `csvgetline` sẽ trả về một con trỏ tới dòng dữ liệu nhập ban đầu, hoặc `NULL` nếu đến cuối tập tin.

Chúng ta sẽ bỏ ký tự sang dòng mới ở cuối dòng trả về bởi hàm `csvgetline`, vì nó có thể được phục hồi dễ dàng khi cần.

Định nghĩa của trường không còn đơn giản nữa: ta cố gắng tạo định nghĩa phù hợp với những quan sát thực tế qua các bảng tính và các chương trình khác. Một trường là một chuỗi ký tự có thể có không ký tự hoặc nhiều ký tự. Các trường ngăn cách nhau bởi dấu phẩy. Các khoảng trắng ở đầu và cuối trường được giữ lại. Một trường có thể chứa các dấu nháy kép ở cả hai đầu, hoặc nó có thể chứa dấu phẩy. Một trường trong dấu nháy có thể chứa các dấu nháy kép, nó được thể hiện bởi một dấu nháy kép đôi: trường CSV `"x""y"` định nghĩa chuỗi `x"y`. Các trường có thể rỗng; trường dạng `""` là rỗng, được đồng nhất với trường trong hai dấu phẩy kề nhau.

Các trường được đánh số từ không. Nếu như người dùng yêu cầu một trường không tồn tại thì sao, chẳng hạn như gọi `csvfield(-1)` hay `csvfield(100000)`? Khi ấy, ta có thể trả về `""` (chuỗi rỗng), vì nó có thể được in ra hay so sánh; các chương trình xử lý liên quan đến nhiều chỉ số trường khác nhau không phải bận tâm nhiều đến các trường không tồn tại. Tuy nhiên cách này không phân biệt được giữa trường rỗng và trường không tồn tại. Cách thứ hai là in ra một thông báo lỗi hoặc thậm chí kết thúc sớm chương trình; chúng ta sẽ thảo luận vì sao cách này không được ưa chuộng. Ta quyết định chọn cách trả về `NULL` khi một trường không tồn tại, giá trị mà lâu nay C vẫn thường dùng để thể hiện chuỗi không tồn tại.

Che giấu thông tin

Thư viện sẽ không giới hạn chiều dài của dòng dữ liệu nhập hay số trường. Để đạt được điều này, hoặc hàm gọi thực hiện phải cung cấp bộ nhớ hoặc hàm được gọi thực hiện phải cấp phát bộ nhớ. Hàm gọi thực hiện của thư viện, `fgets` phải cung cấp một mảng có kích thước tối đa được xác định trước. Nếu dòng dữ liệu nhập dài hơn vùng nhớ chứa dữ liệu nhập thì dòng đó sẽ được cắt ra thành nhiều phần. Cách xử lý này không phù hợp với cách giao tiếp của CSV, do đó thư viện của chúng ta sẽ cấp phát bộ nhớ và sẽ tăng kích thước khi cần thiết.

Vì thế chỉ có hàm `csvgetline` biết cách quản lý bộ nhớ; không cho phép bên ngoài truy cập vào cách thức tổ chức bộ nhớ của nó. Cách hỗ trợ tốt nhất cho việc quản lý cục bộ là thông qua hàm `csvgetline` đọc dòng kế

tiếp, không quan tâm độ lớn của dòng, `csvfield(n)` trả về một con trỏ tới các byte của trường thứ `n` của dòng hiện hành, và `csvnfield(n)` trả về số trường của dòng hiện hành.

Chúng ta sẽ cấp phát thêm bộ nhớ khi có các dòng dài hơn hoặc số trường nhiều hơn. Chi tiết về cách thức thực hiện được che giấu bên trong các hàm `csv`; không một bộ phận khác nào của chương trình biết về cách thức thực hiện, chẳng hạn không thể biết được thư viện sử dụng các mảng có kích thước nhỏ và phát triển thêm khi cần, hay là các mảng có kích thước khổng lồ, hay là một cách nào đó hoàn toàn khác.

Nếu người dùng chỉ gọi hàm `csvgetline`, thì không cần phải cắt dòng ra thành các trường; các dòng có thể được cắt ra khi cần. Việc tách các trường được thực hiện ngay sau khi đọc dòng nhập liệu, hoặc chỉ thực hiện khi nào cần đến một trường hay số trường, hoặc chỉ tách ra trường khi được yêu cầu, đều là các chi tiết cài đặt bên trong che giấu người dùng.

Quản lý tài nguyên

Chúng ta phải quyết định xem ai là người chịu trách nhiệm đối với thông tin chia sẻ. Hàm `csvgetline` nên trả về dữ liệu ban đầu hay tạo một bản sao? Ta quyết định hàm `csvgetline` trả về một con trỏ, trỏ tới dữ liệu nhập ban đầu, nó sẽ được ghi đè lên khi dòng mới được đọc vào. Các trường sẽ được xây dựng lồng vào một bản sao của dòng dữ liệu nhập, và hàm `csvfield` sẽ trả về một con trỏ, trỏ tới một trường trong bản sao. Với cách tổ chức này, người dùng phải tạo ra một bản sao nữa nếu một dòng hoặc một trường nào đó được lưu lại hoặc thay đổi, và trách nhiệm của người dùng là giải phóng vùng lưu trữ khi không còn dùng đến nữa.

Ai sẽ mở và đóng tập tin nhập liệu? Người mở tập tin nhập liệu cũng nên là người đóng nó lại; các tác vụ đi đôi nên được thực hiện ở cùng mức hay cùng nơi. Chúng ta giả định rằng hàm `csvgetline` được gọi với một con trỏ `FILE`, nó trỏ tới một tập tin đã được mở; và đối tượng gọi nó sẽ đóng tập tin lại khi xử lý xong. Quản lý các tài nguyên chia sẻ hay các tài nguyên được chuyển qua nơi giao tiếp giữa một thư viện và các đối tượng gọi nó là

một tác vụ khó khăn, và thường có các lý do thuyết phục nhưng mâu thuẫn nhau để có nhiều cách thiết kế khác nhau. Các lỗi và việc hiểu nhầm về các trách nhiệm chia sẻ thường là nguyên nhân gây ra lỗi.

Xử lý lỗi

Vì hàm `csvgetline` trả về `NULL`, nên không có cách nào để phân biệt việc kết thúc tập tin khi bị xảy ra một lỗi khác, chẳng hạn như tràn bộ nhớ; tương tự thế, việc truy xuất tới một trường không tồn tại sẽ không gây ra lỗi. Ta có thể thêm hàm `csvgeterror` (tương tự như hàm `ferror`) vào phần phương thức giao tiếp để biết được loại lỗi, tuy nhiên để đơn giản ta sẽ bỏ qua nó trong phiên bản này.

Như một nguyên lý, các hàm của thư viện không nên chỉ thuần túy sẽ kết thúc khi xảy ra lỗi, mà phải trả về trạng thái lỗi cho hàm gọi thực hiện để nó có những hành động tương ứng. Cũng không nên in ra các thông báo hay các hộp thoại, vì chúng có thể đang chạy trong môi trường mà ở đó các thông báo có thể gây cản trở các thao tác khác. Việc xử lý lỗi là một vấn đề đáng được trình bày riêng, ta sẽ trình bày lại nó kỹ hơn ở phần sau trong chương này.

Đặc tả

Các cách lựa chọn ở trên nên được gom vào một nơi như là đặc tả các thao tác mà hàm `csvgetline` hỗ trợ và cách sử dụng nó. Trong một đề án lớn, phần đặc tả được thực hiện trước phần cài đặt, vì người đặc tả và người cài đặt thường là những người khác nhau và có thể có các cách tổ chức khác nhau. Tuy nhiên trong thực tế, chúng thường được tiến hành song song với các đặt tả và cài đặt có tác động qua lại.

Cách tiếp cận tốt nhất là viết các đặc tả trước và chỉnh sửa lại nó trong quá trình cài đặt. Đặc tả càng cẩn thận và chính xác bao nhiêu thì kết quả chương trình có khuynh hướng càng tốt bấy nhiêu. Ngay cả các chương trình cá nhân, cũng rất cần được đặc tả một cách đầy đủ và hợp lý, vì nó cho ta cơ hội để cân nhắc giữa các lựa chọn và ghi nhận các chọn lựa của chúng ta.

Với mục đích như thế, đặc tả sẽ gồm các *prototype* của hàm và liệt kê chi tiết các thao tác, và điều kiện thực hiện:

Các trường phân cách nhau bởi dấu phẩy.

Một trường có thể được bao bọc bởi các dấy nháy kép "...".

Một trường trong dấu nháy có thể chứa dấu phẩy nhưng không được chứa các ký tự sang dòng.

Một trường trong dấu nháy có thể chứa các dấu nháy kép ", dạng "".

Các trường có thể rỗng; "" và một chuỗi rỗng đều được thể hiện bởi một trường rỗng.

Các khoảng trắng ở đầu và cuối được giữ lại.

```
char *csvgetline(FILE *f);
```

Đọc một dòng dữ liệu từ một tập tin f đang mở;

giả sử rằng các dòng dữ liệu được kết thúc bởi \r,\n,\r\n, hay EOF.

Trả về con trỏ tới dòng, khi đã bỏ ký tự kết thúc dòng, hoặc NULL nếu đến EOF.

Dòng có chiều dài tùy ý; trả về NULL nếu không đủ bộ nhớ.

Chỉ được đọc từ dòng; không được sửa đổi dòng;

hàm gọi phải tạo ra một bản sao để lưu lại hoặc thay đổi nội dung.

```
char *csvfield(int n)
```

Các trường được đánh số bắt đầu từ 0.

Trả về trường thứ *n* của dòng mới nhất đọc được từ hàm csvgetline;

Trả về NULL nếu n < 0 hoặc ra ngoài phạm vi.

Các trường cách nhau bởi dấu phẩy.

Các trường có thể được bao bọc bởi "..."; các dấu

nháy này được bỏ.

giữa "...", "" được thay thế bởi " và dấu phẩy không phải là dấu phân cách.

Trong các trường không được bao bọc bởi dấu nháy, các dấu nháy là các ký tự thường.

Số trường và chiều dài trường có thể không giới hạn;

trả về NULL nếu không đủ bộ nhớ.

Chỉ được đọc trường; không được sửa đổi trường;

hàm gọi phải tạo ra một bản sao để lưu hoặc thay đổi nội dung.

Hàm này sẽ không xác định nếu nó được gọi trước khi hàm csvgetline được gọi.

```
int csvnfield(void)
```

Trả về số trường của dòng vừa mới đọc bởi hàm csvgetline.

Hàm này sẽ không xác định nếu nó được gọi trước khi hàm csvgetline được gọi.

Các đặc tả này vẫn chưa đầy đủ. Chẳng hạn như, các giá trị nào nên được trả về bởi hàm csvfield và csvnfield nếu chúng được gọi sau khi csvgetline đến EOF? Các trường không theo đúng định dạng nên được xử lý như thế nào? Việc xem xét hết tất cả các rắc rối này không dễ dàng thậm chí đối với một hệ thống nhỏ, và càng phức tạp trong một hệ thống lớn. Thường ta sẽ không phát hiện được những thiếu xót và các vấn đề phát sinh cho đến khi thực hiện cài đặt.

Phần còn lại của phần này bao gồm một bản cài đặt mới của hàm `csvgetline` phù hợp với đặc tả. Thư viện được chia làm hai tập tin, một tập tin `csv.h` chứa các khai báo hàm mô tả phương thức giao tiếp, và một tập tin cài đặt `csv.c` chứa mã nguồn cài đặt phương thức giao tiếp.

Sau đây là tập tin `header`:

```
/* csv.h: header file của thư viện csv */

extern char *csvgetline(FILE *f);    /* đọc dòng
nhập liệu kế tiếp */

extern char *csvfield(int n);        /* trả về
trường thứ n */

extern int csvnfield(void);          /* trả về số
trường */
```

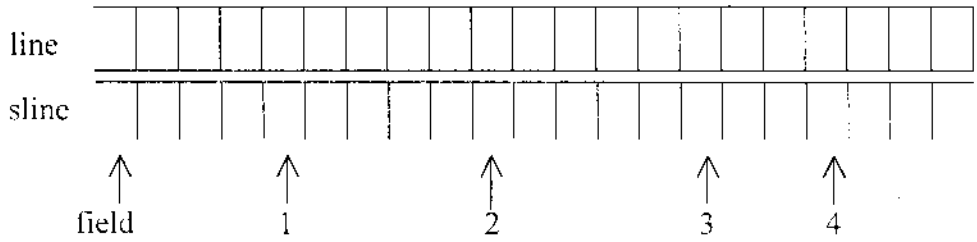
Các biến bên trong nội dung để lưu trữ văn bản và các hàm bên trong như hàm `split` được khai báo tĩnh, nên chúng chỉ được truy xuất trong phạm vi của tập tin chứa chúng. Đây là cách đơn giản nhất để che giấu thông tin trong một chương trình C.

```
enum { NOMEM = -2}

static char *line    = NULL;  /* dòng dữ liệu nhập */

static char *sline   = NULL;  /* dòng được tạo ra để
tách */

static int  maxline  = 0;     /* số dòng tối đa */

static char **field  = NULL;  /* con trỏ đến trường */

static int  maxfield = 0;     /* kích thước mảng chứa
trường */

static int  nfield   = 0;     /* số trường có trong
mảng */
```

```
static char fieldsep[] = ","; /* ký tự phân cách trường
*/
```

Các biến tĩnh cũng được khởi tạo. Các giá trị khởi tạo được dùng để kiểm tra xem nên tạo mới hay mở rộng thêm các mảng.

Các khai báo này mô tả một cấu trúc dữ liệu đơn giản. Mảng `line` lưu dòng dữ liệu nhập; mảng `sline` được tạo ra bằng cách sao chép các ký tự từ mảng `line` và đánh dấu kết thúc mỗi trường. Mảng các con trỏ `field` trỏ tới các bắt đầu của trường trong mảng `sline`. Sơ đồ sau đây cho thấy trạng thái của ba mảng này với dòng nhập liệu `ab,"cd","e""f",,"g,h"` được xử lý. Các phần tử mờ `(shaded)` trong mảng `sline` không phải là thành phần của trường.



Sau đây là hàm `csvgetline`:

```
/* Hàm csvgetline: đọc vào một dòng có chiều dài tùy ý
*/

/* Dữ liệu thử: "LU", 86.25, "11/4/1998", "2:19PM",
+4.0625 */

      char *csvgetline(FILE *fin)

      {

            int i, c;

            char *newl, *news;
```

```
        if (line == NULL) { /* cấp phát cho lần gọi đầu
tiên */
            maxline = maxfield = 1;
            line    (char *) malloc(maxline);
            sline = (char *) malloc(maxline);
            field = (char **)
malloc(maxfield*sizeof(field[0]));
            if (line == NULL || sline == NULL || field
== NULL) {
                reset();
                return NULL; /* out of memory */
            }
        }
        for(i=0;(c=getc(fin))!=EOF &&
!endofline(fin,c); i++) {
            if (i >= maxline-1) {
                maxline *= 2;
                newl    =    (char    *)    realloc(line,
maxline);
                news    =    (char    *)    realloc(sline,
maxline);
                if (newl == NULL || news == NULL) {
                    reset();
                    return NULL; /* out of memory
*/
                }
```

```
            line = newl;
            sline = news;
        }
        line[i] = c;
    }
    line[i] = '\0';
    if (split() == NOMEM) {
        reset();
        return NULL; /* out of memory */
    }
    return (c == EOF && i    0) ? NULL : line;
}
```

Các dòng đọc được từ tập tin được đưa vào mảng `line`; nó sẽ được mở rộng thêm khi cần thiết bằng cách gọi hàm `realloc`; kích thước tăng gấp đôi sau mỗi lần mở rộng thêm, như trong phần 2.6. Kích thước của mảng `sline` được duy trì bằng với kích thước mảng `line`; hàm `csvgetline` gọi hàm `split` để tạo ra các con trỏ trường trong một mảng `field` riêng biệt; mảng này cũng có thể được mở rộng khi cần thiết.

Để thử nghiệm, ta sẽ khởi tạo các mảng với kích thước rất nhỏ và phát triển chúng theo nhu cầu, nhằm bảo đảm rằng mã nguồn mở rộng mảng được thi hành nhiều lần. Nếu cấp phát không thành công, ta gọi hàm `reset` để phục hồi các biến toàn cục về trạng thái khởi dầu, nhờ đó các lời gọi hàm `csvgetline` tiếp theo có cơ hội thành công:

```
/* Hàm reset: gán các biến toàn cục về trạng thái khởi
đầu */
```

```
static void reset (void)
{
      free(line);   /* free(NULL) được phép dùng
trong C*/
      free(sline);
      free(field);
      line = NULL;
      sline = NULL;
      field = NULL;
      maxline = maxfield = nfield = 0;
}
```

Hàm `endofline` xử lý trường hợp dòng dữ liệu nhập kết thúc bằng một ký tự `'\r'`, một ký tự sang dòng, hoặc cả hai, hay thậm chí `EOF`.

```
static int endofline(FILE *fin, int c)
{
      int eol;
      eol = (c =='\r' || c =='\n');
      if (c == '\r' ) {
            c = getc(fin);
            if (c != '\n' && c != EOF)
               ungetc(c, fin);
      }
      return eol;
}
```

Các xử lý được tách ra là cần thiết, bởi vì các hàm nhập chuẩn không xử lý cho rất nhiều định dạng không hợp lý trong các dữ liệu nhập thực tế.

Prototype của chúng ta dùng hàm `strtok` để xác định `token` kế tiếp bằng cách tìm kiếm một ký tự ngăn cách, thường là dấu phẩy, tuy nhiên việc xử lý dấu phẩy được đóng trong dấu nháy là không khả thi. Cần phải có một thay đổi lớn trong cài đặt hàm `split`, mặc dù phương thức giao tiếp của nó không cần thay đổi. Hãy xem các dòng nhập liệu sau:

```
" ", ," "
,"  ",
,,
```

Mỗi dòng có ba trường rỗng. Nhưng hàm `split` phân tích chúng và các dữ liệu nhập lạ khác một cách đúng đắn và rất phức tạp; sau đây là một ví dụ cho thấy các trường hợp đặc biệt và các điều kiện biên có thể chiếm một phần quan trọng trong chương trình như thế nào.

```
/ * Hàm split : chuyển dòng thành trường */
static int split(void)
{
      char *p, **newf;
      char *sepp; /* trỏ đến ký tự phân cách tạm
thời */
      int sepc;   /* ký tự phân cách tạm thời */
      nfield = 0;
      if (line[0] == '\0')
            return 0;
      strcpy(sline, line);
      p = sline;
```

```
        do {
            if (nfield >= maxfield){
                maxfield *= 2;          /* tăng lên
gấp đôi */
                newf = (char **)realloc(field,
                   maxfield  * sizeof(field[0]));
                if (newf == NULL)
                   return NOMEM;
                field  = newf;
            }
            if (*p  == '"')
                sepp = advquoted(++p);   /*bỏ qua
dấu nháy ban đầu*/
            else
                sepp = p + strcspn(p, fieldsep);
            sepc = sepp[0];
            sepp[0] = '\0'; /* trường kết thúc */
            field[nfield++] = p;
            p = sepp + 1 ;
        } while (sepc  == ',');
        return nfield ;
    }
```

Vòng lặp sẽ mở rộng mảng con trỏ các trường khi cần thiết, tiếp đến gọi một trong hai hàm khác để định vị và xử lý trường kế tiếp. Nếu trường bắt đầu bằng một dấu nháy, thì hàm `advquoted` sẽ tìm trường và trả về một

con trỏ tới dấu ngăn cách ở cuối trường. Ngược lại, dùng hàm strcspn(p,s) trong thư viện để tìm dấu phẩy kế tiếp; nó tìm trong chuỗi p một xuất hiện kế tiếp của bất kỳ ký tự nào trong chuỗi s; nó trả về số ký tự đã đi qua.

Các dấu nháy giữa một trường được thể hiện bằng hai dấu nháy liên tiếp, do đó hàm advquote ép chúng lại thành một; nó cũng loại bỏ các dấu nháy bao quanh trường. Một vài phức tạp được thêm vào nhằm giải quyết các dữ liệu nhập gần giống với định nghĩa, chẳng hạn như "abc"def. Trong các trường hợp này, ta mở rộng trường bằng cách thêm tùy ý vào sau dấu nháy thứ hai cho đến khi gặp dấu ngăn cách kế tiếp. Microsoft Excel dùng thuật toán tương tự như thế.

```
/* Hàm advquote: tra về con trỏ của ký tự phân cách kế
tiếp */

      static char *advquoted(char *p)

      {

            int i, j;

            for (i = j = 0; p[j] != '\0'; i++, j++ ) {

               if (p[j] == '"' && p[++j] != '"') {

/*tạo ra bản sao tới ký tự phân cách kế tiếp hoặc \0*/

                  int k = strcspn(p+j, fieldsep);

                  memmove(p+i, p+j, k);

                  i += k;

                  j += k;

                  break;

               }

               p[i] = p[j];
```

```
        }
        p[i]   '\0';
        return p + j;
    }
```

Vì dòng dữ liệu nhập đã được cắt ra, nên hàm `csvfield` và `csvnfield` được cài đặt bình thường:

```
/* Hàm csvfield: trả về con trỏ của trường thứ n */
    char *csvfield(int n)
    {
        if (n < 0 || n>= nfield )
            return NULL;
        return field [n];
    }

    /* Hàm csvfield: trả về số lượng trường */
    int csvnfield (void)
    {
        return nfield;
    }
```

Cuối cùng, ta thay đổi lại hàm `main` để kiểm chứng phiên bản này của thư viện; vì nó giữ một bản sao của dòng dữ liệu nhập mà *prototype* không giữ, nên nó có thể in ra dòng dữ liệu ban đầu trước khi in các trường:

```
/* Hàm main:  kiểm tra các hàm của thư viện CSV */
    int main(void)
    {
```

```
        int i;
        char *line;
        while ((line = csvgetline(stdin)) != NULL)
{
            printf("line = '%s'\n", line);
            for (i = 0; i < csvnfield(); i++)
                printf("field[%d] = '%s'\n", i,
csvfield(i));
        }
        return 0;
    }
```

Đến đây chúng ta đã hoàn thành phiên bản viết bằng ngôn ngữ C. Nó xử lý các dữ liệu nhập lớn tùy ý và thậm chí có một vài xử lý hợp lý với dữ liệu dạng lạ. Giá phải trả là nó dài hơn gấp bốn lần so với phiên bản đầu tiên và một vài chỗ của mã nguồn được làm chi tiết ra. Các mở rộng về kích thước và độ phức tạp như thế là một kết quả đặc thù cho việc chuyển một phiên bản mẫu thành sản phẩm sử dụng được.

Bài tập 4-1. Có nhiều cấp độ trong việc chia trường: một trong các cấp độ đó là chia ngay khi một số trường được yêu cầu nhằm chỉ cắt ra các trường được yêu cầu, hay là chia cho đến trường được yêu cầu. Hãy liệt kê các khả năng, ước lượng những khó khăn và lợi ích của chúng, tiếp đến viết chương trình và tính các tốc độ tương ứng của chúng.

Bài tập 4-2. Hãy mở rộng thêm một số tiện ích để các ký tự phân cách có thể đổi thành:

(a) một lớp tùy ý các ký tự;

(b) các ký tự phân cách khác nhau cho các trường khác nhau;

(c) một diễn đạt bình thường (xem Chương 9).

Một phương thức giao tiếp nên được tổ chức như thế nào?

Bài tập 4-3. Chúng ta chọn cách khởi tạo tĩnh được hỗ trợ bởi ngôn ngữ C như là cơ sở cho một lần chuyển đổi: nếu một con trỏ là `NULL` thì việc khởi tạo sẽ được thực hiện. Một khả năng khác là yêu cầu người dùng gọi một hàm khởi tạo một cách rõ ràng, điều này có thể bao gồm các kích thước khởi đầu cho các mảng. Hãy cài đặt một phiên bản kết hợp các điểm mạnh của hai cách. Vai trò của hàm `reset` trong cài đặt của bạn là gì?

Bài tập 4-4. Hãy thiết kế và cài đặt một thư viện dùng để tạo dữ liệu dạng *CSV*. Phiên bản đơn giản nhất có thể lấy một mảng các chuỗi và in chúng ra với các dấu nháy và dấu phẩy. Phiên bản tốt hơn có thể sử dụng một chuỗi có định dạng tương tự như trong hàm `printf`. Hãy xem một số gợi ý và hệ thống các ký hiệu trong Chương 9.

4.4. Bản cài đặt bằng C++

Trong phần này ta sẽ cài đặt một phiên bản của thư viện *CSV* bằng C++ để giải quyết những hạn chế của phiên bản trên ngôn ngữ C. Ở đây ta cần một vài thay đổi trong phần đặc tả; quan trọng nhất là các hàm sẽ thao tác lên các chuỗi trong C++ thay vì thao tác trên các mảng ký tự như trong C. Việc dùng các chuỗi của C++ sẽ tự động giải quyết các vấn đề về quản lý vùng lưu trữ, vì các hàm trong thư viện sẽ quản lý bộ nhớ giúp chúng ta. Đặc biệt, các hàm thao tác lên trường sẽ trả về các chuỗi cho phép hàm gọi thực hiện có thể sửa đổi, nhờ đó ta có được một thiết kế linh hoạt hơn phiên bản trước.

Lớp `Csv` định nghĩa phần giao tiếp chung, trong khi vẫn che giấu cẩn thận các biến và các hàm cài đặt. Do một đối tượng lớp bao gồm tất cả các trạng thái của một thực thể, nên ta có thể tạo ra nhiều thực thể là các biến kiểu `Csv`; do các phiên bản là độc lập nhau nên các luồng nhập liệu *CSV* có thể hoạt động đồng thời.

```
class Csv {//đọc và phân tích với dấu phẩy là giá trị
phân cách
```

```
    // dữ liệu thu: "LU", 86.25, "11/4/1998",
"2:19PM", +4.0625
        public:
        Csv(istream& fin = cin, string sep = ",") :
            fin(fin), fieldsep(sep) {}
          int getline (string&);
          string getfield(int n);
          int getnfield() const { return nfield; }
        private:
          istream& fin;  //con trỏ file dữ liệu
          string line;   // chuỗi chứa dòng dữ
liệu nhập
           vector<string> field;// các chuỗi thể
hiện trường
          int nfield;          // số lượng trường
          string fieldsep;     // chuỗi các ký tự
phân cách
          int split();
          int endofline(char);
          int endofline(char);
          int advplain(const string& line, string&
fld, int);
         int advquoted(const string& line, string&
fld, int);
```

Các tham số mặc định của `constructor` được định nghĩa nên một đối tượng `Csv` mặc định, nó sẽ đọc từ luồng nhập liệu chuẩn và dùng dấu ngăn cách trường bình thường: cũng có thể được thay thế bởi các giá trị cụ thể nào đó.

Để quản lý các chuỗi, lớp `Csv` dùng các lớp `vector` và `string` trong C++ chuẩn, chứ không dùng các kiểu chuỗi là một mảng các ký tự của C. Không có trạng thái không tồn tại của một chuỗi: *"rỗng"* chỉ có nghĩa là chiều dài bằng không, và không có dạng tương đương với `NULL`, do đó ta không thể sử dụng `NULL` như là tín hiệu kết thúc tập tin. Vì vậy, `Csv::getline` trả về dòng dữ liệu nhập thông qua một đối số tham chiếu, ta dùng giá trị trả về của hàm để cho biết kết thúc tập tin và thông báo lỗi.

```
// Hàm getline: đọc vào một dòng, và được mở rộng khi
cần thiết

      int Csv::getline(string& str)

      {

            char c;

            for    (line    =    "";    fin.get(c)    &&
!endofline(c); )

                  line += c;

            split();

            str = line ;

            return !fin.eof();

      }
```

Toán tử `+=` được định nghĩa chồng để thêm một ký tự vào cuối chuỗi.

Cần phải có vài thay đổi nhỏ trong hàm `endofline`. Một lần nữa, ta phải đọc mỗi lúc một ký tự từ dữ liệu nhập, vì không một hàm nhập chuẩn nào có thể xử lý nhiều loại dữ liệu nhập khác nhau.

```
/* Hàm endofline: kiểm tra dòng dữ liệu nhập được kết
thúc bằng \r, \n, \r\n, hoặc EOF */
int Csv::endofline(char c)
{
      int eol;
      eol = (c=='\r' || c=='\n');
      if (c == '\r') {
            fin.get(c);
            if (!fin.eof() && c != '\n')
                  fin.putback(c);
      }
      return eol;
}
```

Sau đây là phiên bản mới của hàm `split`:

```
// Hàm split: chuyển dòng thành các trường
int Csv::split()
{
      string fld;
      int i, j;
      nfield = 0;
      if (line.length() == 0)
            return 0;
```

```
    i = 0;
    do {
    if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i);
        else
            j = advplain(line, fld, i);
        if (nfield >= field.size())
            field.push_back(fld);
        else
            field[nfield] = fld;
        nfield++;
        i = j + 1;
    } while (j < line.length());
    return nfield;
}
```

Vì hàm `strcspn` không dùng được trên các chuỗi của C++, nên ta thay đổi cả hai hàm `split` và `advquoted`. Phiên bản mới của `advquote` dùng hàm chuẩn của C++ `find_first_of` để định vị một xuất hiện kế tiếp của một ký tự phân cách. Lời gọi `s.find_first_of(fieldsep,j)` sẽ tìm trong chuỗi `s` phần đầu tiên của bất kỳ ký tự nào có trong `fieldsep` và xuất hiện tại hoặc sau vị trí `j`. Nếu tìm thất bại, nó sẽ trả về một chỉ số ngoài phạm vi của chuỗi, do đó ta phải đưa nó vào trong phạm vi. Các lệnh tiếp theo bên trong vòng lặp thêm các ký tự cho tới ký tự phân cách để mảng `field` chứa các chuỗi `fld`.

```
// Hàm advquoted: trả về chỉ số của kí tự phân cách kế
tiếp
```

```
int Csv::advquoted(const string& s, string& fld,
int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
        int k = s.find_first_of(fieldsep, j);
            if (k > s.length()) // không
tìm thấy
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
        fld += s[j];
    }
    return j;
}
```

Hàm `find_first_of` cũng được dùng trong một hàm mới là `advplain`; hàm `advplain` nâng cấp một trường đơn giản không nằm trong dấu nháy. Một lần nữa, thay đổi này là cần thiết bởi vì các hàm xử lý chuỗi trong C như `strcspn` không áp dụng được với các chuỗi của C++, một kiểu dữ liệu hoàn toàn khác.

```
/* Hàm advplain: trả về chỉ số của kí tự phân
cách kế tiếp của trường không nằm trong dấu nháy */
int Csv::advplain(const string& s, string& fld,
int i)
{
      int j;
      j = s.find_first_of(fieldsep, i);// tìm ký
tự phân cách
      if (j > s.length())       // không tìm thấy
          j = s.length();
      fld = string(s, i, j-i);
      return j;
}
```

Cũng giống như phần trước, hàm `Csv::getfield` được viết bình thường, trong khi `Csv::getnfield` thì ngắn đến nỗi nó được cài đặt trong định nghĩa lớp.

```
// Hàm getfield: trả về trường thứ n
string Csv::getfield(int n)
{
      if (n < 0 || n >= nfield)
            return "";
      else
            return field[n];
}
```

Chương trình kiểm chứng của chúng ta không khác hơn trước lắm:

```
// Hàm main: kiểm tra lớp Csv
int main (void)
{
      string line;
      Csv csv;
      while (csv.getline(line) != 0) {
         cout << "line = '" << line <<"'\n";
          for (int i = 0; i < csv.getnfield();
i++)
            cout << "field[" << i << "] = '"
                 << csv.getfield(i) << "'\n";
      }
      return 0;
}
```

Cách dùng thì khác với phiên bản C, mặc dù không nhiều. Tùy thuộc vào trình biên dịch. phiên bản C++ chạy chậm hơn phiên bản C từ 40% đến 4 lần với các tập tin dữ liệu nhập lớn khoảng 30000 dòng và 25 trường mỗi dòng. Như chúng ta đã thấy khi so sánh với các phiên bản của *Markov*, sự khác biệt này phản ánh mức độ hoàn thiện của thư viện. Chương trình nguồn của C++ ngắn hơn khoảng 20%.

Bài tập 4-5. Hãy cải tiến cài đặt C++ bằng cách định nghĩa chồng toán tử `[]` để các trường có thể được truy xuất theo kiểu `csv[i]`.

Bài tập 4-6. Hãy viết một phiên bản của thư viện *CSV* bằng Java, rồi so sánh ba cách cài đặt trên phương diện trong sáng, mạnh mẽ, và tốc độ.

Bài tập 4-7. Hãy đóng gói phiên bản C++ của các mã nguồn *CSV* như một *STL* quen thuộc.

Bài tập 4-8. Phiên bản C++ cho phép nhiều thực thể `Csv` độc lập thi hành đồng thời mà không gây cản trở nhau, một lợi ích của việc đóng gói tất cả các trạng thái vào một đối tượng có thể được nhân bản nhiều lần. Hãy sửa lại phiên bản C nhằm đạt được kết quả tương tự bằng cách thay thế các cấu trúc dữ liệu toàn cục bằng các cấu trúc được cấp phát và khởi tạo bằng một hàm `csvnew` một cách tường minh.

4.5. Nguyên tắc thiết kế phương thức giao tiếp

Trong các phần trước chúng ta đã nêu ra các phần chi tiết của một phương thức giao tiếp. Nó định nghĩa các công việc mà mã nguồn thực hiện cho đối tượng sử dụng nó, và cách thức của các hàm và các thành phần dữ liệu có thể được sử dụng bởi những phần còn lại của chương trình. Phương thức giao tiếp *CSV* của chúng ta hỗ trợ ba hàm: đọc một dòng, lấy một trường, và trả về số trường, và nó cũng là ba điều khiển duy nhất mà nó có thể thực hiện.

Một phương thức giao tiếp tốt phải phù hợp tốt với tác vụ của nó – đơn giản, tổng quát, dễ dùng, và tiện lợi – và nó phải dễ thích ứng khi đối tượng sử dụng nó thay đổi và cách thức cài đặt nó thay đổi. Các phương thức giao tiếp tốt phải tuân theo một số nguyên tắc nhất định. Các nguyên tắc này có thể phụ thuộc nhau hoặc thậm chí phải nhất quán, nhưng chúng giúp ta mô tả các vấn đề xảy trong phần giao tiếp giữa các thành phần trong một chương trình.

Che giấu các chi tiết cài đặt

Các cài đặt bên trong mỗi giao tiếp nên được che giấu để nó có thể được thay đổi mà không làm ảnh hưởng hay phá vỡ cấu trúc chương trình. Có nhiều thuật ngữ dùng cho nguyên tắc tổ chức loại này; che giấu thông tin, tính đóng gói, trừu tượng hoá, môđun hoá; ý nghĩa của các thuật ngữ này có liên quan với nhau. Một phương thức giao tiếp nên che giấu các chi tiết cài đặt không liên quan đến đối tượng sử dụng nó.

Các thư viện cơ sở của hầu hết các ngôn ngữ lập trình chính là các ví dụ tương tự, mặc dù không phải lúc nào chúng cũng là các thiết kế tốt. Thư viện nhập xuất chuẩn của C là một trong số nổi tiếng nhất nó có hàng tá hàm như: mở, đóng, đọc, viết, hay nói cách khác là các hàm thao tác trên tập tin. Cách thức thao tác tập tin được che giấu phía sau kiểu dữ liệu `FILE*`; ta có thể nhìn thấy các thuộc tính của nó (bởi vì chúng thường được liệt kê trong thư viện `<stdio.h>`), tuy nhiên không nên dùng chúng.

Tránh dùng biến toàn cục: cố gắng truyền các tham chiếu tới tất cả các dữ liệu thông qua các đối số hàm nếu có thể được.

Chúng tôi phản đối mạnh mẽ việc dùng chung dữ liệu dưới mọi hình thức; rất khó để bảo đảm tính nhất quán của các giá trị nếu người dùng có thể thay đổi các biến theo ý họ. Các giao tiếp hàm cho phép tăng cường các quy định truy cập dễ dàng hơn, tuy nhiên nguyên tắc này thường không nhất quán. Các luồng nhập xuất được định nghĩa sẵn như `stdin` và `stdout` gần như luôn được định nghĩa như là các thành phần của một mảng toàn cục các cấu trúc `FILE`:

```
extern FILE      __iob[_NFILE];

#define stdin  (&__iob[0])

#define stdout (&__iob[1])

#define stderr (&__iob[2])
```

Tên riêng `__iob` theo quy ước chuẩn ANSI C, hai gạch dưới đặt trước là để quy ước cho các tên riêng được nhìn thấy, điều này nhằm giảm khả năng xung đột với các tên trong chương trình.

Các cơ chế lớp trong C++ và Java tốt hơn trong việc che giấu thông tin; đó là điểm mạnh của hai ngôn ngữ này. Các lớp container trong thư viện mẫu chuẩn của C++ *(C++ Standard Template Library)* mà chúng ta dùng trong Chương 3 thậm chí còn mạnh hơn: không có một thông tin gì về cách thức cài đặt ngoại trừ một số bảo đảm cho việc thực hiện, và các người tạo thư viện có thể sử dụng bất kỳ cơ chế nào họ thích.

Chọn một tập nhỏ chuẩn các hàm cơ sở

Phương thức giao tiếp nên hỗ trợ nhiều chức năng, nhưng không nên nhiều hơn mức cần thiết, và không nên giao nhau quá nhiều chức năng cho một hàm. Thư viện có nhiều hàm có thể dễ sử dụng hơn – bất cứ thứ gì ta cần, luôn có sẵn hàm để dùng. Tuy nhiên, phương thức giao tiếp lớn tốn công sức viết và bảo trì hơn, cũng như việc tìm hiểu và sử dụng nó. Chẳng hạn như, phương thức giao tiếp *API (Application Program Interfaces)* đôi khi quá lớn để chúng ta có thể tận dụng hết tính năng của chúng.

Để tiện lợi, một số phương thức giao tiếp hỗ trợ nhiều cách thực hiện một công việc như nhau; ta không nên theo khuynh hướng này. Thư viện nhập xuất của C hỗ trợ ít nhất bốn hàm khác nhau để xuất một ký tự ra luồng xuất:

```
char c;

putc(c, fp);

fputc(c, fp);

fprintf(fp, "%c", c);

fwrite(&c, sizeof(char), 1, fp);
```

Nếu luồng là `stdout`, thì có rất nhiều khả năng lựa chọn. Điều này thì thuận tiện, nhưng không phải tất cả các hàm đều cần thiết.

Các phương thức giao tiếp hẹp được ưa chuộng hơn các phương thức giao tiếp rộng, đặc biệt khi chúng có nhiều hàm cần thiết: ít hàm, nhưng hỗ trợ tốt. Không nên thêm phương thức giao tiếp chỉ vì nó dễ thực hiện. Việc có một hàm luôn an toàn và chạy nhanh, nếu có thể được, thì tốt hơn là có hai hàm. Chẳng hạn như, hàm `memcpy` và hàm `memmove` đều dùng để sao chép vùng nhớ, nhưng hàm `memcpy` chạy nhanh và hàm `memmove` thực hiện an toàn.

Tránh đặt gánh nặng lên vai người dùng

Các hàm trong một thư viện không nên viết các tập tin hay các biến một cách bí mật, hoặc thay đổi dữ liệu toàn cục, và nên thận trọng về việc sửa đổi dữ liệu trong hàm gọi nó. Hàm `strtok` vi phạm điều này. Chúng ta hơi ngạc nhiên khi hàm `strtok` viết các *byte null* vào giữa chuỗi dữ liệu nhập của nó. Việc con trỏ `null` được dùng như một tín hiệu cho biết vị trí nó đã đi qua trong lần gọi trước đó cho thấy dữ liệu bí mật được cất giữ giữa các lần gọi, dường như đây là nguồn gốc việc phát sinh lỗi, và nó ngăn cản việc sử dụng hàm này một cách đồng thời. Một thiết kế tốt hơn là cung cấp một hàm đơn lẻ có chức năng thiết lập các tín hiệu cho chuỗi dữ liệu nhập. Với những lý do tương tự, phiên bản C thứ hai của chúng ta không thể được dùng cho hai luồng nhập; xem bài tập 4-8.

Dùng cùng cách thức cho cùng công việc ở mọi nơi

Nhất quán và thân thiện là điều quan trọng. Các công việc liên quan nhau nên được thực hiện theo các cùng một liên quan với nhau. Các hàm cơ bản trong C dạng `str...` có thể được sử dụng một cách dễ dàng không cần tài liệu đi kèm, vì chúng thể hiện tương tự nhau: các luồng dữ liệu từ phải sang trái, cùng chiều như trong câu lệnh gán, và tất cả đều trả về chuỗi kết quả. Ngược lại, khó có thể đoán được thứ tự của các đối số truyền cho các hàm trong thư viện nhập xuất chuẩn của C. Một số hàm có đối số `FILE*` đầu tiên, một số hàm khác thì có đối số ở cuối; một số khác thì có rất nhiều thứ tự khác nhau giữa kích thước và số phần tử. Các thuật toán được dùng trong các *STL container* cho thấy phương thức giao tiếp rất nhất quán, do đó rất dễ tiên đoán cách sử dụng của một hàm không quen thuộc.

Các họ hàm `mem...` được thiết kế sau các hàm `str...` trong C nhưng vay mượn cách thức của chúng. Các hàm nhập/xuất chuẩn `fread` và `fwrite` sẽ dễ nhớ hơn nếu nó có dạng giống như các hàm `read` và `write` mà ta thường dùng.

4.6. Quản lý tài nguyên

Một trong những vấn đề khó nhất trong việc thiết kế phương thức giao tiếp cho một thư viện (hay một lớp) là quản lý các tài nguyên sở hữu bởi thư viện hoặc được chia sẻ bởi thư viện và các đối tượng gọi nó. Tài nguyên dễ thấy nhất là bộ nhớ – ai sẽ chịu trách nhiệm trong việc cấp phát và giải phóng vùng lưu trữ? Ngoài ra, còn có các tài nguyên chia sẻ khác bao gồm các tập tin đang mở và trạng thái của các biến có giá trị cần quan tâm. Để đơn giản, ta chỉ xét các vấn đề sau: khởi tạo, duy trì trạng thái, chia sẻ và sao chép, làm sạch.

Prototype của gói *CSV* sử dụng các khởi tạo tĩnh để gán giá khởi tạo cho các con trỏ, biến đếm,... Nhưng cách này hạn chế bởi vì nó ngăn cấm khởi động lại các tác vụ về trạng thái khởi đầu của chúng khi một trong các hàm tương ứng với chúng được gọi. Một phương pháp khác là hỗ trợ một hàm khởi tạo dùng để đưa tất cả các giá trị bên trong về các giá trị khởi tạo chính xác. Điều này cho phép khởi động lại, nhưng người sử dụng phải gọi nó một cách rõ ràng. Hàm `reset` trong phiên bản thứ hai có thể được chuyển thành `public` cho mục đích này.

Trong C++ và Java, các *constructor* được dùng để khởi tạo các thành phần dữ liệu của các lớp. Một *constructor* được định nghĩa đúng phải đảm bảo rằng tất cả các thành phần dữ liệu của nó đều được khởi tạo, và do đó sẽ không có một đối tượng nào được tạo ra mà chưa khởi tạo các thành phần dữ liệu của nó. Có thể có một nhóm các *constructor* hỗ trợ nhiều cách khởi tạo khác nhau: chúng ta có thể cung cấp cho `Csv` một *constructor* với một tên tập tin và một *constructor* khác với dữ liệu nhập như là đối số.

Thư viện sẽ quản lý thông tin các bản sao như thế nào, chẳng hạn như các dòng và các trường nhập liệu? Phiên bản C của chương trình `csvgetline` cho phép truy cập trực tiếp đến các chuỗi dữ liệu nhập (dòng và trường) bằng cách trả về các con trỏ tới chúng. Việc truy cập không hạn chế này gây ra nhiều hậu quả khác nhau. Nó cho phép người dùng ghi đè lên bộ nhớ, và như thế làm sai lệch thông tin; dòng lệnh sau đây là một ví dụ:

```
strcpy(csvfield(1), csvfield(2));
```

có thể gây lỗi theo nhiều cách, thường thấy nhất là ghi đè lên phần đầu của `field2` nếu kích thước của `field2` lớn hơn `field1`. Do đó cần phải tạo ra một bản sao của thông tin trước khi gọi hàm `csvgetline`; phần minh họa dưới đây, con trỏ có thể hoàn toàn bị sai sau khi các dòng lệnh được thực hiện nếu như hàm `csvgetline` thứ hai gây ra một sự cấp phát bộ nhớ lại trên chính vùng bộ nhớ của nó.

```
char *p;

csvgetline(fin);

p = csvfield(1);

csvgetline(fin);

/* con trỏ p có thể bị sai ở đây */
```

Trong phiên bản C++ an toàn hơn do các chuỗi là các bản sao nên có thể được sửa đổi tùy ý.

Còn trong Java dùng các tham chiếu đến các đối tượng – các đối tượng là các thực thể, khác với các kiểu cơ sở chẳng hạn như kiểu `int`. Điều này hiệu quả hơn là tạo ra một bản sao, tuy nhiên người ta có thể nhầm lẫn một tham chiếu với một bản sao; chúng ta mắc phải một lỗi như thế trong phiên trước của chương trình Java *Markov*; và đây cũng chính là nguồn gốc của vô số lỗi liên quan đến các chuỗi trong C. Các phương thức `Clone` cho phép tạo ra một bản sao khi cần thiết.

Ngược với các *constructor* là *destructor* – làm sạch và khôi phục lại các tài nguyên khi một thực thể khi không dùng đến nó nữa. Điều này đặc biệt quan trọng đối với bộ nhớ, bởi vì một chương trình nếu không khôi phục các vùng nhớ không dùng nữa thì có thể sẽ kết thúc không như mong muốn. Nhiều phần mềm hiện nay dễ bị lỗi này. Các vấn đề tương tự cũng xảy ra khi đóng các tập tin đang mở: nếu dữ liệu còn nằm trên *buffer* thì *buffer* nên được làm sạch (và vùng nhớ của nó được giải phóng). Đối với các hàm thư viện chuẩn của C, việc giải phóng vùng nhớ được thực hiện tự

động khi chương trình kết thúc bình thường, vì nó đã được lập trình sẵn. Hàm `atexit` chuẩn trong C và C++ cung cấp một cách thức lấy quyền điều khiển ngay trước khi chương trình kết thúc bình thường; do đó những người viết phương thức giao tiếp có thể dùng công cụ này để lập kế hoạch làm sạch vùng nhớ.

Cấp phát và giải phóng tài nguyên trong cùng một nơi

Một cách để quản lý việc cấp phát và giải phóng tài nguyên là cấp phát và giải phóng phải được thực hiện trong cùng một thư viện, hoặc một phương thức giao tiếp. Nói cách khác, trạng thái cấp phát của một tài nguyên không nên thay đổi trước và sau khi dùng phương thức giao tiếp. Các thư viện *CSV* của chúng ta đọc dữ liệu từ tập tin đã được mở, vì thế tập tin vẫn được mở khi chúng đã thực hiện xong. Hàm gọi thư viện phải có nhiệm vụ đóng tập tin.

Các *constructor* và *destructor* trong C++ giúp tăng cường nguyên tắc này. Khi một thực thể của lớp vượt ngoài phạm vi kiểm soát hoặc bị hủy hoàn toàn thì *destructor* được gọi để làm sạch *buffer*, giải phóng vùng nhớ, đặt lại các giá trị, và thực hiện những việc cần thiết khác. Trong Java không hỗ trợ một cơ chế tương tự như vậy. Mặc dù Java cho phép định nghĩa những hàm hủy trong một lớp, nhưng không chắc là nó sẽ được gọi thực hiện, chứ huống hồ chi trong các trường hợp đặc biệt, vì thế những thao tác làm sạch vùng nhớ không bảo đảm được thực hiện, mặc dù điều đó có vẻ như vô lý.

Nhưng Java có hỗ trợ đáng kể trong việc quản lý bộ nhớ nhờ vào bộ dọn dẹp được xây dựng sẵn. Khi một chương trình chạy, nó sẽ cấp phát những đối tượng mới. Không cách nào có thể giải phóng hoàn toàn những đối tượng này, nhưng hệ thống nhận biết được những đối tượng nào vẫn còn hữu dụng và những đối tượng không được dùng nữa để giải phóng chúng một cách hợp lý.

Có nhiều kỹ thuật được dùng trong bộ dọn dẹp. Một số thuật toán dùng kỹ thuật ghi nhớ số lần sử dụng của mỗi đối tượng bằng cách dùng

biến đếm tham chiếu. và sẽ giải phóng một đối tượng khi biến tham chiếu của nó xuống không. Kỹ thuật này hoàn toàn có thể được áp dụng trong C và C++ để quản lý những đối tượng được chia sẻ. Các thuật toán khác dựa vào toàn bộ không gian cấp phát để theo vết theo định kỳ tất cả các đối tượng được tham chiếu. Các đối tượng được tìm thấy ở đây thì vẫn còn hữu dụng; các đối tượng không được tham chiếu bởi bất kỳ đối tượng nào thì không còn được dùng nữa và có thể được giải phóng.

Việc bộ dọn dẹp tồn tại một cách tự động không có nghĩa là ta không xem xét các vấn đề về quản lý bộ nhớ trong một thiết kế. Chúng ta vẫn còn phải quyết định xem các thao tác trong phương thức giao tiếp sẽ trả về các tham chiếu đến các đối tượng được chia sẻ hay trả về các bản sao của chúng; và điều này ảnh hưởng đến toàn bộ chương trình. Cũng không có nghĩa là bộ dọn dẹp không hoạt động - nó bảo quản thông tin và giải phóng những vùng nhớ không được sử dụng, và việc này được thực hiện vào những lúc không đoán trước được.

Các vấn đề này trở nên phức tạp hơn khi một thư viện được dùng trong môi trường có nhiều tiểu trình chạy cùng lúc như trong chương trình đa tiểu trình của Java.

Để tránh xảy ra rắc rối, chúng ta phải viết mã nguồn cho phép chạy đồng thời, nghĩa là nó vẫn thực hiện đúng kể cả trong trường hợp có nhiều lần gọi nó thực hiện cùng lúc. Mã nguồn cho phép chạy đồng thời sẽ tránh dùng các biến toàn cục, các biến cục bộ tĩnh, và bất kỳ các biến nào có thể bị thay đổi trong các tiểu trình khác. Để có một thiết kế tốt dùng trong môi trường đa tiểu trình, thì cần phải phân chia các thành phần ra sao cho chúng không có phần dùng chung ngoại trừ các phương thức giao tiếp được định nghĩa tốt. Các thư viện cho phép chia sẻ các biến một cách không cố ý sẽ phá vỡ mô hình. (Trong một chương trình đa tiểu trình, hàm `strtok` gây ra rất nhiều phiền toái, và các hàm trong thư viện C lưu giữ các giá trị trong bộ nhớ trong tĩnh cũng thế). Những biến được dùng chung phải được bảo vệ bằng các cơ chế khóa để bảo đảm rằng tại một thời điểm chỉ có một tiểu trình được phép truy xuất chúng. Các lớp tỏ ra rất hữu ích trong trường hợp

này, vì chúng có được thiết kế nhắm vào các mô hình cho phép chia sẻ và có cơ chế khóa. Các phương thức đồng bộ hóa trong Java cho phép một tiểu trình khóa toàn bộ một lớp hay chỉ đối tượng của một lớp để tránh việc sửa đổi đồng thời giữa các tiểu trình; các khối đồng bộ hoá cho phép chỉ một tiểu trình chạy một phần mã nguồn tại một thời điểm.

Đa tiểu trình làm phức tạp hóa các vấn đề lập trình một cách đáng kể, và là một đề tài lớn mà chúng ta không thể thảo luận chi tiết ở đây được.

4.7. Thoát, thực hiện lại, dừng thất bại?

Trong các chương trước, ta đã dùng các hàm như `eprintf` và `estrdup` để xử lý lỗi bằng cách hiển thị một thông báo trước khi kết thúc chương trình. Ví dụ, hàm `eprintf` tương tự như hàm `fprintf(stderr, ...)`, nhưng chỉ khác là nó thông báo lỗi trước khi thoát chương trình do lỗi gây ra. Nó dùng tập tin *header* `<stdarg.h>` và thư viện `vfprintf` để in ra các đối số được mô tả bằng ba dấu chấm như trong *prototype* dưới dây. Thư viện `stdarg` phải được khởi tạo bằng cách gọi `va_start` và được kết thúc bằng cách gọi `va_end`. Chúng ta sẽ dùng phương thức giao tiếp này nhiều hơn trong Chương 9.

```
#include <stdarg.h>
#include <string.h>
#include <errno.h>
/* Hàm eprintf: in những thông báo lỗi và kết thúc */
void eprintf(char *fmt, ...)
{
    va_list args;
    fflush(stdout);
    if (progname() != NULL)
        fprintf(stderr, "%s: ", progname());
```

```
        va_start(args, fmt);
        vfprintf(stderr, fmt, args);
        va_end(args);
    if (fmt[0] != '\0' && fmt[strlen(fmt)-1] == ':')
        fprintf(stderr, " %s", strerror(errno));
        fprintf(stderr, "\n");
        exit(2);
    }
```

Nếu định dạng chuỗi đối số kết thúc bằng một dấu hai chấm ':' thì hàm `eprintf` gọi hàm `strerror` trong thư viện chuẩn của C, hàm `strerror` trả về một chuỗi chứa tất cả các thông tin về lỗi của hệ thống nếu có. Ta cũng đã viết hàm `weprintf`, tương tự như hàm `eprintf` , để hiển thị lỗi nhưng chỉ để cảnh báo chứ không thoát chương trình.

Tương tự như thế, hàm `estrdup` tạo ra bản sao của một chuỗi, hay kết thúc với một thông thông điệp (thông qua hàm `eprintf`) nếu không đủ bộ nhớ:

```
/* Hàm estrdup: tạo ra bản sao của một chuỗi, hay thông
báo khi có lỗi */
    char *estrdup(char *s)
    {
        char *t;
        t = (char *) malloc(strlen(s)+1);
        if (t == NULL)
          eprintf("estrdup(\"%.20s\") failed:", s);
        strcpy(t, s);
        return t;
```

```
}
```

và hàm `emalloc` hỗ trợ một dịch vụ tương tự như hàm `malloc`:

```
/* Hàm emalloc: gọi đến hàm malloc, sẽ hiển thị thông
báo khi có lỗi */

void *emalloc(size_t n)

{

      void *p;

      p = malloc(n);

      if (p == NULL)

        eprintf("malloc of %u bytes failed:", n);

      return p;

}
```

Tập tin *header* `eprintf.h` tương ứng khai báo các hàm sau:

```
/* eprintf.h: chứa các hàm về xử lý lỗi */

extern void eprintf(char *, ...);

extern void weprintf(char *, ...);

extern char *estrdup(char *);

extern void *emalloc(size_t);

extern void *erealloc(void *, size_t);

extern char *progname(void);

extern void setprogname(char *);
```

Tập tin *header* này sẽ được đưa vào bất kỳ một tập tin nào gọi một trong các hàm xử lý lỗi này. Mỗi thông báo lỗi cũng chứa tên chương trình nếu nó được gán bởi người sử dụng: nó được gán và được lấy ra bởi hàm thông dụng `setprogname` và `progname`, hai hàm này được khai báo trong

tập tin *header* và được định nghĩa trong tập tin nguồn cùng với eprintf:

```
static char *name = NULL;
/* Hàm setprogname: gán tên cho chương trình */
void setprogname(char *str)
{
      name = estrdup(str);
}
/* Hàm progname: trả về tên của chương trình */
char *progname(void)
{
      return name;
}
```

Và cách dùng nó như sau:

```
int main (int argc, char *argv[])
{
      setprogname("markov");
      ...
      f = fopen(argv[i], "r");
if (f == NULL)
            eprintf("can't open %s:", argv[i]);
      ...
}
```

kết quả của chương trình như sau:

```
markov: can't open psalm.txt: No such file or directory
```

Ta nhận thấy rằng các hàm được viết ở trên thì tiện lợi cho việc lập trình của chúng ta, vì chúng có phần xử lý lỗi thay vì bỏ qua. Không có gì đặc biệt trong thiết kế của chúng ta, tuy nhiên có lẽ bạn thích có một vài biến thể cho các chương trình của bạn hơn.

Giả sử ta đang viết một thư viện cho người khác dùng trong các chương trình của họ. Các hàm đó sẽ xử lý như thế nào trong trường hợp có những lỗi không phát hiện được? Các hàm mà ta đã viết ở phần đầu của chương này hiển thị một thông báo và thoát khỏi chương trình. Cách xử lý như thế có thể chấp nhận được trong nhiều chương trình, đặc biệt là trong các công cụ và ứng dụng nhỏ độc lập. Tuy nhiên, đối với các chương trình khác thì thoát như thế sẽ gây ra lỗi nặng, vì nó làm cho phần còn lại của chương trình không thể phục hồi được; ví dụ như, một chương trình xử lý văn bản phải có chức năng phục hồi khi xảy ra lỗi để dữ liệu đang nhập vào không bị mất. Trong một vài tình huống, một tác vụ của thư viện thậm chí không nên hiển thị thông báo, vì chương trình có thể đang chạy trong một môi trường mà ở đó các thông điệp có thể gây phiền toái cho dữ liệu đã được hiển thị hoặc sẽ biến mất không để lại dấu vết. Một phương pháp hữu dụng là xuất các lỗi được phát hiện vào một *log file* để có thể xem một cách độc lập.

Phát hiện lỗi ở mức thấp và xử lý lỗi ở mức cao

Nguyên lý chung là các lỗi nên được phát hiện ở mức thấp nhất có thể, và được xử lý ở mức cao. Trong hầu hết các trường hợp, hàm gọi xác định cách thức xử lý lỗi, chứ không phải là hàm được gọi thực hiện việc này. Các hàm của thư viện thực hiện nguyên lý này bằng cách tránh xử lý; đó là lý do để chúng ta trả về giá trị `NULL` cho một trường không tồn tại thay vì kết thúc chương trình. Tương tự như vậy, hàm `csvgetline` luôn trả về giá trị `NULL` khi kết thúc tập tin, bất kể giá trị kết thúc tập tin được đọc bao nhiêu lần.

Việc lựa chọn các giá trị trả về thích hợp không phải lúc nào cũng dễ dàng, như trường hợp giá trị trả về của hàm `csvgetline` đã được trình bày ở phần trước. Ta luôn muốn trả về nhiều thông tin hữu ích nhất có thể, nhưng phải ở trong dạng thức dễ dùng và phải đáp ứng được nhu cầu của chương trình. Trong C, C++, và Java điều này có nghĩa là trả về giá trị hàm hay các giá trị khác thông qua các đối số tham chiếu (hay con trỏ). Nhiều thư viện hàm dựa trên khả năng phân biệt giữa các giá trị bình thường và các giá trị lỗi. Các hàm nhập liệu như hàm `getchar` trả về một ký tự cho dữ liệu hợp lệ, hoặc các giá trị đặc biệt như `EOF` cho biết kết thúc tập tin hoặc bị gây ra lỗi.

Cơ chế trên sẽ không hoạt động nếu miền giá trị của các giá trị trả về hợp lệ lấy hết tất cả các giá trị có thể. Ví dụ như hàm toán học `log` có thể trả về bất kỳ một số chấm động nào. Trong bảng chuẩn *IEEE* của các số chấm động, có một giá trị đặc biệt là `NaN` (*"not a number"*) biểu thị một lỗi và có thể trả về một tín hiệu lỗi.

Một vài ngôn ngữ như Perl và Tcl cho phép gom nhóm hai hay nhiều giá trị vào một bộ. Các ngôn ngữ này cho phép trả về một cặp (giá trị hàm, trạng thái lỗi) một cách dễ dàng. STL của C++ hỗ trợ kiểu dữ liệu `pair` có thể được áp dụng theo cách này.

Việc phân biệt giữa các giá trị ngoại lệ như kết thúc tập tin, và các trạng thái lỗi được ưa chuộng hơn là gom chúng lại thành một giá trị đơn. Nếu các giá trị không thể được phân biệt một cách dễ dàng thì cách lựa chọn khác có thể là trả về duy nhất một giá trị ngoại lệ và cung cấp một hàm khác để trả về chi tiết hơn về lỗi mới nhất.

Cách này đã được dùng trong Unix và trong thư viện chuẩn của C, ở đây nhiều lời gọi hệ thống và hàm của thư viện trả về giá trị `-1`, đồng thời gán cho biến toàn cục `errno` một giá trị lỗi đặc biệt; hàm `strerror` trả về một chuỗi kết hợp với chỉ số lỗi, chương trình này được viết như sau:

```
#include <stdio.h>

#include <string.h>
```

```
#include <errno.h>
#include <math.h>
/* Hàm main, kiểm tra tính đúng đắn của hàm errno */
int main(void)
{
     double f;
     errno = 0; /* Khởi động giá trị của errno */
      f = log(-1.23);
 printf("%f %d %s\n", f, errno, strerror(errno));
      return 0;
}
```

cho kết quả như sau:

```
nan0x10000000 33 Domain error
```

Ta nhận thấy, `errno` phải được khởi động bằng không trước; và sau đó nếu xảy ra lỗi thì `errno` sẽ được gán giá trị khác không.

Chỉ dùng các ngoại lệ trong các trường hợp ngoại lệ

Một số ngôn ngữ hỗ trợ các ngoại lệ để bắt các tình huống bất thường và phục hồi chúng về trạng thái trước đó; các ngoại lệ hỗ trợ một cách xử lý lỗi. Không nên dùng ngoại lệ để xử lý các giá trị trả về như mong muốn. Chẳng hạn như, giá trị cuối cùng đọc được từ tập tin là `EOF`, khi ấy ta nên dùng giá trị trả về này thay vì dùng ngoại lệ.

Trong Java, có người viết:

```
String fname = "someFileName"
try {
 FileInputStream in = new FileInputStream(fname);
```

```
            int c;
            while ((c = in.read()) != -1)
                System.out.print((char) c);
            in.close();
        } catch (FileNotFoundException e) {
            System.err.println(fname + "not found");
        } catch (IOException e) {
            System.err.println("IOException: " + e);
            e.printStackTrace();
        }
```

Vòng lặp trên đọc tất cả các ký tự cho đến ký tự kết thúc tập tin; khi đọc ký tự kết thúc tập tin, giá trị trả về như mong muốn là `-1`. Tuy nhiên, nếu không mở được tập tin thì sẽ phát sinh một ngoại lệ, chứ không gán giá trị `NULL` cho dữ liệu nhập như trong C hay C++. Cuối cùng, nếu có lỗi nhập/xuất xảy ra trong khối `try`, đây cũng là một ngoại lệ, thì nó sẽ được bắt bởi cụm `IOException`.

Các ngoại lệ thường hay bị lạm dụng. Các ngoại lệ có thể gây phá vỡ các cấu trúc, điều này rất dễ gây ra lỗi, vì chúng làm sai lệch luồng điều khiển. Ta có thể nhận biết được việc mở một tập tin không thành công thông qua giá trị trả về; phát sinh một ngoại lệ trong trường hợp này là quá lạm dụng. Chỉ nên dùng ngoại lệ cho việc xử lý các lỗi không đoán trước được, chẳng hạn như các lỗi về quá tải số tập tin hoặc về dấu chấm động.

Đối với các chương trình C, cặp hàm `setjmp` và `longjmp` hỗ trợ một dịch vụ ở mức thấp hơn nhiều mà trên đó một cơ chế ngoại lệ có thể được xây dựng, tuy nhiên do độ phức tạp của chúng nên ta sẽ không bàn bạc ở đây.

Việc phục hồi tài nguyên khi xảy ra lỗi sẽ được giải quyết như thế nào? Có nên để cho thư viện thực hiện phục hồi khi xảy ra lỗi hay không? Thường thì không, tuy nhiên nó sẽ có ích khi thực hiện các công việc nhằm bảo đảm cho thông tin ở trạng thái rõ ràng, và vô hại nhất có thể. Vùng lưu trữ không được dùng nên được phục hồi. Nếu các biến vẫn còn được truy xuất thì chúng nên được gán các giá trị hợp lý. Nguyên nhân thông thường gây ra lỗi là cố gắng sử dụng một con trỏ đang trỏ tới vùng lưu trữ đã được giải phóng. Nếu mã nguồn có xử lý lỗi gán các con trỏ trỏ tới giá trị không sau khi giải phóng vùng chúng trỏ tới, thì điều này sẽ được phát hiện. Hàm `reset` trong phiên bản thứ hai của thư viện *CSV* là một ví dụ cụ thể về các vấn đề này. Một cách tổng quát, mục tiêu là bảo đảm cho thư viện vẫn hữu dụng sau khi xảy ra lỗi.

4.8. Giao diện người dùng

Cho đến bây giờ, chủ đề chính mà ta đã thảo luận là các giao tiếp giữa các thành phần của một chương trình hay giữa các chương trình với nhau. Nhưng vẫn còn một loại giao tiếp quan trọng khác nữa là giao tiếp giữa chương trình và người sử dụng nó.

Hầu hết các chương trình ví dụ trong sách này là dạng văn bản, do đó phần giao tiếp người dùng có khuynh hướng đơn giản. Như chúng ta đã thảo luận trong phần trước, các lỗi nên được phát hiện và thông báo, và phục hồi nên được thực hiện ở những chỗ hợp lý. Lỗi xuất ra nên bao gồm tất cả các thông tin có giá trị và nên làm rõ nghĩa nhất có thể trong ngữ cảnh; không nên viết như sau khi thông báo lỗi:

```
estrdup failed
```

trong khi ta có thể viết:

```
markov: estrdup("Derrida") failed: Memory limit reached
```

Việc bổ sung thêm thông tin như trong `estrdup` được thực hiện rất dễ dàng, và nó có thể giúp người sử dụng xác định lỗi hoặc cung cấp các dữ liệu nhập hợp lệ.

Các chương trình nên hiển thị thông tin về cách sử dụng một cách hợp lý khi xảy ra lỗi như trong hàm sau:

```
/* Hàm usage: thông báo lỗi sử dụng và thoát */
void usage(void)
{
  fprintf(stderr, "usage: %s [-d] [-n nwords]"
      " [-s seed] [files ...]\n", progname());
      exit(2);
}
```

Tên chương trình cho chúng ta biết nguồn gốc của thông báo, điều này đặc biệt quan trọng nếu chương trình là thành phần của một tiến trình lớn hơn. Nếu chương trình chỉ xuất ra thông báo như `syntax error` hay `estrdup failed`, thì người dùng sẽ không biết được nguồn gốc của nó.

Các thông báo lỗi, việc nhập liệu, các hộp thoại nên đưa ra dạng dữ liệu nhập mẫu. Không nên nói là đối số quá lớn, mà nên đưa ra miền giá trị cụ thể. Dữ liệu nhập nên tự điều chỉnh cho hợp lý nếu có thể được, chẳng hạn một dòng lệnh đầy đủ với bộ đối số chính xác. Để hướng dẫn người dùng thao tác đúng, dữ liệu xuất có thể được đưa vào một tập tin hay được đánh dấu để có thể được nhận biết dễ dàng, và sau đó chúng được dùng để chạy một số xử lý tương ứng. Điều này cho thấy điểm yếu của các hộp thoại: nội dung của chúng khó có thể được lưu để dùng lại sau này.

Một cách hiệu quả để tạo ra một giao diện người dùng tốt cho việc nhập liệu là thiết kế một ngôn ngữ chuyên dụng cho việc gán các đối số, điều khiển các hành động, v.v...; một hệ thống các ký hiệu tốt có thể làm cho chương trình dễ dùng trong khi nó giúp tổ chức việc cài đặt. Chương 9 sẽ thảo luận vấn đề này.

Phương pháp lập trình bảo vệ *(defensive programming)*, đảm bảo cho chương trình hạn chế dữ liệu nhập vào không chính xác, quan trọng

trong việc giúp người dùng tránh sai sót khi sử dụng và đây cũng là một cơ chế an toàn. Vấn đề này sẽ được thảo luận nhiều hơn trong Chương 6 nói về kiểm chứng chương trình.

Đối với nhiều người, các giao diện đồ họa là giao diện người dùng trên máy tính của họ. Giao diện người dùng đồ họa là một đề tài lớn vì thế chúng ta chỉ đề cập đến một số vấn đề có liên quan trong quyển sách này. Thứ nhất, rất khó để tạo ra các giao diện đồ họa và đưa ra các *"chuẩn"* đúng đắn, vì sự phù hợp và thành công của chúng phụ thuộc rất nhiều vào bản tính và nhu cầu của người dùng. Thứ hai, thực tế cho thấy nếu một hệ thống hỗ trợ giao diện người dùng thì thường sẽ dùng nhiều mã nguồn cho việc xử lý các tương tác với người dùng hơn là cho công việc chính.

Tuy nhiên, các quy tắc tương tự được áp dụng cho cả thiết kế bên ngoài lẫn cài đặt bên trong của một phần mềm giao diện người dùng. Theo quan điểm của người dùng, các vấn đề như đơn giản, dễ hiểu, thân thiện, nhất quán, và đúng đắn đều góp phần làm cho giao diện dễ sử dụng; thiếu các đặc tính này thường dẫn đến giao diện tồi hoặc khó sử dụng.

Tính nhất quán và thân thiện sẽ được ưa chuộng, bao gồm việc sử dụng thống nhất các thuật ngữ, đơn vị, định dạng, cách thể hiện, phông chữ, màu sắc, kích cỡ, và tất cả các lực chọn bổ sung khác mà một hệ thống đồ họa có thể hỗ trợ. Có bao nhiêu từ tiếng Anh khác nhau được dùng để chỉ việc thoát khỏi chương trình hoặc đóng cửa sổ? Có hàng tá cách chọn lựa khác nhau. Sự không thống nhất này có thể sẽ gây lẫn lộn cho người bản xứ và khó khăn cho những người khác.

Với các mã đồ họa, các giao diện là đặc biệt quan trọng vì những hệ thống này lớn, phức tạp, và được điều khiển bởi một mô hình nhập liệu khác hơn nhiều so với việc xử lý văn bản một cách tuần tự. Lập trình hướng đối tượng rất thích hợp cho các giao diện người dùng, bởi vì nó cung cấp cách thức làm giảm bớt việc quản lý các trạng thái và hành vi của các cửa sổ bằng cách sử dụng thừa kế để gom các điểm chung vào các lớp cơ sở trong khi đưa các điểm khác nhau vào các lớp dẫn xuất.

Chương 5

GỠ RỐI

Trong bốn chương trước đã trình bày nhiều đoạn mã nguồn và chúng ta lầm tưởng rằng chúng đã chạy tốt ngay từ lần đầu tiên. Tất nhiên điều này không đúng bởi lẽ chúng đã có nhiều lỗi. Từ "bug" không có nguồn gốc cùng lúc với sự ra đời của lập trình viên, nhưng nó là một trong những thuật ngữ thông dụng trong lĩnh vực tin học. Tại sao phần mềm lại khó như vậy?

Sự phức tạp của chương trình liên quan tới cách thức tương tác giữa các thành phần của chương trình đó, đồng thời một phần mềm lại bao gồm nhiều thành phần và các tương tác giữa chúng. Nhiều kỹ thuật làm giảm mối liên hệ giữa các thành phần trong một chương trình nhờ thế làm giảm số lượng các thành phần tương tác; chẳng hạn như kỹ thuật che giấu thông tin, trừu tượng hóa và giao tiếp, và các đặc trưng của ngôn ngữ hỗ trợ cho các kỹ thuật này. Cũng có các kỹ thuật nhằm đảm bảo tính toàn vẹn của một bản thiết kế phần mềm – các tài liệu của chương trình, việc lập mô hình, phân tích các yêu cầu, kiểm tra hình thức – nhưng chưa có một kỹ thuật nào trong số này làm thay đổi cách thức xây dựng phần mềm; chúng chỉ giải quyết được một số vấn đề nhỏ. Thực tế cho thấy rằng lỗi luôn xuất hiện thông qua quá trình kiểm chứng và được loại bỏ bằng việc gỡ rối.

Các lập trình viên có khả năng tốt thường nhận thấy rằng họ tốn cùng lượng thời gian cho việc gỡ rối và việc viết chương trình, do đó họ cố gắng rút kinh nghiệm từ các lỗi trước đó. Mỗi một lỗi khi được tìm ra sẽ giúp bạn tránh lặp lại lỗi tương tự, cũng như giúp bạn nhận ra lỗi đó khi gặp lại.

Công việc gỡ rối thường khó khăn và có thể tốn một lượng thời gian

không thể đoán trước được. Do đó, mục tiêu cần đạt được là tránh gây ra nhiều lỗi. Một số cách giúp làm giảm thời gian gỡ rối là: thiết kế cho thật tốt, phong cách lập trình tốt, kiểm tra các điều kiện biên, kiểm tra các khẳng định và tính đúng đắn trong mã nguồn, thiết kế các giao tiếp tốt, giới hạn việc sử dụng dữ liệu toàn cục và dùng các công cụ kiểm tra. Phương châm là phòng lỗi hơn chữa lỗi.

Vai trò của ngôn ngữ lập trình là gì? Động lực chính cho việc cải tiến các ngôn ngữ lập trình là cố gắng ngăn chặn các lỗi thông qua các đặc trưng của ngôn ngữ. Một số đặc trưng hạn chế bớt lỗi là: việc kiểm tra các giới hạn biên của các chỉ số, hạn chế hoặc không dùng con trỏ, bộ dọn dẹp, các kiểu dữ liệu chuỗi, xác định kiểu nhập/xuất, và kiểm tra kiểu dữ liệu chặt chẽ. Ngược lại, có một đặc tính dễ gây ra lỗi như: các lệnh `goto`, các biến toàn cục, các con trỏ trỏ đến những vùng không biết trước, và việc chuyển kiểu tự động. Các lập trình viên nên biết được các phần gây ra lỗi tiềm ẩn trong các ngôn ngữ lập trình và đặc biệt thận trọng khi sử dụng chúng. Họ cũng nên kích hoạt tất cả các chức năng kiểm tra của trình biên dịch và cũng nên quan tâm đến các cảnh báo.

Mỗi đặc trưng ngăn chặn lỗi của ngôn ngữ đều có ưu điểm và khuyết điểm. Nếu một ngôn ngữ cấp cao hơn tránh được các lỗi đơn giản một cách tự động, thì giá phải trả là nó dễ gây ra các lỗi ở mức cao hơn. Không một ngôn ngữ nào có thể ngăn hoàn toàn việc gây ra lỗi của bạn.

Phần lớn thời gian lập trình được dùng cho việc kiểm chứng và sửa lỗi mặc dù chúng ta không muốn như vậy. Chương này sẽ trình bày những phương pháp giúp bạn giảm thời gian gỡ rối một cách có hiệu quả; vấn đề kiểm chứng sẽ được trình bày trong Chương 6.

5.1. Trình gỡ rối

Các trình biên dịch cho các ngôn ngữ lớn thường kèm theo các trình gỡ rối phức tạp và được đóng gói như là một phần của môi trường phát triển phần mềm. Môi trường này tích hợp tất cả các công việc tạo lập, hiệu chỉnh mã nguồn, biên dịch, thực thi, và gỡ rối chương trình vào một hệ thống duy

nhất. Các trình gỡ rối với các giao diện đồ họa cho phép chạy chương trình từng bước qua từng lệnh hoặc từng hàm tại một thời điểm, dừng lại ở các dòng lệnh đặc biệt hay dừng lại khi xuất hiện một điều kiện đặc biệt. Chúng cũng hỗ trợ các công cụ cho phép định dạng và hiển thị các giá trị của các biến.

Trình gỡ rối có thể được kích hoạt một cách trực tiếp khi có một lỗi được biết trước. Một số trình gỡ rối kích hoạt một cách tự động khi xảy ra lỗi bất ngờ trong quá trình chạy chương trình. Thường để tìm ra được nơi mà chương trình bị hỏng khi đang chạy, chúng ta sẽ xem xét thứ tự các hàm đã được kích hoạt (theo vết ngăn xếp) và hiển thị các giá trị của các biến cục bộ và toàn cục. Những thông tin đó có thể đủ để xác định được lỗi. Nếu như vẫn không tìm ra được lỗi, thì dùng các điểm ngắt và việc chạy từng bước cho phép chúng ta chạy lại chương trình lỗi từng bước để xác định nơi đầu tiên gây ra lỗi.

Với một môi trường phù hợp và một người có kinh nghiệm thì một trình gỡ rối tốt là một phương tiện giúp gỡ rối hiệu quả và ít tốn kém. Với các công cụ mạnh như thế trong tay, tại sao ta không dùng chúng? Tại sao ta phải cần đến cả một chương nói về vấn đề gỡ rối?

Có nhiều lý do khách quan và chủ quan (dựa trên kinh nghiệm cá nhân) để trả lời cho câu hỏi trên. Một số ngôn ngữ ngoài luồng không có trình gỡ rối hoặc chỉ hỗ trợ các khả năng gỡ rối thô sơ. Các trình gỡ rối lại phụ thuộc hệ thống, nên có thể bạn sẽ không tìm thấy một trình gỡ rối quen thuộc khi làm việc trên một hệ thống khác. Các trình gỡ rối cũng hoạt động không tốt trên một số chương trình: các chương trình đa xử lý hay đa tiến trình, các hệ điều hành và các hệ thống phân tán thường phải được gỡ rối thông qua các tiếp cận ở mức thấp hơn. Trong các tình huống như vậy, bạn sẽ phải tự xoay sở lấy, bạn chỉ có thể dựa vào các câu lệnh xuất ra, kinh nghiệm và khả năng suy luận dựa trên mã nguồn của bạn.

Theo kinh nghiệm cá nhân, ta không nên dùng các trình gỡ rối bằng cách theo vết ngăn xếp hoặc lấy giá trị của một hoặc hai biến. Nguyên nhân là do chúng ta rất dễ đi lạc vào các chi tiết của các cấu trúc dữ liệu và các

luồng điều khiển phức tạp; chúng ta nhận thấy rằng việc chạy từng bước trong một chương trình thì không hiệu quả bằng việc suy nghĩ sâu hơn và thêm các lệnh xuất kết quả ra và tự kiểm tra mã nguồn tại những nơi có nguy cơ gây ra lỗi. Chạy qua các lệnh sẽ tốn nhiều thời gian hơn là duyệt qua các kết quả xuất ra của nó được đặt cẩn thận ở nơi nhạy cảm với lỗi. Quá trình quyết định vị trí đặt các lệnh xuất kết quả ra sẽ tốn ít thời gian hơn so với việc nhảy từng bước tới vùng có nguy cơ gây lỗi của mã nguồn ngay cả khi giả sử là ta đã biết trước được vùng gây lỗi. Và điều quan trọng là việc sửa lỗi các câu lệnh luôn tồn tại cùng với chương trình; còn các phiên bản của trình gỡ rối chỉ được dùng nhất thời.

Việc sử dụng trình gỡ rối một cách tự phát sẽ không có lợi. Sẽ hữu ích hơn nếu sử dụng trình gỡ rối để khám phá trạng thái của một chương trình khi nó có lỗi, tiếp đến suy nghĩ đến cách thức gây ra lỗi. Các trình gỡ rối có thể bí hiểm và khó dùng, và đặc biệt là đối với những người mới bắt đầu, chúng gây ra nhiều bối rối hơn là giúp đỡ họ. Nếu bạn đưa ra một câu hỏi sai, chúng có thể sẽ cho bạn một câu trả lời, tuy nhiên bạn có thể không biết được là nó đang bị sai.

Tuy nhiên, một trình gỡ rối có thể rất có giá trị, và bạn nên đưa một trình gỡ rối vào bộ công cụ gỡ rối của bạn. Nhưng nếu bạn không có một trình gỡ rối, hay nếu bạn phải đối phó với một vấn đề đặc biệt khó khăn thì các kỹ thuật trong chương này sẽ giúp bạn gỡ rối hiệu quả và đỡ tốn kém. Chúng cũng giúp bạn sử dụng hiệu quả hơn trình gỡ rối vì chúng tập trung chủ yếu vào việc cung cấp cho bạn các suy luận về lỗi và các nguyên nhân có thể gây ra lỗi.

5.2. Có đầu mối, phát hiện ra lỗi dễ dàng

Khi chương trình bị hỏng, hay xuất hiện các thông báo vô nghĩa, hoặc bị treo. Vì sao lại như vậy? Những người bắt đầu thường có khuynh hướng đổ lỗi cho trình biên dịch, thư viện, hay bất kỳ một nguyên nhân nào khác thay vì phải xem lại mã nguồn của họ. Các lập trình viên kinh nghiệm cũng thích làm như thế, nhưng họ biết rằng thật ra hầu hết các lỗi là do

chính các sai sót của họ.

May mắn là hầu hết các lỗi đều đơn giản và có thể được tìm thấy bằng các kỹ thuật đơn giản. Hãy khảo sát các đầu mối của việc xuất ra kết quả có lỗi và cố gắng suy ra nguyên nhân gây ra nó. Hãy quan sát bất kỳ việc gỡ rối nào của thao tác xuất ra kết quả trước khi xảy ra lỗi, nếu có thể hãy theo vết ngăn xếp từ trình gỡ rối. Bây giờ bạn có được một số thông tin về lỗi, và nơi xảy ra lỗi. Tạm dừng để ngẫm nghĩ. Lỗi xảy ra như thế nào? Suy luận ngược trở lại trạng thái của chương trình bị hỏng để xác định nguyên nhân gây ra lỗi này.

Gỡ rối liên quan đến việc lập luận lùi, giống như việc tìm kiếm các bí mật của một vụ ám sát. Một số vấn đề không thể xảy ra và chỉ có những thông tin xác thực mới đáng tin cậy. Do đó, chúng ta phải suy nghĩ ngược từ kết quả để khám phá ra các nguyên nhân. Khi ta có lời giải thích đầy đủ, ta sẽ biết được vấn đề cần sửa và có thể phát hiện ra một số vấn đề khác mà ta chưa dự trù từ trước.

Tìm các lỗi tương tự

Hãy tự hỏi liệu đây có phải là một lỗi tương tự hay không? "Tôi đã gặp qua rồi" thường là câu khởi đầu cho việc hiểu vấn đề, hay thậm chí là đã biết toàn bộ câu trả lời. Các lỗi phổ biến thì có các đặc điểm phân biệt. Chẳng hạn như, các lập trình viên C thiếu kinh nghiệm thường viết

```
?       int n;
?       scanf("%d", n);
```

thay vì

```
int n;
scanf("%d", &n);
```

và đây là một ví dụ đặc trưng về nguyên nhân gây ra một cố gắng truy cập tới bộ nhớ ngoài phạm vi khi đọc dữ liệu vào. Người dạy C sẽ nhận ra vấn

đề này ngay lập tức.

Các chuyển đổi kiểu và các kiểu không hợp lý trong hàm `printf` và `scanf` là một thường là nguyên nhân gây ra các lỗi đơn giản.

```
?       int n = 1;

?       double d = PI;

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

Một lỗi khác thường thấy nữa là sử dụng định dạng `%f` thay vì dùng định dạng `%lf` để đọc một giá trị có kiểu `double` bằng hàm `scanf`. Một vài trình biên dịch bắt các lỗi như thế bằng cách xác minh rằng các kiểu của các đối số của hàm `scanf` và `printf` phải phù hợp với các định dạng của chúng: nếu tất cả các cảnh báo được bật lên, đối với hàm `print` ở trên, thì trình biên dịch sẽ thông báo như sau:

```
x.c:9: warning: int format, double arg (arg 2)

x.c:9: warning: double format, different type arg (arg 3)
```

Không khởi tạo biến cục bộ dẫn đến một dạng lỗi đặc biệt khác, kết quả thường là một giá trị cực kỳ lớn, giá trị rác trước đó được lưu lại trong cùng vị trí bộ nhớ. Một số trình biên dịch sẽ cảnh báo bạn, mặc dù có thể bạn phải kích hoạt chức năng kiểm tra tại thời điểm biên dịch, và chúng có thể không bao giờ bắt hết tất cả các trường hợp. Bộ nhớ được trả về bởi các hàm cấp phát như `malloc`, `realloc`, và `new` cũng thường chứa giá trị rác; phải nhớ khởi tạo nó.

Kiểm tra sự thay đổi mới nhất

Thay đổi mới nhất là gì? Nếu bạn chỉ thay đổi mỗi lúc một vấn đề khi phát triển chương trình thì lỗi thường có khuynh hướng xảy ra hoặc nằm trong đoạn mã mới hoặc do đoạn mã mới tác động đến chương trình. Khảo sát cẩn thận các thay đổi mới sẽ có ích cho việc khoanh vùng. Nếu lỗi xuất hiện trong phiên bản mới và không xuất hiện trong phiên bản cũ thì đoạn mã mới chính là phần gây ra lỗi. Điều này có nghĩa là bạn nên giữ lại ít nhất là

phiên bản ngay trước đó của chương trình, mà bạn tin rằng nó chạy đúng, và như thế bạn có thể so sánh giữa các phiên bản với nhau. Nó cũng có nghĩa là bạn nên lưu lại việc sửa đổi và các lỗi được sửa, như thế bạn không phải xét lại các thông tin này trong khi cố gắng sửa lỗi. Các hệ thống quản lý mã nguồn và các cơ chế lưu lại quá trình sửa đổi sẽ rất hữu ích trong trường hợp này.

Tránh mắc cùng một lỗi hai lần

Sau khi sửa một lỗi, bạn phải nghĩ xem bạn có phạm lỗi như thế ở nơi nào khác nữa không. Xét đoạn chương trình minh họa sau:

```
?    for (i = 1; i < argc; i++) {
?          if (argv[i][0] != '-')  /*các chọn lựa được
kết thúc */
?                break;
?          switch (argv[i][1]) {
?          case 'o':   /* tên tập tin output */
?                outname = argv[i];
?                break;
?          case 'f':
?                from = atoi(argv[i]);
?                break;
?          case 't':
?                to = atoi(argv[i]);
?                break;
?          ...
```

luôn có `-o` gắn vào trước tên tập tin xuất ra. Lỗi này được sửa dễ dàng, mã nguồn được viết lại như sau:

```
outname = &argv[i][2];
```

và sau đó lại phát hiện ra chương trình bị sai khi xử lý đối số có dạng `-f123` một cách chính xác: giá trị số được chuyển đổi luôn là `0`. Đây là lỗi giống như lần trước; trường hợp tiếp theo trong lệnh `switch` nên viết là:

```
from = atoi(&argv[i][2]);
```

Mã nguồn dễ có thể có các lỗi nếu như chúng ta viết cẩu thả vì thấy nó quen thuộc. Thậm chí khi mã nguồn đơn giản đến mức bạn có thể viết trong giấc ngủ thì cũng đừng có ngủ gật trong lúc viết.

Gỡ lỗi ngay khi gặp

Quá vội vã cũng có thể gây ảnh hưởng đến các tình huống khác. Đừng bỏ qua lỗi khi chúng xuất hiện; hãy sửa ngay, vì nó có thể không xuất hiện ra lại cho đến khi quá trễ để thực hiện điều này.

Theo vết ngăn xếp

Mặc dù các trình gỡ rối có thể kiểm tra các chương trình đang chạy, nhưng một trong các ứng dụng phổ biến nhất của chúng ta là kiểm tra trạng thái của một chương trình sau khi hỏng. Số dòng mã nguồn có lỗi (thông thường được lưu vết trong ngăn xếp) là một phần hữu ích nhất cung cấp thông tin gỡ rối; các giá trị bất thường của các đối số cũng là một đầu mối lớn (các con trỏ 0, các số nguyên có giá trị lớn mà lẽ ra nó phải có giá trị nhỏ hoặc nó có giá trị âm thay vì nó phải có giá trị dương, các chuỗi ký tự không thuộc bảng mẫu tự).

Sau đây là một ví dụ đặc trưng, dựa vào phần trình bày về việc sắp xếp trong Chương 2. Để sắp xếp một mảng các số nguyên, chúng ta nên gọi hàm `qsort` với hàm so sánh số nguyên `icmp`:

```
int arr[N];

qsort(arr, N, sizeof(arr[0]), icmp);
```

nhưng giả sử tên của hàm so sánh chuỗi `scmp` được truyền vào một cách vô ý như sau:

```
?       int arr[N];
?       qsort(arr, N, sizeof(arr[0]), scmp);
```

Trình biên dịch không thể nhận biết được sự không phù hợp kiểu ở đây, do đó sẽ xảy ra lỗi. Việc chạy chương trình sẽ bị hỏng do cố gắng truy cập tới một vị trí bộ nhớ không hợp lệ.

Đọc trước khi gõ vào

Một kỹ thuật gỡ rối hiệu quả nhưng không được ủng hộ lắm là đọc đoạn mã cẩn thận và bỏ ra một chút thời gian để suy nghĩ về nó mà không thực hiện thay đổi. Nếu như có một sự thôi thúc khiến chúng ta chạm ngay vào bàn phím và bắt đầu sửa đổi chương trình thì hãy thử xem có sửa được lỗi hay không? Nhưng khi chúng ta không biết được điều gì thực sự gây ra lỗi và sửa không đúng chỗ sẽ có nguy cơ gây ra các lỗi khác. Việc liệt kê các phần có khuynh hướng gây lỗi của chương trình ra giấy có thể cho chúng ta một cái nhìn khác hơn là xem trên màn hình, và khuyến khích chúng ta bỏ ra nhiều thời gian hơn cho việc ngẫm nghĩ. Không nên thực hiện các liệt kê như vậy cho các phần không có khuynh hướng gây ra lỗi. In ra toàn bộ chương trình sẽ phá vỡ cây cấu trúc vì rất khó để thấy được cấu trúc khi nó được trải qua nhiều trang và các liệt kê trở nên không còn hữu ích khi bạn bắt đầu soạn thảo lại.

Hãy nghỉ ngơi một chút; đôi khi những thứ được viết trong mã nguồn không đúng như những gì bạn nghĩ, và những lúc nghỉ giải lao giúp bạn tỉnh táo trở lại và khi quay trở lại viết bạn sẽ nhận ra điều đó.

Hãy chống lại sự thôi thúc gõ vào; hãy nên suy nghĩ nhiều.

Giải thích cho người khác về đoạn mã của bạn

Một kỹ thuật hiệu quả khác là giải thích cho người khác về đoạn mã của bạn. Điều này thường chỉ ra được lỗi của bạn. Đôi khi chỉ cần một vài câu mở đầu thì theo sau sẽ là một câu đầy bối rối: "Ồ! Tôi phát hiện ra lỗi rồi. Xin lỗi đã làm phiền anh". Cách này hoạt động đặc biệt hiệu quả; thậm chí bạn có thể giải thích cho những người không phải là lập trình viên.

5.3. Không có đầu mối, khó phát hiện ra lỗi

Làm cho lỗi xuất hiện lại

Bước đầu tiên phải chắc chắn rằng bạn có thể làm cho lỗi xuất hiện khi cần. Thật là phiền toái để bắt các lỗi không xảy ra trong khi chạy chương trình. Bỏ ra một ít thời gian để cấu trúc lại các thiết lập dữ liệu nhập và các đối số được cho là gây ra lỗi, tiếp đến gom các cách trên lại sao cho có thể chạy nó thông qua nút nhấn hay một vài cái gõ phím. Nếu nó là một lỗi khó, bạn có thể làm cho nó lặp đi lặp lại nhiều lần trong khi bạn tìm nguyên nhân gây ra nó, và như thế bạn sẽ tiết kiệm được thời gian bằng cách làm cho nó xuất hiện lại một cách dễ dàng.

Nếu không thể làm cho lỗi xuất hiện lại qua mỗi lần chạy thì cố gắng tìm nguyên nhân tại sao lại không được. Có phải lỗi sẽ nhạy cảm trên một tập các điều kiện nào đó hay không? Thậm chí nếu bạn không thể làm cho lỗi xuất hiện qua mỗi lần chạy, nếu bạn có thể làm giảm được thời gian chờ xử lý thì bạn sẽ tìm thấy lỗi nhanh hơn.

Nếu một chương trình có hỗ trợ việc xuất kết quả gỡ lỗi, thì hãy kích hoạt nó lên. Các chương trình giả lập như chương trình chuỗi *Markov* trong Chương 3 nên đưa vào các chọn lựa như thế để biết thêm nhiều thông tin chẳng hạn như việc chọn số của bộ phát sinh số ngẫu nhiên để kết quả xuất ra có thể được tái sử dụng; các tùy chọn khác nên cho phép việc tạo lại số được chọn. Nhiều chương trình thêm vào các tùy chọn và một ý tưởng hay nữa là thêm vào các công cụ tương tự vào chương trình của chính bạn.

Chia để trị

Có thể thu hẹp phạm vi hơn hoặc chỉ tập trung vào dữ liệu nhập gây lỗi cho chương trình không? Thu hẹp các khả năng bằng cách giới hạn lại tối đa dữ liệu nhập sao cho lỗi vẫn còn xuất hiện. Cần những sửa đổi gì để giải quyết được lỗi? Cố gắng tìm ra các trường hợp kiểm tra tiêu biểu tập trung vào lỗi. Mỗi trường hợp kiểm tra nên hướng đến một kết quả cụ thể để từ đó xác nhận hay từ chối một vấn đề đặc thù nào đó được cho là gây ra lỗi.

Tiến trình thực hiện theo cách tìm kiếm nhị phân. Giới hạn dữ liệu nhập còn một nửa và xem kết quả có còn sai nữa không; nếu không, quay về trạng thái ngay trước đó và kiểm tra trên nửa kia của dữ liệu nhập. Cùng một tiến trình tìm kiếm nhị phân có thể được áp dụng trên toàn bộ chương trình: loại bỏ một số phần của chương trình không liên quan tới lỗi và kiểm tra xem có còn lỗi hay không.

Hiển thị kết quả để định vị khu vực tìm lỗi

Nếu bạn không hiểu chương trình đang làm gì, việc thêm vào các lệnh để hiển thị thêm thông tin là cách dễ dàng và hiệu quả nhất để tìm được lỗi. Đưa chúng vào để xác định sự hiểu biết của bạn hoặc để hình thành ý tưởng cho bạn về vấn đề gây lỗi. Ví dụ, hiển thị "không thể đến được đây" nếu bạn nghĩ là không có khả năng để thâm nhập vào một điểm nào đó trong đoạn mã; tiếp đến nếu bạn thấy xuất hiện thông điệp đó thì chuyển các câu lệnh xuất ngược về đầu để tìm ra nơi đầu tiên gây lỗi. Hoặc là hiển thị các thông điệp "đến đây" và di chuyển xuống dưới để xác định nơi cuối cùng mà ở đó công việc dường như là hoạt động đúng. Mỗi thông điệp nên được phân biệt để bạn có thể nhận biết được thông điệp bạn đang quan sát.

Hiển thị các thông điệp theo một định dạng cố định rõ ràng sao cho chúng có thể được duyệt qua bằng mắt hay bằng các chương trình như công cụ xác định mẫu `grep`. (Một chương trình như `grep` rất hữu ích cho việc tìm kiếm văn bản. Chương 9 có trình bày một cài đặt đơn giản). Nếu bạn cho hiển thị giá trị của một biến thì hãy định dạng nó cùng một cách qua mỗi lần hiển thị. Trong ngôn ngữ C và C++ trình bày các con trỏ như là các số thập lục phân với `%x` hay `%p`: điều này giúp bạn nhận ra việc hai con trỏ có cùng giá trị hay có liên quan nhau. Hãy học cách đọc các giá trị con trỏ và nhận biết điểm giống nhau và khác nhau, như 0, các số âm, các số lẻ, và các số nhỏ. Sự quen thuộc định dạng của các địa chỉ cũng sẽ hữu ích khi bạn sử dụng một trình gỡ rối.

Nếu kết quả xuất ra có khuynh hướng rất lớn thì có thể chỉ cần in ra các ký tự đơn như A, B, C,... để đánh dấu nơi mà chương trình đi qua.

Viết mã tự kiểm tra

Nếu cần nhiều thông tin hơn thì bạn có thể viết hàm kiểm tra cho riêng bạn để kiểm tra một điều kiện, bỏ đi các biến có liên quan, và kết thúc chương trình

```
/* Hàm check: kiểm tra điều kiện, in ra và hỏng
*/
void check(char *s)
{
      if (var1 > var 2) {
            printf("%s: var1 %d var2 %d\n", s,
var1, var2);
           fflush(stdout) /* bảo đảm tất cả kết
quả xuất ra được sạch */
          abort();        /* tín hiệu kết thúc
bất thường*/
      }
}
```

Ta viết hàm `check` để gọi hàm `abort`, một hàm trong thư viện chuẩn C có chức năng làm cho chương trình đang thi hành kết thúc một cách bất thường. Nhưng trong các ứng dụng khác, bạn có thể muốn hàm `check` vẫn thi hành sau khi in ra kết quả.

Kế tiếp, thêm vào các lời gọi tới hàm `check` vào bất kỳ nơi đâu bạn thấy là hữu ích trong mã nguồn của bạn:

```
check("trước điểm nghi ngờ");
/*...mã nguồn nghi ngờ...*/
check("sau điểm nghi ngờ");
```

Sau khi một lỗi được sửa, đừng vứt bỏ hàm `check`. Cứ lưu nó trong mã nguồn chương trình, đóng nó lại bằng dấu chú thích hay điều khiển nó bằng một lựa chọn gỡ rối, để sau này nó có thể được kích hoạt lại khi xuất hiện vấn đề khó khác.

Đối với các vấn đề khó hơn, hàm `check` có thể được phát triển lên để thực hiện việc kiểm tra và hiển thị của các cấu trúc dữ liệu. Cách tiếp cận này có thể được tổng quát hoá thành các hàm thực hiện kiểm tra tính thống nhất của các cấu trúc dữ liệu và các thông tin khác. Trong một chương trình có các cấu trúc dữ liệu phức tạp, một ý tưởng hay là viết các kiểm tra này trước khi các vấn đề xảy ra, như các thành phần của chương trình hoàn chỉnh; như thế chúng có thể được kích hoạt khi bắt đầu phát sinh lỗi. Đừng dùng chúng chỉ khi gỡ rối; cài đặt chúng trong suốt các giai đoạn phát triển chương trình. Nếu không quá tốn kém thì tốt nhất là nên để nó luôn trong trạng thái hoạt động. Các chương trình lớn như các hệ thống chuyển điện thoại thường được dùng nhiều mã nguồn để "kiểm tra tính đúng đắn" của các hệ thống con có chức năng theo dõi thông tin và thiết bị, và thông báo hoặc thậm chí sửa lỗi nếu xảy ra.

Tạo log file

Một chiến lược khác là tạo ra *log file* chứa một loạt các kết quả gỡ rối với định dạng cố định. Khi xảy ra lỗi, *log file* sẽ lưu lại những việc thực hiện ngay trước khi xảy ra lỗi. Các *web server* và các chương trình mạng khác duy trì các *log file* dài của việc lưu thông tin giúp chúng theo dõi chính chúng và các *client* của chúng; phân đoạn sau đây (đã được hiệu chỉnh cho vừa) được trích ra từ một hệ thống cục bộ:

```
[Sun Dec 27 16:19:24 1998]
HTTPd: access to /usr/local/httpd/cgi-bin/test.htm
        failed for ml.cs.bell-labs.com,
        reason: client denied by server (CGI non-executable)
```

```
from        http://m2.cs.bell-labs.com/cgi-
bin/test.pl
```

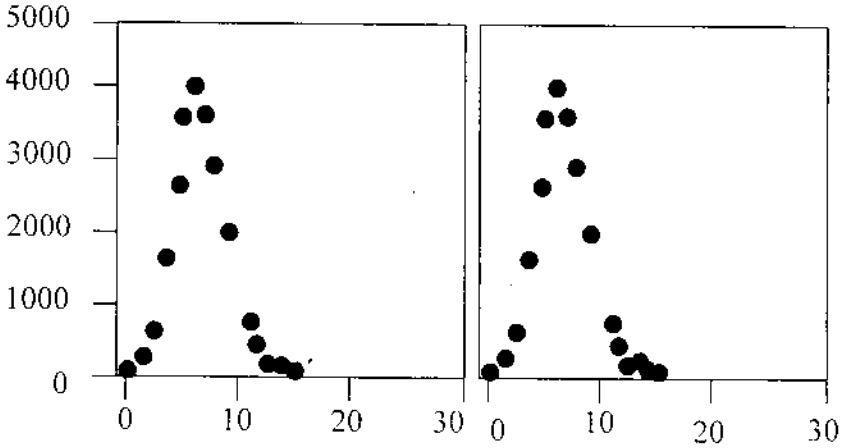
Hãy chắc chắn là các `buffer` nhập/xuất được làm sạch để các thông tin mới nhất được lưu vào *log file*. Các hàm xuất kết quả như `printf` thông thường đưa kết quả của chúng vào `buffer` để in ra hiệu quả hơn; việc kết thúc bất thường có thể tiêu huỷ kết quả đã được đưa vào `buffer` này. Trong C một lời gọi tới hàm `fflush` bảo đảm rằng tất cả các kết quả được xuất ra trước khi chương trình hỏng; trong C++ và Java có các hàm `flush` tương tự như thế cho việc xuất kết quả. Hoặc nếu bạn không sợ tốn kém thì có thể tránh làm sạch toàn bộ vấn đề bằng cách sử dụng các hàm nhập/xuất không sử dụng `buffer` cho các *log file*. Các hàm chuẩn `setbuf` và `setvbuf` điều khiển việc sử dụng `buffer`; `setbuf(fp, NULL)` làm tắt chức năng dùng `buffer` trên luồng `fp`. Các luồng xử lý lỗi chuẩn `(stderr, cerr, System.err)` mặc định thường là tắt chức năng dùng `buffer`.

Vẽ hình

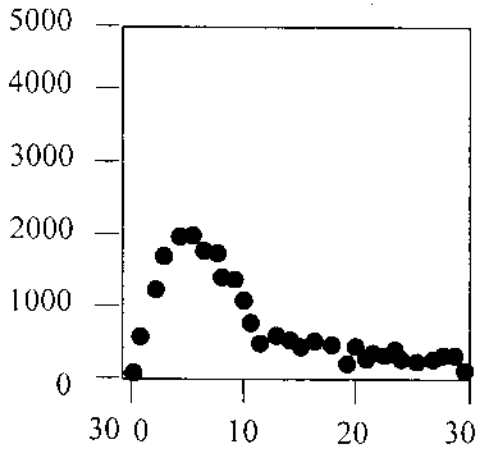
Đối với việc kiểm chứng và gỡ rối đôi khi các hình ảnh hiệu quả hơn là văn bản. Trong Chương 2, ta đã thấy các hình ảnh rất hữu ích để thể hiện các cấu trúc dữ liệu một cách dễ hiểu, và dĩ nhiên là khi đang viết phần mềm đồ họa, nhưng thật ra chúng có thể được dùng cho tất cả các loại chương trình. Các dấu chấm phân bố lộn xộn minh họa các giá trị không chính xác hiệu quả hơn là các cột của các con số. Một biểu đồ của dữ liệu cho thấy các sự bất thường trong điểm thi, các số ngẫu nhiên, các kích thước khối trong các điều khiển cấp phát và bảng băm, và v.v...

Nếu bạn không hiểu được điều gì đang diễn ra bên trong chương trình của bạn thì thử thêm chú thích cho các cấu trúc dữ liệu với các thống kê và vẽ đồ thị biểu diễn kết quả. Các đồ thị sau đây minh họa cho chương trình *Markov* trong C ở Chương 3, trục x biểu thị các chiều dài của chuỗi băm, và trục y biểu thị số lượng tương ứng của các phần trong các chuỗi của chiều dài đó. Dữ liệu vào là quyển sách 42685 từ, và 22482 tiếp đầu ngữ. Hai đồ thị đầu tiên minh họa cho hàm băm tốt ứng với các số nhân là 31 và 37, và đồ thị thứ ba minh họa cho hàm băm tồi với số nhân là 128. Trong

hai trường hợp đầu tiên, không có chuỗi nào có nhiều hơn 15 hoặc 16 phần tử và hầu hết các phần tử thuộc các chuỗi có chiều dài 5 hoặc 6. Trong trường hợp thứ 3, sự phân bố trải rộng hơn, chuỗi dài nhất có 187 phần tử, và có hàng ngàn phần tử trong các chuỗi dài hơn 20.



Số nhân 31 Số nhân 37



Số nhân 128

Sử dụng các công cụ

Hãy tận dụng các công cụ trong môi trường bạn đang gỡ rối. Ví dụ, một chương trình so sánh tập tin như `diff` so sánh các kết quả xuất ra từ các lần gỡ rối thành công và thất bại để bạn có thể tập trung vào điều được thay đổi. Nếu kết quả của việc gỡ rối dài, hãy dùng `grep` để tìm kiếm nó hoặc dùng một trình soạn thảo để kiểm tra nó. Tránh gửi các kết quả gỡ rối ra máy in; máy tính duyệt qua lượng kết quả lớn tốt hơn con người. Sử dụng các giao diện *script* và các công cụ khác để thực hiện tự động tiến trình xử lý dữ liệu xuất ra từ các lần chạy gỡ rối.

Viết các chương trình đơn giản để kiểm tra tính đúng đắn hay xác nhận sự hiểu biết của bạn về vấn đề hoạt động như thế nào. Ví dụ, việc giải phóng một con trỏ `NULL` có hợp lý hay không?

```
int main(void)

{

        free(NULL);

        return 0;

}
```

Các chương trình điều khiển mã nguồn như RCS theo dõi các phiên bản của mã nguồn sao cho bạn có thể nhìn thấy điều được thay đổi và có thể chuyển nó về các phiên bản trước đó để phục hồi một trạng thái trước đó. Bên cạnh việc chỉ ra những điều được thay đổi gần đây, chúng cũng có thể xác định được các phần của mã nguồn thường xuyên bị sửa đổi; các phần này thường là nơi tiềm tàng của lỗi.

Lưu vết

Nếu việc tìm lỗi được tiến hành sau một thời gian thì bạn sẽ bắt đầu mất dấu vết về những gì bạn đã thử qua và những gì bạn đã xem xét. Nếu bạn lưu lại các kiểm tra và các kết quả của bạn thì bạn ít khi xem lại vấn đề bạn đã xét hoặc bạn nghĩ rằng đã kiểm tra một số khả năng mà thực ra bạn

chưa kiểm tra. Thao tác lưu vết sẽ giúp bạn ghi nhớ được vấn đề trong lần kế tiếp khi có xuất hiện vấn đề tương tự, và nó cũng sẽ hữu ích khi bạn giải thích vấn đề cho người khác.

5.4. Phương sách cuối cùng

Bạn sẽ làm gì nếu không có cách nào giúp bạn giải quyết được vấn đề? Đã đến lúc bạn phải dùng một trình gỡ rối tốt để chạy chương trình từng bước. Nếu bạn nghĩ rằng cách thức hoạt động của chương trình là sai quá rõ ràng, như vậy bạn hoàn toàn xem xét không đúng chỗ, hoặc đúng chỗ nhưng không thấy được vấn đề, thì trình gỡ rối sẽ giúp bạn suy nghĩ một cách đúng đắn. Các lỗi do "mô hình trí óc" này là một trong số các lỗi khó phát hiện nhất; cơ chế sửa lỗi này là vô giá.

Sự quan niệm sai đôi khi rất đơn giản: độ ưu tiên toán tử không đúng, hay toán tử sai, hay việc canh hàng không khớp với cấu trúc thực tế, hay lỗi về phạm vi của biến, hay việc lẫn lộn cách dùng giữa biến cục bộ và biến toàn cục. Ví dụ, các lập trình viên thường quên rằng toán tử `&` và `|` có độ ưu tiên thấp hơn toán tử `==` và `!=`. Khi viết một đoạn mã nguồn như sau:

```
?       if (x & 1 ==0)
?           ...
```

và không biết được tại sao điều này luôn luôn sai. Đôi khi một cái trượt ngón trên bàn phím cũng làm chuyển đổi một dấu = thành hai hay ngược lại:

```
?       while ((c == getchar()) != EOF)
?           if (c = '\n')
?               break;
```

Hay mã nguồn dư thừa được bỏ qua trong quá trình soạn thảo

```
?       for (i   0; i < n ; i++);
?           a[i++] = 0;
```

Hay gây ra lỗi do gõ vào cấu thả:

```
?     switch (c) {
?           case '<':
?                 mode   LESS;
?                 break;
?           case '>':
?                 mode = GREATER;
?                 break;
?           defualt:
?                 mode   EQUAL;
?                 break;
?     }
```

Đôi khi lỗi liên quan đến các đối số được đưa vào không đúng thứ tự trong một tình huống mà việc kiểm tra kiểu không phát hiện được, như sau:

```
?     memset(p, n, 0);  /* luu n số 0 vào p*/
```

thay vì

```
      memset(p, 0, n);  /* luu n số 0 vào p*/
```

Đôi khi có một vài thay đổi bạn không kiểm soát được – các biến toàn cục bị thay đổi và bạn không nhận biết được rằng một vài tiến trình khác có thể truy xuất đến chúng.

Đôi khi giải thuật hoặc cấu trúc dữ liệu của bạn có một rất nghiêm trọng và bạn không thể nhìn thấy được. Trong khi chuẩn bị các thao tác trên các danh sách liên kết, ta đã viết một tập các hàm về danh sách cho phép tạo mới một phần tử ở đầu hoặc cuối các danh sách...; các hàm này được trình bày trong Chương 2. Dĩ nhiên chúng tôi đã viết một chương trình kiểm tra để chắc chắn rằng mọi việc được thực hiện chính xác. Một số các kiểm tra

đầu tiên hoạt động tốt nhưng sau đó có một kiểm tra không chạy được. Nguyên nhân là do chương trình kiểm tra:

```
?       while (canf("%s %d", name, &value) != EOF)
{
?           p = newitem(name, value);
?           list1 = addfront(list1, p);
?           list2 = addend(list2, p);
?       }
?       for (p = list1; p != NULL; p = p->next)
?           printf("%s %d\n", p->name, p->value);
```

Thật ngạc nhiên khi thấy rằng vòng lặp đầu tiên đặt cùng nút `p` vào cả hai danh sách, do đó các con trỏ được cố gắng truy cập một cách vô vọng vào lúc chúng ta thực hiện in.

Cũng thật khó khăn để tìm thấy được lỗi loại này bởi vì ý nghĩ của bạn vạch ra một hướng suy nghĩ sai lệch, bạn cứ cho điều không đúng là đúng. Vì thế trình biên dịch sẽ giúp bạn, nó giúp bạn đi theo một hướng khác, theo những gì mà chương trình thực hiện, chứ không phải những gì bạn nghĩ là nó sẽ thực hiện. Thông thường vấn đề được chú ý là những gì không đúng so với cấu trúc của toàn bộ chương trình, và để tìm được lỗi bạn cần phải quay trở lại các suy luận ban đầu của bạn.

Hãy chú ý rằng trong ví dụ về danh sách này, lỗi xảy ra trong mã nguồn kiểm tra, khiến cho việc tìm lỗi khó hơn nhiều. Rất hay gặp trường hợp tìm kiếm lỗi ở nơi mà lỗi không xuất hiện, vì chương trình kiểm tra sai, hoặc do kiểm tra trên phiên bản sai của chương trình, hoặc do không cập nhật hay biên dịch lại trước khi kiểm tra.

Nếu bạn không thể tìm thấy lỗi sau khi đã tìm kiếm một cách kỹ lưỡng, hãy nghỉ giải lao. Hãy thư giãn và làm chuyện khác. Thảo luận với

một người bạn và yêu cầu được giúp đỡ. Bạn có thể có được câu trả lời, nhưng nếu không, bạn sẽ không đi theo con đường cũ ở lần gỡ rối kế tiếp.

Nếu như trong một thời gian dài mà vẫn không phát hiện được lỗi thì vấn đề thực sự là trình biên dịch hay thư viện hay hệ điều hành hay thậm chí phần cứng, đặc biệt là nếu có một vài thay đổi trong môi trường ngay trước khi một lỗi xuất hiện. Bạn đừng bao giờ vội vàng đổ lỗi cho một trong các vấn đề trên, nhưng sau khi đã loại ra mọi thứ, thì mới có thể nghĩ tới chúng. Có lần chúng tôi phải di chuyển một chương trình định dạng văn bản lớn từ máy Unix nguyên thủy của nó sang một máy PC. Chương trình được biên dịch không có một sự khác biệt nào, nhưng lại hoạt động theo một cách thức rất khác thường: nó cắt xuống dòng gần như là mỗi hai ký tự của dữ liệu nhập. Suy nghĩ đầu tiên của chúng tôi là một vài thuộc tính có kiểu số nguyên 16 bit thay vì 32 bit, hay có thể là một vài vấn đề liên quan đến thứ tự khác thường của các byte. Nhưng nhờ vào việc in ra các ký tự trong vòng lặp chính, cuối cùng chúng tôi đi dần đến một lỗi trong tập tin `ctype.h`, được cung cấp bởi nhà cung cấp trình biên dịch. Nó cài đặt `isprint` như là một macro hàm

```
?       #define isprint(c) ((c) >= 040 && (c) <
0177)
```

và vòng lặp nhập liệu như sau:

```
?       while (isprint(c = getchar()))
?           ...
```

Mỗi khi một ký tự nhập vào là khoảng trắng (040, một cách viết tối là `' '`) hay lớn hơn, điều này hầu như lúc nào cũng gặp, hàm `getchar` được gọi lần thứ hai bởi vì macro tính toán đối số của nó hai lần, và ký tự nhập vào đầu tiên luôn luôn mất. Mã nguồn nguyên thủy không được rõ ràng như chúng ta nghĩ -- có quá nhiều sai sót trong điều kiện lặp – nhưng sai sót trong tập tin `header` của nhà cung cấp là không thể tha thứ được.

Ngày nay bạn vẫn có thể tìm thấy các phiên bản của vấn đề này; macro sau đây được lấy từ các tập tin `header` hiện tại của một nhà cung cấp

khác:

```
?       #define _iscsym(c) (isalnum(c) || ((c) == '_'))
```

Bộ nhớ "rò rỉ" – không thể trả về bộ nhớ không còn được dùng đến nữa – là nguồn gốc của các thao tác bất thường. Một vấn đề nữa là quên đóng các tập tin, đến lúc bảng quản lý tập tin đang mở đầy thì chương trình không thể mở thêm một tập tin nào nữa. Các chương trình có các lỗi "rò rỉ" thường bị treo một cách bí ẩn bởi vì chúng cạn kiệt một số tài nguyên và ta không thể tạo ra lại được các lỗi đặc trưng này.

Đôi khi chính phần cứng có vấn đề. Năm 1994, lỗi dấu chấm động trong bộ xử lý Pentium làm cho một số tính toán nào đó tạo ra các kết quả sai, là một lỗi được phổ biến rộng rãi và đắt giá trong thiết kế phần cứng, nhưng khi nó được phát hiện, thì dĩ nhiên nó được tạo lại một cách đúng đắn. Một trong các loại lạ nhất mà chúng tôi từng thấy có liên quan đến một trình tính toán, xảy ra trên một hệ thống có hai bộ xử lý, cách đây lâu rồi. Đôi khi diễn đạt ½ sẽ in ra 0,5 và đôi khi nó lại in ra giá trị vẫn đảm bảo tính thống nhất nhưng hoàn toàn sai 0,7432: không có một quy luật nào để cho biết kết quả có được là đúng hay sai. Vấn đề này cuối cùng được truy ra là một lỗi của đơn vị chấm động trong một trong các bộ xử lý. Khi trình tính toán được chạy ngẫu nhiên trên hoặc một bộ xử lý này hoặc một bộ xử lý còn lại, thì các kết quả là hoặc chính xác hoặc vô lý.

5.5. Các lỗi không có khả năng xuất hiện lại

Các lỗi không xuất hiện theo một cách thức nào hết thì rất khó giải quyết, và thông thường vấn đề sẽ không rõ ràng như là lỗi phần cứng. Tuy nhiên hành vi này thật ra là do thông tin của chính chương trình; nghĩa là lỗi này ít có khả năng do sai sót trong thuật toán của bạn, mà do theo một cách thức nào đó, mã nguồn của bạn sử dụng thông tin bị thay đổi qua mỗi lần chạy chương trình.

Hãy kiểm tra xem tất cả các biến được khởi tạo hết chưa; có thể là bạn đã lấy một giá trị ngẫu nhiên từ một giá trị bất kỳ được lưu trước đó trong cùng vị trí bộ nhớ. Các biến cục bộ của các hàm và bộ nhớ được tạo ra

từ các chức năng cấp phát thường là các thủ phạm trong C và C++. Gán các giá trị được biết cho các biến.

Nếu lỗi làm thay đổi hành vi hay thậm chí không xuất hiện khi mã nguồn gỡ rối được thêm vào, thì nó có thể là do lỗi cấp phát bộ nhớ - bạn đã viết ra ngoài vùng nhớ được cấp phát ở một chỗ nào đó, và việc thêm vào mã gỡ rối làm thay đổi hình thức lưu trữ đủ để làm thay đổi sự tác động của lỗi. Phần lớn các hàm xuất dữ liệu, từ hàm `printf` cho đến các hộp thoại, là tự cấp phát bộ nhớ, góp phần làm cho vấn đề phức tạp thêm.

Nếu dường như không có một vấn đề gì có thể bị sai ở nơi xảy ra lỗi, thì vấn đề thường là viết chồng lên bộ nhớ bằng cách lưu vào một vị trí bộ nhớ không được sử dụng trong suốt một khoảng thời gian dài. Đôi khi đây là một vấn đề về quản lý con trỏ lỏng lẻo, ở đó một con trỏ trỏ tới một biến cục bộ được trả về một cách vô tình từ một hàm, và nó được dùng đến. Việc trả về điều khiển của một biến cục bộ là một trong những nguyên nhân dẫn đến loại lỗi này.

```
?     char *msg(int n, char *s)
?     {
?           char buf[100];
?
?           sprintf(buf, "error %d: %s\n", n, s);
?           return buf;
?     }
```

Vào lúc con trỏ được trả về bởi hàm `msg` được sử dụng, thì nó không còn trỏ tới vùng lưu trữ có ý nghĩa nữa. Bạn phải cấp phát vùng nhớ bằng hàm `malloc`, dùng một mảng tĩnh, hay yêu cầu đối tượng gọi cung cấp không gian.

Việc dùng một giá trị đã được cấp phát một cách linh động sau khi nó được giải phóng sẽ có các triệu chứng tương tự. Chúng ta đã đề cập điều

này trong Chương 2 khi viết hàm `freeall`. Đoạn mã sau đây là sai:

```
?       for (p = listp; p != NULL; p = p->next)
?           free(p);
```

Khi bộ nhớ được giải phóng, nó phải không được dùng tới vì các nội dung của nó có thể thay đổi và không đảm bảo `p->next` vẫn còn trỏ tới một vị trí đúng.

Trong một số cài đặt của hàm `malloc` và `free`, việc giải phóng một phần tử hai lần sẽ làm phá vỡ cấu trúc dữ liệu bên trong nhưng phải sau một thời gian thì vấn đề rắc rối mới phát sinh khi một lời gọi sau đó rơi vào các hư hỏng xảy ra trước đó. Một số trình cấp phát có kèm theo các lựa chọn gỡ rối, các lựa chọn này có thể được kích hoạt để kiểm tra tính thống nhất cục bộ tại mỗi lần gọi; chúng được kích hoạt nếu bạn gặp một lỗi không kiểm soát được. Nếu không có chức năng này, bạn có thể viết trình cấp phát của riêng bạn để thực hiện một vài kiểm tra tính thống nhất của chính nó hay để lưu trữ tất cả các lời gọi cho việc phân tích riêng. Một trình cấp phát mà không cần phải chạy nhanh thường dễ viết, do đó chiến lược này là khả thi khi tình huống trở nên quá tồi tệ. Cũng có một số sản phẩm thương mại tốt cho việc kiểm tra việc quản lý bộ nhớ và bắt các lỗi và các "rò rỉ": việc viết hàm `malloc` và `free` cho riêng bạn có thể cho bạn một vài lợi ích nếu bạn không có các hàm này hay không thể truy cập vào bên trong chúng.

Khi một chương trình hoạt động tốt cho một người này nhưng không tốt cho một người khác, là do một số vấn đề phụ thuộc vào môi trường bên ngoài của chương trình. Nó có thể là do tập tin đưa vào làm dữ liệu nhập cho chương trình, các thao tác trên tập tin, các biến môi trường, đường dẫn tới các lệnh, các mặc định, hay các tập tin khởi tạo. Rất khó để thảo luận về các tình huống này.

Bài tập 5-1. Viết một phiên bản của hàm `malloc` và `free` có thể được dùng cho việc gỡ rối các vấn đề về quản lý lưu trữ. Một cách tiếp cận là kiểm tra toàn bộ vùng làm việc trên mỗi lời gọi tới hàm `malloc` và `free`; cách tiếp cận khác là viết các thông tin lưu lại có thể được xử lý bởi một

chương trình khác. Với mỗi cách, thêm các đánh dấu vào đầu và cuối của mỗi khối được cấp phát để phát hiện ra các trường hợp vượt ra ngoài giới hạn của hai đầu.

5.6. Công cụ gỡ rối

Các trình gỡ rối không phải là các công cụ duy nhất để tìm kiếm lỗi. Nhiều chương trình có thể giúp chúng ta tìm trong đống dữ liệu xuất để lấy ra các bit quan trọng, tìm ra những điều bất thường hoặc sắp xếp lại dữ liệu để xem xét được dễ dàng hơn những gì đang xảy ra. Nhiều chương trình loại này là một phần của các bộ công cụ chuẩn: một số được viết để giúp tìm kiếm lỗi hoặc để phân tích một chương trình nào đó.

Trong phần này ta sẽ mô tả một chương trình đơn giản gọi là các chuỗi mà chúng đặc biệt hữu ích cho việc tìm kiếm các tập tin có chứa nhiều ký tự không thuộc bảng chữ cái, chẳng hạn các tập tin thực thi được (*.EXE) hoặc các tập tin nhị phân được một số chương trình xử lý văn bản hỗ trợ. Thông thường có các thông tin ẩn bên trong như một đoạn văn bản hay các thông báo lỗi và các tùy chọn không thông báo, hoặc tên các tập tin, thư mục hoặc các tên của các hàm mà chương trình có thể gọi.

Ta cũng tìm thấy các chuỗi hữu ích cho việc định vị các văn bản trong các tập tin nhị phân. Các tập tin ảnh thường chứa các chuỗi ASCII để chỉ định chương trình tạo ra chúng, và các tập tin nén và các tập tin lưu trữ (chẳng hạn các tập tin nén) có thể chứa các tên tập tin, các chuỗi cũng có thể tìm thấy các văn bản trong các tập tin này.

Các hệ thống Unix cung cấp sẵn một chương trình các chuỗi, cho dù nó hơi khác so với các chuỗi ở trên. Nó nhận ra khi dữ liệu nhập của nó là một chương trình và chỉ xét những đoạn văn bản và dữ liệu, bỏ qua các bảng biểu tượng. Nhưng nó có tùy chọn, cho phép chọn đọc hết toàn bộ tập tin.

Để hiệu quả, chuỗi trích văn bản ASCII từ tập tin nhị phân vì vậy văn bản có thể được các chương trình khác đọc hay xử lý. Nếu một thông báo lỗi không mang một dấu hiệu nhận diện nào, nó có thể không xác định được chương trình nào đã phát sinh ra nó. Trong trường hợp đó, dùng lệnh

để tìm kiếm trong các thư mục có thể có:

```
% strings *.exe *.dll | grep 'mystery message'
```

có thể tìm ra chương trình tạo ra nó.

Chức năng của chuỗi đọc một tập tin và in ra tất cả các đường chạy với ít nhất có `MINLEN = 6` các ký tự có thể in ra.

```
        /* Hàm strings: trích các chuỗi có thể in ra từ
stream */
        void strings(char *name, FILE *fin)
        {
              int c, i;
              char buf[BUFSIZ]
              do {
                    for (i = 0; (c = getc(fin)) != EOF;
){
                          if (!isprint(c))
                              break;
                          buf[i++] = c;
                          if (i >= BUFSIZ)
                              break;
                    }
                    if (i >= MINLEN) /* in chuỗi buf nếu nó
đủ dài */
                        printf("%s:%.*s\n", name, i, buf);
              } while (c != EOF);
```

Lệnh `printf` định dạng in chuỗi `%.*s` lấy chiều dài chuỗi từ tham số `i` kế tiếp, vì chuỗi `buf` không kết thúc bằng *null*.

Vòng lặp `do-while` tìm và sau đó in ra từng chuỗi, kết thúc khi gặp `EOF`. Kiểm tra kết thúc tập tin ở cuối giúp cho vòng lặp `getc` và vòng lặp `string` cùng chung điều kiện kết thúc và để cho hàm in chuỗi xử lý kết thúc chuỗi, kết thúc tập tin và chuỗi quá dài.

Một vòng lặp ngoài theo chuẩn với việc kiểm tra ở đầu hoặc một vòng lặp đơn `getc` với thân chương trình phức tạp hơn, sẽ đòi hỏi thêm lệnh `printf`. Hàm này bắt đầu hoạt động theo cách đó, nhưng nó có một lỗi trong câu lệnh `printf`. Ta đã chỉnh sửa ở một chỗ nhưng quên chỉnh ở hai chỗ khác. (Ta có phạm sai lầm tương tự ở chỗ nào khác không?). Đến đây, điều đó trở nên rõ ràng rằng chương trình cần phải được viết lại để có ít đoạn mã lặp lại hơn: điều này dẫn đến vòng lặp `do-while`.

Hàm `main` của `strings` gọi hàm `strings` đối với mỗi tập tin đối số của nó:

```
        /* strings main: tìm chuỗi có thể in được trong
    tập tin */
        int main(int argc, char *argv[])
        {
              int i;
              FILE *fin;

              setprogname("strings");
              if (argc == 1)
                    eprintf("usage: strings filenames");
              else {
                  for (i = 1; i < argc; i++) {
```

```
        if ((fin = fopen(argv[i], "rb")) == NULL )
            weprintf("can't open %s:", argv[i] );
        else {
            strings(argv[i], fin);
            fclose(fin);
        }
    }
    return 0;
}
```

Bạn có thể ngạc nhiên rằng hàm `strings` không đọc dữ liệu nhập chuẩn của nó nếu không có tập tin. Để giải thích tại sao nó không thực hiện, ta cần mô tả việc gỡ rối.

Trường hợp kiểm tra rõ ràng cho hàm `strings` là chạy chính chương trình này. Chương trình chạy tốt trên Unix, tuy nhiên trên nền Windows 95 lệnh

```
C:\> strings <strings.exe
```

cho ra đúng 5 dòng kết quả:

```
!This program cannot be run in DOS mode.
`.rdata
@.data
.idata
.reloc
```

Dòng đầu tiên trông giống như một thông báo lỗi và ta phải tốn một ít thời gian trước khi nhận ra rằng nó chính là một dòng của kết quả chương trình, theo như nó chạy. Nó không biết có một quá trình cố gắng gỡ rối được thực hiện vì hiểu nhầm đó là câu thông báo lỗi.

Nhưng có nhiều kết quả được xuất ra hơn. Nó ở đâu ra? Ta dần dần hiểu được vấn đề. Đây chính là vấn đề tính khả chuyển được mô tả chi tiết hơn trong Chương 8. Ban đầu, chúng ta đã viết chương trình chỉ để đọc từ dữ liệu nhập chuẩn sử dụng hàm `getchar`. Tuy nhiên, trên Windows hàm `getchar` trả về `EOF` khi nó gặp phải byte đặc biệt (`0x1A` hoặc Control-Z) trong dữ liệu nhập dạng văn bản và điều này gây ra việc chương trình dừng sớm.

Đây tất nhiên là một cách xử lý hợp lý, nhưng nó không giống như những gì chúng ta mong đợi trong môi trường Unix. Kết quả là việc mở một tập tin theo nhị phân dùng `"rb"`. Nhưng `stdin` đã mở rồi và không có lệnh C chuẩn nào để thay đổi trạng thái này. (Các hàm như `fdopen` hay `setmode` có thể được dùng nhưng chúng không phải lệnh C chuẩn). Cuối cùng ta gặp phải một loạt các thay thế khó chấp nhận ép buộc người dùng nhập tên tập tin để nó có thể chạy tốt trên môi trường Windows nhưng không tương thích trên Unix; và mặc nhiên đưa ra các kết quả sai nếu người sử dụng Windows thử đọc từ dữ liệu nhập chuẩn; hoặc dùng sự biên dịch có điều kiện để tạo ra các kết quả thích hợp với các hệ thống khác nhau, trả giá cho việc giảm đi tính tương thích. Ta chọn sự lựa chọn đầu vì vậy cùng một chương trình có thể làm việc như nhau ở mọi nơi.

Bài tập 5-2. Chương trình `strings` in ra các chuỗi với `MINLEN` hay nhiều ký tự hơn, điều này thỉnh thoảng xuất ra nhiều kết quả hơn. Hãy nhập các chuỗi với tham số tùy chọn để định nghĩa chiều dài chuỗi tối thiểu.

Bài tập 5-3. Viết chương trình `vis`, chép dữ liệu nhập vào dữ liệu xuất, ngoại trừ nó hiển thị các byte không in được như các ký tự *backspace*, các ký tự điều khiển và các ký tự không phải ASCII như `\Xhh` trong đó `hh` là thể hiện hệ 16 của byte không in được. Trái ngược với `strings`, `vis` thật sự hữu ích cho việc xem xét các dữ liệu nhập chỉ chứa vài ký tự không in được.

Bài tập 5-4. `vis` đưa ra kết quả gì nếu dữ liệu nhập là `\XCA`? Bạn làm thế nào để có thể tạo kết quả của `vis` trở nên mơ hồ?

Bài tập 5-5. Mở rộng chương trình `vis` để xử lý một loạt các tập tin, nhóm các dòng dài tại bất cứ cột nào và loại bỏ các ký tự không in được một cách hoàn toàn. Những đặc trưng nào khác có thể thích hợp với vai trò của chương trình?

5.7. Lỗi của người khác

Thực tế hầu hết các lập trình viên đều không mong muốn việc tạo ra một hệ thống hoàn toàn mới từ đầu. Thay vào đó, họ bỏ ra nhiều giờ sử dụng, bảo trì, thay đổi và sau đó không tránh khỏi việc gỡ rối các đoạn mã do người khác viết.

Khi gỡ rối mã nguồn của người khác, ta sẽ áp dụng mọi thứ như khi ta gỡ rối mã nguồn của chính mình. Trước khi bắt đầu, ta phải hiểu được một chương trình được tổ chức như thế nào và các lập trình viên ban đầu đã nghĩ và viết như thế nào.

Đây là nơi mà các công cụ có thể giúp một cách đáng kể. Các chương trình tìm kiếm văn bản như `grep` có thể tìm thấy tất cả các xuất hiện của các tên. Các tham khảo chéo cho một số ý tưởng cho cấu trúc chương trình. Sự hiển thị của đồ thị của các lời gọi hàm là có giá trị nếu nó không quá lớn. Từng bước duyệt qua chương trình, một lời gọi hàm tại thời điểm khi gỡ rối có thể cho thấy một chuỗi các biến cố. Một quá trình chạy của chương trình có thể có một số mẫu chốt thông qua việc hiển thị những gì được thực hiện bởi chương trình qua thời gian. Các thay đổi thường xuyên là một dấu hiệu của dòng lệnh khó hiểu hoặc phải chịu các yêu cầu thay đổi và do đó dễ bị lỗi.

Thỉnh thoảng bạn cần phải theo dõi các lỗi trong phần mềm mà bạn không phải chịu trách nhiệm và không có mã nguồn của nó. Trong trường hợp này, nhiệm vụ của bạn là định rõ đặc điểm của lỗi một cách hiệu quả và bạn có thể thực hiện thông báo lỗi một cách chính xác, và khi đó bạn nên tránh các vấn đề khác nảy sinh.

Nếu bạn nghĩ rằng bạn đã tìm thấy một lỗi trong một chương trình của ai khác, bước đầu tiên là đảm bảo rằng đó là lỗi thực sự, vì thế bạn không phí thời gian của tác giả và uy tín của mình.

Khi bạn tìm thấy một lỗi biên dịch thì đảm bảo rằng lỗi là thật sự của trình biên dịch và không phải trong mã nguồn của bạn. Chẳng hạn, phép toán dịch bit sang phải sẽ điền vào các bit trống (dịch chuyển logic) hoặc truyền bit dấu (dịch số học) là không xác định trong C và C++. Do đó những người mới học thỉnh thoảng nghĩ rằng có lỗi nếu có lệnh như sau:

```
?       i = -1;

?       printf("%d\n", i >> 1);
```

cho ra một kết quả không xác định được. Nhưng điều này là vấn đề tương thích, vì phát biểu trên có thể hoạt động hợp lý nhưng khác nhau trên các hệ thống khác nhau. Thử kiểm tra sự hiểu của bạn trên nhiều hệ thống khác nhau và đảm bảo rằng bạn hiểu những gì xảy ra.

Đảm bảo lỗi là mới. Bạn có phiên bản mới nhất của chương trình không? Bạn đã có danh sách các lỗi đã sửa không? Hầu hết các phần mềm có nhiều phiên bản: nếu bạn tìm thấy một lỗi trong phiên bản 4.0b1, nó có thể được sửa hoặc thay thế bởi phiên bản mới như 4.04b2.

Cuối cùng, bạn phải tự đặt mình trong hoàn cảnh của người nhận báo cáo của bạn. Bạn muốn cung cấp cho người sở hữu nó một chương trình được kiểm tra tốt nhất có thể. Điều này không thật sự hữu ích nếu lỗi có thể được thể hiện với dữ liệu nhập lớn; hoặc trong một môi trường phức tạp, hoặc dữ liệu nhập từ nhiều tập tin. Đối với phiên bản bị lỗi của `isprint` được đề cập trong phần 5.4, ta có thể cung cấp một chương trình kiểm tra như sau:

```
/* Kiểm tra chương trình bị lỗi ở hàm isprint */

int main(void)

{

        int c;
```

```
        while (isprint(c = getchar()) || c != EOF)
        printf("%c", c);
        return 0;
}
```

Bất kỳ dòng nào của văn bản in được cũng đều là trường hợp để kiểm tra, vì dữ liệu xuất chỉ chứa một nửa dữ liệu nhập:

```
% echo 1234567890 | isprint_test
24680
%
```

Các thông báo lỗi tốt là các thông báo mà chỉ cần một hoặc hai dòng dữ liệu nhập trên một hệ thống mà có thể biểu diễn được lỗi và nó bao gồm cả việc sửa lỗi.

5.8. Tổng kết

Với thái độ đúng mực, việc gỡ rối có thể vui, giống như giải quyết một câu đố cho dù bạn có thích hay không, gỡ rối là một nghệ thuật mà chúng ta phải luyện tập thường xuyên. Thế nhưng điều đó rất tốt đẹp nếu các lỗi không xảy ra, vì vậy chúng ta cố gắng tránh chúng bằng cách viết các lệnh thật tốt ngay từ đầu. Mã nguồn được viết tốt có ít lỗi hơn từ khi bắt đầu và chúng sẽ dễ tìm thấy hơn.

Một khi thấy lỗi, điều đầu tiên phải làm là nghĩ kỹ về các nguồn gốc sinh ra nó. Làm thế nào mà gây ra lỗi? Nó có quen không? Có điều gì đó vừa mới đưa vào chương trình không? Có điều gì đặc biệt với dữ liệu nhập mà phát sinh ra lỗi không?

Nếu không tìm được nguồn gốc gây ra lỗi thì việc suy nghĩ kỹ vẫn là bước khởi đầu tốt nhất, sau đó thực hiện các thao tác thử sai một cách hệ thống để thu hẹp lại vị trí của vấn đề gây ra lỗi. Tất cả những điều đó là các

trường hợp của chiến lược tổng quát, chia để trị. điều này rất hữu ích trong quá trình gỡ rối giống như trong chính trị và trong chiến tranh.

Sử dụng các nguồn trợ giúp khác. Giải thích mã lệnh của bạn cho một ai đó là điều thật sự hiệu quả. Sử dụng chương trình gỡ rối để quan sát ngăn sếp. Sử dụng một số công cụ thương mại để kiểm tra việc rò rỉ bộ nhớ, tràn mảng, mã nguồn có khả năng gây ra lỗi và những thứ tương tự. Chạy từng bước chương trình của bạn để xác minh rằng bạn đã có suy nghĩ đúng hoặc sai về mục đích của mã nguồn.

Biết chính mình và tất cả các lỗi mình sai phạm. Một khi bạn tìm thấy và sửa lỗi, đảm bảo rằng bạn loại bỏ tất cả các lỗi tương tự khác. Suy nghĩ về những gì đã xảy ra để mà bạn có thể tránh việc lặp lại các lỗi như thế.

Chương 6

KIỂM CHỨNG

Kiểm chứng và gỡ rối thường được hiểu là như nhau, nhưng thực ra chúng không phải là như nhau. Để đơn giản, gỡ rối là những việc bạn cần phải làm khi chương trình bị lỗi. Kiểm chứng là sự kiểm tra có hệ thống nhằm tìm ra các lỗi của chương trình mà bạn nghĩ là nó đang thực hiện đúng.

Edsger Dijkstra nhận xét rằng việc kiểm chứng có thể giải thích sự hiện diện của các lỗi nhưng không thể cho biết được sự tiềm ẩn của chúng. Ông hy vọng rằng các chương trình có thể được viết một cách đúng đắn trong quá trình phát triển chúng để chương trình không còn lỗi và vì vậy không cần kiểm chứng nữa. Mặc dù đây là một mục tiêu tốt đẹp nhưng nó không thực tế. Do vậy, trong chương này chúng ta sẽ chú trọng đến những cách thức kiểm chứng để tìm ra lỗi nhanh chóng và hiệu quả.

Nghĩ đến các lỗi tiềm ẩn trong khi viết chương trình là một bước khởi đầu tốt. Việc kiểm chứng có hệ thống từ đơn giản đến các phức tạp bảo đảm rằng các chương trình có thể thực hiện đúng đắn và vẫn còn đúng khi mở rộng chúng. Sự tự động hóa giúp loại bỏ các thao tác thủ công và khuyến khích việc kiểm chứng bao quát. Các nhà lập trình đã và đang tích lũy nhiều mẹo lập trình từ kinh nghiệm thực tế.

Một cách để viết mã nguồn không còn lỗi là dùng một chương trình phát sinh ra nó. Nếu nhiệm vụ lập trình được hiểu rõ đến mức có thể viết mã nguồn tự động thì chúng nên được phát sinh một cách tự động. Thông thường, một chương trình có thể được phát sinh từ một đặc tả bằng một số ngôn ngữ đặc trưng. Chẳng hạn, ta biên dịch ngôn ngữ cấp cao sang hợp

ngữ; ta sử dụng những biểu thức thông thường để định rõ những mẫu của văn bản; sử dụng các ký hiệu như `SUM(A1:A50)` để biểu diễn các phép toán trên một các phần tử trong bảng tính. Trong trường hợp này, nếu bộ phát sinh hoặc bộ biên dịch đúng và nếu đặc tả đúng thì kết quả của chương trình cũng sẽ đúng. Chương 9 sẽ trình bày kỹ hơn về vấn đề này; ở đây ta sẽ tóm tắt các cách thực hiện kiểm chứng từ những bản đặc tả đầy đủ.

6.1. Kiểm tra trong khi viết mã nguồn

Một lỗi được tìm thấy càng sớm càng tốt. Nếu bạn nghĩ một cách có hệ thống về những gì bạn sẽ viết, bạn sẽ xác định được các thuộc tính đơn giản của chương trình ngay khi đang xây dựng nó. Kết quả là mã nguồn của bạn sẽ được kiểm chứng ít nhất một lần trước khi biên dịch. Nhờ đó một số loại lỗi xác định sẽ không bao giờ xuất hiện.

Kiểm chứng mã nguồn ở các giá trị biên của nó

Một kỹ thuật được gọi là kiểm tra điều kiện biên: khi viết một đoạn mã nguồn, ví dụ như một vòng lặp hay một biểu thức điều kiện, nếu việc kiểm tra điều kiện là đúng thì rẽ theo nhánh đúng hoặc vòng lặp sẽ thực hiện tiếp. Quá trình này gọi là việc kiểm tra điều kiện biên bởi vì bạn kiểm tra tại những biên tự nhiên bên trong chương trình và dữ liệu, chẳng hạn như giá trị nhập không tồn tại hoặc rỗng, một mảng đầy. Có ý kiến cho rằng hầu hết các lỗi xuất hiện tại biên. Nếu một đoạn mã nguồn sai, nó thường sai ở biên. Ngược lại, nếu nó có thể thực hiện đúng tại biên, nó cũng sẽ thực hiện tại các vị trí khác.

Đoạn chương trình này, mô phỏng hàm `fgets` đọc chuỗi các ký tự cho đến khi gặp ký tự kết thúc chuỗi hay đầy *buffer*:

```
?     int i;
?     char s[MAX];
?
?     for (i = 0; (s[i] = getchar()) != '\n' && i
```

```
< MAX-1; ++i)
        ?
        ?       s[--i] = '\0';
```

Tưởng tượng rằng bạn vừa viết xong vòng lặp này. Vòng lặp này sẽ thực hiện việc đọc vào một dòng. Kiểm tra biên đầu tiên là trường hợp đơn giản nhất: một dòng trống. Nếu bạn bắt đầu với một dòng không có ký tự nào, vòng lặp sẽ dừng ngay lần đầu tiên với giá trị của `i` là `0`, do đó giá trị của `i` sẽ giảm xuống và mang giá trị `-1` ở dòng lệnh cuối cùng và do đó sẽ gán giá trị rỗng vào `s[-1]` (`s[-1]` nằm trước vị trí bắt đầu của mảng). Việc kiểm tra điều kiện biên sẽ phát hiện ra lỗi này.

Nếu ta viết lại vòng lặp bằng cách gán các phần tử của mảng với các ký tự nhập vào từ bàn phím như sau:

```
        ?       for (i = 0; i < MAX-1; i++)
        ?               if ((s[i] = getchar()) == '\n')
        ?                       break;
        ?       s[i] = '\0';
```

Thực hiện lại kiểm tra biên ban đầu, ta dễ dàng chứng minh một dòng chỉ có một ký tự xuống dòng đã được xử lý chính xác khi đó giá trị của `i` là `0`, ký tự đọc được đầu tiên sẽ thoát khỏi vòng lặp, và `'\0'` sẽ được gán vào `s[0]`. Việc kiểm tra tương tự với 1 hoặc 2 ký tự nhập và kết thúc bằng dấu xuống dòng chứng tỏ vòng lặp làm việc tốt gần với biên đó.

Tuy nhiên, có những điều kiện biên khác cần phải kiểm tra. Nếu ta nhập vào một dòng dài hoặc dòng không có ký tự xuống dòng thì ta sẽ kiểm tra giá trị của `i` phải nhỏ hơn `MAX-1`. Nếu nhập vào dòng rỗng thì hàm `getchar()` sẽ trả về giá trị `EOF` ở lần gọi hàm đầu tiên. Ta buộc phải kiểm tra điều này:

```
        ?       for (i = 0; i < MAX-1; i++)
        ?           if ((s[i] = getchar()) == '\n' |  s[i] == EOF)
```

```
?                   break;
?       s[i]='\0';
```

Việc kiểm tra điều kiện biên có thể tìm ra được nhiều lỗi, nhưng không phải là tìm ra được tất cả. Ta sẽ trở lại ví dụ này trong Chương 8, và chỉ ra rằng vẫn còn những khi đúng khi sai.

Bước tiếp theo là kiểm tra dữ liệu nhập vào tại những biên khác, chẳng hạn tại những biên mà mảng gần đầy, đầy hoặc quá đầy, đặc biệt là nếu ký tự xuống dòng được nhập vào cùng thời điểm. Ta sẽ không viết ra chi tiết ở đây mà xem đây là bài tập hữu ích dành cho các bạn. Suy nghĩ về các biên sẽ đưa ra câu hỏi là cần phải làm gì khi gán các giá trị cho *buffer* trước khi nhập vào ký tự xuống dòng '\n'. Sự thiếu sót này trong đặc tả nên được giải quyết sớm, và các biên dùng để kiểm tra sẽ giúp dễ phát hiện nó.

Việc kiểm tra điều kiện biên sẽ có hiệu quả trong việc tìm và loại bỏ từng lỗi một. Trong thực tế nhiều lỗi được loại bỏ trước khi nó xảy ra.

Kiểm tra điều kiện trước và điều kiện sau

Một cách khác để tránh những lỗi là xác định những thuộc tính cần thiết đi trước (điều kiện trước) và sau (điều kiện sau) mã nguồn được thi hành. Các giá trị đầu vào phải thuộc một phạm vi cho trước là một ví dụ thông thường để kiểm tra điều kiện trước. Hàm tính giá trị trung bình của một mảng gồm n phần tử sẽ có lỗi nếu n nhỏ hơn hoặc bằng 0:

```
?       double avg(double a[], int n)
?       {
?               int i;
?               double sum;
?
?               sum = 0.0;
?               for (i = 0; i < n; i++)
?                       sum += a[i];
```

```
}                   return sum / n;
}
```

Hàm `avg` sẽ làm gì nếu `n` có giá trị `0`? Một mảng không có phần tử nào vẫn đầy đủ ý nghĩa mặc dù giá trị trung bình của nó không được xác định. Hàm `avg` có nên để cho hệ thống bắt những lỗi chia cho `0`? Chấm dứt? Thông báo lỗi? Trả về một giá trị bất kỳ nào đó? Còn nếu `n` có giá trị âm, thật vô lý nhưng vẫn có thể xảy ra? Như đã đề cập trong Chương 4, ta nên trả về giá trị `0` nếu `n` nhỏ hơn hay bằng `0`:

```
return n <= 0 ? 0.0 : sum/n;
```

nhưng đó không phải là câu trả lời đúng duy nhất.

Chắc chắn bạn sẽ nhận được câu trả lời sai khi bỏ việc kiểm tra tiền điều kiện. Một bài báo của *Scientific American* vào tháng 11 năm 1998 mô tả sự cố của một chiến hạm có chứa các tên lửa điều khiển *USS Yorktown*. Một thủy thủ đã nhầm lẫn khi nhập `0` cho một giá trị dữ liệu, dẫn đến lỗi chia cho `0`, lỗi xảy ra đã làm chao đảo và thậm chí làm hệ thống đẩy của tàu ngưng hoạt động. Yorktown bị chìm dần sau hai giờ vì chương trình đã không kiểm tra được tính hợp lệ của dữ liệu nhập vào.

Sử dụng các khẳng định (assert)

C và C++ cung cấp cơ chế khẳng định các điều kiện trước và sau dễ dàng trong thư viện `<assert.h>`. Nếu một sự khẳng định bị sai thì nó làm kết thúc chương trình, nó được dành riêng cho các tình huống xảy ra lỗi không ngờ tới hoặc không thể khắc phục được. Ta có thể làm tăng tính đúng đắn của mã nguồn bằng việc thêm khẳng định trước khi vào vòng lặp như sau:

```
assert(n > 0);
```

Nếu sự khẳng định bị vi phạm, nó sẽ làm cho chương trình ngừng thi hành với thông báo chuẩn sau:

```
Assertion failed: n > 0, file avgtest.c, line 7
Abort(crash)
```

Cơ chế khẳng định đặc biệt hữu ích trong việc kiểm tra tính đúng đắn của các thuộc tính của giao tiếp vì nó chú ý đến sự không nhất quán giữa hàm gọi thực hiện và hàm được gọi thực hiện, thậm chí có thể chỉ ra lỗi do hàm nào gây ra. Nếu sự khẳng định `n>1` là sai khi hàm được gọi, nó chỉ ra rằng hàm gây ra lỗi là hàm gọi thực hiện chứ không chỉ là lỗi của hàm `avg`. Nếu một giao tiếp thay đổi nhưng ta quên sửa các hàm phụ thuộc vào nó, sự khẳng định có thể bắt những lỗi này trước khi nó gây ra lỗi thực sự.

Hoàn thiện chương trình

Một kỹ thuật hữu ích là thêm mã nguồn để xử lý các trường hợp "không thể xảy ra", các tình huống không logic vì một trường hợp nào đó không thể xảy ra nhưng nó vẫn có thể xảy ra (do một vài lỗi xảy ra ở một nơi nào đó trong chương trình). Chẳng hạn như, việc kiểm tra số lượng phần tử của mảng được gán trong biến `avg` có giá trị là 0 hoặc âm. Một ví dụ khác, chương trình xử lý việc xếp loại học sinh cần phải kiểm tra là không có điểm âm hoặc điểm quá lớn:

```
    if (grade < 0 || grade > 100) /* các trường hợp không
thể xảy ra */
                letter = '?';
    else if (grade >= 90)
                letter = 'A';
    else
            ...
```

Đây là một ví dụ về lập trình bảo vệ: bảo đảm chương trình có thể ngăn chặn việc dùng sai hoặc dữ liệu không hợp lệ. Các con trỏ `NULL`, chỉ số vượt ra ngoài giới hạn, phép chia cho 0, và những lỗi khác có thể được phát hiện sớm và cảnh báo. Lập trình bảo vệ có thể bắt được lỗi của phép chia cho không trong sự cố *Yorktown*.

Kiểm tra lỗi dựa vào giá trị trả về

Một cách ngăn chặn lỗi tổng quát thường được sử dụng là kiểm tra

lỗi dựa trên giá trị trả về của các hàm thư viện và các lời gọi hệ thống. Giá trị trả về của các hàm đọc dữ liệu như `fread`, `fscanf` luôn được kiểm tra lỗi, cũng như việc kiểm tra lỗi của hàm thực hiện mở tập tin như hàm `fopen`. Nếu thao tác đọc hoặc mở tập tin có lỗi, quá trình tính toán không thể nào thực hiện chính xác được.

Kiểm tra giá trị trả về của các hàm xuất kết quả như hàm `fprintf` hay `fwrite` sẽ bắt những lỗi khi thực hiện ghi dữ liệu lên tập tin không còn chỗ trống trên đĩa. Ta cũng cần phải kiểm tra giá trị trả về của hàm `fclose`, *nó* trả về giá trị là `EOF` nếu có bất cứ lỗi nào xảy ra trong quá trình thao tác hoặc trả về giá trị `0` nếu không có lỗi.

```
fp = fopen(outfile, "w");
while (…)          /* ghi kết quả ra file */
      fprintf(fp,…);
if (fclose(fp) == EOF) {       /* xảy ra lỗi? */
      /* xuất kết quả thông báo lỗi */
}
```

Những lỗi đầu ra có thể rất nghiêm trọng. Nếu tập tin đang được ghi là phiên bản mới của tập tin chứa dữ liệu quan trọng thì việc kiểm tra này sẽ giúp bạn không xóa tập tin cũ nếu tập tin mới chưa được ghi thành công.

Chi phí dành cho việc kiểm chứng khi viết chương trình sẽ thấp và cho kết quả đáng kể. Nghĩ đến việc kiểm chứng trong khi viết chương trình sẽ cho bạn những đoạn mã nguồn tốt hơn, vì khi đó bạn biết chính xác đoạn chương trình đó nên thực hiện công việc như thế nào. Nếu bạn đợi cho đến khi lỗi phát sinh thì có thể bạn đã quên đoạn chương trình đó làm việc ra sao. Vì áp lực công việc, bạn buộc phải tính lại nó, điều này làm mất thời gian, và việc sửa chữa sẽ không triệt để vì bạn không hoàn toàn hiểu lại đoạn chương trình đó như lúc ban đầu.

Bài tập 6-1. Hãy kiểm tra các ví dụ sau ở các biên của chúng, sau đó sửa lại (nếu cần thiết) theo những nguyên tắc chuẩn trong Chương 1 và những lời khuyên trong chương này.

(a) Tính giai thừa:

```
?     int factorial(int n)
?     {
?           int fac;
?           fac = 1;
?           while (n--)
?                 fac *= n;
?           return fac;
?     }
```

(b) In từng ký tự của một chuỗi trên từng dòng:

```
?     i = 0;
?     do {
?           putchar(s[i++]);
?           putchar('\n');
?     } while (s[i] != '\0');
```

(c) Sao chép một chuỗi từ nguồn sang đích:

```
?     void strcpy(char *dest, char *src)
?     {
?           int i;
?
?           for (i = 0; src[i] != '\0'; i++)
?                 dest[i] = src[i];
?     }
```

(d) Sao chép *n* ký tự từ chuỗi `s` sang chuỗi `t`:

```
?     void strncpy(char *t, char *s, int n)
?     {
?           while (n>0 && *s != '\0') {
```

```
?                   *t = *s;
?                   t++;
?                   s++;
?                   n--;
?             }
?       }
```

(e) So sánh các số:

```
?       if (i > j)
?             printf("%d is greater than %d.\n", i, j);
?       else
?             printf("%d is smaller than %d.\n", i, j);
```

(f) Kiểm tra lớp ký tự:

```
?       if (c >= 'A' && c <= 'Z') {
?             if (c <= 'L')
?                   cout <<"first half of alphabet";
?             else
?                   cout <<"second half of alphabet";
?       }
```

Bài tập 6-2. Như ta đã biết, sự cố năm 2000 dường như là sự cố có điều kiện biên lớn nhất.

(a) Ngày tháng nào bạn sử dụng để kiểm tra liệu hệ thống có làm việc vào năm 2000 không? Giả sử rằng việc thực hiện kiểm tra này tốn nhiều chi phí, bạn thực hiện việc kiểm tra theo trật tự nào sau ngày 1.1.2000?

(b) Bằng cách nào để kiểm chứng hàm chuẩn `ctime`, nó trả về một chuỗi đại diện cho ngày tháng theo dạng sau:

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

Giả sử chương trình của bạn gọi hàm `ctime`. Bạn sẽ viết mã nguồn như thế nào để khắc phục lỗi cài đặt này.

(c) Hãy mô tả cách bạn kiểm chứng một chương trình in lịch như sau:

January 2000

S	M	Tu	W	Th	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

(d) Những biên thời gian nào khác mà bạn nghĩ bạn sử dụng trong hệ thống, và bằng cách nào bạn kiểm chứng chúng có được xử lý chính xác hay không?

6.2. Kiểm chứng có hệ thống

Việc kiểm chứng một chương trình một cách có hệ thống rất quan trọng, vì vậy bạn cần phải biết tại mỗi bước bạn cần kiểm chứng cái gì và kết quả bạn muốn làm gì. Bạn cần phải thực hiện một cách có thứ tự và bạn phải lưu lại những việc bạn đã làm để biết bạn đã làm được những gì.

Kiểm chứng tăng dần

Việc kiểm chứng cần phải đi đôi với việc xây dựng chương trình. Một người viết toàn bộ chương trình sau đó mới kiểm chứng tất cả sẽ gặp nhiều khó khăn và tốn nhiều thời gian hơn cách tiếp cận tăng dần. Viết một

phần của chương trình, kiểm chứng nó, viết thêm mã nguồn, kiểm chứng nó và cứ tiếp tục như vậy. Nếu bạn có hai phần được viết và kiểm chứng độc lập, khi bạn tích hợp chúng với nhau bạn chỉ cần kiểm tra chúng có hoạt động tốt với nhau không.

Chẳng hạn như, khi ta kiểm chứng chương trình *CSV* ở Chương 4, bước đầu tiên là viết mã nguồn vừa đủ để đọc giá trị nhập vào: điều này cho phép xử lý thực hiện dữ liệu nhập vào một cách đúng đắn. Bước tiếp theo là chia các dòng dữ liệu nhập này tại dấu phẩy. Khi những phần này hoạt động, ta chuyển sang xử lý phần trích dẫn, và dần dần tiến tới kiểm chứng mọi thứ.

Kiểm chứng những phần đơn giản trước

Cách tiếp cận tăng dần cũng được áp dụng để kiểm chứng các tính năng của chương trình. Việc kiểm chứng cần phải tập trung vào những phần đơn giản và những phần thường được thực hiện nhất của chương trình; chỉ khi chúng thực thi đúng bạn mới chuyển sang phần khác. Bằng cách này, ở mỗi giai đoạn, bạn đưa ra nhiều phần hơn để kiểm chứng và xây dựng với sự tin tưởng rằng nó sẽ hoạt động tốt. Những kiểm chứng đơn giản sẽ giúp tìm ra những lỗi đơn giản. Mỗi lần kiểm chứng góp phần tìm ra lỗi tiềm ẩn tiếp theo. Mặc dù khó phát hiện các lỗi về sau hơn nhưng việc sửa lỗi không hẳn là khó hơn.

Trong phần này, ta bàn về việc chọn những biện pháp kiểm chứng hiệu quả và thứ tự áp dụng chúng; trong hai phần tiếp theo, ta sẽ đề cập đến vấn đề làm sao để tự động hóa quy trình để việc thực hiện có hiệu quả. Đối với những chương trình nhỏ hoặc những hàm riêng lẻ, bước đầu tiên là việc mở rộng các kiểm chứng điều kiện biên ở phần trước và kiểm chứng có hệ thống các trường hợp nhỏ.

Giả sử rằng ta có một hàm thực hiện việc tìm kiếm nhị phân trên một mảng các số nguyên. Ta sẽ bắt đầu với những kiểm chứng sau, được sắp xếp theo mức độ phức tạp tăng dần:

- Tìm kiếm trên mảng không có phần tử nào

- Tìm kiếm trên mảng chỉ có một phần tử và giá trị cần tìm:
  - Nhỏ hơn phần tử duy nhất của mảng
  - Bằng phần tử duy nhất
  - Lớn hơn phần tử duy nhất
- Tìm kiếm trên mảng có hai phần tử và một giá trị cần tìm:
  - Kiểm tra tất cả năm vị trí có khả năng
- Kiểm tra việc thực hiện trên mảng gồm hai phần tử giống nhau và giá trị cần tìm
  - Nhỏ hơn giá trị trong mảng
  - Bằng giá trị trong mảng
  - Lớn hơn giá trị trong mảng
- Tìm kiếm trên mảng có ba phần tử tương tự như với hai phần tử
- Tìm kiếm trên mảng có bốn phần tử tương tự như với hai, ba phần tử

Nếu những kiểm chứng được thực hiện mà chương trình không có lỗi phát sinh thì có vẻ chương trình này chạy tốt nhưng nó vẫn cần được kiểm chứng nhiều hơn nữa.

Vì tập hợp những kiểm chứng này nhỏ nên ta có thể tiến hành thủ công, nhưng tốt hơn là nên tự động hóa quy trình này. Ta có thể quản lý chương trình điều khiển đơn giản. Nó đọc vào các dòng dữ liệu nhập bao gồm một khóa để tìm kiếm và kích thước của mảng. Đồng thời, nó sẽ tạo ra một mảng có kích thước đã cho chứa các giá trị 1, 3, 5, ... và thực hiện việc tìm khóa trên mảng này.

```
/* Hàm main của việc kiểm chứng hàm tìm kiếm nhị phân*/
int main(void)
{
      int i, key, nelem, arr[1000];
```

```
    while (scanf("%d %d", &key, &nelem) !=EOF {
        for (i = 0; i < nelem; i++)
            arr[i] = 2*i + 1;
        printf("%d\n", binsearch(key, arr, nelem));
    }
    return 0;
}
```

Điều này thật đơn giản nhưng nó cho thấy rằng việc thực hiện tự động thì hữu ích không phải cồng kềnh, và dễ dàng mở rộng về sau để thực hiện các kiểm chứng tương tự, và ít đòi hỏi con người can thiệp hơn.

Biết trước dữ liệu xuất

Trong tất cả các kiểm chứng, bạn cần phải biết kết quả đúng là gì; nếu không thì thật là lãng phí thời gian. Điều này dường như hiển nhiên, bởi vì trong nhiều chương trình, thật dễ dàng để xét xem chương trình có làm việc hay không. Chẳng hạn như, thao tác sao chép một tập tin thực hiện việc sao chép một tập tin hoặc không, dữ liệu xuất của một hàm sắp xếp có được sắp xếp hoặc không.

Quả thật là khó khăn để chỉ ra đặc điểm của hầu hết các chương trình - trình biên dịch (dữ liệu đầu ra có thực sự biên dịch đúng dữ liệu đầu vào không?), các thuật toán số học (kết quả có chứa những lỗi có thể bỏ qua hay không?), đồ họa (các pixel có ở đúng vị trí của nó không?)... Vì những lý do này, để đảm bảo dữ liệu xuất đúng ta thực hiện so sánh chúng với các giá trị đã biết.

- Để kiểm chứng một trình biên dịch, ta cần biên dịch và thực thi các tập tin thử nghiệm. Chương trình thử nghiệm ngược lại sẽ phát sinh các dữ liệu xuất, và các kết quả này sẽ được so sánh với những kết quả đã biết.
- Để kiểm chứng một chương trình số học, ta phát sinh các trường hợp thử nghiệm bao quát mọi trường hợp của thuật toán, từ những trường hợp đơn giản cho đến các trường hợp phức tạp. Nếu có thể, bạn viết

đoạn chương trình để kiểm tra tính đúng đắn các thuộc tính của dữ liệu xuất. Chẳng hạn như, giá trị xuất của một phép tính tích phân có thể được kiểm tra tính liên tục, và tính bị chặn của nó.

- Để kiểm chứng một chương trình đồ họa, nếu ta chỉ kiểm tra xem chương trình có thể vẽ một hình hộp hay không thì chưa đủ, thay vào đó ta phải đọc từ màn hình và kiểm tra xem các cạnh của chúng có nằm đúng vị trí hay không.

Nếu một chương trình có khả năng thực hiện theo chiều ngược lại thì cần kiểm tra rằng nó có thể khôi phục lại giá trị nhập hay không. Mã hóa và giải mã là hai quá trình ngược nhau, vì vậy nếu bạn có thể thực hiện mã hóa nhưng lại không thể giải mã được thì chứng tỏ có điều gì đó sai đã xảy ra. Tương tự, thuật toán nén không mất thông tin và giải nén là ngược nhau. Nhờ đó sau khi nén tập tin thì vẫn có thể giải nén tập tin đúng với ban đầu tập tin. Đôi khi có nhiều phương pháp đảo ngược; cần phải kiểm tra tất cả các trường hợp này.

Kiểm tra các thuộc tính lưu trữ

Nhiều chương trình dành lưu trữ các tính chất của dữ liệu nhập. Những công cụ như `wc` (đếm số hàng, số từ, số ký tự) và `sum` (tính tổng) có thể kiểm chứng được các dữ liệu xuất có cùng kích thước, có cùng số từ, có các byte giống nhau theo một thứ tự nào đó. Những chương trình so sánh các tập tin (`cmp`) hoặc đưa ra sự khác nhau của các tập tin (`diff`). Các chương trình này và các chương trình tương tự thích hợp với hầu hết các môi trường, và thu được kết quả tốt.

Một chương trình tuần tự theo byte có thể được dùng để kiểm tra sự lưu trữ dữ liệu và cũng để phát hiện những sự khác thường như những ký tự phi văn bản trong những tập tin văn bản. Sau đây là một phiên bản của chương trình `freq`:

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

```
unsigned long count[UCHAR_MAX+1];
/*Hàm main của freq: hiển thị tần xuất theo từng byte*/
int main(void)
{
      int c;
      while ((c = getchar()) != EOF)
            count[c]++;
      for (c = 0; c <= UCHAR_MAX; c++)
            if (count[c] != 0)
                printf("%.2x   %c   %lu\n",
                  c, isprint(c) ? c : '-', count[c]);
      return 0;
}
```

Các thuộc tính lưu trữ cũng có thể được kiểm tra bên trong một chương trình. Một hàm đếm các thành phần trong cấu trúc dữ liệu sẽ cho phép kiểm tra tính nhất quán giản đơn. Một bảng băm có thuộc tính là mỗi phần tử sau khi thêm vào bảng băm đều có thể truy xuất trở lại được. Ta có thể dễ dàng kiểm tra điều kiện này bằng cách tạo một hàm sao chép nội dung của bảng băm vào một tập tin hay một mảng. Tại bất kỳ thời điểm nào, số lượng các phần tử thêm vào trong cấu trúc dữ liệu trừ cho số lượng phần tử bị xoá bỏ phải bằng với số lượng phần tử hiện có trong cấu trúc dữ liệu đó. Ta dễ dàng kiểm tra được điều kiện này.

So sánh các quá trình cài đặt độc lập

Các quá trình cài đặt độc lập của một thư viện hay của một chương trình phải cho ra cùng kết quả. Chẳng hạn như, trong hầu hết các trường hợp, hai trình biên dịch sẽ phải tạo ra các chương trình hoạt động giống nhau trên cùng một máy.

Đôi khi một kết quả có thể được tính theo hai cách khác nhau, hoặc bạn có thể viết một phiên bản bình thường của chương trình sử dụng phép

so sánh chậm nhưng thực hiện độc lập. Nếu hai chương trình độc lập cho ra cùng kết quả thì có thể khẳng định cả hai chương trình đó đều đúng; nhưng nếu hai chương trình đó cho ra các kết quả khác nhau thì có ít nhất một chương trình bị sai.

Kiểm tra tính bao quát của việc kiểm chứng

Một mục tiêu của kiểm chứng là đảm bảo mọi câu lệnh trong chương trình đều được thi hành tại một thời điểm nào đó trong suốt quá trình kiểm chứng; việc kiểm chứng không thể xem là hoàn tất nếu mỗi dòng của chương trình chưa được kiểm tra ít nhất một lần. Tính bao quát hoàn toàn thường khó đạt được. Ngay cả khi bỏ qua các trường hợp "không thể xảy ra", việc sử dụng các dữ liệu nhập thông thường để buộc chương trình thực hiện các câu lệnh nào đó cũng gặp nhiều khó khăn.

Có một số phần mềm thương mại dùng để đo lường tính bao quát. Nó cung cấp cách tính tần xuất thực hiện của mỗi câu lệnh trong chương trình, và cho biết tính bao quát trong các lần kiểm chứng cụ thể.

Ta đã kiểm chứng chương trình *Markov* ở Chương 3 bằng cách kết hợp những kỹ thuật này. Phần cuối cùng của chương này mô tả chi tiết các quá trình kiểm chứng đó.

Bài tập 6-3. Hãy mô tả cách kiểm chứng chương trình `freq`.

Bài tập 6-4. Hãy thiết kế và cài đặt một phiên bản của chương trình `freq` để đo tần số của các kiểu giá trị dữ liệu khác, chẳng hạn các số nguyên 32 bit hoặc các số thực dấu chấm động. Bạn có thể tạo một phiên bản của chương trình để xử lý một số kiểu dữ liệu khác nhau hay không?

6.3. Kiểm chứng tự động

Quá trình kiểm chứng chương trình thủ công rất nhàm chán và không đáng tin cậy; ba quá trình kiểm chứng theo đúng nghĩa gồm: thực hiện việc kiểm chứng nhiều lần, nhiều bộ dữ liệu nhập và nhiều lần so sánh dữ liệu xuất. Vì vậy, quá trình kiểm chứng cần được thực hiện bằng chương trình, nó giúp ta không bị mệt mỏi cũng như tránh được sự bất cẩn có thể

xảy ra. Ta nên viết một chương trình bao gồm tất cả các quá trình kiểm chứng, vì vậy, một bộ chương trình kiểm chứng đầy đủ có thể được thực thi (theo cả nghĩa đen lẫn nghĩa bóng) bằng cách nhấn một nút duy nhất. Bộ chương trình kiểm chứng càng dễ chạy bao nhiêu thì nó càng được sử dụng thường xuyên bấy nhiêu, và càng ít khi bỏ qua nó khi có ít thời gian. Ta viết một bộ kiểm chứng để kiểm tra tất cả chương trình đã viết trong cuốn sách này, và chạy nó mỗi khi có sự thay đổi; các thành phần của bộ phần mềm này chạy một cách tự động sau khi biên dịch thành công.

Tự động hóa việc kiểm chứng lùi

Một dạng cơ bản của tự động là việc kiểm chứng lùi, nó thực hiện tuần tự các kiểm chứng so sánh phiên bản mới với những phiên bản cũ tương ứng. Khi sửa lỗi, khuynh hướng tự nhiên là kiểm tra việc sửa đổi đó có đúng hay không; thật dễ dàng để nhận ra được khả năng các thao tác sửa đổi làm hỏng những thứ khác. Mục đích của việc kiểm chứng lùi là đảm bảo việc sửa lỗi sẽ không làm ảnh hưởng những phần khác trừ khi chúng ta muốn như thế.

Một số hệ thống có nhiều công cụ trợ giúp tự động hóa; các ngôn ngữ *script* cho phép viết các đoạn *script* để thực hiện kiểm chứng tuần tự. Trong hệ điều hành Unix, các trình thao tác trên tập tin như `cmp` và `diff` so sánh các dữ liệu xuất, `sort` sắp xếp các phần tử, `grep` kiểm chứng dữ liệu xuất; `wc`, `sum` và `freq` tổng kết dữ liệu xuất. Nếu kết hợp chúng lại với nhau thì sẽ dễ dàng tạo ra được một mô hình kiểm chứng, nó có thể không kiểm chứng được đầy đủ cho các chương trình lớn nhưng hầu như đủ cho các chương trình được bảo trì bởi từng cá nhân hay một nhóm nhỏ.

Đây là bản mô tả cho việc kiểm chứng lùi một chương trình xóa gọi là `ka`. Nó chạy phiên bản cũ (`old_ka`) và một phiên bản mới (`new_ka`) trên một số lượng lớn các tập tin dữ liệu khác nhau, và đưa ra thông báo về trường hợp mà dữ liệu xuất không đồng nhất. Nó được viết trên giao diện *shell* của Unix nhưng có thể chuyển qua Perl hoặc các ngôn ngữ mô tả khác:

```
for i in ka_data.*          # lặp trên các file dữ
```

```
liệu thu
      do
            old_ka $i > out1          # chạy phiên bản cũ
            new_ka $i > out2          # chạy phiên bản mới
            if ! cmp -s out1 out2     # so sánh các file dữ
liệu xuất
            then
                  echo $i : BAD       # khác nhau: xuất thông
báo lỗi
            fi
      done
```

Đoạn chương trình kiểm chứng thường chạy một cách thầm lặng, việc xuất các thông báo chỉ khi có lỗi xuất hiện. Ta có thể in ra tên tập tin đang được kiểm chứng hoặc đưa ra một thông báo lỗi nếu có lỗi phát sinh. Những sự báo hiệu này giúp xác định các lỗi như vòng lặp vô hạn, hoặc một đoạn chương trình kiểm chứng chạy sai trên những trường hợp đúng, nhưng khi thêm những báo hiệu khác vào thì rất phiền phức nếu việc kiểm chứng thực hiện đúng.

Tham số `-s` buộc `cmp` đưa ra trạng thái nhưng không đưa ra output. Nếu các tập tin so sánh bằng nhau, `cmp` trả về `true`, nên `!cmp` là `false`, và không có gì được in ra. Tuy nhiên, nếu giá trị xuất mới và cũ khác nhau thì `cmp` trả về `false`, khi đó tên tập tin và lời cảnh báo được in ra.

Trong việc kiểm chứng lùi cần phải giả sử rằng phiên bản trước của chương trình cho ra kết quả đúng. Điều này cần được kiểm chứng cẩn thận tại thời điểm bắt đầu, và sự bất biến phải được duy trì một cách thận trọng. Nếu có một kết quả sai bên trong việc kiểm chứng lùi thì rất khó để phát hiện, và mọi thứ phụ thuộc vào nó sẽ không chính xác. Do đó, ta cần phải kiểm tra chính việc kiểm chứng lùi một cách có định kỳ để đảm bảo nó vẫn hợp lệ.

Tạo ra những kiểm chứng độc lập

Các kiểm chứng độc lập có chứa giá trị nhập và giá trị xuất mong đợi sẽ bổ sung cho việc thực hiện kiểm chứng lùi. Nhiều ngôn ngữ có cấu trúc được kiểm chứng bằng cách chạy thử một vài chương trình nhỏ với các bộ dữ liệu thử đặc biệt và thấy rằng dữ liệu xuất ra đúng. Phần kiểm chứng hỗn hợp sau đây sẽ trình bày cách kiểm chứng một biểu thức có độ phức tạp tăng dần. Nó chạy trên phiên bản mới của ngôn ngữ Awk *(newawk)* thực hiện kiểm chứng một chương trình ngắn dữ liệu xuất được ghi vào một tập tin, kết quả đúng được ghi vào một tập tin khác bằng hàm echo, sau đó so sánh 2 tập tin, thông báo lỗi nếu 2 tập tin này khác nhau.

```
# field increment test : $i++ means  ($i)++, not $(i++)


echo 3 5 | newawk '{i =1; print $i++; print $1 ,i}'
>out1
echo '3
4 1' >out2                  # kết quả đúng
if ! cmp -s out1 out2   # nếu việc so sánh file là khác
nhau
then
      echo'BAD: field increment test failed'
fi
```

Câu lệnh đầu tiên là một phần dữ liệu nhập của việc kiểm chứng; nó giải thích việc kiểm chứng đang thực hiện những gì.

Ta có thể thực hiện việc kiểm chứng nhiều lần với chi phí vừa phải. Đối với những biểu thức đơn giản, ta tạo ra một ngôn ngữ chuyên biệt đơn giản để mô tả các kiểm chứng, giá trị dữ liệu nhập, giá trị dữ liệu xuất mong muốn. Sau đây là một mô tả ngắn cho việc kiểm chứng một vài cách thể hiện số 1 trong Awk:

```
try {if ($1 == 1) print "yes"; else print "no"}
1       yes
1.0     yes
1E0     yes
0.1E1   yes
10E-1   yes
01      yes
+1      yes
10E-2   no
10      no
```

Dòng đầu tiên là chương trình cần được kiểm chứng (tất cả những từ sau từ `try`). Mỗi dòng tiếp theo là một tập các đầu vào và đầu ra mong muốn cách nhau bởi dấu *tab*. Kiểm chứng đầu tiên cho thấy nếu giá trị vào là 1 thì giá trị xuất là `yes`. Bảy hàng kiểm chứng đầu tiên đều in ra `yes` và hai hàng còn lại in ra `no`.

Một chương trình Awk chuyển mỗi kiểm chứng thành một chương trình Awk hoàn chỉnh, sau đó chạy với những dữ liệu nhập, so sánh dữ liệu xuất thực tế với dữ liệu xuất mong muốn, và chỉ thông báo những trường hợp có câu trả lời sai.

Những cơ chế tương tự được sử dụng để kiểm chứng những biểu thức thông dụng và những câu lệnh thay thế. Một số ngôn ngữ được dùng để viết kiểm chứng tạo ra các mô tả cho việc kiểm chứng một cách dễ dàng; đó chính là việc dùng một chương trình để viết ra chương trình thực hiện việc kiểm chứng cho chương trình ở mức cao (Chương 9 đề cập nhiều về các ngôn ngữ đặc tả và dùng một số chương trình để viết các chương trình khác).

Tóm lại, có khoảng 1000 kiểm chứng Awk; toàn bộ tập này có thể được chạy với từng lệnh đơn, và nếu mọi thứ đều tốt thì sẽ không xuất ra gì

cả. Bất cứ lúc nào khi một chức năng được thêm vào hay một lỗi được sửa, các kiểm chứng mới được thêm vào để chứng tỏ các thao tác vừa mới được thay đổi là đúng? Mỗi khi một chương trình thay đổi, ngay cả theo những cách thông thường, toàn bộ các kiểm chứng sẽ được thực hiện; nó chỉ mất vài phút thôi. Nhưng đôi khi bắt được những lỗi hoàn toàn không ngờ trước được, và tiết kiệm được nhiều thời gian.

Bạn nên làm gì khi bạn phát hiện ra một lỗi? Nếu nó không được tìm thấy bởi những thao tác kiểm chứng có sẵn, bạn phải tạo một kiểm chứng mới để phát hiện vấn đề và thực hiện việc kiểm chứng bằng cách chạy nó trên phiên bản có lỗi. Một lỗi có thể cần nhiều thao tác kiểm chứng hoặc phải kiểm tra toàn bộ các lớp mới. Hoặc có thể thêm vào những đoạn chương trình bảo vệ để có thể bắt được những lỗi bên trong chương trình.

Đừng bao giờ bỏ qua giai đoạn kiểm chứng. Nó có thể giúp bạn chọn thông báo lỗi đưa ra hợp lý, hoặc mô tả những gì vừa được sửa chữa. Lưu vết các lỗi, các thay đổi và sửa chữa. Trong hầu hết các chương trình thương mại, những bản ghi chép đó là bắt buộc. Đối với các chương trình cá nhân thì những bản này tốn ít chi phí và cho ra kết quả tốt.

Bài tập 6-5. Hãy thiết kế một bộ kiểm chứng cho hàm `printf`, sử dụng các cơ chế hỗ trợ có thể.

6.4. Mô hình kiểm chứng

Những thảo luận của chúng ta phần lớn dựa vào việc kiểm chứng các chương trình độc lập trong dạng hoàn chỉnh của nó. Tuy nhiên, điều này không phải là một loại kiểm chứng tự động, cũng không phải là cách kiểm chứng một chương trình trong lúc xây dựng, nhất là khi bạn thuộc một nhóm lập trình. Nó cũng không là cách hiệu quả nhất để kiểm chứng các thành phần nhỏ ẩn trong các thành phần lớn hơn.

Để kiểm chứng một bộ phận độc lập, ta cần phải tạo ra một số loại mô hình để cung cấp đầy đủ sự hỗ trợ cũng như các giao tiếp cho tất cả các phần có liên quan của hệ thống mà việc kiểm chứng được thực hiện. Như chúng ta đã trình bày một ví dụ nhỏ để kiểm chứng việc tìm kiếm nhị phân ở

phần đầu của chương này.

Thật dễ dàng xây dựng một mô hình để kiểm chứng các hàm toán học, các hàm liên quan đến chuỗi, các hàm sắp xếp .... vì việc tạo mô hình bao gồm phần lớn sự thiết lập các tham số đầu vào, gọi các hàm sẽ được kiểm chứng, sau đó kiểm chứng kết quả. Nhưng để tạo ra một mô hình để kiểm chứng từng phần của một chương trình hoàn chỉnh thì quả thật là một việc tốn nhiều thời gian và công sức.

Để minh họa, ta sẽ xây dựng một kiểm chứng cho hàm `memset`, một trong những hàm liên quan đến bộ nhớ trong thư viện chuẩn của C/C++. Những hàm này thường được viết bằng hợp ngữ cho từng loại máy khác nhau bởi vì sự thực hiện của chúng rất quan trọng. Tuy nhiên, nếu chúng càng được thực hiện cẩn thận thì chúng lại càng sai do vậy chúng cần được kiểm chứng một cách kỹ lưỡng.

Bước đầu tiên là cung cấp những phiên bản C đơn giản nhất có thể chạy được; những phiên bản này cho thấy một sự đánh giá cho việc thực thi và quan trọng hơn là tính chính xác. Khi chuyển sang một môi trường mới chỉ lấy sang những phiên bản đơn giản và chúng chỉ được dùng khi đã được hiệu chỉnh cho phù hợp.

Hàm `memset(s,c,n)` gán ký tự `c` vào *n byte* trong bộ nhớ bắt đầu từ đầu địa chỉ `s`, trả về `s`. Hàm này được thực hiện dễ dàng nếu không chú ý tới tốc độ:

```
/* Hàm memset : gán kí tự c vào n byte đầu tiên của s*/
void *memset(void *s , int c, size_t n)
{
      size_t i;
      char *p;
      p = (char *) s;
      for (i = 0; i < n; i++)
            p[i] = c;
```

```
    return s;
}
```

Nhưng khi quan tâm đến vấn đề tốc độ thì những mẹo như viết đầy đủ các *word* 32 hoặc 64 bit tại cùng thời điểm được sử dụng. Những điều này có thể gây ra lỗi vì thế bắt buộc phải có những kiểm chứng tổng quát.

Việc kiểm chứng dựa trên sự kết hợp toàn diện và kiểm tra điều kiện biên tại những nơi dễ gây ra lỗi. Đối với hàm `memset`, biên của nó bao gồm các giá trị hiển nhiên của `n` như 0, 1, 2 nhưng cũng bao gồm các giá trị là lũy thừa của hai hoặc những giá trị lân cận, bao gồm các giá trị nhỏ và lớn như 2^{16}, nó thích hợp với giá trị biên tự nhiên của nhiều máy một *word* có 16 *bit*. Lũy thừa của hai cần được chú ý bởi vì một cách để làm cho hàm `memset` thực hiện nhanh hơn là gán nhiều *byte* cùng lúc, điều này được thực hiện bằng các lệnh đặc biệt hoặc bằng cách lưu một *word* tại một thời điểm thay vì chỉ lưu một *byte*. Một cách tương tự, ta muốn kiểm tra mảng với những cách sắp xếp khác nhau trong trường hợp có lỗi phát sinh dựa trên địa chỉ bắt đầu hoặc chiều dài mảng. Ta sẽ đặt mảng đích vào trong một mảng có kích thước lớn hơn, nhờ đó tạo ra một vùng đệm hoặc một vùng an toàn trên mỗi chiều và chỉ ra một cách dễ dàng để thay đổi cách sắp xếp.

Ta cũng muốn kiểm chứng nhiều giá trị của `c`, bao gồm 0, `0x7F` (giá trị có dấu lớn nhất), `0x80` và `0xFF` (tìm những lỗi tiềm ẩn có thể liên quan đến các ký tự có dấu và không dấu), và các giá trị lớn hơn một *byte* (để chắc chắn chỉ có một *byte* được sử dụng). Ta cũng cần phải khởi tạo bộ nhớ theo vài giá trị khác những giá trị đó để có thể kiểm tra hàm `memset` có ghi ra ngoài vùng hợp lệ hay không.

Chúng ta có thể dùng một sự cài đặt đơn giản như một tiêu chuẩn so sánh trong việc kiểm chứng hai mảng, sau đó so sánh các sự thực hiện trên sự kết hợp của `n`, `c` và *offset (địa chỉ tương đối)* trong mảng:

```
big = maximum left margin + maximum n + maximum right
margin
```

```
        s0 = malloc(big)
        s1 = malloc(big)
        for each combination of test parameters n, c and
offset:
            set all of s0 and s1 to know pattern
            run slow memset(s0 + offset, c, n)
            run fast memset(s1 + offset, c, n)
            check return values
            compare all of s0 and s1 byte by byte
```

Một lỗi gây ra do hàm `memset` là ghi bên ngoài giới hạn của mảng thì ảnh hưởng đến những *byte* ở vị trí bắt đầu hoặc kết thúc của mảng, vì vậy dành ra vùng đệm để dễ dàng phát hiện những *byte* hư hỏng và để phát hiện ra lỗi ghi đè lên một số phần khác của chương trình. Để kiểm chứng việc ghi ra ngoài giới hạn đã định, ta so sánh tất cả các *byte* của `s0` và `s1`, chứ không chỉ *n byte* cần ghi.

Như vậy, tập các kiểm chứng hợp lệ phải bao gồm tất cả các kết hợp sau:

```
offset = 10, 11, …, 20
c    0, 1, 0x7F, 0x80, 0xFF, 0x11223344
n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
    31, 32, 33, …, 65535, 65536, 65537
```

Giá trị của `n` bao gồm ít nhất $2^i - 1$, 2^i, $2^i + 1$ với `i` từ 0 đến 16.

Những giá trị này không cần phải kết với phần chính của mô hình kiểm tra, nhưng nó cần xuất hiện trong mảng được tạo một cách thủ công hay bằng chương trình. Phát sinh ra chúng một cách tự động sẽ tốt hơn: điều này làm cho dễ định rõ các lũy thừa của hai hoặc có được nhiều địa chỉ

offset và nhiều ký tự hơn.

Những kiểm chứng này làm cho hàm `memset` làm việc một cách đúng đắn, tốn ít thời gian để tạo ra, thực hiện độc lập, vì có khoảng 3500 trường hợp cho các giá trị ở trên. Những kiểm chứng này có thể mang chuyển được, vì vậy chúng ta có thể chuyển nó sang một môi trường khác nếu cần thiết.

Những hàm như `memset` có thể có được các kiểm chứng một cách toàn diện vì chúng đơn giản đến nỗi một người có thể chứng minh rằng tất cả trường hợp kiểm chứng có thể được thực hiện thông qua mã nguồn, vì thế nó có thể được kiểm chứng một cách hoàn chỉnh. Chẳng hạn như, có thể kiểm chứng hàm `memmove` trong mọi cách kết hợp chồng chéo, theo hướng, và theo hàng. Ta không có được sự kiểm chứng toàn diện trong trường hợp phải kiểm chứng tất cả những thao tác sao chép có thể, nhưng đó là sự kiểm chứng toàn diện đối với mỗi loại khác nhau của dữ liệu nhập.

Bài tập 6-6. Hãy tạo ra mô hình kiểm chứng cho hàm `memset` theo những đường thẳng mà ta chỉ định.

Bài tập 6-7. Hãy tạo ra những kiểm chứng cho phần còn lại của họ `mem...`.

Bài tập 6-8. Hãy định rõ các cơ chế kiểm chứng cho các hàm số học như `sqrt`, `sin`... trong thư viện `math.h`. Giá trị dữ liệu nhập có ý nghĩa gì? Những việc kiểm tra độc lập nào có thể được thực thi.

Bài tập 6-9. Hãy xác định các cơ chế kiểm chứng các hàm của họ `str...` trong C, chẳng hạn như `strcmp`. Một số hàm này, đặc biệt các hàm như `strtok` và `strcspn` thì phức tạp đáng kể so với họ `mem...` vì thế các kiểm chứng phức tạp hơn sẽ được gọi thực hiện.

6.5. Kiểm chứng tự động với tập dữ liệu có giá trị lớn

Một kỹ thuật kiểm chứng hiệu quả khác là phát sinh tập dữ liệu nhập có giá trị lớn. Tập dữ liệu nhập này gây khó khăn cho chương trình hơn là dữ liệu do người dùng đưa vào. Giá trị lớn có khuynh hướng tạo ra lỗi vì

nhiều dữ liệu nhập có thể làm tràn các vùng đệm bộ nhớ nhập liệu, mảng, biến đếm, và hiệu quả trong việc phát hiện ra những bộ lưu trữ có kích thước cố định chưa được kiểm tra trong chương trình. Hầu hết, mọi người có khuynh hướng tránh những trường hợp *không bình thường* chẳng hạn như dữ liệu nhập vào rỗng, hoặc nó không theo trật tự nào cả hoặc vượt quá giới hạn, và cũng không muốn tạo ra tên quá dài hoặc những giá trị dữ liệu khổng lồ. Ngược lại, máy tính, tạo ra những dữ liệu xuất đúng theo chương trình và không tránh bất kỳ trường hợp nào hết.

Để minh họa, sau đây là một dòng của dữ liệu xuất ra bởi trình biên dịch Microsoft Visual C++ 5.0 khi biên dịch bản cài đặt trên C++ STL của *Markov*, nó đã được hiệu chỉnh có dạng sau:

```
        xtree(114) : warning C4786:
'std::_Tree<std::deque<std::
        basic_string<char,
std::char_traits<char>,std::allocator
        <char>>,std::allocator<std::basic_string<char,std
::                                                    .
        …1420 characters omitted
        allocator <char>>>>>>::iterator' : indentifier
was
        truncated to '255' characters in the debug
information
```

Trình biên dịch đã cảnh báo rằng chương trình đã phát sinh một tên biến có chiều dài 1594 ký tự, trong khi chỉ có 255 ký tự được dành cho thông tin này. Không phải tất cả các chương trình đều tự bảo vệ đối với những chuỗi có chiều dài bất thường như vậy.

Dữ liệu nhập ngẫu nhiên (không cần thiết phải hợp lý) là một cách khác để tấn công vào một chương trình với hy vọng sẽ làm hỏng một cái gì đó. Chẳng hạn như, một số trình biên dịch thương mại C được kiểm chứng với các giá trị phát sinh ngẫu nhiên chứ không là những chương trình có cú

pháp hợp lệ. Trong trường hợp này (trong C chuẩn), mẹo là dùng các đặc tả của vấn đề để xử lý một chương trình đưa ra những dữ liệu kiểm chứng hợp lý nhưng kỳ lạ.

Những kiểm chứng này dựa trên việc phát hiện các kiểm tra gắn liền với chương trình, bởi vì không thể chứng minh rằng chương trình tạo ra các dữ liệu xuất đúng; mục đích là tạo ra một sự hư hỏng hoặc một trường hợp "không thể xảy ra" hơn là tìm ra dễ dàng các lỗi. Nó cũng là một cách tốt để kiểm chứng mã nguồn xử lý lỗi có làm việc hay không. Với dữ liệu nhập thích hợp, hầu hết các lỗi không xảy ra và đoạn mã nguồn xử lý chúng không làm việc; bản chất tự nhiên, lỗi có xu hướng ẩn ở những nơi kín đáo. Tuy nhiên, trong một số trường hợp loại kiểm chứng này cũng làm giảm bớt được lỗi: nó tìm ra được những vấn đề không giống như những gì xảy ra trong thực tế nên ta có thể không cần phải tốn công để sửa chữa.

Một số việc kiểm chứng dựa vào dữ liệu nhập có chủ định. Việc phá vỡ sự an toàn thường dùng các giá trị nhập lớn hoặc không hợp lệ gây nên việc ghi đè lên những dữ liệu quan trọng, do đó việc tìm ra những điểm yếu như vậy là vô cùng đáng giá. Một số hàm thư viện chuẩn có thể bị hư hỏng vì những loại tấn công này. Chẳng hạn như, hàm thư viện chuẩn `gets` không đưa ra kích thước giới hạn của dòng dữ liệu nhập vào, vì vậy không nên sử dụng nó; thay vào đó nên dùng hàm `fgets(buf, sizeof(buf), stdin)`. Hàm `scanf("%s", buf)` nguyên mẫu cũng không giới hạn chiều dài của dữ liệu nhập, vì vậy nó chỉ được dùng với kiểu dữ liệu có kích thước rõ ràng, chẳng hạn `scanf("%20s", buf)`. Trong phần 3.3, ta đã trình bày cách đánh địa chỉ cho vấn đề này đối với *buffer* có kích thước tổng quát.

Bất kỳ hàm nào có thể nhận giá trị bên ngoài chương trình một cách trực tiếp hoặc gián tiếp, thì cần phải kiểm tra tính hợp lệ của dữ liệu nhập trước khi sử dụng chúng. Chương trình sau đây đọc vào một số nguyên do người dùng nhập vào, và đưa ra thông báo nếu số nguyên quá lớn. Nhằm minh họa cách khắc phục những vấn đề gây ra của hàm `gets`, nhưng giải pháp này không phải lúc nào cũng thực hiện được.

```
?     #define MAXNUM 10
?
?     int main(void)
?     {
?           char num[MAXNUM];
?
?           memset(num, 0, sizeof(num));
?           printf("Type a number");
?           gets(num);
?           if (num[MAXNUM-1] != 0)
?                 printf("Number too big.\n");
?           /* ... */
?     }
```

Nếu số nhập vào dài hơn 10 ký số thì nó sẽ ghi đè một giá trị khác không lên số không cuối cùng trong mảng `num`, và theo giả thiết điều này được ngăn chặn sau khi kiểm tra giá trị trả về của hàm `gets`. Nhưng thật không may là điều này không đủ. Những người cố tình có thể nhập vào một chuỗi dài hơn sẽ ghi đè lên một số dữ liệu quan trọng, có thể là giá trị trả về của hàm `gets`, vì vậy chương trình không trở về thực hiện câu lệnh `if` và thay vào đó là thực hiện một điều gì đó bất hợp lệ. Như vậy, dữ liệu nhập không được kiểm chứng loại này là vấn đề tiềm ẩn ảnh hưởng đến sự an toàn.

Chương trình phân tích các đạng của HTML cũng có thể bị ảnh hưởng khi lưu trữ chuỗi có kích thước lớn trong mảng nhỏ:

```
?     static char query[1024];
?
?     char *read_form(void)
```

```
?       {
?               int qsize;
?
?               qsize = atoi(getenv("CONTENT_LENGTH"));
?               fread(query, qsize, 1, stdin);
?               return query;
?       }
```

Đoạn mã nguồn giả sử rằng dữ liệu đưa vào không bao giờ vượt quá 1024 *byte*, tương tự như hàm `gets`, nó dẫn đến tình trạng tràn vùng đệm.

Những sự tràn tương tự cũng gây ra nhiều rắc rối. Nếu một số nguyên bị tràn một cách không biết được thì kết quả có thể rất tai hại. Xét sự khai báo sau:

```
?       char *p;
?       p = (char *) malloc(x * y * z);
```

Nếu tích số của `x`, `y`, `z` bị tràn, lời gọi hàm `malloc` có thể tạo ra một mảng có kích thước hợp lý, nhưng `p[x]` có thể tham chiếu tới một vùng nhớ bên ngoài vùng đã khai báo. Giả sử mỗi số nguyên là 16 bit và `x`, `y`, `z` đều có giá trị 41. Như vậy `x*y*z` là 68921, bằng 3385 modulo 2^{16}. Do đó việc gọi hàm `malloc` chỉ định vị 3385 *byte;* bất kỳ sự tham chiếu nào với một chỉ số ngoài giá trị đó sẽ vượt qua ngoài giới hạn.

Các giá trị dữ liệu nhập ở dạng nhị phân đôi khi làm hỏng các chương trình chỉ nhận các giá trị dữ liệu nhập ở dạng văn bản, đặc biệt nếu ta giả sử rằng dữ liệu nhập thuộc tập các ký tự ASCII 7-bit. Đó là một bài học bổ ích khi bỏ qua các giá trị dữ liệu nhập ở dạng nhị phân đối với các chương trình chỉ nhận giá trị dữ liệu ở dạng văn bản.

Những trường hợp kiểm chứng tốt có thể áp dụng trong nhiều chương trình. Chẳng hạn như, bất kỳ chương trình nào đọc tập tin đều phải kiểm chứng được tập tin có rỗng hay không. Các chương trình đọc tập tin

văn bản phải thực hiện kiểm chứng với các tập tin nhị phân. Các chương trình đọc một dòng văn bản phải thực hiện kiểm chứng với một dòng có kích thước rất lớn, dòng trống hoặc dòng không có ký tự kết thúc chuỗi. Thật là một ý tưởng hay để hình thành một bộ sưu tầm các tập tin để thực hiện kiểm chứng một cách thuận tiện, khi đó bạn có thể kiểm chứng bất cứ chương trình nào mà không phải tạo lại những kiểm chứng. Hoặc viết một chương trình để tạo ra những kiểm chứng theo yêu cầu.

Bài tập 6-10. Hãy tạo ra một tập tin mà nó làm hỏng chương trình xử lý văn bản, trình biên dịch hoặc những chương trình khác mà bạn thích nhất.

6.6. Những mẹo trong việc kiểm chứng

Những người kiểm chứng nhiều kinh nghiệm sử dụng nhiều mẹo và kỹ thuật để làm cho công việc có hiệu quả hơn: mục này bao gồm nhiều mẹo được ưa chuộng.

Chương trình nên kiểm tra các biên của mảng (nếu ngôn ngữ lập trình không thực hiện việc này) nhưng mã nguồn có thể không thực hiện kiểm chứng nếu kích thước mảng rất lớn so với số phần tử nhập vào. Để thực hiện việc kiểm chứng, tạm thời làm cho kích thước của mảng thật nhỏ để dễ dàng tạo ra một lượng lớn các trường hợp kiểm chứng. Ta sử dụng một mẹo có liên quan đến đoạn mã nguồn mở rộng mảng ở Chương 2 và trong thư viện *CSV* ở Chương 4. Thật ra, ta đặt các giá trị khởi tạo nhỏ đúng vị trí, và chi phí cho việc khởi tạo là không đáng kể.

Hàm băm trả về một hằng số, do đó mỗi phần tử được đặt vào vị trí trong một bảng băm. Điều này minh họa cho cơ chế móc xích, nó cũng cung cấp sự chỉ định trong những trường hợp xấu nhất.

Hãy viết một phiên bản của hàm cấp phát bộ nhớ và phát hiện ra lỗi sớm, để kiểm chứng đoạn mã nguồn của bạn có lỗi thiếu bộ nhớ. Phiên bản này trả về giá trị `NULL` sau 10 lần gọi:

```
    /* Hàm testmalloc: trả về giá trị NULL sau 10 lần gọi
*/
```

```
void *testmalloc(size_t n)
{
        static int count = 0;

        if (++count > 10)
                return NULL;
        else
                return malloc(n);
}
```

Trước khi chạy đoạn mã nguồn của bạn, nên vô hiệu hóa các giới hạn kiểm chứng gây ảnh hưởng đến việc thực hiện. Ta đã bắt theo dấu lỗi thực thi của trình biên dịch đối với một hàm băm mà luôn trả về giá trị không vì mã nguồn kiểm chứng chưa được cài đặt.

Ta nên khởi tạo các biến và máng với những giá trị khác biệt nào đó, thay vì giá trị mặc nhiên thường được chọn là không; nếu bạn truy cập ra ngoài giới hạn hoặc tham chiếu đến một biến chưa được khởi tạo thì bạn phải chú ý đến nó. Hằng số `0xDEADBEEF` thì dễ được nhận ra trong một trình gỡ rối, trình cấp phát bộ nhớ đôi khi dùng những giá trị này để tìm ra những dữ liệu chưa được khởi tạo.

Thay đổi các cách kiểm chứng của bạn, đặc biệt khi thực hiện các kiểm chứng nhỏ một cách thủ công, nếu không ta dễ dàng cảm thấy nhàm chán khi luôn kiểm chứng cùng một điều, và bạn có thể không biết được những thứ khác đã hư hỏng.

Đừng tiếp tục cài đặt các tính năng mới hoặc kiểm chứng một tính năng sẵn có khi chương trình đang bị lỗi; nó có thể ảnh hưởng đến kết quả kiểm chứng.

Kết quả kiểm chứng phải bao gồm tất cả các thiết lập tham số nhập, nhờ đó các kết quả kiểm chứng có thể được tái tạo một cách chính xác. Nếu chương trình của bạn có tập dữ liệu nhập từ những số ngẫu nhiên thì ta có được một cách để thiết lập và in ra kết quả được chọn, nó không phụ thuộc vào các kiểm chứng ngẫu nhiên. Hãy đảm bảo rằng tập dữ liệu đầu vào và đầu ra tương ứng được xác định thích hợp, để chúng có thể tái tạo được.

Việc cung cấp những cách tạo ra một tập các dữ liệu xuất có thể điều khiển được khi chương trình đang chạy là rất hữu ích; các dữ liệu xuất thêm vào có thể giúp ích trong suốt quá trình kiểm chứng.

Nên thực hiện kiểm chứng trên nhiều máy, nhiều trình biên dịch và thậm chí trên nhiều hệ điều hành. Mỗi sự kết hợp sẽ phát hiện một số lỗi mà chúng không được tìm thấy trên những sự kết hợp khác, chẳng hạn như, sự phụ thuộc vào thứ tự *byte*, kích thước của số nguyên, việc xử lý con trỏ *null*, cách xử lý các giá trị trả về và ký tự kết thúc chuỗi, việc các tập tin *header* và thư viện. Kiểm chứng trên nhiều máy có thể phát hiện những vấn đề liên quan đến việc kết hợp các thành phần của chương trình, ta sẽ trình bày vấn đề này trong Chương 8 nhằm phát hiện sự phụ thuộc vào môi trường phát triển.

6.7. Ai sẽ thực hiện kiểm chứng?

Việc kiểm chứng được thực hiện bởi người cài đặt hay một người có thể truy cập tới mã nguồn, thường được gọi là phương pháp kiểm chứng hộp trắng *(white box testing)*. (Thuật ngữ gần tương tự với phương pháp kiểm chứng hộp đen *(black box testing)*, người kiểm chứng không biết bằng cách nào các thành phần được cài đặt.) Kiểm chứng mã nguồn của bạn là rất quan trọng; đừng cho rằng các bộ phận kiểm chứng hoặc người dùng sẽ tìm nó cho bạn. Nhưng cần phải kiểm tra rằng bạn đang kiểm chứng cẩn thận như thế nào, do đó hãy bỏ qua mã nguồn và nghĩ về những trường hợp phức tạp, những trường hợp không dễ một chút nào cả. Trích dẫn Don Knuth mô tả cách ông ta tạo ra những cho bộ định dạng TEX , "*Tôi nghĩ đến những trường hợp có ý nghĩa và khó khăn nhất mà tôi có thể xoay xở, và viết*

những mã nguồn phức tạp mà tôi nghĩ ra, sau đó tôi thay đổi hoàn toàn và ghi nhớ những chỉ thị hầu như là rất kinh khủng". Lý do của việc kiểm chứng là để tìm ra những lỗi, chứ không phải tuyên bố chương trình đã làm việc. Vì thế cần kiểm chứng ở những phần phức tạp, và khi tìm ra được lỗi, thì đó là một sự chứng minh cho phương pháp của bạn.

Phương pháp kiểm chứng hộp đen nghĩa là người thực hiện kiểm chứng không hiểu hoặc không có quyền truy cập vào từng chi tiết của mã nguồn. Nó tìm ra nhiều loại lỗi khác nhau, bởi vì người thực hiện kiểm chứng có những giả thuyết khác nhau khi bắt đầu thực hiện kiểm chứng. Các điều kiện biên là một vị trí tốt để bắt đầu sử dụng phương pháp hộp đen, các dữ liệu nhập với giá trị hoặc số lượng lớn và không hợp lệ là những bộ thử tốt. Dĩ nhiên bạn cũng cần phải kiểm chứng các *"trường hợp lưng chừng"* hoặc dùng các quy ước của chương trình để xác định những chức năng cơ bản.

Người sử dụng thật sự là bước kế tiếp. Người sử dụng mới tìm thấy nhiều lỗi mới, bởi vì họ thực hiện chương trình theo những cách chưa được đoán trước. Việc thực hiện những kiểm chứng này quan trọng trước khi chương trình được đưa ra thị trường, đáng buồn là nhiều chương trình được tung ra thị trường mà chưa được kiểm chứng kỹ lưỡng. Phiên bản Beta có nhiều người sử dụng thật sự để kiểm chứng chương trình trước khi nó được hoàn thành, nhưng phiên bản Beta không nên được sử dụng để thay thế cho việc kiểm chứng kỹ càng. Tuy nhiên, khi hệ thống phần mềm trở nên lớn và phức tạp hơn, và thời gian phát triển ngắn lại thì áp lực để đưa ra một chương trình chưa được kiểm chứng đầy đủ lại tăng lên.

6.8. Kiểm chứng chương trình Markov

Chương trình *Markov* ở Chương 3 khá rắc rối nên nó cần được kiểm chứng một cách cẩn thận. Nó tạo ra các giá trị vô nghĩa nên khó phân tích tính hợp lệ, và ta đã viết nhiều phiên bản trên một số ngôn ngữ. Do sự phức tạp ở bước cuối cùng, việc phát sinh dữ liệu của chúng là ngẫu nhiên và mỗi lần lại khác nhau. Bằng cách nào ta có thể áp dụng những bài học trong

chương này để kiểm chứng chương trình *Markov*?

Kiểm chứng đầu tiên xét tập dữ liệu bao gồm một số tập tin nhỏ được dùng kiểm tra các điều kiện biên, để đảm bảo chương trình sẽ phát sinh ra kết quả đúng đối với những dữ liệu nhập chỉ chứa vài từ. Đối với các tiếp vĩ ngữ có chiều dài là hai, ta sử dụng năm tập tin lưu trữ riêng biệt (với một từ trên một hàng)

```
(empty file)
a
a b
a b c
a b c d
```

Đối với mỗi tập tin, dữ liệu xuất phải đồng nhất với giá trị dữ liệu nhập. Những kiểm chứng này phát hiện ra nhiều lỗi trong quá trình khởi tạo bảng, khởi động và kết thúc bộ phát sinh.

Kiểm chứng thứ hai nhằm xác minh các thuộc tính lưu trữ. Đối với các tiếp vĩ ngữ hai từ, mỗi từ, mỗi cặp từ và mỗi bộ ba từ xuất hiện trong kết quả được phát sinh ra phải xuất hiện trong dữ liệu nhập. Ta đã viết chương trình Awk đọc các giá trị dữ liệu nhập chuẩn vào một mảng lớn, tạo ra các mảng của những bộ đôi, bộ ba, sau đó đọc các kết quả được phát sinh của chương trình *Markov* vào một mảng khác và so sánh hai mảng này:

```
# Kiểm chứng chương trình Markov
# Thực hiện kiểm tra trên tất cả các từ, cặp, và bộ ba
từ trong
# kết quả phát sinh ARGV[2] từ dữ liệu nhập ARGV[1]
BEGIN {
      while (getline <ARGV[1] > 0)
            for (i = 1; i <= NF; i++) {
             wd[++nw] = $i      # nhập từ vào
```

```
                single[$i]++
            }
        for (i = 1; i < nw; i++)
                pair[wd[i],wd[i+1]]++
        for (i = 1; i < nw-1; i++)
                triple[wd[i],wd[i-1],wd[i+2]]++
        while (getline < ARGV[2]>0) {
                outwd[++ow] = $0          #phát sinh kết quả
                if (!($0 in single))
                        print "unexpected word", $0
        }
        for (i = 1; i < ow; i++)
                if (!((outwd[i],outwd[i+1]) in pair))
                        print "unexpected pair", outwd[i],
outwd[i+1]
        for (i = 1; i < ow-1; i++)
                if(!((outwd[i],outwd[i+1], outwd[i+2]) in
triple))
                        print"unexpected triple",
                                outwd[i], outwd[i+1],
outwd[i+2]
    }
```

Chúng ta không cần phải tạo ra một kiểm chứng hiệu quả mà chỉ cần tạo ra một chương trình kiểm chứng càng đơn giản càng tốt. Nó mất khoảng 6 hoặc 7 giây để kiểm tra tập tin dữ liệu được phát sinh khoảng 10000 từ tương ứng tập tin dữ liệu nhập 42685 từ, không lâu hơn thời gian của một số phiên bản của *Markov* dùng để phát sinh ra kết quả. Việc kiểm tra sự duy trì gặp một lỗi chính của bản cài đặt bằng Java: thỉnh thoảng chương trình ghi đè lên một số mục của bảng băm bởi vì nó sử dụng tham chiếu thay vì tạo ra

những bản sao các tiếp đầu ngữ.

Kiểm chứng này minh họa cho nguyên tắc xác định một tính chất của kết quả thì dễ hơn việc tạo ra nó. Chẳng hạn như, việc kiểm tra một tập tin có được sắp xếp hay không thì dễ hơn thực hiện sắp xếp tập tin.

Kiểm chứng thứ ba là việc thống kê trong tự nhiên. Dữ liệu nhập bao gồm chuỗi tuần tự

```
a b c a b c ... a b d ...
```

với 10 lần xuất hiện của `abc` rồi sau đó là `abd` và cứ tiếp tục như thế. Dữ liệu xuất có số lần xuất hiện của `c` gấp 10 lần của `d` nếu sự chọn lựa ngẫu nhiên được thực hiện tốt. Tất nhiên, ta khẳng định điều này nhờ hàm `freq`.

Kiểm chứng thống kê trình bày một phiên bản sơ lược của chương trình Java, gắn một bộ đếm vào mỗi tiếp vĩ ngữ, tạo ra 20 ký tự `c` cho mỗi ký tự `d`, gấp đôi số lượng cần có. Sau vài thảo luận sơ bộ, ta nhận thấy bộ phát sinh số ngẫu nhiên của Java tạo ra cả số nguyên âm lẫn số dương, thừa số của hai xuất hiện bởi vì phạm vi của các giá trị thì gấp đôi như những gì ta muốn, vì vậy sẽ có gấp đôi giá trị sẽ trả về là không khi thực hiện chia lấy dư của bộ phát sinh ngẫu nhiên; điều này đã xảy ra với phần tử đầu tiên trong danh sách, đó chính là `c`. Biện pháp sửa chữa là lấy giá trị tuyệt đối trước khi thực hiện phép chia lấy dư. Nếu không thực hiện kiểm chứng loại này thì sẽ không bao giờ phát hiện ra lỗi này, vì dữ liệu xuất có vẻ là tốt.

Cuối cùng, ta tìm được một văn bản tiếng Anh đơn giản để thấy rằng chương trình *Markov* đã phát sinh ra văn bản không có nghĩa. Dĩ nhiên, ta cũng thực hiện việc kiểm chứng này trong quá trình phát triển chương trình. Nhưng ta không chỉ dừng lại kiểm chứng khi chương trình xử lý những dữ liệu nhập thông thường, mà còn thực hiện nó với những trường hợp khó khăn sẽ xảy ra trong thực tế. Kiểm chứng các trường hợp đúng và dễ thì rất hấp dẫn, nhưng các trường hợp khó cũng cần phải được kiểm chứng. Việc kiểm chứng có hệ thống được tự động hóa là cách tốt nhất để tránh các lỗi này.

Tất cả các kiểm chứng đều được tự động hóa. Một giao diện *script*

phát sinh các dữ liệu nhập cần thiết, chạy và định giờ các kiểm chứng, và in ra những dữ liệu xuất bất thường. *Script* có thể được định dạng do đó các kiểm chứng giống nhau có thể được áp dụng trong bất kỳ phiên bản nào của *Markov*, và mỗi khi chương trình thay đổi thì ta chạy lại các kiểm chứng để chắc chắn rằng không có gì bị hư hỏng.

6.9. Tổng kết

Mã nguồn của bạn được viết càng tốt thì sẽ càng có ít lỗi và chắc chắn bạn càng thực hiện kiểm chứng được kỹ lưỡng hơn. Kiểm chứng điều kiện biên khi bạn viết chương trình là một cách hiệu quả để loại ra nhiều lỗi nhỏ ngớ ngẩn. Việc kiểm chứng có hệ thống thực hiện việc dò tìm tại những điểm rắc rối một cách tuần tự, hơn nữa, những lỗi thông thường được tìm thấy ở biên, có thể được dò tìm một cách thủ công hoặc bằng chương trình. Thật lý tưởng để tự động hóa quá trình kiểm chứng, bởi vì máy tính không gây ra nhiều lỗi, không mệt mỏi và cũng không gây ra ngộ nhận một chức năng nào đó đang thực hiện trong khi nó không thực hiện. Sự kiểm chứng lùi kiểm chứng phiên bản sau của chương trình vẫn cho ra những kết quả như được thực thi bởi phiên bản trước. Thực hiện kiểm chứng sau mỗi thay đổi nhỏ là một kỹ thuật tốt để cục bộ hóa mã nguồn của mọi vấn đề, bởi vì hầu hết những lỗi mới thường xuất hiện trong các mã nguồn mới.

Một quy tắc cơ bản của việc kiểm chứng là hãy thực hiện kiểm chứng.

Chương 7

TỐC ĐỘ THỰC THI

Từ khi máy tính mới ra đời, các nhà lập trình phải nỗ lực rất nhiều để viết chương trình sao cho thật sự hiệu quả vì máy tính thời đó vừa đắt tiền vừa có cấu hình yếu. Ngày nay, máy tính rẻ hơn và có cấu hình mạnh hơn nhiều, nên nhu cầu viết chương một cách hiệu quả tuyệt đối đã giảm xuống rõ rệt. Có đáng bận tâm về tốc độ thực thi chương trình hay không?

Vẫn còn, khi chương trình thực sự chạy quá chậm, ta vẫn có thể hy vọng làm cho chương trình chạy nhanh hơn trong khi vẫn giữ được tính đúng đắn, và tính sáng sủa của nó. Một chương trình chạy nhanh mà lại cho kết quả sai thì không có ý nghĩa gì cả..

Như vậy, nguyên tắc tối ưu hóa đầu tiên là đừng tối ưu hóa. Chương trình thực thi đã đủ tốt chưa? Trước hết, cần tìm hiểu cách thức sử dụng chương trình và môi trường mà nó thực thi để từ đó xác định xem việc tăng tốc độ của chương trình có lợi hay không. Các chương trình viết để ra bài tập trong trường học sẽ không bao giờ được dùng lại lần nữa: tốc độ rất ít khi là vấn đề. Đối với các chương trình ra đề bài tập cho học sinh thì tốc độ thực thi rất ít khi gặp khó khăn. Cũng tương tự đối với các chương trình cá nhân, các công cụ ít dùng, khung trắc nghiệm, thí nghiệm, và chương trình mẫu. Tuy nhiên, vì thời gian thực thi (run-time) của một sản phẩm thương mại hay của một thư viện đồ họa có thể vô cùng quan trọng nên cần phải quan tâm đến tốc độ thực thi của chúng, do vậy cần phải hiểu cách suy tính về vấn đề tốc độ thực thi.

Khi nào ta nên tăng tốc cho một chương trình? Chúng ta làm như thế bằng cách nào? Chúng ta mong muốn có được những gì? Chương này sẽ nói

về những cách làm cho chương trình chạy nhanh hơn hoặc dùng ít bộ nhớ hơn. Tốc độ thường là điều quan tâm lớn nhất nên chủ yếu sẽ nói về vấn đề này. Không gian (bộ nhớ, đĩa) ít khi là vấn đề gây trở ngại nhưng cũng có thể đóng vai trò quyết định nên ta cũng giành một phần để nói về nó.

Như đã thấy ở Chương 2, chiến lược tốt nhất là dùng những thuật toán đơn giản nhất, rõ ràng nhất cũng như những cấu trúc dữ liệu thích hợp. Sau đó đo lường tốt độ thực thi để xét xem liệu có cần thay đổi chỗ nào không; cho phép lựa chọn sinh mã nhanh nhất của trình biên dịch; đánh giá những thay đổi nào của chương trình hiệu quả nhất; mỗi lần chỉ đưa vào một thay đổi và đánh giá lại; giữ phiên bản đơn giản nhất để kiểm nghiệm.

Đo lường là một thành phần quan trọng trong việc cải thiện tốc độ thực thi chương trình vì suy luận và trực giác có thể là những định hướng sai lầm và cần phải được bổ sung bằng những công cụ đo thời gian thực thi và lập sơ đồ thời gian của chương trình. Việc cải thiện tốc độ thực thi chương trình có nhiều điểm chung với việc kiểm chứng, bao gồm những kỹ thuật như tự động hóa, lưu vết, và các kiểm chứng lùi để bảo đảm cho các thay đổi vẫn giữ được sự đúng đắn và không làm mất đi những cải tiến trước đó.

Nếu đã chọn tốt được thuật toán cũng như đã viết tốt ngay từ đầu thì có thể không cần tăng tốc thêm. Mã lệnh thiết kế tốt chỉ cần những thay đổi nhỏ thường gặp để sửa chữa các vấn đề về tốc độ thực thi. Trong khi đó, mã lệnh thiết kế kém sẽ cần phải viết lại nhiều hơn.

7.1. Hiện tượng nghẽn cổ chai

Chúng ta hãy bắt đầu bằng việc mô tả cách loại trừ hiện tượng nghẽn cổ chai ra khỏi một chương trình đang bị lỗi trong môi trường cục bộ.

Thư điện tử đến với chúng ta phải thông qua một máy tính gọi là gateway, có chức năng nối mạng nội bộ với Internet. Thư điện tử từ bên ngoài gửi đến gateway của một mạng nội bộ chỉ vài nghìn người có thể lên đến hàng chục nghìn bức một ngày, và sau đó được phát đến từng người trong mạng nội bộ; sự ngăn cách này cô lập mạng nội bộ của chúng ta với Internet và chỉ cho phép ta công bố một tên máy (là tên của gateway) cho tất

cả mọi người trong mạng nội bộ.

Một trong số các dịch vụ của gateway là lọc các *"spam"*, là các thư rác kèm theo các quảng cáo về những dịch vụ mà lợi ích chưa được kiểm chứng. Sau một số thử nghiệm thành công ban đầu bộ lọc các *"spam"* đã được cài đặt như một đặc tính không thể thiếu cho tất cả người dùng trong mạng nội bộ, và rồi rắc rối lập tức xuất hiện. Gateway, vốn cũ kỹ và thường rất bận, bị quá tải vì chương trình lọc chiếm quá nhiều thời gian – nhiều hơn hẳn so với thời gian cần thiết cho tất cả các thao tác xử lý khác đối với bức thư – đến nỗi hàng đợi thư bị đầy và việc truyền phát thư tín bị đình hoãn hàng giờ trong khi hệ thống cố xoay xở để kịp phân phối thư.

Sau đây là một ví dụ về vấn đề tốc độ thực thi chương trình: chương trình không đủ nhanh để hoàn thành nhiệm vụ, và sự chậm trễ này tạo ra trở ngại cho người dùng. Do đó chương trình cần phải chạy nhanh hơn nhiều so với trước.

Một cách đơn giản, bộ lọc *"spam"* hoạt động như sau. Mỗi thông điệp gửi tới được xem như một chuỗi đơn, và một bộ so mẫu ký tự kiểm tra chuỗi đó xem có chứa bất kỳ một cụm từ nào trong số các cụm từ thường gặp trong các "spam" hay không, chẳng hạn như: "Make millions in your spare time" (kiếm hàng triệu đồng trong thời gian rảnh của bạn) hay "XXX-rated" (các trang khiêu dâm). Các thông điệp có xu hướng lặp lại, vì vậy kỹ thuật này có hiệu quả đáng kể, và nếu như một thông điệp *"spam"* lọt lưới thì một cụm từ đặc trưng cho *"spam"* đó sẽ được thêm vào danh sách các *"spam"* để chặn nó vào lần sau.

Do không có một công cụ so sánh chuỗi nào, ví dụ như *grep*, vừa chạy nhanh vừa đáng tin cậy, nên người ta viết ra một bộ lọc đặc biệt dùng cho mục đích lọc các *"spam"*. Mã nguồn rất đơn giản; nó tìm xem mỗi bức thông điệp có chứa một trong số các cụm từ (các mẫu) nào đó hay không:

```
        /* Hàm isspam: kiểm tra chuỗi thông điệp
mesg có chứa spam nào không */
        int isspam(char *mesg)
```

```
        int i;
        for (i = 0; i < npat; i++)
            if (strstr(mesg, pat[i]) !=NULL){
                printf("spam: match for
'%s'\n", pat[i]);
                return 1;
            }
        return 0;
    }
```

Làm sao để thực hiện đoạn chương trình trên nhanh hơn? Cần phải tìm kiếm chuỗi ký tự mẫu trong bức thông điệp, dùng hàm `strstr` của thư viện C là cách tốt nhất để tìm kiếm vì nó vừa đúng chuẩn vừa hiệu quả.

Dùng kỹ thuật lập sơ đồ sử dụng thời gian, kỹ thuật này sẽ được nói kỹ hơn ở mục sau, ta thấy rõ rằng việc thực thi hàm `strstr` có những tính chất không thích hợp khi dùng cho một bộ lọc *spam*. Việc thay đổi cách thức làm việc của hàm `strstr` có thể tăng hiệu quả lên *trong trường hợp cụ thể này*. Hàm `strstr` được cài lại như sau:

```
    /* Hàm strstr: dùng hàm strchr để tìm kiếm kí tự
đầu tiên */
    char *strstr(const char *s1, const char *s2)
    {
        int n;
        n = strlen(s2);
        for (;;) {
            s1 = strchr(s1, s2[0]);
```

```
            if (s1 == NULL)
                return NULL;
            if (strncmp(s1, s2, n) == 0)
                return (char *) s1;
            s1++;
        }
    }
```

Đoạn chương trình này đã được viết hướng tới mục tiêu hiệu quả, và trên thực tế thì nó chạy nhanh nếu dùng cho một mục đích tiêu biểu nào đó, vì chỉ dùng các hàm được tối ưu hóa cao để thực hiện công việc. Đoạn chương trình này gọi hàm `strchr` để tìm vị trí tiếp theo của ký tự đầu tiên trong chuỗi, sau đó gọi để xem liệu phần còn lại của chuỗi có so khớp với phần còn lại của chuỗi mẫu. Như vậy, đoạn chương trình này sẽ bỏ qua một phần lớn của bức thông điệp để tìm chỉ ký tự đầu tiên của chuỗi mẫu, rồi quét nhanh để tìm kiếm phần còn lại. Nhưng tại sao tốc độ thực thi chương trình lại chậm?

Có vài lý do sau đây. Trước hết, hàm `strncmp` lấy chiều dài chuỗi mẫu bằng `strlen` làm tham số đầu vào. Nhưng vì chiều dài chuỗi mẫu là cố định nên không cần tính lại khi xử lý mỗi thông điệp.

Thứ hai, hàm `strncmp` có chứa một vòng lặp phức tạp. Thực tế không những phải so từng *byte* của hai chuỗi mà phải tìm *byte* cuối `'\0'` trên cả hai chuỗi trong khi vẫn đếm lùi tham số chiều dài. Vì chiều dài của tất cả các chuỗi đều đã biết trước (dù không cần dùng tới hàm `strncmp`) nên sự phức tạp này là không cần thiết; ta biết rằng phép đếm là đúng nên việc kiểm tra *byte* `'\0'` là phí phạm thời gian.

Thứ ba, hàm `strnchr` cũng phức tạp, bởi vì đồng thời tìm ký tự và kiểm tra *byte* `'\0'` kết thúc bức thông điệp. Trong một lần gọi hàm `isspam`

thì bức thông điệp là cố định, vì vậy thời gian dùng để kiểm tra *byte* `'\0'` là uổng phí vì ta đã biết bức thông điệp kết thúc ở đâu.

Cuối cùng, dù cho hàm `strncmp`, `strchr`, và `strlen` đều rất hiệu quả khi làm việc độc lập, tổng thời gian cần để gọi các hàm này cũng xấp xỉ với thời gian tính toán của chúng. Sẽ hiệu quả hơn nếu làm mọi việc bằng chỉ một hàm `strstr` được viết lại thật cẩn thận và tránh gọi tất cả những hàm khác.

Những loại vấn đề này là một nguồn gốc chung làm chậm tốc độ thực thi chương trình – một thủ tục hoặc một phương thức giao tiếp làm việc tốt trong một tình huống điển hình nhưng lại thực hiện kém trong tình huống bất thường xảy ra trở thành trung tâm của chương trình. Hàm `strstr` có sẵn thực hiện tốt khi chuỗi mẫu và chuỗi cần tìm ngắn và thay đổi ở mỗi lần hàm được gọi. Nhưng khi chuỗi cần tìm dài và cố định thì thời gian tiêu tốn quá nhiều.

Đã nắm vững điều nói trên, `strstr` được viết lại để xử lý trên chuỗi mẫu cùng với chuỗi thông điệp nhằm tìm ra chỗ so khớp mà không cần gọi thủ tục con. Khả năng thực thi thu được hoàn toàn có thể biết trước: có thể hơi chậm trong một số trường hợp, song rất nhanh trong việc lọc các *"spam"* và quan trọng nhất là không bao giờ chạy quá lâu. Để kiểm tra lại tính đúng đắn của bản cài đặt mới cũng như tốc độ thực thi của chương trình mới, ta xây dựng một bộ kiểm chứng tốc độ thực thi. Bộ thử này bao gồm không chỉ các ví dụ đơn giản như tìm một từ trong câu, mà còn gồm cả một số trường hợp đặc biệt như tìm chuỗi mẫu chỉ có một ký tự '`x`' trong chuỗi có một ngàn ký tự '`e`', và chuỗi mẫu có một ngàn ký tự '`x`' trong một chuỗi chỉ có một ký tự '`e`'. Những trường hợp đặc biệt như vậy là phần chính yếu trong việc đánh giá khả năng thực thi.

Thư viện được cập nhật hàm `strstr` mới và bộ lọc *"spam"* chạy nhanh hơn 30%, đây là kết quả đáng giá cho việc viết lại.

Tuy nhiên, tốc độ thực thi vẫn còn quá chậm. Khi giải quyết các khó khăn, điều quan trọng là xác định đúng vấn đề cần giải quyết. Cho đến nay,

ta đang đặt vấn đề tìm cách nhanh nhất để truy soát một chuỗi ký tự mẫu trong một chuỗi ký tự. Nhưng thực ra thì vấn đề là tìm kiếm một tập hợp lớn, cố định các chuỗi ký tự mẫu trong một chuỗi ký tự dài thay đổi. Trong trường hợp đó, hàm `strstr` không thực sự là giải pháp đúng.

Cách hiệu quả nhất để làm cho một chương trình chạy nhanh hơn là dùng một giải thuật tốt hơn. Bây giờ khi đã nhìn rõ hơn vấn đề cần giải quyết, ta cần suy nghĩ xem thuật toán nào sẽ làm việc tốt nhất.

Vòng lặp cơ bản:

```
for (i = 0; i < npat; i++)
if (strstr(mesg, pat[i]) != NULL)
     return 1;
```

duyệt qua bức thông điệp `npat` lần độc lập; giả thiết rằng nó không tìm ra bất kỳ sự so khớp nào, nó phân tích từng *byte* của bức thông điệp `npat` lần trong `strlen(mesg)*npat` phép so sánh.

Một cách tiếp cận tốt hơn là đảo ngược vòng lặp, quét qua bức thông điệp một lần ở vòng lặp ngoài trong khi tìm tất cả các chuỗi mẫu song song ở vòng lặp trong:

```
for (j = 0; mesg[j] != '\0'; j++)
   if(có chuỗi mẫu nào có phần đầu trùng với ký
tự đầu của mesg[j])
     return 1;
```

Sự cải thiện tốc độ bắt nguồn từ một cách nhìn đơn giản. Để nhận ra liệu có chuỗi mẫu nào so khớp với bức thông điệp ở vị trí `j` không, ta không cần phải xét tất cả các chuỗi mẫu mà chỉ cần xét các chuỗi có cùng ký tự đầu tiên với `mesg[j]`. Đại khái, với 52 ký tự chữ hoa và chữ thường ta chỉ cần thực hiện `strlen(mesg)*npat/52` phép so sánh. Vì các chữ cái không phân bố đều, có nhiều từ bắt đầu bằng ký tự '`s`' hơn là bằng ký tự '`x`', ta sẽ không có được một sự cải thiện 52 lần tốt hơn, nhưng ắt là phải có ở một

mức độ nào đó. Để thực hiện, ta xây dựng một bảng băm dùng ký tự đầu tiên của chuỗi mẫu làm khóa.

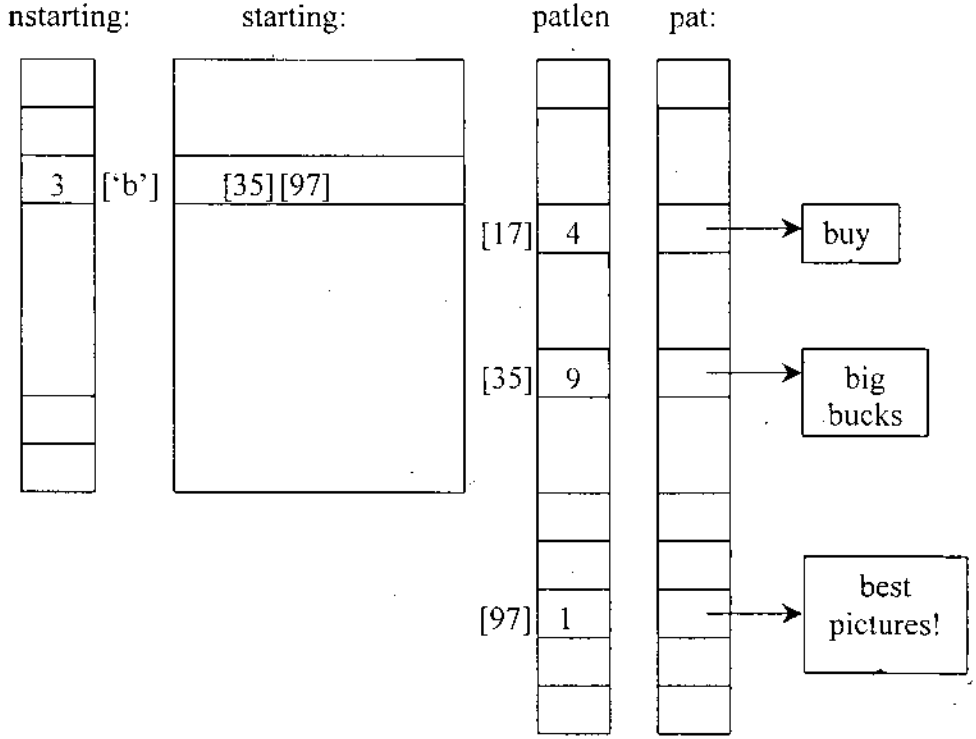
Dựa vào một vài tính toán trước để xây dựng một bảng những chuỗi mẫu bắt đầu với từng ký tự, hàm `isspam` vẫn ngắn như sau:

```
int patlen[NPAT];    /* mảng chiều dài của các
chuỗi mẫu */
int starting[UCHAR_MAX+1][NSTART];    /* danh mục
các chuỗi mẫu bắt bằng ký tự tương ứng */
int nstarting[UCHAR_MAX+1];     /* số chuỗi mẫu
bắt đầu bằng ký tự tương ứng */
      ...
/* Hàm isspam: kiểm tra chuỗi thông điệp mesg có
chứa spam nào không */
int isspam(char *mesg)
{
      int i, j, k;
      unsigned char c;

           for (j = 0; (c = mesg[j]) != '\0'; j++)
 {
               for (i = 0; i < nstarting[c]; i++)
 {
                    k = staring[c][i];
                    if (memcmp(mesg+j, pat[k],
 patlen[k]) == 0) {
                       printf("spam: match for
 '%s'\n", pat[k]);
                       return 1;
```

```
            }
        }
        return 0;
    }
```

Với mỗi ký tự 'c', mảng hai chiều `starting[c][]` chứa danh mục các chuỗi mẫu bắt đầu bằng ký tự 'c' đó. Mảng `nstarting[c]` ghi nhận bao nhiêu chuỗi mẫu bắt đầu bằng ký tự 'c'. Nếu không có các bảng đó, vòng lặp trong sẽ chạy từ 0 đến `npat`, khoảng 1000, thay vì chỉ chạy từ 0 đến khoảng 20. Còn phần từ mảng `patlen[k]` chứa kết quả tính toán trước của `strlen(pat[k])`.

Hình dưới đây phác họa những cấu trúc dữ liệu này dùng một tập hợp ba chuỗi mẫu bắt đầu bằng ký tự 'b':



Đoạn chương trình để xây dựng các bảng trên rất dễ:

```
int i;
unsigned char c;

for (i = 0; i < npat; i++){
    c = pat[i][0];
    if (nstarting[c] >= NSTART)
        eprintf("too many patterns (>=%d)
begin '%c'",
                                NSTART, c);
    starting[c][nstarting[c]++] = i;
    patlen[i] = strlen(pat[i]);
}
```

Phụ thuộc vào dữ liệu nhập, bộ lọc *"spam"* bây giờ cải tiến chạy nhanh hơn từ 5 đến 10 lần khi dùng hàm `strstr`, và nhanh hơn từ 7 đến 15 lần khi dùng phương pháp đầu tiên. Ta không có được sự cải thiện đến 52 lần, một phần là do sự phân bố không đều của bảng chữ cái, một phần do vòng lặp trong chương trình mới phức tạp hơn, và một phần khác do vẫn còn phải thực hiện quá nhiều phép so chuỗi không đạt, nhưng bộ lọc *"spam"* không còn là nút cổ chai gây nghẽn việc phân phối thư nữa. Vấn đề tốc độ thực thi chương trình đã được giải quyết.

Phần còn lại của chương này sẽ đi sâu vào các kỹ thuật được dùng để phát hiện các vấn đề về tốc độ thực thi chương trình, tách các đoạn mã chậm và tăng tốc cho chúng. Trước khi tiếp tục, ta cần xem lại ví dụ về bộ lọc *"spam"* nhằm rút ra những bài học cần thiết. Nếu bộ lọc *"spam"* không phải là một nút cổ chai thì tất cả các cố gắng của ta đều không có ý nghĩa. Một khi ta biết nó gây ra vấn đề, ta đã dùng kỹ thuật *profiling* và những kỹ thuật khác để nghiên cứu cách thực thi của chương trình nhằm tìm ra vấn đề

thực sự nằm ở đâu. Sau đó ta phải chắc chắn rằng ta đã giải quyết đúng vấn đề, thử nghiệm toàn bộ chương trình chứ không chỉ tập trung vào hàm `strstr`, là mồi nghi ngờ ban đầu, thoạt đầu tưởng là đã rõ nhưng thực ra lại là không đúng. Cuối cùng, ta giải quyết đúng vấn đề bằng một giải thuật tốt hơn, và kiểm nghiệm lại xem chương trình đã chạy nhanh hơn chưa. Một khi chương trình đã chạy đủ nhanh, ta ngưng lại. Tại sao phải suy nghĩ quá mức cần thiết?

Bài tập 7.1. Cần phải cải tiến tầm mức của một bảng sơ đồ các chuỗi mẫu bắt đầu bằng một ký tự. Hãy cài đặt một phiên bản `isspam` dùng hai ký tự làm chỉ mục. Có thể cải tiến được đến mức nào? Đây là những trường hợp đơn giản của một kiểu cấu trúc dữ liệu được gọi là *trie*. Đa số những kiểu cấu trúc dữ liệu như vậy, đều chấp nhận tốn không gian lưu trữ để thời gian chạy chương trình nhanh hơn.

7.2. Kỹ thuật lập sơ đồ sử dụng thời gian

Các cách đo thời gian tự động

Đa số hệ thống đều có lệnh để đo thời gian chạy một chương trình. Trên Unix, lệnh này là `time`:

```
%time slowprogram
real        7.0
user        6.2
sys         0.1
%
```

Đoạn chương trình trên chạy lệnh và trả lại 3 giá trị đều đo bằng giây: thời gian "thực", là thời gian đã trôi qua cho đến khi chương trình kết thúc; thời gian CPU của người dùng, là thời gian dùng để thực thi chương trình; và thời gian CPU của hệ thống, là thời gian chương trình giữ quyền điều khiển trong hệ thống. Nếu hệ thống bạn đang dùng có một lệnh tương tự, hãy dùng lệnh đó; giá trị đo được chứa đựng nhiều thông tin hơn, tin cậy hơn, và dễ theo dõi hơn là thời gian đo bằng một đồng hồ bấm giờ. Và hãy

ghi chú cẩn thận. Trong khi bạn làm việc trên chương trình, sửa đổi, đo đạc, bạn sẽ tích lũy được nhiều dữ liệu đến nỗi chỉ một hai ngày sau là sẽ gây nhầm lẫn. (Chẳng hạn như phiên bản nào của chương trình chạy nhanh hơn 20%?). Nhiều kỹ thuật đã nghiên cứu trong Chương 6 có thể được đưa vào để đo đạc và cải tiến tốc độ thực thi chương trình. Dùng máy chạy và đo bộ chương trình kiểm tra, và điều quan trọng nhất là dùng kỹ thuật kiểm tra hồi quy để bảo đảm các sửa đổi không làm gãy chương trình.

Nếu hệ thống của bạn không có lệnh `time`, hay nếu bạn đang đo thời gian của riêng một hàm, xây dựng một cấu trúc đo thời gian tương tự với một cấu trúc kiểm tra là không khó. Trong C và C++ đều có cung cấp một hàm chuẩn, hàm `clock`, trả ra giá trị thời gian CPU mà chương trình đã dùng cho đến thời điểm hiện tại. Hàm này có thể được gọi trước và sau một hàm để đo thời gian sử dụng CPU:

```
#include <time.h>
#include <stdio.h>
    ...
    clock_t  before;
    double  elapsed;

    before = clock();
    long_running_function();
    elapsed = clock() - before;
    printf("function used %.3f seconds\n",
            elapsed/CLOCKS_PER_SEC);
```

Số hạng đếm gộp `CLOCKS_PER_SEC` ghi lại kết quả của chương trình bấm giờ do hàm `clock` báo ra. Nếu hàm chỉ thực hiện trong một phần nhỏ của một giây, cho hàm chạy trong một vòng lặp, nhưng cần nhớ phải tính phần trước vòng lặp nếu phần này tốn thời gian đáng kể:

```
before = clock();
for (i = 0; i < 1000; i++)
     short_running_function();
elapsed = (clock()-before)/(double)i;
```

Trong Java, các hàm trong lớp `Date` cho sẵn thời gian đồng hồ biên, là một xấp xỉ của thời gian CPU:

```
Date before = new Date();
long_running_function();
Date after  = new Date();
long elapsed   after.getTime() -
before.getTime();
```

Giá trị trả về của `getTime` là miligiây.

Dùng công cụ lập sơ đồ sử dụng thời gian

Bên cạnh một phương pháp đo thời gian đáng tin cậy, công cụ quan trọng nhất trong phép phân tích tốc độ thực thi chương trình là một hệ thống giúp thiết lập sơ đồ sử dụng thời gian. Nó xác định cách chương trình sử dụng thời gian vào từng thao tác trong chương trình. Một số sơ đồ sử dụng thời gian liệt kê ra số các hàm, số lần chúng được gọi, và phần thời gian đã dùng để thực thi từng hàm. Số khác lại đếm số lần mỗi phát biểu được thực thi. Những phát biểu nào được thực thi nhiều lần sẽ chiếm thời gian thi hành nhiều hơn, trong khi đó những phát biểu không bao giờ được thực thi chính là những đoạn mã hoặc vô dụng hoặc chưa được kiểm tra đúng đắn.

Kỹ thuật lập sơ đồ sử dụng thời gian là công cụ hiệu quả để tìm các đoạn mã chính trong chương trình, tức là những hàm hoặc những đoạn mã chiếm phần lớn thời gian tính toán. Tuy nhiên, sơ đồ sử dụng thời gian cần phải được diễn giải một cách thận trọng. Do sự tinh tế của trình biên dịch và sự phức tạp trong các hiệu ứng của bộ nhớ đệm và bộ nhớ chính cũng như sự kiện là việc lập sơ đồ sử dụng thời gian của chương trình sẽ làm ảnh hưởng đến tốc độ thực thi, các thống kê trong sơ đồ sử dụng thời gian chỉ là

xấp xỉ.

Kỹ thuật lập sơ đồ sử dụng thời gian thường được kích hoạt bằng một cờ của trình biên dịch hoặc chức năng đặc biệt. Chương trình được cho chạy, và sau đó một công cụ phân tích biểu diễn kết quả. Trên Unix, cờ thường là cờ `-p` và công cụ được gọi là `prof` như sau:

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

Bảng sau đây mô tả một sơ đồ sử dụng thời gian được tạo ra bởi một phiên bản đặc biệt của bộ lọc *spam* chúng tôi đã tạo ra để tìm hiểu cách thức hoạt động của nó. Chương trình dùng một bức thông điệp cố định và một tập hợp cố định gồm 217 cụm từ, tập hợp này so khớp với bức thông điệp 10000 lần. Chương trình chạy trên máy 250 MHz MIPS R1000 dùng bản gốc của hàm `strstr` trong đó gọi các hàm chuẩn khác. Kết quả xuất ra đã được biên tập và định dạng lại cho vừa với khổ giấy. Hãy chú ý cách thể hiện kích cỡ của dữ liệu đầu vào (217 cụm từ) và số lần chạy (10000) dưới dạng những phép kiểm tra không đổi trong cột "*số lần gọi*", cột này đếm số lần gọi mỗi hàm.

12234768552: Tổng số lệnh thực thi

13961810001: Tổng số chu kỳ tính toán

55.847: Tổng thời gian tính toán

1.141: Số chu kỳ trung bình/lệnh

Rõ ràng là hàm `strchr` và hàm `strncmp`, cả hai do hàm `strstr` gọi, đã chi phối hoàn toàn tốc độ thực thi. Chỉ dẫn của Knuth tỏ ra hoàn toàn đúng: chỉ một phần nhỏ của chương trình đã chiếm gần hết thời gian chạy chương trình. Khi một chương trình lần đầu tiên được xây dựng *profile*, thường có thể thấy hàm được gọi nhiều nhất chiếm đến 50% hoặc hơn, như ở trường hợp này, và giúp ta dễ dàng nhận ra nơi cần tập trung chú ý.

secs	%	cum%	chu kỳ	lệnh	số lần gọi	hàm
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	96.6%	648080000	85450000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	2170435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memcc py
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	_malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfre e
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

Tập trung quan tâm đến các đoạn mã nguồn chính

Sau khi viết lại hàm `strstr`, ta lập sơ đồ sử dụng thời gian của chương trình `spamtest` lần nữa và thấy rằng 99.8% thời gian bây giờ chiếm chỉ bởi `strstr`, mặc dù cả chương trình đã nhanh hơn rõ rệt. Khi có chỉ một hàm gây ra hiện tượng nghẽn cổ chai áp đảo, thì chỉ có hai con đường để lựa chọn: cải tiến hàm bằng một thuật toán tốt hơn, hoặc bỏ toàn bộ hàm đó bằng cách viết lại cả chương trình.

Trong trường hợp này, ta viết lại toàn bộ chương trình. Ở đây là một vài dòng đầu tiên trong sơ đồ sử dụng thời gian của chương trình `spamtest` dùng bản cài đặt có tốc độ cao của hàm `isspam`. Hãy chú ý rằng thời gian toàn bộ ít hơn nhiều, trong đó hàm `memcmp` là *hot spot*, và rằng hàm `isspam` bây giờ dùng một phần đáng kể trong thời gian tính toán. Bản này phức tạp hơn so với bản có gọi hàm `strstr`, nhưng ít tốn chi phí hơn nhờ vào sự loại bỏ hàm `strlen` và hàm `strchr` trong `isspam` và việc thay thế hàm `strncmp` bằng hàm `memcmp`, ít tốn *byte* hơn.

secs	%	cum%	chu kỳ	lệnh	số lần gọi	Hàm
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcpy
2.662	43.0%	100.0%	665550000	902920000	10000	Isspam
0.001	0.0%	100.0%	140304	106043	652	Strlen
0.000	0.0%	100.0%	100025	100028	1	Main

Hãy so sánh số chu kỳ đếm được và số lần gọi trong hai sơ đồ sử dụng thời gian. Lưu ý rằng hàm `strlen` đi từ vài triệu lần gọi đến còn 652 và `strncmp` cùng với `memcmp` có cùng số lần gọi. Cũng lưu ý rằng hàm `isspam`, bây giờ hợp nhất với hàm `strchr`, vẫn dùng số chu kỳ ít hơn nhiều so với hàm `strchr` trước đó vì hàm này chỉ kiểm tra các chuỗi mẫu thích hợp ở mỗi bước. Các bước thực thi chi tiết có thể hiểu rõ thông qua phân tích định lượng.

Một đoạn mã nguồn chính thường có thể được loại trừ, hoặc ít nhất cũng làm yếu đi bằng một kỹ thuật đơn giản hơn so với cách ta đã dùng cho bộ lọc *spam*. Trước đây, một sơ đồ sử dụng thời gian của Awk cho thấy rằng một hàm được gọi khoảng một triệu lần qua diễn biến của một phép thử hồi quy trong vòng lặp sau:

```
?       for(j = i; j < MAXFLD; j++)
?           clear(j);
```

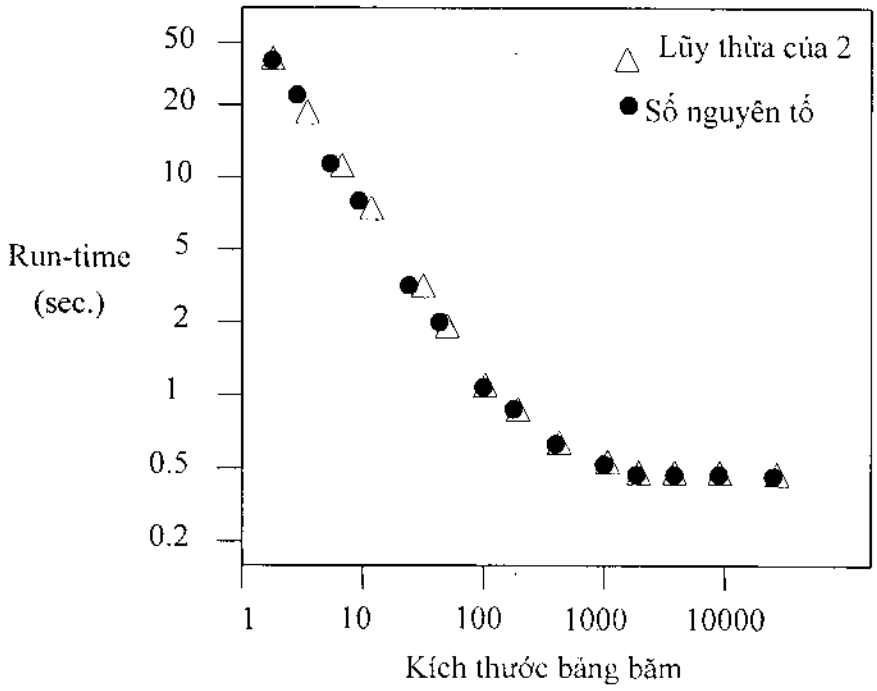
Vòng lặp này, thực hiện việc xóa các trường trước khi đọc vào một dòng mới, đã dùng hơn 50% thời gian thi hành. Hằng số `MAXFLD`, số trường tối đa cho phép trong một dòng, bằng 200. Nhưng trong hầu hết các ứng dụng của Awk, số trường thực sự chỉ là hai hay ba. Do vậy một khoảng thời gian rất dài đã bị sử dụng lãng phí vào việc xóa đi các trường không bao giờ được xác lập. Thay hằng số trên bằng giá trị trước đó của số trường tối đa cho phép tăng tốc toàn bộ lên chừng 25%. Sự sửa chữa được thực hiện bằng cách thay đổi cận trên của vòng lặp:

```
for(j = i; j < maxfld; j++)
    clear(j);
maxfld = i;
```

Vẽ hình

Hình vẽ thường rất hữu ích trong việc trình bày sự đo lường tốc độ thực thi chương trình. Các hình vẽ có khả năng chuyển tải thông tin về hiệu quả của việc thay đổi các tham số, so sánh các thuật toán và các cấu trúc dữ liệu, và đôi khi còn chỉ ra được những chỗ chương trình thực thi theo cách không mong muốn. Các đồ thị chiều dài dây xích rất có tác dụng trong một hàm băm số nhân ở Chương 5 đã cho thấy rất rõ rằng một số nhân tốt hơn số còn lại.

Đồ thị sau đây cho thấy ảnh hưởng do kích cỡ của mảng các bảng băm lên thời gian thi hành chương trình của giải thuật *Markov* viết bằng C dùng quyển sách 42685 từ, 22482 tiếp đầu ngữ làm dữ liệu nhập. Ta hãy làm hai thử nghiệm. Một thử nghiệm gồm một số các lần chạy chương trình dùng các cỡ mảng là lũy thừa của 2 từ 2 đến 16384; thử nghiệm kia dùng các cỡ mảng là số nguyên tố lớn nhất nhỏ hơn mỗi lũy thừa của 2. Ta xem liệu kích thước của mảng là số nguyên tố có làm nên sự khác biệt nào trong tốc độ thực thi chương trình hay không.



Đồ thị cho thấy thời gian chạy ứng với đầu vào trên đây không thay đổi nhiều so với kích thước bảng khi bảng có trên 1000 phần tử, cũng không có sự khác biệt rõ rệt nào giữa hai loại kích thước bảng theo số nguyên tố và theo lũy thừa của 2.

Bài tập 7.2. Cho dù hệ thống bạn đang dùng có lệnh `time` hay không, hãy dùng hàm `clock` hay hàm `getTime` viết một tiện ích đo thời gian cho mục đích dùng riêng của bạn. Hãy so sánh sự đo đạc thời gian ở tiện ích của bạn với một đồng hồ treo tường. Những hoạt động khác trong máy ảnh hưởng đến việc đo thời gian như thế nào?

Bài tập 7.3. Ở sơ đồ sử dụng thời gian đầu tiên, hàm `strchr` được gọi 48350000 lần và hàm `strcmp` chỉ gọi 46180000 lần. Hãy giải thích sự khác biệt.

7.3. Chiến lược tăng tốc độ

Trước khi thay đổi một chương trình để nó chạy nhanh hơn, hãy chắc rằng chương trình chạy thực sự quá chậm, và hãy dùng các công cụ đo thời gian và công cụ lập sơ đồ sử dụng thời gian để tìm xem thời gian đã dùng vào những việc gì. Một khi đã biết sự thể ra sao, ta có nhiều chiến lược để thực hiện mục đích. Ta liệt kê ra dưới đây một vài chiến lược theo thứ tự hiệu quả giảm dần.

Dùng thuật toán hoặc cấu trúc dữ liệu tốt hơn

Yếu tố quan trọng nhất làm cho chương trình chạy nhanh hơn là sự lựa chọn thuật toán và cấu trúc dữ liệu; có thể có một khác biệt rất lớn giữa một thuật toán hiệu quả và một thuật toán không hiệu quả. Bộ lọc *spam* của ta như đã thấy, một sự thay đổi cấu trúc dữ liệu đã đem lại một hệ số tốc độ 10 lần, và còn có thể có một sự cải thiện lớn hơn nếu như thuật toán mới giảm bớt số lệnh tính toán từ $O(n^2)$ còn $O(n\log n)$. Ta đã xem xét điều này ở Chương 2 nên không nhắc lại ở đây.

Hãy hiểu rằng điều mà ta chắc chắn sẽ gặp là sự phức tạp; nếu không, có thể có một lỗi ẩn đâu đó. Thuật toán tựa tuyến tính dùng để duyệt

chuỗi sau đây:

```
?     for (i = 0; i < strlen(s); i++)
?         if (s[i] == c)
?             ...
```

thực tế là bậc hai, nếu `s` có *n* ký tự, mỗi lệnh gọi hàm `strlen` sẽ đi qua *n* ký tự của chuỗi và vòng lặp thì được thực hiện *n* lần.

Làm hữu hiệu mọi khả năng tối ưu hóa của trình biên dịch

Một sự thay đổi không có phí tổn nào mà thường đem lại một sự cải thiện đáng kể là cho phép mọi khả năng tối ưu hóa mà trình biên dịch có. Các trình biên dịch hiện đại đều làm việc rất hiệu quả và cho phép loại bỏ nhiều những yêu cầu sửa đổi không lớn đối với lập trình viên.

Theo ngầm định thì hầu hết các trình biên dịch C và C++ không đưa vào nhiều khả năng tối ưu hóa cho chương trình. Một tùy chọn của trình biên dịch cho phép làm hữu hiệu bộ tối ưu hóa. Điều này nên là tùy chọn ngầm định trừ khi khả năng tối ưu hóa có xu hướng gây xáo trộn trình bắt lỗi ở mức độ nguồn, do vậy lập trình viên dứt khoát phải kích hoạt bộ tối ưu hóa một khi tin là chương trình đã được bắt lỗi.

Sự tối ưu hóa trình biên dịch thường cải thiện thời gian chạy chương trình từ vài phần trăm cho đến hai trăm phần trăm. Nhưng đôi khi lại có thể làm giảm tốc độ của chương trình, do vậy cần đo lường sự cải thiện trước khi đóng gói sản phẩm. Đối với bộ lọc *spam*, trong dãy các thử nghiệm dùng phiên bản thuật toán so trùng cuối cùng, thời gian thi hành ban đầu là 8.1 giây, giảm xuống còn 5.9 giây khi làm hữu hiệu khả năng tối ưu hóa, mức độ cải thiện như vậy là 25%. Ngược lại, ở phiên bản dùng hàm `strstr` đã sửa chữa, sự tối ưu hóa không đem lại một cải thiện nào bởi lẽ hàm `strstr` đã được tối ưu hóa khi đưa vào thư viện: bộ tối ưu hóa chỉ áp dụng vào mã nguồn đang được biên dịch trong chương trình chứ không phải vào các thư viện của hệ thống. Tuy nhiên, một số trình biên dịch có các bộ tối ưu hóa

toàn cục có khả năng phân tích toàn bộ chương trình tìm các chỗ còn có thể cải thiện được. Nếu một trình biên dịch như thế có trong hệ thống của bạn, hãy thử dùng; nó có thể giúp giảm được một vài chu kỳ.

Một điều cần phải hiểu rõ là càng cho phép trình biên dịch tối ưu hóa nhiều bao nhiêu thì khả năng đưa vào lỗi trong chương trình càng nhiều bấy nhiêu. Sau khi làm hữu hiệu khả năng tối ưu hóa, chạy lại dãy thử nghiệm hồi quy để có thể kịp thời thêm những sửa đổi cần thiết.

Tinh chỉnh mã

Chọn lựa thuật toán đúng đắn là điều rất quan trọng khi kích thước dữ liệu lớn. Hơn nữa, sự cải thiện do thuật toán có tác dụng ở mọi máy, mọi trình biên dịch, mọi ngôn ngữ lập trình. Nhưng trong trường hợp đã có thuật toán đúng mà tốc độ vẫn còn là vấn đề thì tiếp theo nên thử tinh chỉnh mã: điều chỉnh chi tiết các vòng lặp và phát biểu sao cho tốc độ thực thi nhanh hơn.

Phiên bản `isspam` ta đã xét ở phần cuối của mục 7.1 chưa được tinh chỉnh. Ở đây ta sẽ làm rõ những cải thiện nào có thể thu được bằng cách ngắt vòng lặp. Để nhắc lại, dưới đây là đoạn mã ban đầu:

```
        for (j = 0; (c = mesg[j]) != '\0'; j++) {
            for (i = 0; i < nstarting[c]; i++) {
                k = staring[c][i];
                if (memcmp(mesg+j, pat[k], patlen[k])
== 0) {
                    printf("spam: match for
'%s'\n", pat[k]);
                    return 1;
                }
            }
        }
```

Phiên bản ban đầu này trong chuỗi thử nghiệm chạy trong 6.6 giây khi được biên dịch có dùng bộ tối ưu hóa. Vòng lặp trong có một chỉ mục mảng (`nstarting[c]`) trong điều kiện lặp, mà chỉ mục này có giá trị không đổi ở mỗi lần lặp của vòng lặp ngoài. Ta có thể tránh tính lại giá trị đó bằng cách lưu vào một biến cục bộ:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = staring[c][i];
        ...
```

Cách làm này giảm thời gian thực thi chương trình xuống còn 5.9 giây, nhanh hơn được 10%, một sự tăng tốc điển hình cho khả năng cải thiện tốc độ mà của việc tinh chỉnh mã. Còn có thể đưa ra một biến khác: `starting[c]` cũng cố định. Có vẻ như loại bỏ phép tính khỏi vòng lặp như đã làm ở trên sẽ có ích, nhưng thực tế khi thử nghiệm ta không thấy một khác biệt rõ rệt nào. Điều này lại cũng là một kết quả điển hình của việc tinh chỉnh mã: đôi khi có ích, đôi khi lại không, và người ta phải tự đo đạc để tìm xem sự tinh chỉnh nào là có ích. Kết quả thu được có thay đổi theo từng máy cũng như từng trình biên dịch khác nhau.

Còn có một thay đổi khác ta có thể đưa vào cho bộ lọc *spam*. Vòng lặp trong so toàn bộ chuỗi mẫu với chuỗi cần xét, nhưng thuật toán bảo đảm là ký tự đầu tiên đã so trùng. Do vậy ta có thể tinh chỉnh mã để bắt đầu hàm `memcmp` ở *byte* tiếp theo. Thử nghiệm cho thấy sự tinh chỉnh này giúp tăng tốc độ 3%, tuy ít nhưng chỉ phải điều chỉnh 3 dòng của chương trình, mà một đã ở trước đoạn mã tính toán.

Không tối ưu hóa những gì không gây vấn đề

Đôi khi sự tinh chỉnh mã không đem lại kết quả gì bởi vì nó được sử dụng vào nơi không thích hợp. Hãy chắc chắn lại rằng đoạn mã đang được tối ưu hóa thực sự là chỗ ngốn nhiều thời gian chạy trong chương trình. Câu chuyện sau đây có thể là ngụy tạo, nhưng cũng nên kể ra làm ví dụ. Một

công ty này không còn tồn tại, dùng một công cụ theo dõi tốc độ thực thi của phần cứng ở một chiếc máy đời đầu và thấy rằng máy dùng đến 50% thời gian thực thi cùng một dãy vài chỉ thị. Các kỹ sư đã tạo một chỉ thị để đóng gói lại thành một hàm, chạy lại, và thấy rằng họ không thu được kết quả gì; họ đã tối ưu hóa vòng lặp nghỉ của hệ thống.

Cần nỗ lực đến mức nào trong việc tăng tốc chương trình? Điều kiện chủ yếu là liệu các thay đổi có sinh lợi đến mức đáng giá hay không? Có một chỉ dẫn, đó là tổng thời gian đã dùng trong việc tăng tốc cho chương trình không được vượt quá thời gian mà sự tăng tốc có thể đem lại được suốt thời gian tồn tại của chương trình. Theo quy tắc này, sự cải tiến thuật toán đối với hàm `isspam` là đáng giá: mất một ngày làm việc nhưng lại làm lợi được hàng giờ mỗi ngày. Bỏ chỉ số mảng khỏi vòng lặp trong có vẻ kém nghệ thuật hơn, nhưng vẫn có giá trị, vì chương trình cung cấp dịch vụ cho một cộng đồng lớn. Tối ưu hóa các dịch vụ công cộng như bộ lọc *spam* hay một thư viện hầu như lúc nào cũng có giá trị, còn tăng tốc cho các chương trình trắc nghiệm hầu như không bao giờ có giá trị. Đối với một chương trình chạy suốt năm, chắt lọc tất cả những gì có thể được. Có thể khởi động lại sau một tháng chạy chương trình nếu tìm được cách tăng tốc cho chương trình đến 10%.

Các chương trình có tính cạnh tranh như trò chơi, trình biên dịch, xử lý văn bản, bảng tính, hệ thống cơ sở dữ liệu cũng đều thuộc nhóm này, vì sự thành công thương mại thường đến với phần mềm chạy nhanh nhất, ít nhất trong các kết quả chấm điểm chính thức.

Việc đo thời gian của chương trình khi thực hiện các thay đổi là rất quan trọng để chắc chắn rằng chương trình vẫn đang được cải thiện. Đôi khi hai cải tiến khác nhau có thể có ích một cách riêng rẽ nhưng tác động cùng lúc của cả hai lại loại trừ lẫn nhau. Cũng có trường hợp cơ chế đo thời gian hoạt động thất thường đến mức khó mà rút ra được kết luận chính xác về tác dụng của các thay đổi. Ngay cả trên một hệ thống chỉ có một người dùng, thời gian cũng có thể dao động ngoài dự tính. Nếu sự biến thiên của đồng hồ thời gian nội bộ (hay ít nhất là những gì được báo lại) đến 10%, những thay

đổi đem lại sự cải thiện 10% rất khó phân biệt với độ nhiễu đó.

7.4. Tinh chỉnh mã

Có nhiều kỹ thuật làm giảm thời gian chạy khi tìm thấy một đoạn mã chiếm phần lớn thời gian của chương trình. Sau đây là một số đề xuất, nên được áp dụng rất cẩn thận, và cần có các thử nghiệm hồi quy để bảo đảm chương trình vẫn chạy tốt. Cần nhớ rằng một số trình biên dịch tốt có thể làm giúp bạn một số phần, và thực tế là bạn có thể vô tình ngăn cản điều đó bằng cách làm phức tạp chương trình. Bất cứ những gì bạn thử, hãy đo lường hiệu quả để bảo đảm rằng điều đó có ích.

Tập hợp những biểu thức chung

Nếu một phép tính tốn nhiều thời gian xuất hiện nhiều lần, hãy thực hiện chỉ một lần rồi lưu lại kết quả. Ví dụ như ở Chương 1, ta đã gặp một macro dùng để tính khoảng cách bằng cách gọi hàm `sqrt` hai lần trong một dòng lệnh với cùng các giá trị, phép tính là như sau:

```
?      sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0)?
...)
```

Hãy tính căn chỉ một lần và dùng giá trị đó ở hai nơi.

Nếu một phép tính thực hiện bên trong vòng lặp nhưng không phụ thuộc vào những thay đổi bên trong vòng lặp, hãy đưa ra ngoài, như thay đoạn chương trình sau:

```
for (i = 0; i < nstarting[c]; i++){
```

bằng đoạn chương trình:

```
n = nstarting[c];
for (i = 0; i < n; i++){
```

Thay các phép tính chạy chậm bằng các phép tính chạy nhanh hơn

Dùng các phép tính tối ưu hóa ít tốn thời gian thay thế các phép tính tốn thời gian. Ngày trước, kỹ thuật này được dùng để chỉ sự thay phép nhân

bằng phép cộng hoặc phép dịch bit, nhưng nay thì không cần nữa. Phép chia và phép lấy phần dư còn chậm hơn nhiều so với phép nhân, tuy nhiên, có thể cải thiện được khi thay phép chia bằng phép nhân với số nghịch đảo, hoặc phép lấy phần dư bằng một toán tử mặt nạ nếu số chia là lũy thừa của 2. Thay chỉ số mảng bằng con trỏ trong C hay C++ có thể tăng tốc được, dù hầu hết các trình biên dịch tự động làm việc này. Thay một phép gọi hàm bằng một phép tính đơn giản hơn cũng vẫn có thể có giá trị. Khoảng cách trong mặt phẳng được xác định theo công thức `sqrt(dx*dx+dy*dy)`, như vậy để xét xem điểm nào ở xa hơn trong hai điểm thường cần hai phép lấy căn bậc hai. Tuy nhiên ta có thể chỉ cần xét bình phương của khoảng cách:

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
    ...
```

cũng cho cùng kết quả như khi so sánh các căn bậc hai.

Ví dụ tương tự xảy ra trong các bộ so mẫu văn bản như ở bộ lọc *spam*. Nếu chuỗi mẫu bắt đầu với một chữ cái trong bảng chữ cái, một phép tìm kiếm được thực hiện đến cuối đoạn văn bản đầu vào đối với chữ cái đó; nếu không có sự trùng lặp nào, một cơ chế tìm kiếm tốn nhiều thời gian hơn sẽ không được gọi.

Khử hay loại bỏ các vòng lặp

Có những phí tổn thời gian nhất định trong xác lập và chạy một vòng lặp. Nếu thân vòng lặp không quá dài và không có nhiều bước lặp, có thể sẽ có hiệu quả hơn khi viết rõ từng bước lặp thành một dãy lệnh. Ví dụ như chuyển:

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

thành

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

Điều này giúp loại bỏ phí tổn thời gian cho vòng lặp, đặc biệt trong trường hợp rẽ nhánh mà thường làm chậm các bộ xử lý hiện đại bằng cách ngắt dòng thực thi chương trình.

Nếu vòng lặp dài hơn, có thể dùng cách hoán chuyển tương tự để giảm bớt số lần lặp, chuyển đoạn chương trình sau:

```
for (i = 0; i < 3*n; i++)
     a[i] = b[i] + c[i];
```

thành

```
for (i = 0; i < 3*n; i += 3) {
     a[i+0] = b[i+0] + c[i+0];
     a[i+1] = b[i+1] + c[i+1];
     a[i+2] = b[i+2] + c[i+2];
}
```

Chú ý rằng điều này chỉ có kết quả khi chiều dài của mảng là bội số của bước lặp; nếu không thì lại cần thêm một đoạn mã để sửa đoạn cuối vòng lặp, đó là chỗ để các sai sót lọt vào và các hiệu quả đã thu được trôi mất lần nữa.

Lưu trữ lại các giá trị thường dùng

Các giá trị lưu trữ không cần phải tính lại. Việc lưu trữ có lợi thế ở tính nội bộ, là xu hướng chương trình dùng lại các mục mới truy cập (hoặc ở gần) có liên quan tới các dữ liệu cũ hơn (hoặc ở xa hơn).

Điều tốt nhất nên làm là làm sao để hoạt động lưu trữ không thể thấy được từ bên ngoài, như vậy sẽ không gây ảnh hưởng đến phần chương trình còn lại trừ tác động tăng tốc cho chương trình. Nghĩa là trong trường hợp chương trình xem trước trang in trên, phần chung của hàm vẽ ký tự

không đổi, luôn luôn là

```
drawchar(c);
```

Phiên bản gốc của `drawchar` gọi hàm `show(lookup(c))`. Việc thực thi phép lưu trữ dùng các giá trị biến nội bộ tĩnh để lưu ký tự trước, mã như sau:

```
if (c != lastc) { /*cập nhật cache */
    lastc = c;
    lastcode = lookup(c);
}
show(lastcode);
```

Viết một cấp phát đặc biệt

Một đoạn mã đơn lẻ chiếm phần lớn thời gian trong một chương trình thường là một cấp phát bộ nhớ, trong đó gặp rất nhiều phép gọi các hàm `malloc` hay dùng toán tử `new`. Khi hầu hết yêu cầu đều đòi hỏi các ô nhớ có cùng kích thước, có thể thu được một sự tăng tốc đáng kể nhờ thay các lệnh gọi đến bộ cấp phát tổng quát bằng lệnh gọi đến một cấp phát đặc biệt. Bộ cấp phát đặc biệt này thực hiện một lệnh gọi hàm `malloc` để xin cấp phát một mảng lớn các phần tử, rồi cấp phát cho mỗi phần tử khi cần, làm theo cách này ít tốn thời gian hơn. Sau khi các phần tử dùng xong vùng nhớ, vùng nhớ sẽ được giải phóng và được đặt trong *danh sách vùng nhớ tự do* để có thể được dùng lại nhanh chóng.

Nếu các phần tử khác nhau có cùng cỡ, ta có thể đánh đổi không gian lưu trữ lấy thời gian bằng cách luôn luôn cấp phát đủ cho cỡ lớn nhất được yêu cầu. Điều này khá hiệu quả trong việc quản lý các chuỗi ký tự ngắn nếu dùng cùng độ dài cho mọi chuỗi ở một giá trị xác định.

Một số thuật toán có thể dùng cách cấp phát dựa trên *stack*, trong đó toàn bộ trình tự cấp phát được thực hiện cùng một lúc và sau khi dùng xong thì tất cả chúng sẽ được giải phóng cùng một lần. Bộ cấp phát chiếm giữ một vùng nhớ cho mình và quản lý ở dạng *stack*, đưa vào (*pushing*) ra các

phần tử đã được cấp phát khi cần và cuối cùng lấy ra (*poping*) toàn bộ chỉ bằng một thao tác. Một số thư viện C có hàm `alloca` cho loại cấp phát này, cho dù đó không phải là cách làm đúng tiêu chuẩn. Hàm này gọi *stack* cục bộ và xem *stack* như là tài nguyên của bộ nhớ, và giải phóng toàn bộ các phần tử khi nó gọi trả về hàm `alloca`.

Dùng bộ nhớ trung gian cho việc nhập và xuất dữ liệu

Dùng bộ nhớ trung gian (hay bộ nhớ đệm) cho các khối lệnh để các hoạt động thường xuyên được thực thi với phí tổn ít nhất có thể, và các thao tác chiếm nhiều thời gian chỉ được thực hiện khi cần thiết. Chi phí cho các thao tác do vậy sẽ trải ra trên nhiều giá trị dữ liệu. Ví dụ như khi một chương trình C gọi hàm `printf`, các ký tự sẽ được lưu trong một bộ nhớ đệm chứ không được chuyển cho hệ điều hành cho đến khi bộ nhớ đệm đầy. Hệ điều hành đến lượt nó sẽ lại tiếp tục hoãn việc ghi dữ liệu vào đĩa. Điều trở ngại là việc làm đầy bộ nhớ đệm xuất để giúp thấy được dữ liệu; trong trường hợp xấu nhất, thông tin vẫn nằm trong bộ nhớ đệm xuất sẽ bị mất khi chương trình bị treo.

Quản lý một cách riêng các tình huống đặc biệt

Bằng cách quản lý các đối tượng có cùng kích thước bằng các đoạn mã riêng, các bộ cấp phát đặc biệt làm giảm phí tổn thời gian và không gian trong bộ cấp phát tổng quát và tinh cờ làm giảm sự phân mảnh.

Tính trước kết quả

Một đôi khi có thể làm cho chương trình chạy nhanh hơn bằng cách tính trước một số kết quả để sẵn dùng khi cần. Ta đã thấy điều này trong bộ lọc *spam*, ở đó đã tính trước giá trị `strlen(pat[i])` và lưu lại trong mảng `patlen[i]`. Nếu một hệ thống đồ họa cần tính nhiều lần một hàm toán học như hàm `sin` chẳng hạn nhưng chỉ với một số tập hợp giá trị rời rạc, ví dụ như với số đo độ bằng số nguyên, chương trình sẽ chạy nhanh hơn nếu tính trước một bảng gồm 360 phần tử (hoặc cung cấp nó như nguồn dữ liệu) và tham chiếu vào bảng đó khi cần. Đây là ví dụ cho sự đánh đổi không gian lấy thời gian. Có nhiều dịp để thay mã lệnh bằng dữ liệu hay bằng cách thực

hiện tính toán trong quá trình biên dịch để tiết kiệm thời gian hay cả không gian nữa. Ví dụ, các hàm `ctype` cũng như hàm `isdigit` hầu như luôn luôn được cài đặt bằng cách tham chiếu vào bảng cờ `bit` hơn là việc thực hiện một chuỗi kiểm tra.

Dùng các giá trị xấp xỉ

Nếu sự chính xác không phải là vấn đề lớn, hãy dùng các kiểu dữ liệu có độ chính xác ít hơn. Trên các máy đời cũ hay cỡ nhỏ, hay các cơ cấu mô phỏng dấu chấm động trong phần mềm, số học dấu chấm động độ chính xác đơn thường nhanh hơn độ chính xác kép, do vậy thường dùng kiểu `float` thay cho kiểu `double` để tiết kiệm thời gian.

Viết lại bằng một ngôn ngữ cấp thấp hơn

Các ngôn ngữ cấp thấp hơn có xu hướng hiệu quả hơn dù sẽ tốn thời gian của lập trình viên hơn. Như vậy viết lại các phần trọng yếu của một chương trình C++ hay Java trong C hay thay một *script* dạng thông dịch bằng một chương trình viết bằng một ngôn ngữ ở dạng biên dịch có thể làm cho chương trình chạy nhanh hơn nhiều.

Đôi khi, ta có thể tăng tốc rất nhiều nếu dùng các mã lệnh có tính phụ thuộc phần cứng. Đây là phương cách cuối cùng, không phải là một bước nên làm hời hợt, vì nó sẽ phá hủy sự linh động của chương trình và làm cho việc bảo trì và sửa chữa trong tương lai trở nên khó khăn hơn nhiều. Hầu như các thao tác diễn đạt trong ngôn ngữ assembly là các hàm tương đối nhỏ nên được nhúng trong một thư viện; hàm `memset` và `memmove` hay các thao tác đồ họa là các thí dụ điển hình. Cách tiếp cận là việc viết mã lệnh rõ ràng đến mức tối đa có thể bằng một ngôn ngữ cấp cao và bảo đảm rằng chương trình đã viết đúng bằng cách kiểm nghiệm như đã mô tả trong cho hàm `memset` trong Chương 6. Đây là bản linh động của chương trình, có thể làm việc trên bất kỳ hệ thống nào, dù chậm. Khi chuyển sang một môi trường mới, ta có thể bắt đầu với một phiên bản đã biết chắc là làm việc được. Bây giờ khi viết một phiên bản bằng ngôn ngữ assembly, hãy kiểm tra hết mọi mặt của chương trình so với phiên bản linh động ở trên. Khi có lỗi,

hãy nghĩ ngay đến các mã lệnh không linh động; đó là cách thích hợp nhất để so sách các bản cài đặt với nhau.

Bài tập 7-4. Một cách để làm cho một hàm như hàm `memset` chạy nhanh hơn là viết hàm ở dạng các đoạn có kích cỡ từng từ thay vì từng *byte*, điều này có thể phù hợp tốt hơn với phần cứng và giảm phí tổn thời gian cho các vòng lặp đến một hệ số từ 4 tới 8. Điểm yếu là ở chỗ bây giờ có nhiều tác động đầu cuối cần phải giải quyết nếu mục tiêu không được sắp hàng trên biên của một từ và nếu chiều dài không phải là bội số của độ lớn của từ. Hãy viết một phiên bản của hàm `memset` thực hiện tối ưu hóa chỗ này. So sánh tốc độ thực thi với phiên bản thư viện hiện có và với một vòng lặp từng-*byte*-tại-một-thời-điểm trực tiếp.

Bài tập 7-5. Hãy viết một bộ cấp phát bộ nhớ `smalloc` cho các chuỗi ký tự C dùng bộ cấp phát đặc biệt cho các chuỗi ngắn nhưng gọi hàm `malloc` cho các chuỗi dài. Sẽ cần định nghĩa một `struct` để biểu diễn chuỗi ký tự trong cả hai trường hợp. Làm thế nào để quyết định khi nào chuyển hàm `smalloc` sang dùng hàm `malloc`?

7.5. Dùng hiệu quả không gian

Bộ nhớ đã luôn là tài nguyên tính toán quý giá nhất, luôn luôn thiếu hụt, và rất nhiều chương trình kém đã được viết ra với cố gắng chiếm phần lớn nhất của nguồn tài nguyên ít ỏi. Sự cố năm 2000 khét tiếng thường được trích dẫn làm ví dụ cho điều này: khi bộ nhớ thực sự khan hiếm, chỉ hai *byte* cần thiết để lưu trữ cặp số 19 đã được cho là quá đắt giá. Cho dù không gian có đúng là nguyên nhân thực sự của vấn đề hay không thì những mã lệnh như vậy đơn giản chỉ phản ánh cách con người ghi ngày tháng trong cuộc sống thường nhật, trong đó phần chỉ thế kỷ thường bị lược bỏ, điều này minh họa mối nguy hiểm cố hữu trong một sự tối ưu hóa thiển cận.

Trong bất kỳ tình huống nào, thời thế cũng đã thay đổi, và cả bộ nhớ chính cũng như các phương tiện nhớ thứ cấp nay đã rẻ một cách đáng ngạc nhiên. Do vậy hướng tiếp cận đầu tiên để tối ưu hóa không gian nên giống như để cải thiện tốc độ: đừng lo lắng.

Vẫn có những hoàn cảnh trong đó sự dùng hiệu quả không gian là một vấn đề. Nếu một chương trình không đủ với lượng bộ nhớ hiện có, vài phần sẽ được đưa vào phân trang, và điều này sẽ làm cho tốc độ thực thi trở nên không thể chấp nhận được. Ta gặp điều này ở các phiên bản mới của phần mềm lãng phí bộ nhớ; đó là một thực tế đáng buồn khi việc nâng cấp phần mềm thường kéo theo việc mua thêm bộ nhớ.

Tiết kiệm không gian bằng cách dùng kiểu dữ kiệu có kích thước nhỏ nhất

Một bước tiến tới việc dùng hiệu quả không gian là thực hiện những thay đổi nhỏ để dùng lượng bộ nhớ hiện hữu cách tốt hơn, ví dụ như bằng cách dùng kiểu dữ liệu nhỏ nhất có thể làm việc được. Điều này có thể có nghĩa là thay kiểu `int` bằng kiểu `short` nếu khớp được với dữ liệu; đây là kỹ thuật chung cho các tọa độ trong các hệ thống đồ họa hai chiều, vì 16 bit có thể quản lý bất kỳ tọa độ nào trên màn hình. Hay cũng có thể có nghĩa là thay kiểu `double` bằng kiểu `float`: vấn đề tiềm ẩn là mất đi độ chính xác, vì kiểu `float` thường chỉ giữ 6 hay 7 số thập phân.

Trong những trường hợp này cũng như những trường hợp tương tự, những thay đổi khác cũng có thể cần đến, đáng chú ý nhất là các định dạng chi tiết trong hàm `printf` và đặc biệt là các phát biểu `scanf`.

Sự mở rộng hợp lôgic của hướng tiếp cận này là mã hóa thông tin trong một *byte* hay với một số bit ít hơn, ngay chỉ với một bit nếu có thể được. Không nên dùng trường bit của C hay C++: chúng rất không linh động và có xu hướng sinh mã khối lượng lớn và không hiệu quả. Thay vào đó, gói gọn các phép toán muốn có trong các hàm tác động đến từng bit đơn trong từ hay một mảng các từ với các toán tử dời và toán tử mặt nạ. Hàm này trả về một nhóm các bit liên tục từ giữa một từ:

```
/* Hàm getbits: lấy n bit từ vị trí của p */
unsigned int getbits(unsigned int x, int p, int
n)
{
```

```
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

Nếu những hàm như vậy trở nên quá chậm, có thể cải tiến bằng kỹ thuật đã được mô tả trước đây trong chương này. Trong C++, sự định nghĩa chồng toán tử có thể được dùng để tạo ra sự truy cập vào bit giống như chỉ số dưới thông thường.

Không lưu trữ những gì có thể tính lại dễ dàng

Những thay đổi loại này là thiểu số, tuy nhiên, chúng tương tự như việc tinh chỉnh mã. Các cải tiến lớn thường do cấu trúc dữ liệu tốt hơn, có khi gắn liền với thay đổi thuật toán.

7.6. Ước tính

Khó mà ước tính được chương trình sẽ chạy nhanh như thế nào, và còn khó khăn gấp đôi để ước tính phí tổn thời gian của các phát biểu trong một ngôn ngữ hay chỉ thị trong một máy nhất định. Dù vậy, rất dễ tạo ra một mô hình phí tổn cho một ngôn ngữ hay cho một hệ thống ít nhất có thể cho một ý tưởng sơ bộ về những thao tác quan trọng kéo dài bao lâu.

Một cách tiếp cận thường được dùng cho các ngôn ngữ lập trình cổ điển là một chương trình đo thời gian các dãy mã lệnh đại diện. Chẳng hạn như, ta có một mô hình ước tính phí tổn, thời gian đo bằng nano-giây cho mỗi phép tính như sau:

Các thao tác trên số nguyên

```
i1++                    8
i1 = i2 + i3           12
i1 = i2 - i3           12
i1 = i2 * i3           12
i1 = i2 / i3          114
i1 = i2 % i3          114
```

Các thao tác trên số thực `float`

```
f1 = f2                    8
f1 = f2 + f3              12
f1 = f2 - f3              12
f1 = f2 * f3              11
f1 = f2 / f3              28
```

Các thao tác trên số thực `double`

```
d1 = d2                    8
d1 = d2 + d3              12
d1 = d2 - d3              12
d1 = d2 * d3              11
d1 = d2 / d3              58
```

Chuyển đổi số

```
i1 = f1                   8
f1 = i1                   8
```

Các thao tác trên số nguyên rất nhanh, ngoại trừ phép chia và phép chia lấy dư. Các thao tác trên dấu chấm động nhanh hoặc còn nhanh hơn, điều ngạc nhiên cho những ai đã quen với thời kỳ các phép tính dấu chấm động chậm hơn nhiều so với phép tính số nguyên.

Các thao tác cơ bản khác cũng rất nhanh, gồm có thao tác gọi hàm, ba hàng cuối cùng trong nhóm dưới đây:

Thao tác trên mảng số nguyên

```
v[i]  = i                 49
v[v[i]]  = i              81
v[v[v[i]]]  = i           100
```

Cấu trúc điều khiển

```
if (i == 5)  i1++              4
if (i != 5)  i1++              12
while (i < 0)  i1++            3
i1 = sum1(i2)                  57
i1 = sum1(i2, i3)              58
i1 = sum1(i2, i3, i4)          54
```

Nhưng nhập và xuất không nhanh lắm, cũng như hầu hết các hàm thư viện khác:

Các thao tác nhập xuất

```
fputs(s, fp)              270
fget(s, 9, fp)            222
fprintf(fp, "%d\n", i)    1820
fscanf(fp, "%d", &i1)     2070
```

Các thao tác gọi hàm `malloc`

```
free(malloc(8))           342
```

Các hàm về chuỗi

```
strcpy(s, "0123456789")          157
i1 = strcmp(s, s)                176
i1 = strcmp(s, "a123456789")     64
```

Chuyển đổi kiểu chuỗi/số

```
i1 = atoi("12345")               402
sscanf("12345", "%d", &i1)       2376
sprintf(s, "%d", i)              1492
f1 = atof("123.45")              4098
sscanf("123.45", "%f", &f1)      6438
sprintf(s, "%6.2f", 123.45)      3902
```

Thời gian cho các hàm `malloc` và `free` có lẽ không thể hiện đúng tốc độ chạy của chúng, vì sự giải phóng ngay lập tức sau khi cấp phát không phải là một kiểu mẫu điển hình.

Cuối cùng là các hàm toán học:

```
i1 = rand()                    135
f1 = log(f2)                   418
f1 = exp(f2)                   462
f1 = sin(f2)                   514
f1 = sqrt(f2)                  112
```

Những giá trị này tất nhiên sẽ khác ở những phần cứng khác nhau, nhưng chiều hướng chung có thể được dùng cho các ước tính nháp về thời gian chạy, hoặc cho việc so sánh phí tổn tương đối giữa xuất nhập và các phép tính cơ bản, hoặc cho việc quyết định liệu có nên viết lại một biểu thức hay nên dùng một hàm nội tuyến.

Có nhiều nguyên nhân gây ra các biến thể. Một trong số đó là mức tối ưu hóa trình biên dịch. Các trình biên dịch hiện đại có thể tìm được những sự tối ưu hóa vượt tầm đa số lập trình viên. Hơn nữa, các CPU hiện nay phức tạp đến nỗi chỉ có một trình biên dịch tốt là có thể tận dụng các khả năng phát ra nhiều chỉ thị đồng thời, phân luồng sự thi hành, tìm kiếm chỉ thị và dữ liệu trước khi cần đến, và những khả năng tương tự như vậy.

Kiến trúc của bản thân máy tính là một nguyên nhân chủ yếu khác làm cho giá trị bằng số của tốc độ thực thi khó dự đoán. Bộ nhớ truy cập nhanh (cache memory) làm nên một sự khác biệt lớn trong tốc độ, và phần lớn sự khôn ngoan trong thiết kế phần cứng đưa đến chỗ che giấu sự thực là bộ nhớ chính thực sự chậm hơn bộ nhớ truy cập nhanh một ít. Tốc độ đồng hồ thô của bộ xử lý như "400 MHz" có thể coi là một gợi ý nhưng không phải là tất cả. Một trong số các máy Pentium 200 MHz cũ của chúng tôi chậm hơn rõ rệt so với một máy Pentium 100 MHz cũ hơn vì chiếc máy sau có một lượng bộ nhớ truy cập nhanh lớn trong khi chiếc máy tính đầu không có. Và các thế hệ khác nhau của bộ xử lý, ngay cả có cùng cấu trúc, dùng

một số chu kỳ đồng hồ khác nhau cho một phép tính nhất định.

Bài tập 7-6. Tạo một tập hợp các trắc nghiệm để ước tính phí tổn thời gian của các thao tác cơ bản cho các máy tính và trình biên dịch bạn có, và nghiên cứu những sự tương tự và sự khác biệt trong tốc độ thực thi.

Bài tập 7-7. Tạo một mô hình phí tổn cho các hoạt động bậc cao hơn trong C++. Một số chức năng có thể bao gồm xây dựng, sao chép, xóa một lớp đối tượng; lệnh gọi các hàm thành viên; các hàm thực; các hàm nội tuyến; thư viện `iostream`; và STL. Bài tập này là bài tập mở, do đó chỉ nên xoay quanh một tập hợp nhỏ các hoạt động đại diện.

Bài tập 7-8. Làm lại bài tập trên trong Java.

7.7. Tổng kết

Một khi đã chọn được thuật toán đúng, tối ưu hóa tốc độ thực thi thường là điều lo ngại cuối cùng khi viết chương trình. Tuy nhiên, nếu buộc phải thực hiện thì chu trình cơ bản là đo đạc, tập trung vào một số chỗ mà sự thay đổi sẽ đem lại nhiều khác biệt nhất, kiểm tra lại các sửa chữa đã thêm vào, và đo lại. Nên dừng lại sớm nhất có thể, và dùng phiên bản đơn giản nhất làm cơ sở cho việc đo thời gian và sửa chữa.

Khi đang nỗ lực cải tiến tốc độ hoặc không gian tiêu thụ của chương trình, một ý tưởng hay là làm các trắc nghiệm chấm điểm hay ghi nhận các vấn đề trục trặc để có thể ước tính và nắm được tốc độ thực thi của chương trình.

Việc chấm điểm có thể được quản lý theo cùng một kiểu khung ta đã gặp trong Chương 6 về kiểm tra. Các trắc nghiệm thời gian được chạy tự động; ngõ xuất chứa đủ các nét nhận dạng để có thể hiểu và tái tạo được; các bản ghi được giữ lại để có thể quan sát các xu hướng và các thay đổi có ý nghĩa.

.Chương 8

TÍNH KHẢ CHUYỂN

Rất khó khăn để viết được một phần mềm chạy chính xác và hiệu quả. Khi một chương trình đã chạy tốt trong một môi trường nào đó, bạn không muốn phải thay đổi nhiều khi chuyển nó sang một trình biên dịch, bộ xử lý hay hệ điều hành khác. Lý tưởng nhất là chẳng cần thay đổi gì cả.

Ý tưởng này được gọi là tính khả chuyển. Thực tế "tính khả chuyển" được hiểu đơn giản hơn, đó là việc dễ dàng điều chỉnh chương trình khi có nhu cầu chuyển chúng tới môi trường khác hơn là phải viết lại từ đầu. Sự thay đổi càng ít thì tính khả chuyển càng cao.

Có lẽ bạn sẽ tự hỏi tại sao phải lo lắng về tính khả chuyển. Nếu phần mềm chỉ chạy trong một môi trường, dưới những điều kiện nhất định, tại sao phải mất thời gian làm cho nó có thể sử dụng rộng hơn? Thứ nhất, bất kỳ một chương trình tốt nào, theo định nghĩa có thể được dùng bất kỳ ở đâu dưới bất kỳ hình thức nào. Xây dựng phần mềm tổng quát hơn đặc tả ban đầu của nó thì sự bảo trì sẽ đơn giản và tiện lợi hơn khi sử dụng. Thứ nhì, môi trường luôn thay đổi. Khi trình biên dịch, hệ điều hành hay phần cứng thay đổi hoặc nâng cấp, đặc tính của chúng thay đổi. Chương trình càng ít phụ thuộc vào các đặc tính cụ thể nó càng ít bị trục trặc và thích ứng với các tình huống thay đổi. Cuối cùng, quan trọng nhất, một chương trình có tính khả chuyển thì vẫn tốt hơn. Sự nỗ lực làm cho một chương trình có tính khả chuyển cũng làm cho việc thiết kế, cấu trúc chương trình được dễ dàng hơn, cũng như việc kiểm tra được thông suốt hơn. Nói chung những kỹ thuật làm cho một chương trình có tính khả chuyển cũng gần nghĩa với những kỹ thuật làm cho một chương trình chạy tốt hơn.

Dĩ nhiên mức độ khả chuyển cũng tùy theo tình hình thực tế. Không có chương trình có tính khả chuyển tuyệt đối, chỉ là một chương trình chưa được thử trong những môi trường khác nhau. Nhưng tính khả chuyển là mục đích của chúng ta khi viết phần mềm để chúng có thể chạy khắp nơi mà không cần phải thay đổi nhiều. Ngay cả khi mục đích này không được thỏa mãn hoàn toàn, thời gian dành cho việc giải quyết tính khả chuyển sẽ được đền đáp khi nâng cấp phần mềm.

Hãy cố gắng viết phần mềm có thể chạy trên phần chung của nhiều tiêu chuẩn, giao diện và môi trường khác nhau. Đừng sửa các lỗi liên quan đến tính khả chuyển bằng cách thêm vào đoạn mã nguồn riêng, thay vì vậy hãy sửa lại phần mềm cho nó hoạt động với các ràng buộc mới. Dùng tính trừu tượng và tính đóng gói dữ liệu để giới hạn và điều khiển phần mã nguồn không khả chuyển. Bằng cách đứng bên trong phần chung của các ràng buộc và cục bộ hóa sự phụ thuộc hệ thống, mã nguồn của bạn sẽ trở nên rõ ràng và tổng quát hơn khi sử dụng chúng.

8.1. Ngôn ngữ

Tuân theo chuẩn

Dĩ nhiên bước đầu tiên để viết mã nguồn có tính khả chuyển là lập trình với ngôn ngữ cấp cao, ngôn ngữ chuẩn càng tốt. Tập tin nhị phân không có tính khả chuyển tốt, nhưng mã nguồn thì ngược lại. Ngay cả khi trình biên dịch chuyển một chương trình sang mã máy không hoàn toàn theo định nghĩa, thậm chí đó là ngôn ngữ chuẩn. Một vài ngôn ngữ sử dụng rộng rãi chỉ có một các cài đặt duy nhất: do đó có nhiều nhà cung cấp, hoặc có nhiều phiên bản cho các hệ điều hành khác nhau. Cách thức thông dịch mã nguồn cũng khác nhau.

Tại sao không có chuẩn hay một định nghĩa chặt chẽ? Đôi lúc chuẩn chưa hoàn thiện và không thể định nghĩa các hành vi khi các tính năng tương tác lẫn nhau. Đôi khi sự ràng buộc lại không chặt chẽ: ví dụ kiểu dữ liệu char trong C và C++ lúc có dấu, lúc không có dấu, và không cần chính xác là 8 bit. Tùy theo người viết trình biên dịch bỏ qua những vấn đề như

vậy để cho việc cài đặt được hiệu quả hơn và tránh bị giới hạn về phần cứng khi dùng ngôn ngữ này, hậu quả là rủi ro sẽ tăng lên khi viết chương trình. Các nguyên tắc và các vấn đề kỹ thuật tương thích nảy sinh có thể dẫn đến sự thỏa hiệp không rõ ràng. Cuối cùng, vì ngôn ngữ khó hiểu và trình biên dịch thường rắc rối nên dễ dàng phát sinh lỗi trong quá trình thông dịch và có lỗi trong khi quá trình cài đặt.

Vì vậy, mặc dù các tiêu chuẩn và các sổ tay tham khảo đã nêu ra các đặc tả chặt chẽ, chúng chưa bao giờ định nghĩa một ngôn ngữ đầy đủ, và các cách thức cài đặt khác nhau có thể thông dịch hợp lệ nhưng không tương thích. Xét khai báo không hợp lệ sau trong C và C++:

```
?       *x[] = {"abc"};
```

qua kiểm tra hàng chục trình biên dịch cho thấy một số chẩn đoán đúng lỗi thiếu khai báo kiểu `char` cho `x`, một số báo lỗi *"mismatched types"*, và một số đã biên dịch thành công đoạn mã nguồn không hợp lệ này.

Lập trình theo xu hướng chính

Cần phải tránh những điểm mơ hồ của ngôn ngữ lập trình – ví dụ như các trường bit trong C và C++. Chỉ nên sử dụng những tính năng mà ngôn ngữ đã định nghĩa rõ ràng và dễ hiểu. Vì những tính năng này thông dụng và xử lý như nhau trong nhiều ngôn ngữ. Chúng ta gọi đó là *xu hướng chính* của ngôn ngữ.

Thật khó biết được đâu là xu hướng chính, nhưng có thể dễ dàng nhận ra thông qua các cấu trúc tốt bên ngoài của ngôn ngữ. Những đặc trưng mới như dấu chú thích (`//`) và kiểu `complex` trong C, hoặc những đặc trưng cụ thể cho một kiến trúc như từ khóa `near` và `far` chắc chắn là nguyên nhân của vấn đề này. Nếu như có một đặc tính mà bạn không rõ thì đừng dùng nó và nên tham khảo ý kiến chuyên gia về ngôn ngữ mà bạn đang quan tâm.

Để thảo luận vấn đề này, chúng ta sẽ tập trung vào C và C++, những ngôn ngữ đa dụng và đã được sử dụng rộng rãi để viết phần mềm có tính khả chuyển.

Xu hướng chính của ngôn ngữ C là gì? Cách nói này thường đề cập đến phong cách của ngôn ngữ. Xét phiên bản C gốc không cần phải khai báo *prototype*, hàm `sqrt` được khai báo như sau:

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

hàm này định nghĩa kiểu trả về nhưng không có các tham số. ANSI C đã thêm vào cách khai báo *prototype* như sau:

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

các trình biên dịch ANSI C chấp nhận kiểu khai báo không có *prototype*, nhưng tốt nhất là nên khai báo *prototype*, như thế sẽ an toàn hơn vì các lời gọi hàm sẽ được kiểm tra đầy đủ, và nếu có bất kỳ thay đổi nào trong quá trình giao tiếp giữa các hàm thì trình biên dịch sẽ nhận ra. Nếu bạn gọi hàm

```
func(7, PI);
```

nếu hàm `func` không khai báo *prototype* thì trình biên dịch sẽ không kiểm tra rằng hàm `func` này có được gọi chính xác hay không. Nếu sau đó hàm `func` bị sửa trở thành có ba tham số, thì việc sửa lại phần mềm có thể bị sai vì kiểu cú pháp cũ không có khả năng kiểm tra các thông số của hàm.

C++ là một ngôn ngữ rộng hơn với nhiều chuẩn, do đó khó xác định xu hướng chính của nó. Ví dụ, mặc dù chúng ta mong đợi STL sẽ là xu hướng chính, nhưng điều đó sẽ không xảy ra ngay, và nhiều phiên bản chưa hỗ trợ STL hoàn toàn.

Kích thước của các kiểu dữ liệu

Kích thước của các kiểu dữ liệu cơ bản trong C và C++ không được định nghĩa rõ ràng; chỉ có một quy luật đơn giản:

```
sizeof(char)  ≤  sizeof(short)  ≤  sizeof(int)  ≤
sizeof(long)

sizeof(float) ≤ sizeof(double)
```

và char ít nhất là 8 bit, short và int ít nhất là 16 bit, và long ít nhất là 32 bit, tuy nhiên không có gì bảo đảm cho các quy định này. Thậm chí không yêu cầu giá trị của con trỏ phù hợp với kiểu dữ liệu int.

Chúng ta dễ dàng biết được kích thước của các kiểu dữ liệu trên một trình biên dịch cụ thể:

```
/* sizeof: hiển thị kích thước của các kiểu dữ liệu cơ
bản */
int main(void)
{
      printf("char %d, short %d, int %d, long %d,",
            sizeof(char),sizeof(short),
            sizeof(int),sizeof(long));
      printf("float %d, double %d, void* %d, %d\n",
            sizeof(float),sizeof(double),
sizeof(void*));
            return 0;
}
```

Hầu hết các kết quả là như sau:

```
char 1, short 2, int 4, long 4, float 4, double 8,
void* 4
```

tuy nhiên cũng có ngoại lệ. Một số hệ thống 64-bit cho kết quả:

```
char 1, short 2, int 4, long 8, float 4, double 8,
void* 8
```

và các máy PC thời kỳ đầu lại cho kết quả:

```
char 1, short 2, int 2, long 4, float 4, double 8,
void* 2
```

Trước đây, phần cứng của những máy PC hỗ trợ một vài kiểu dữ liệu con trỏ. Sự lộn xộn này nảy sinh ra hai từ khóa `far` và `near` cho con trỏ, tuy nhiên chúng không phải là chuẩn, vấn đề này đã gây không ít khó khăn cho các trình biên dịch hiện tại. Nếu như trình biên dịch của bạn có thể thay đổi kích thước của các kiểu dữ liệu cơ bản, hay nếu máy của bạn có các kiểu dữ liệu cơ bản với kích thước khác nhau thì cứ việc biên dịch và chạy thử chương trình của bạn trong các cấu hình khác nhau này.

Tập tin chuẩn `stddef.h` định nghĩa một số kiểu dữ liệu có thể giúp ta viết chương trình có tính khả chuyển. Thông dụng nhất là kiểu `size_t`, đó là kiểu dữ liệu số nguyên không dấu trả về bởi toán tử `sizeof`. Những giá trị của kiểu này được trả về bởi một số hàm như `strlen` và được dùng như là đối số của nhiều hàm khác, `malloc` chẳng hạn.

Trong Java kích thước của các kiểu dữ liệu cơ bản được định nghĩa rõ ràng: `byte` là 8 bit, `char` và `short` là 16 bit, `int` là 32 và `long` là 64 bit.

Chúng ta sẽ không đề cập đến những rắc rối tiềm tàng của việc tính toán dấu chấm động bởi vì đó là một vấn đề lớn. May mắn là hầu hết các máy tính hiện nay đều hỗ trợ chuẩn IEEE cho các phần cứng có khả năng tính toán dấu chấm động, như vậy các thuật toán cho việc tính toán dấu chấm động được thực hiện khá hợp lý.

Thứ tự thực hiện

Trong C và C++, thứ tự khi thực hiện của các thao tác trong một biểu thức, và các tham số hàm không được định nghĩa rõ ràng. Chẳng hạn như, trong phép gán sau:

```
?       n = (getchar() << 8) | getchar();
```

hàm `getchar()` thứ hai có thể được gọi trước; vì thế phép gán có thể được xử lý không như mong đợi. Xem đoạn mã nguồn sau:

```
?       ptr[count] = name[++count];
```

biến `count` có thể tăng trước hoặc sau khi nó làm chỉ mục cho `ptr` , và trong

```
?       printf("%c %c\n", getchar(), getchar());
```

ký tự đầu có thể được in ra sau thay vì được in ra trước. Tương tự thế, cách viết:

```
?       printf("%f %s\n", log(-1.23), strerror(errno));
```

giá trị của lỗi có thể được tính trước khi hàm `log` được gọi.

Có một số quy luật khi đánh giá các biểu thức. Theo quy định tất cả các hiệu ứng lề và lời gọi hàm phải được hoàn thành tại dấu chấm phẩy, hoặc khi một hàm được gọi. Toán tử `&&` và `||` được thi hành từ trái sang phải cho đến khi giá trị cuối cùng được xác định (gồm cả các hiệu ứng lề).

Thứ tự thực hiện trong Java được định nghĩa rất chặt chẽ. Nó yêu cầu các biểu thức, gồm cả các hiệu ứng lề, phải được thực hiện từ trái sang phải, tuy nhiên một số chuyên gia cho rằng cũng không nhất thiết phụ thuộc vào điều này khi viết chương trình. Lời khuyên này có giá trị trong trường hợp có nhu cầu chuyển mã nguồn từ Java sang C hoặc C++. Chuyển mã nguồn từ ngôn ngữ này sang ngôn ngữ khác là điều bất đắc dĩ, nhưng lại hữu ích khi muốn kiểm tra tính khả chuyển.

Dấu của kiểu dữ liệu `char`

Trong C và C++ không xác định rõ kiểu dữ liệu `char` có dấu hay không. Điều này gây ra rắc rối khi kết hợp kiểu dữ liệu `char` với `int`, chẳng hạn trong đoạn mã nguồn sau, trả về giá trị `int` từ hàm `getchar()`. Nếu bạn viết:

```
?       char c; /*nên là kiểu int*/

?       c = getchar();
```

theo thông thường kiểu dữ liệu ký tự là 8-bit, thì giá trị của `c` nằm giữa 0 và 255 nếu `char` không dấu, còn nếu có dấu thì `c` có giá trị từ -128 tới 127. Điều này có thể khi kiểu dữ liệu character được dùng như một mảng hoặc được dùng để kiểm tra `EOF`, có giá trị là `-1` trong thư viện `stdio`. Ví dụ như, chúng ta đã phát triển đoạn mã nguồn trong phần 6.1 sau khi sửa đổi một vài

điều kiện về giới hạn so với bản gốc. Việc so sánh `s[i] == EOF` sẽ luôn sai nếu kiểu `char` không dấu:

```
?       int i;
?       char s[MAX];
?
?       for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?                   break;
?       s[i] = '\0';
```

Khi hàm `getchar` trả về `EOF`, giá trị 255 (`0xFF`, kết quả của việc chuyển đổi từ `-1` sang kiểu dữ liệu `char` không dấu) sẽ được giữ trong `s[i]`. Nếu `s[i]` là không dấu, giữ giá trị là 255 khi so sánh với `EOF` luôn sai.

Tuy nhiên ngay cả khi kiểu `char` có dấu, đoạn mã nguồn trên vẫn không chính xác. Việc so sánh sẽ thành công tại `EOF`, nhưng một giá trị *byte* hợp lệ `0xFF` nhập vào giống như `EOF` sẽ ngắt vòng lặp sớm. Vì vậy để tránh vấn đề rắc rối này, bạn dùng kiểu dữ liệu `int` để lưu giá trị trả về của hàm `getchar` để so sánh với `EOF`. Dưới đây là cách viết vòng lặp có tính khả chuyển:

```
int c, i;
      char s[MAX];
for (i = 0; i < MAX-1; i++) {
      if ((c = getchar()) == '\n' || c == EOF)
            break;
      s[i] = c;
}
s[i] = '\0';
```

Java không có từ khóa `unsigned`; kiểu dữ liệu số nguyên là có dấu còn kiểu dữ liệu `char` (16-bit) thì không.

Phép dịch chuyển số học hay logic

Phép dịch phải của các đại lượng với toán tử >> có thể là phép dịch chuyển số học (một bản sao các bit dấu được sinh ra trong khi dịch chuyển) hay logic (số 0 diền vào các bit trống trong khi dịch chuyển). Một lần nữa, Java đã rút kinh nghiệm vấn đề này từ C và C++, Java dùng >> cho phép dịch phải số học và dùng >>> cho phép dịch phải logic.

Sự canh chỉnh cấu trúc và các thành phần lớp

Canh chỉnh lại các thành phần trong cấu trúc, và lớp không được xác định rõ ràng, ngoại trừ cách sắp đặt trật tự các thành viên của một khai báo. Ví dụ:

```
struct X {
      char c;
      int i;
};
```

địa chỉ trong bộ nhớ của `i` có thể là 2, 4, hay 8 byte từ vị trí bắt đầu của cấu trúc. Một vài máy cho phép `int` được lưu trữ tại các vùng lẻ, nhưng hầu hết yêu cầu các kiểu dữ liệu chính *n*-byte phải được lưu trữ tại một vùng biên *n*-byte, ví dụ kiểu dữ liệu `double` có chiều dài 8 byte được lưu trữ tại các địa chỉ chia hết cho 8. Tuy nhiên do nhu cầu về hiệu suất, trình biên dịch có thể được thay đổi sao cho phù hợp.

Bạn không nên cho rằng các thành phần của một cấu trúc nằm liên tiếp nhau trong bộ nhớ. Khi cấp phát bộ nhớ cho cấu trúc `X` thì ít nhất một byte được cấp phát dư. Những byte cấp phát dư này ý là kích thước thật của một cấu trúc lớn hơn tổng kích thước của các thành viên cộng lại, và thay đổi từ máy này sang máy khác. Kích thước thật của cấu trúc `x` là `sizeof(struct x)` chứ không phải là `sizeof(char) + sizeof(int)`.

Các trường bit

Các trường bit phụ thuộc rất nhiều vào phần cứng do đó không nên dùng chúng.

Tóm lại, những lỗi tiềm tàng này có thể khắc phục được nếu tuân theo một số quy tắc. Không sử dụng các hiệu ứng lề ngoại trừ một vài cách lập trình như:

```
a[i++] = 0;

c = *p++;

*s++ = *t++;
```

Không so sánh một ký tự với `EOF`. Luôn dùng `sizeof` để tính kích thước của một kiểu dữ liệu hay một đối tượng. Không bao giờ dịch phải một giá trị có dấu. Bảo đảm kiểu dữ liệu đủ lớn cho phạm vi của các giá trị mà bạn muốn lưu trữ trong nó.

Thử dùng một số trình biên dịch

Cũng đơn giản để nghĩ rằng bạn hiểu được tính khả chuyển, nhưng các trình biên dịch lại gặp một số vấn đề mà bạn không ngờ, và các trình biên dịch khác nhau có cách nhìn không giống nhau về chương trình của bạn. Vì vậy, bạn nên quan tâm đến sự hỗ trợ của các trình biên dịch. Đọc kỹ tất cả các cảnh báo của trình biên dịch. Cần thử nghiệm nhiều trình biên dịch trên một máy và trên các máy khác nhau. Thử dùng trình biên dịch C++ cho chương trình C.

Do các trình biên dịch chấp nhận các chương trình một cách khác nhau, việc chương trình của bạn biên dịch với một trình biên dịch là không bảo đảm ngay cả với sự chính xác về cú pháp. Tuy nhiên nếu một vài trình biên dịch chấp nhận mã nguồn của bạn thì khả năng thành công rất cao.

Tất nhiên, nhiều trình biên dịch cũng là vấn đề của tính khả chuyển, do có nhiều sự chọn lựa khác nhau cho các hành vi không rõ ràng. Nhưng chúng ta vẫn còn hy vọng vào cách tiếp cận của chúng ta. Thay vì viết mã

nguồn khác nhau cho các hệ thống, môi trường và trình biên dịch khác nhau, chúng ta cố gắng tạo ra các phần mềm không phụ thuộc vào sự khác biệt này.

8.2. Tập tin tiêu đề (header) và thư viện

Các tập tin tiêu đề và các thư viện cung cấp các dịch vụ bổ sung thêm cho các ngôn ngữ cơ bản. Ví dụ, việc nhập dữ liệu và xuất dữ liệu thông qua thư viện `stdio` của C, `iostream` của C++, và `java.io` của Java. Thật ra chúng không phải thành phần của ngôn ngữ, nhưng chúng đi kèm một cách khăng khít với ngôn ngữ và được chấp nhận như là một thành phần của môi trường có hỗ trợ ngôn ngữ này. Nhưng bởi vì các thư viện chứa đựng rất lớn các tiện ích viết sẵn, và thường giải quyết cụ thể cho một hệ điều hành nào đó, do đó chúng tiềm ẩn các vấn đề về tính khả chuyển.

Dùng các thư viện chuẩn

Đối với mỗi ngôn ngữ nên: bám sát chuẩn, và các thành phần đã được xây dựng ổn định. Ngôn ngữ C định nghĩa một thư viện chuẩn các hàm cho việc nhập và xuất dữ liệu; xử lý chuỗi, ký tự; cấp phát, lưu trữ vùng nhớ; và nhiều tác vụ khác. Nếu bạn hạn chế hệ điều hành của bạn tương tác với các hàm này, thì mã nguồn của bạn có khả năng xử lý và thi hành giống nhau khi di chuyển từ hệ thống này sang hệ thống khác. Nhưng bạn cũng nên cẩn thận, bởi vì có nhiều phiên bản khác nhau cho các thư viện và một số có các đặc tính không tuân theo chuẩn.

ANSI C không định nghĩa hàm sao chép chuỗi `strdup` (tuy thế hầu hết các môi trường hỗ trợ nó) ngay cả khi điều này được yêu cầu thành chuẩn. Một lý do nữa là các lập trình viên có thói quen dùng hàm `strdup` mà không biết rằng đó không phải là chuẩn. Do đó chương trình trình sẽ thất bại khi biên dịch ở môi trường khác không hỗ trợ hàm này. Loại vấn đề này là nguyên nhân chính gây rắc rối cho tính khả chuyển khi dùng các thư viện; giải pháp duy nhất là bám sát chuẩn và kiểm tra chương trình của bạn trên nhiều môi trường khác nhau.

Các tập tin tiêu đề và các thao tác khai báo phương thức giao tiếp cho các hàm chuẩn. Một vấn đề là các tiêu đề thường có khuynh hướng lộn xộn bởi vì chúng cố gắng đưa một số ngôn ngữ vào cùng một tập tin. Ví dụ, ta thường tìm thấy một tập tin tiêu đề như `stdio.h` dùng chung cho các trình biên dịch – ANSI C cũ, ANSI C, và ngay cả với C++. Trong những trường hợp này, tập tin đó sẽ chứa lộn xộn các cú pháp định hướng như `#if` và `#ifdef`. Bởi vì các ngôn ngữ lập trình trước đây không được uyển chuyển lắm, nên các tập tin thường phức tạp và khó đọc, đôi khi còn chứa lỗi.

Xét đoạn trích sau từ một tập tin tiêu đề:

```
?     #ifdef _OLD_C
?           extern int fread();
?           extern int fwrite();
?     #else
?     # if defined(__STDC__) || defined(__cplusplus)
?  extern size_t fread( void*, size_t, size_t, FILE*);
?  extern size_t fwrite( const void*, size_t, size_t,
FILE*);
?     #else /*not __STDC__ || __cplusplus */
?  extern size_t fread();
?  extern size_t fwrite();
?     # endif /*else not __STDC__ || __cplusplus */
?     #endif
```

mặc dù chúng được viết khá rõ ràng, nhưng nó cũng chỉ ra rằng các tập tin tiêu đề có dạng như trên rất khó hiểu và khó bảo trì. Dễ dàng hơn nếu ta dùng các tập tin tiêu đề khác nhau cho các trình biên dịch và môi trường khác nhau. Điều này đòi hỏi phải bảo trì các tập tin tiêu đề riêng biệt, nhưng

điều này thích hợp cho các hệ thống cụ thể, và nó cũng giảm thiểu các lỗi tương tự như lỗi `strdup` trong môi trường ANSI C.

8.3. Tổ chức chương trình

Có hai hướng chính cho tính khả chuyển: sự hợp nhất và sự giao nhau. Hướng tiếp cận sự hợp nhất, dùng các đặc tính tốt nhất của các hệ thống riêng biệt, dựa vào đó tạo ra các tiến trình cài đặt và biên dịch phù hợp cho các môi trường cục bộ. Kết quả mã nguồn giải quyết được thống nhất tất cả các tình huống, tận dụng được thế mạnh của các hệ thống. Tuy nhiên một điều trở ngại là khối lượng và độ phức tạp của tiến trình cài đặt, cũng như mã nguồn sẽ khó hiểu hơn do có các điều kiện biên dịch.

Chỉ sử dụng các đặc tính có sẵn ở nhiều nơi

Hướng tiếp cận sự giao nhau chỉ dùng các đặc trưng tồn tại ở tất cả các hệ thống; đừng dùng một đặc trưng nào mà nó không tồn tại ở mọi nơi. Một điều nguy hiểm là sự yêu cầu phải có sẵn các đặc trưng ở mọi nơi có thể làm giới hạn số lượng các hệ thống hoặc khả năng của chương trình biên dịch; và phải tổn thất về hiệu suất ở một số môi trường.

Để so sánh hai hướng tiếp cận này, ví dụ sau dùng mã nguồn theo hướng sự hợp nhất và hướng sự giao nhau. Bạn sẽ thấy, mã nguồn hợp nhất được viết không có tính khả chuyển, bất chấp mục tiêu đã định, trong khi mã nguồn giao nhau cũng không có tính khả chuyển nhưng thường đơn giản hơn.

Ví dụ nhỏ sau cố gắng phù hợp với các môi trường, không có tập tin chuẩn `stdlib.h`:

```
?     #if defined (STDC_HEADERS) || defined (_LIBC)
?     #include <stdlib.h>
?     #else
?     extern void *malloc(unsigned int);
?     extern void *realloc(void *, unsigned int);
```

```
?       #endif
```

Kiểu đối phó này chấp nhận được nếu tình huống trên ít khi xảy ra, nhưng thất bại nếu tình huống trên xảy ra thường xuyên. Nó cũng đặt ra câu hỏi hóc búa đó là cần khai báo bao nhiêu hàm theo cách này? Nếu chương trình dùng hàm `malloc` và `realloc` thì chắc chắn nó cũng dùng hàm `free`. Chuyện gì xảy ra nếu các tham số `unsigned int` của `malloc` và `realloc` không bằng `size_t`? Ngoài ra làm sao ta biết `STDC_HEADERS` hay `_LIBC` đã được định nghĩa đúng? Và làm sao ta chắc chắn không có một trong những cái tên vô tình gọi đến một số hàm đặc biệt nào đó trong một số môi trường? Bất cứ đoạn mã có điều kiện nào giống như trên đều không đầy đủ - không khả chuyển - bởi vì tồn tại những hệ thống không thỏa những điều kiện mà ta nghĩ là đã tổng quát, khi đó ta lại phải điều chỉnh `#ifdef`. Nếu chúng ta có thể giải quyết vấn đề mà không dùng các phát biểu điều kiện phức tạp, chúng ta có thể loại trừ được các vấn đề bảo trì phức tạp.

Vẫn còn những vấn đề tương tự cần giải quyết, và làm thế nào chúng ta giải quyết được chúng trọn vẹn? Một cách lý tưởng ta luôn cho rằng các tập tin tiêu đề chuẩn đã tồn tại; đây cũng chính là một vấn đề nếu ta nghĩ vậy. Sẽ dễ dàng hơn khi phát hành phần mềm ta kèm theo một tập tin tiêu đề định nghĩa chính xác các hàm `malloc`, `realloc` và `free` như ANSI C. Tập tin này có thể luôn luôn được dùng, thay vì phải dán thẳng vào mã nguồn. Do đó ta luôn chắc rằng các interface luôn có sẵn.

Tránh điều kiện biên dịch

Điều kiện biên dịch với `#ifdef` và các từ khóa tương tự rất khó quản lý.

```
?       #ifdef NATIVE
?               char *astring = "convert ASCII to native
character set";
?       #else
?       #ifdef MAC
```

```
?           char *astring = "convert to Mac textfile
format";
?       #else
?       #ifdef DOS
?           char *astring = "convert to DOS textfile
format";
?       #else
?           char *astring = "convert to Unix textfile
format";
?       #endif /* ?DOS */
?       #endif /* ?MAC */
?       #endif /* ?NATIVE */
```

Đoạn trích trên sẽ tốt hơn với `#elif` sau mỗi lời định nghĩa, hơn là một đống các `#endif` ở phía dưới. Nhưng vấn đề thực sự là đoạn mã nguồn trên tính khả chuyển rất kém (mặc dù ý định của nó là tính khả chuyển) bởi vì nó xử lý khác nhau trên mỗi hệ thống và cần phải cập nhật thêm `#ifdef` cho các môi trường mới. Một chuỗi diễn đạt tổng quát, đơn giản hơn, hoàn toàn có tính khả chuyển, và cung cấp nhiều thông tin hơn là:

```
char *astring = "convert to local text format";
```

Không cần phải thêm các đoạn mã nguồn điều kiện do nó giống nhau trên tất cả các hệ thống.

Việc trộn lẫn các dòng điều khiển biên dịch (dùng câu phát biểu `#ifdef`) với dòng điều khiển trong lúc thực thi làm vấn đề tệ hơn, vì rất khó đọc.

```
?       #ifndef DISKSYS
?           for (i = 1; i <= msg->dbgmsg.msg_total; i++)
```

```
?       #endif
?       #ifdef DISKSYS
?             i = dbgmsgno;
?             if (i <= msg->dbgmsg.msg_total)
?       #endif
?             {
?                   ...
?                   if ( msg->dbgmsg.msg_total == i)
?       #ifndef DISKSYS
?                   break; /*không còn message nào để đợi
*/
?       khoảng 30 dòng, với nhiều điều kiện biên dịch
?       #endif
?             }
```

Ngay cả khi có vẻ ổn, các điều kiện biên dịch có thể được thay thế bằng cách khác rõ ràng hơn. Ví dụ, `#ifdefs` thường dùng cho việc điều khiển debug mã nguồn:

```
?       #ifdef DEBUG
?             printf(...);
?       #endif
```

nhưng thông thường câu lệnh `if` với một hằng điều kiện sẽ tiện hơn như trong ví dụ sau:

```
enum { DEBUG = 0 };
...
if (DEBUG) {
```

```
        printf(...);
    }
```

Nếu DEBUG bằng không thì hầu hết các trình biên dịch không phát sinh mã gì, nhưng chúng sẽ kiểm tra cú pháp của đoạn mã nguồn thêm vào. Nhưng phát biểu `#ifdef` có thể che giấu lỗi cú pháp cho đến khi nào phát biểu `#ifdef` được thực hiện.

Đôi khi điều kiện biên dịch gộp vào các đoạn mã nguồn rất lớn:

```
#ifdef notdef /* undefined symbol */
    ...
#endif
```

hay

```
#if 0
    ...
#endif
```

nhưng đoạn mã nguồn điều kiện có thể được bỏ qua bằng cách cho chúng vào các tập tin riêng trong khi biên dịch. Chúng ta sẽ quay lại vấn đề này trong phần sau.

Khi bạn phải điều chỉnh chương trình cho phù hợp với một môi trường mới, đừng bắt đầu bằng cách tạo một bản sao của chương trình mà nên sửa trực tiếp trên mã nguồn luôn. Có thể bạn sẽ phải thay đổi phần chính của đoạn mã nguồn, và nếu như bạn thay đổi trên bản sao thì về sau sẽ có các phiên bản không đồng nhất. Tốt hơn là chỉ có một mã nguồn cho mỗi chương trình; nếu như bạn thay đổi một vài thứ khi chuyển tới một môi trường cụ thể nào đó, thì nên tìm cách thay đổi sao cho chúng cũng chạy tốt ở nơi khác. Thay đổi các phương thức giao tiếp bên trong nếu cần, nhưng giữ cho mã nguồn được nhất quán. Điều này làm cho mã nguồn có tính khả chuyển cao, hơn là chỉ tập trung vào một vài môi trường cụ thể. Thu hẹp thành sự giao nhau hơn là mở rộng tính thống nhất.

Chúng ta đã nhấn mạnh vấn đề gây ra bởi các điều kiện biên dịch thông qua một số ví dụ. Nhưng vấn đề chính chúng ta chưa để ý: đó là rất khó kiểm tra các điều kiện biên dịch. Một `#ifdef` biến một chương trình đơn thành hai chương trình riêng biệt khi biên dịch. Rất khó biết khi nào các chương trình khác nhau được biên dịch và kiểm tra. Nếu chúng ta thay đổi trong khối `#ifdef` , có thể phải thay đổi ở nơi khác, nhưng sự thay đổi này chỉ có thể kiểm tra được trong môi trường nào đó thực sự thi hành đoạn `#ifdef`. Nếu một thay đổi tương tự cho các cấu hình khác, thì không thể kiểm tra được. Cũng vậy, khi ta thêm một khối `#ifdef` vào thì rất khó cách ly sự thay đổi, để xác định điều kiện nào cần phải thỏa cũng như nơi nào trong chương trình phải điều chỉnh cho phù hợp với sự thay đổi này. Cuối cùng, nếu điều gì trong đoạn mã nguồn bị bỏ qua do điều kiện biên dịch, thì trình biên dịch không thể nhìn thấy chúng được. Điều này gây ra những tình huống rất khó lường trước, nếu như một môi trường nào đó kích hoạt các điều kiện biên dịch này. Chương trình sau biên dịch thành công khi `_MAC` được định nghĩa và thất bại nếu ngược lại:

```
#ifdef _MAC

      printf("Là máy Macintosh\r");

#else

      printf("Không đúng cú pháp Macintosh");

#endif
```

Vì thế mục tiêu của chúng ta là chỉ dùng các đặc trưng thông dụng trên tất cả các môi trường đích. Chúng ta có thể biên dịch và kiểm chứng tất cả các đoạn mã nguồn. Nếu có thứ gì đó gây ra vấn đề về tính khả chuyển, ta có thể viết lại nó hay hơn cách thêm vào các điều kiện biên dịch; cách này làm tăng tính khả chuyển và cải tiến chương trình tốt hơn.

Một số hệ thống phân tán với một script tùy biến mã nguồn cho môi trường cục bộ. Vào thời điểm biên dịch, đoạn script kiểm tra các thuộc tính môi trường - định vị các thư viện, các tập tin tiêu đề, thứ tự `byte` trong các dữ liệu `word`, kích thước các kiểu dữ liệu... và phát sinh ra các tham số cấu

hình phù hợp với tình huống đang gặp. Các đoạn script này có thể lớn và rắc rối, một phần đáng kể của các phần mềm phân tán, và đòi hỏi phải thường xuyên bảo trì cho chúng làm được việc. Đôi khi những kỹ thuật này là cần thiết cho mã nguồn càng khả chuyển thì sự cấu hình và cài đặt càng đơn giản.

8.4. Sự cô lập

Mặc dù chúng ta mong muốn có một mã nguồn duy nhất có thể biên dịch trên tất cả các hệ thống mà không phải thay đổi gì, tiếc thay điều này là không thực tế. Nhưng đó sẽ là một sai phạm nếu ta để các đoạn mã nguồn không khả chuyển trong chương trình; một trong các vấn đề là do điều kiện biên dịch tạo ra.

Cục bộ hóa các hệ thống phụ thuộc vào các tập tin riêng biệt. Khi các hệ thống khác nhau cần mã nguồn khác nhau, thì sự khác nhau này nên để trong các tập tin riêng biệt, một tập tin cho mỗi hệ thống.

Che giấu các phụ thuộc hệ thống đằng sau các phương thức giao tiếp

Trừu tượng hóa là một kỹ thuật tốt để tạo ra các ranh giới giữa các phần mềm khả chuyển và không khả chuyển của một chương trình. Các thư viện nhập/xuất đi kèm các ngôn ngữ lập trình là một minh họa tốt; chúng thể hiện sự trừu tượng cho các thành phần lưu trữ thứ cấp trong các tập tin được mở, đóng, đọc và ghi, mà không cần phải tham chiếu tới cấu trúc và vị trí vật lý của chúng. Những chương trình tuân theo các phương thức giao tiếp sẽ chạy trên bất kỳ hệ thống nào hiện thực các phương thức giao tiếp này.

Hướng tiếp cận tính khả chuyển của Java là một ví dụ tốt. Một chương trình trình Java được dịch sang thi hành trong một "máy ảo", mô phỏng một máy tính và có thể thi hành để chạy trên bất kỳ máy thực nào. Các thư viện của Java cung cấp sự truy xuất thống nhất các đặc tính phía dưới của hệ thống như đồ họa, giao tiếp người dùng, mạng và nhiều thứ khác.

8.5. Chuyển đổi dữ liệu

Việc chuyển các dữ liệu văn bản từ hệ thống này sang hệ thống khác rất dễ và là cách đơn giản nhất để chuyển đổi các thông tin tùy ý giữa các hệ thống.

Việc chuyển đổi văn bản cũng còn có một rắc rối: các hệ thống PC dùng kí hiệu trở về đầu dòng `'\r'` và xuống dòng kế tiếp `'\n'` để ngắt dòng, trong khi các hệ thống Unix chỉ dùng newline. Xuống dòng mới là một khái niệm cho máy điện báo đánh chữ ngày xưa, nó có một thao tác trở về đầu dòng (CR) và một thao tác xuống dòng (LF) riêng cho việc xuống dòng kế tiếp.

Mặc dù các máy tính ngày nay đã khác chúng vẫn dùng khái niệm (CRLF) sự kết hợp của CR và LF cho mỗi dòng. Nếu trong tập tin không có các ký hiệu này thì nó được xem là một dòng dài. Dẫn đến việc đếm dòng và các kí tự có thể sai. Một số phần mềm tránh được vấn đề này nhưng đa số là không. Một điều may mắn là các giao thức mạng hiện đại tương thích với vấn đề này chẳng hạn HTTP cũng dùng CRLF để phân định dòng.

Nên dùng các phương thức giao tiếp chuẩn, chúng xử lý vấn đề CRLF trên bất kỳ hệ thống nào, trên các hệ thống PC chúng bỏ `\r` khi nhập dữ liệu và thêm vào khi xuất dữ liệu, còn trên các hệ thống Unix thì không tạo ra `\r` . Với các tập tin di chuyển giữa các hệ thống khác nhau thường xuyên thì cần phải có một chương trình đặc biệt để chuyển đổi.

Bài tập 8-1. Viết một chương trình bỏ các `\r` ra khỏi một tập tin. Viết một chương trình thứ hai thêm chúng vào bằng các thay thế các dòng mới với một CR và LF. (Làm sao bạn kiểm tra được các chương trình này?).

8.6. Nâng cấp và tính khả chuyển

Một trong các nguyên nhân gây ra vấn đề về tính khả chuyển là sự thay đổi của các phần mềm trong vòng đời của nó. Các thay đổi có thể xảy ra tại bất kỳ các phương thức giao tiếp trong hệ thống và gây ra sự không tương thích giữa phiên bản đã có của chương trình.

Thay đổi tên nếu bạn thay đổi đặc tả

Ví dụ ưu chuộng là việc thay đổi các thuộc tính của lệnh `echo` trong Unix, thông qua cách khởi động các thông số của nó:

```
% echo hello, world
hello, world
%
```

Tuy nhiên, lệnh `echo` là một phần chính của giao diện script trên Unix, và sự cần thiết phát sinh ra các định dạng kết quả trở nên quan trọng. Do đó lệnh `echo` được thay đổi để thông dịch các thông số của nó, giống như hàm `printf`:

```
% echo 'hello\nworld'
hello
world
%
```

Điểm mới này rất hữu ích, nhưng lại gây ra vấn đề về tính kha chuyển cho bất kì các giao diện script nào phụ thuộc vào lệnh `echo`, với ý định chỉ dùng để `echo` mà thôi. Cách xử sự của lệnh:

```
%echo $PATH
```

bây giờ phụ thuộc vào phiên bản nào của `echo`. Nếu trên dòng lệnh vô tình có các dấu sổ ngược (\), thường có trong DOS và Windows, nó có thể bị lệnh `echo` thông dịch nhưng là một lệnh đặc biệt. Sự khác biệt tương tự như kết quả từ hàm `printf(str)` và `printf("%s", str)` nếu chuỗi `str` có chứa kí hiệu phần trăm (%).

Chúng ta chỉ mới nói một phần nhỏ về lệnh `echo`, nhưng nó cũng chỉ ra vấn đề cơ bản: thay đổi các hệ thống có thể phát sinh ra các phiên bản khác nhau của phần mềm, vô tình gây ra vấn đề về tính khả chuyển. Và các vấn đề loại này rất khó giải quyết. Có thể giảm bớt rắc rối bằng cách đặt tên

khác nhau cho các phiên bản của `echo`. Ví dụ dễ thấy nhất là lệnh `sum` của Unix nó in ra kích thước và ký số của một tập tin. Nó dùng để kiểm tra các thông tin truyền đi có thành công hay không:

```
% sum file

52313 2 file

%

% copy file to other machine

%

% telnet othermachine

$

$ sum file

52313 2 file

$
```

Ta thấy kí số của tập tin giống nhau sau khi truyền đi, chứng tỏ bản chính và bản sao là một.

Khi hệ thống giãn nở, các phiên bản biến đổi, và một vài người đã quan sát thấy thuật toán dùng kí số là không hoàn hảo, do đó lệnh `sum` được thay đổi để dùng thuật toán tốt hơn. Một vài người khác cũng quan sát thấy điều tương tự và lại thay đổi để dùng thuật toán tốt hơn. Cứ thế mà hiện nay có rất nhiều phiên bản của lệnh `sum`, mỗi lệnh cho kết quả khác nhau. Chúng ta thử chép một tập tin sang nhiều máy khác nhau để xem kết quả của lệnh `sum` ra sao:

```
% sum file

52313 2 file

%

% copy file to machine 2
```

```
% copy file to machine 3
% telnet machine2
$
$ sum file
eaa0d468   713   file
$ telnet machine3
>
> sum file
62992  1  file
>
```

Phải chăng tập tin đã bị sai, hay là do các phiên bản của sum khác nhau? Có thể là cả hai.

Vậy lệnh sum thực sự là tai họa cho tính khả chuyển; một chương trình dự định giúp đỡ cho việc sao chép một phần mềm từ máy này sang máy khác đã có sự không tương thích về phiên bản, kết quả là dùng lệnh sum không còn có ý nghĩa gì.

Với nhiệm vụ đơn giản của nó, lệnh sum ban đầu là tốt. Việc thay đổi nó để tạo ra một chương trình tốt hơn, nhưng chẳng tốt hơn bao nhiêu, và vô tình đã gây ra không ít phiền toái. Vấn đề ở đây không phải do tính mở rộng mà là do sự không tương thích của chương trình khi dùng cùng tên.

Bảo trì tính tương thích với các chương trình và dữ liệu có sẵn

Khi một phiên bản mới của phần mềm chẳng hạn như xử lý văn bản được phát hành, thông thường có nhu cầu phải xử lý được các tập tin do phiên bản cũ tạo ra. Điều mong đợi là các đặc tính mới được thêm vào sẽ không gây ảnh hưởng xấu. Nhưng các phiên bản mới đôi khi lại gây lỗi khi xử lý các định dạng cũ. Người dùng các phiên bản mới, ngay cả khi họ không dùng đặc tính mới, không thể chia sẻ tập tin với người khác nếu họ

còn dùng phiên bản cũ, vậy là họ buộc phải nâng cấp. Dù là một sự thiếu sót hay chiến lược tiếp thị, thì đây cũng được coi là một kiểu thiết kế rất đáng tiếc.

Tương thích ngược là khả năng của một chương trình có thể chạy được với các phiên bản cũ của nó. Nếu bạn định thay đổi một chương trình thì phải chắc không gây tổn hại cho các phần mềm hay dữ liệu phụ thuộc vào nó. Lập tài liệu cho các thay đổi, và cung cấp các phương thức giao tiếp để nó có thể thích ứng với các đặc tả cũ. Quan trọng nhất là phải xem xét sự thay đổi đó có thực sự cải tiến chương trình so với cái giá phải trả cho tính tương thích không?

8.7. Tổng kết

Phấn đấu để viết mã nguồn có tính khả chuyển là một điều đáng làm, vì mất rất nhiều thời gian để thay đổi một chương trình để cho nó có thể chạy được trên một hệ thống khác, hay khi chính hệ thống mà chương trình đang chạy có sự thay đổi. Tất nhiên tính khả chuyển không tự nhiên mà có, mà đòi hỏi phải cẩn thận khi cài đặt cũng như phải có kiến thức về tính khả chuyển trên tất cả các hệ thống mà chương trình có khả năng được triển khai.

Có hai hướng tiếp cận tính khả chuyển mà chúng ta gọi là hướng hợp nhất và hướng giao nhau. Hướng tiếp cận hợp nhất yêu cầu viết các phiên bản có thể chạy được trên mỗi hệ thống, kết hợp các đoạn mã nguồn càng nhiều càng tốt theo cơ chế biên dịch có điều kiện. Nhược điểm là phải viết nhiều mã nguồn và mã nguồn thường phức tạp, khó cập nhật và kiểm tra.

Hướng tiếp cận giao nhau là viết nhiều mã nguồn có thể chạy mà không thay đổi gì trên các hệ thống càng nhiều càng tốt. Sự phụ thuộc bắt buộc của từng hệ thống có thể đưa vào một tập tin duy nhất, có chức năng như một phương thức giao tiếp giữa chương trình và hệ thống phía dưới. Hướng tiếp cận giao nhau cũng có nhược điểm là làm mất nhiều hiệu suất cũng như đặc trưng riêng của mỗi hệ thống, tuy nhiên xét về mặt lâu dài hướng tiếp cận này có nhiều lợi ích hơn.

Chương 9

KÝ HIỆU

Một ngôn ngữ lập trình theo đúng nghĩa của nó luôn nâng cao tính tiện dụng để giúp cho việc viết chương trình được dễ dàng. Điều này lý giải vì sao mà trong bộ công cụ lập trình của những người lập trình kinh nghiệm không chỉ có những ngôn ngữ lập trình đa dụng như C và các công cụ liên quan mà còn có các ngôn ngữ script, các bộ công cụ tạo giao diện có thể lập trình được và rất nhiều những ngôn ngữ lập trình ứng dụng đặc thù khác.

Sức mạnh của tập ký hiệu tốt thoát ra khỏi phương pháp lập trình truyền thống để đi vào các lĩnh vực giải quyết các bài toán đặc thù. Các biểu thức thông thường cho phép ta viết các định nghĩa cô đọng (đôi khi còn mang vẻ tối nghĩa) của các lớp chuỗi ký tự; HTML cho phép định nghĩa cách bố trí cho các tài liệu tương tác, thường sử dụng các chương trình được nhúng vào các ngôn ngữ khác chẳng hạn như JavaScript, PostScript để biểu diễn toàn bộ tài liệu, chẳng hạn như việc biểu diễn cuốn sách này. Các chương trình bảng tính và xử lý văn bản thường có các ngôn ngữ lập trình như Visual Basic để tính toán các biểu thức, truy xuất thông tin hoặc là điều khiển việc hiển thị.

Nếu bạn cảm thấy rằng mình phải viết quá nhiều mã nguồn để làm một công việc lặp đi lặp lại một cách tẻ nhạt hoặc gặp khó khăn trong việc biểu diễn một quy trình nào đó, thì có lẽ bạn đang sử dụng một ngôn ngữ không phù hợp. Nếu chưa có một ngôn ngữ phù hợp thì đó là cơ hội cho bạn tạo ra một ngôn ngữ riêng cho chính mình. Việc tạo ra một ngôn ngữ không nhất thiết là xây dựng một ngôn ngữ như Java; thông thường thì việc thay đổi các ký hiệu có thể giải quyết các vấn đề rắc rối. Chẳng hạn như, các hàm định dạng chuỗi trong họ hàm `printf` là các hàm điều khiển sự hiển thị các

giá trị một cách cô đọng và đầy đủ ý nghĩa.

Chương này sẽ trình bày cách thức giải quyết các vấn đề thông qua tập ký hiệu, và minh họa một số kỹ thuật có thể được sử dụng để cài đặt những ngôn ngữ mang tính đặc thù. Chúng ta sẽ được xem xét việc sử dụng một chương trình để viết một chương trình khác, cùng với cách sử dụng tập ký hiệu thường dùng.

9.1. Định dạng dữ liệu

Trong thực tế thường có một khoảng cách giữa những gì chúng ta muốn máy tính làm với những gì ta cần phải làm để hoàn tất công việc. Khoảng cách này càng hẹp càng tốt. Tập ký hiệu tốt sẽ giúp ta giải quyết các bài toán dễ dàng hơn. Đôi khi, tập ký hiệu tốt giúp ta có cách hiểu mới, cho phép ta giải quyết các bài toán khó, thậm chí còn đưa đến những khám phá mới.

Các ngôn ngữ nhỏ gọn thường có các tập ký hiệu chuyên biệt dành cho các lĩnh vực hẹp. Các ngôn ngữ này không chỉ cung cấp một giao diện tốt mà còn giúp tổ chức cách cài đặt chương trình. Xét câu lệnh sau:

```
printf("%d %6.2f %-10.10s\n", i, f, s);
```

Mỗi dấu `%` trong chuỗi ký hiệu định dạng đại diện cho một giá trị của một tham số của lệnh `printf`, tiếp theo là một số cờ tùy chọn và độ rộng các trường, ký tự kết thúc biểu diễn kiểu tham số cần xuất. Các ký hiệu này cô đọng, trực quan và dễ viết, việc cài đặt cũng rõ ràng. Các thay đổi trong C++ (`iostream`) và Java (`java.io`) dường như khó khăn hơn bởi vì chúng không cung cấp tập ký hiệu đặc biệt, mặc dù chúng cho phép mở rộng các kiểu dữ liệu do người dùng định nghĩa và hỗ trợ việc kiểm tra kiểu dữ liệu.

Một số cài đặt không chuẩn của hàm `printf` cho phép ta tự thêm vào các quy ước vào các tập chức năng có sẵn của nó. Ví dụ, một trình biên dịch có thể dùng `%L` để biểu diễn số hàng và tên tập tin; một hệ thống đồ họa có thể dùng `%P` để biểu diễn điểm và `%R` để biểu diễn hình chữ nhật. Chuỗi cô đọng các ký tự và số dùng để đọc chỉ số chứng khoán được trình bày

trong Chương 4 cũng dựa trên tinh thần đó, tập ký hiệu cô đọng để sắp xếp các kết hợp dữ liệu chứng khoán.

Ta có thể tổng hợp các ví dụ tương tự như thế trong C và C++. Giả sử ta cần chuyển các gói chứa những kết hợp của các kiểu dữ liệu khác nhau từ một hệ thống này sang hệ thống khác. Như đã thấy trong Chương 8, giải pháp rõ ràng nhất là chuyển thành một biểu diễn văn bản. Khi đó, với một giao thức mạng chuẩn, định dạng biểu diễn sẽ ở dạng nhị phân để tăng hiệu suất chuyển đổi. Làm sao chúng ta có thể viết các đoạn mã nguồn xử lý gói tin trở nên khả chuyển, hiệu quả và dễ dùng?

Hãy tưởng tượng rằng ta dự tính gửi các gói tin chứa 8-bit, 16-bit và 32-bit dữ liệu từ hệ thống này sang hệ thống khác. Trong ANSI C, ta có thể lưu trữ tối thiểu 8 bit dữ liệu kiểu `char` (kiểu ký tự), 16 bit dữ liệu `short` (số nguyên ngắn), 32 bit cho `long` (số nguyên dài), vì thế ta sẽ sử dụng những kiểu dữ liệu này để biểu diễn các giá trị. Sẽ có nhiều loại gói tin: gói tin loại 1 sẽ có 1-byte chỉ định, 2-byte đếm, 1-byte giá trị và 4-byte dữ liệu:

0×01	cnt_1	cnt_0	va_1	$data_3$	$data_2$	$data_1$	$data_0$

Gói tin loại 2 sẽ chứa 1 số nguyên ngắn và 2 từ dữ liệu số nguyên dài:

)1×02	cnt_1	cnt_0	dwl_3	dwl_2	dwl_1	dwl_0	$dw2_3$	$dw2_2$	$dw2_1$	$dw2_0$

Một cách tiếp cận là viết các hàm nén và giải nén cho mỗi loại gói tin tương ứng:

```
int pack_type1(unsigned char *buf, unsigned short
cout,
        unsigned char val, unsigned long data)
```

```
{
        unsigned char *bp;
        bp = buf;
        *bp++ = 0x01;
        *bp++ = count >> 8;
        *bp++ = count;
        *bp++ = val;
        *bp++ = data >> 24;
        *bp++ = data >>16;
        *bp++ = data >> 8;
*bp++ = data;
        return bp - buf;
}
```

Đối với giao thức thực tế, sẽ có hàng tá thủ tục, tất cả các biến thể liên quan đến cùng một chủ đề. Các thủ tục có thể được đơn giản hoá bằng cách sử dụng các macro hay các hàm để kiểm soát những kiểu dữ liệu cơ bản (số nguyên ngắn, số nguyên dài, v.v...), nhưng nếu như vậy thì các đoạn mã lặp đi lặp lại rất dễ bị sai, khó đọc và khó bảo trì.

Sự lặp đi lặp lại vốn có của các đoạn mã nguồn là một đầu mối mà các ký hiệu có thể có ích. Mượn ý tưởng từ hàm `printf`, chúng ta có thể định nghĩa một một ngôn ngữ đặc tả nhỏ cho phép mỗi gói tin được định nghĩa bằng một chuỗi ngắn gọn, nắm bắt định dạng (layout) của gói tin. Các thành phần kế tiếp của gói tin được mã hoá với `c` cho một ký tự 8-bit, `s` cho một số nguyên ngắn 16-bit và `l` cho một số nguyên dài 32-bit. Vì vậy, chẳng hạn như một gói tin loại 1 tạo ra từ ví dụ trên, bao gồm cả byte kiểu dữ liệu, sẽ được mô tả bởi chuỗi định dạng `cscl`. Sau đó, chúng ta có thể sử dụng

một hàm nén đơn để tạo ra các gói tin của bất kỳ loại nào; gói tin này sẽ được tạo ra với

```
pack(buf, "cscl", 0x01, count, val, data);
```

Do chuỗi định dạng dữ liệu chỉ bao gồm các định nghĩa dữ liệu cho nên không cần phải dùng các ký tự % như trong hàm printf.

Trong thực tế, các thông tin ở đầu gói tin sẽ thông báo cho phía nhận gói tin cách giải mã những phần còn lại, nhưng chúng ta giả sử rằng byte đầu tiên của gói tin có thể được dùng để xác định định dạng. Người gửi mã hoá dữ liệu theo định dạng này và gửi đi; người nhận đọc gói tin, lấy ra byte đầu tiên và dùng nó để giải mã những phần còn lại.

Sau đây là cài đặt của gói tin sẽ lấp đầy buf với các biểu diễn đã mã hoá của các tham số như đã xác định bằng định dạng trên. Chuyển tất cả các dữ liệu thành không dấu, bao gồm các byte trong bộ nhớ đệm gói tin, để tránh các vấn đề về dấu mở rộng (sign-extension). Ở đây cũng sử dụng một số kiểu định nghĩa dữ liệu quy ước để cho việc khai báo ngắn gọn hơn:

```
typedef unsigned char uchar;

typedef unsigned short ushort;

typedef unsigned long ulong;
```

Cũng giống như hàm printf, strcpy và các hàm tương tự khác, hàm nén (pack) giả sử rằng bộ nhớ đệm đủ lớn để lưu kết quả, và trách nhiệm của hàm thực hiện lời gọi là đảm bảo điều này. Không cần phải kiểm tra tính không hợp kiểu dữ liệu giữa định dạng và danh sách tham số.

```
#include <stdarg.h>

/* pack: nén các mục nhị phân vào buf, và trả về
chiều dài */

int pack(uchar *buf, char *fmt,...)

{
```

```
va_list args;
char *p;
uchar *bp;
ushort s;
ulong l;

bp = buf;
va_start(args, fmt);
for  (p = fmt; *p != '\0'; p++) {
      switch (*p) {
      case 'c': /* char  */
            *bp++ = va_arg(args, int);
            break;
      case 's': /* short  */
            s = va_arg(args, int);
            *bp++ = s >> 8;
            *bp++ = s;
            break;
      case 'l':  /* long */
            l = va_arg(args, ulong);
            *bp++ = l >> 24;
            *bp++ = l >> 16;
            *bp++ = l >> 8;
            *bp++ = l;
```

```
                    break;
                default:  /* ký tự không hợp lệ */
                    va_end(args);
                    return -1;
                }
            }
            va_end(args);
            return bp - buf;
        }
```

Hàm `pack` dùng `stdarg.h` một cách mở rộng hơn hàm `eprintf` trong Chương 4. Các tham số trước đó được tách ra sử dùng macro `va_arg`, tham số đầu tiên có kiểu `va_list` thiết lập bằng cách gọi `va_start` và toán hạng tiếp theo là kiểu của tham số (đó là lý do tại sao `va_arg` phải là một macro, không phải là một hàm). Khi thực hiện xong, `va_end` phải được gọi. Mặc dù các tham số đối với `'c'` và `'s'` đại diện cho các giá trị kiểu `char` và `short` nhưng chúng cũng phải được tách ra thành các số nguyên `int` bởi vì C chuyển `char` và `short` thành `int` khi chúng được biểu diễn bằng một tham số có dạng tham số... (ba dấu chấm).

Mỗi thủ tục `pack_type` có chiều dài một dòng, sắp xếp các tham số thành một hàm gọi hàm `pack`:

```
    /* pack_type1: nén gói tin có kiểu định dạng 1 */
    int pack_type1 (uchar *buf, ushort count, uchar
val, ulong data)
    {
        return pack(buf, "cscl", 0x01, count, val ,
data);
    }
```

Để giải nén, chúng ta có thể làm giống như vậy: thay vì viết các đoạn mã nguồn riêng biệt để cắt mỗi định dạng gói tin, ta gọi một hàm `unpack` với một chuỗi định dạng. Điều này tập trung sự chuyển đổi vào một nơi:

```
/* unpack: giải nén các mục từ buf, trả về chiều dài */
    int unpack(uchar *buf, char *fmt, ...)
    {
        va_list args;
        char *p;
        uchar *bp, *pc;
        ushort  *ps;
        ulong  *pl;

        bp = buf;
        va_start(args, fmt);
        for  (p = fmt; *p != '\0'; p++) {
            swicht (*p) {
            case 'c': /* char  */
                pc = va_arg(args, uchar*);
                *pc = *bp++;
                break;
            case 's': /* short  */
                ps = va_arg(args, ushort*);
                *ps  = *bp++ << 8;
                *ps |= *bp++;
```

```
                    break;
                case 'l':  /* long */
                    pl = va_arg(args, ulong*);
                    *pl  = *bp++ << 24;
                    *pl |= *bp++ << 16;
                    *pl |= *bp++ << 8;
                    *pl |= *bp++;
                    break;
                default:  /* ký tự không hợp lệ */
                    va_end(args);
                    return -1;
                }
            }
            va_end(args);
            return bp - buf;
        }
```

Giống như hàm `scanf`, hàm `unpack` phải trả về nhiều giá trị cho hàm gọi nó, do đó các tham số của nó là các con trỏ chỉ đến các biến lưu trữ các kết quả. Giá trị của hàm là số byte trong gói tin, có thể dùng giá trị này để kiểm tra lỗi.

Do các giá trị dữ liệu không mang dấu và do ANSI C định nghĩa kích thước kiểu dữ liệu cho nên đoạn mã nguồn này có thể chuyển đổi dữ liệu một cách khả chuyển và thậm chí giữa hai máy có kích thước dữ liệu `short` và `long` khác nhau. Nếu như chương trình sử dụng hàm `pack` không gửi một số `long` (chẳng hạn), một giá trị không thể biểu diễn bằng 32 bit, thì giá trị sẽ nhận được một cách chính xác. Kết quả là chúng ta chuyển được

32 bit thấp của giá trị đó. Nếu muốn những giá trị lớn hơn thì chúng ta phải định nghĩa một định dạng khác.

Các hàm giải nén theo từng kiểu dữ liệu gọi đến hàm unpack rất dễ viết:

```
/* unpack_type2: giải nén và xử lý gói tin loại 2
*/
int unpack_type2(int n, uchar *buf)
{
        uchar c;
        ushort count;
        ulong dw1, dw2;

        if (unpack(buf, "csll", &c, &count, &dw1,
&dw2) != n)
            return -1;
        assert(c == 0x02);
        return process_type2(count, dw1, dw2);
}
```

Để gọi hàm unpack_type2, trước tiên chúng ta phải biết rằng chúng ta có một gói tin loại 2, sử dụng một vòng lặp nhận như sau:

```
while ((n = readpacket(network, buf, BUFSIZ)) >
0) {
        switch (buf[0]) {
        default:
```

```
            eprintf("bad packet type 0x%x", buf[0]);
                break;
        case 1:
                unpack_type1(n, buf);
                break;
        case 2:
                unpack_type2(n, buf);
                break;
        ...
        }
}
```

Phong cách lập trình này còn có thể phát triển hơn nữa. Một phương thức cô đọng hơn là định nghĩa một bảng các con trỏ hàm mà các đầu vào là các hàm giải nén được đánh chỉ mục theo loại:

```
int (*unpackfn[])(int, uchar *) = {
        unpack_type0,
        unpack_type1,
        unpack_type2,
};
```

Mỗi hàm trong bảng thực hiện phân tích một gói tin, kiểm tra kết quả trả về và khởi tạo các xử lý tiếp theo cho gói tin đó. Bảng này giúp cho công việc của người nhận tiến triển dễ dàng hơn:

```
/* receive: đọc các gói tin từ mạng, xử lý chúng */
void receive(int network)
{
```

```
        uchar type, buf[BUFSIZ];
        int n;
        while ((n = readpacket(network, buf, BUFSIZ)) >
0) {
        type = buf[0];
        if (type >= NELEMS(unpackfn))
                eprintf("bad packet type 0x%x", type);
                        if ((*unpackfn[type])(n, buf) <0)
                        eprintf ("protocol error, type %x
length %d", type, n);
                }
        }
```

Mỗi đoạn mã nguồn xử lý gói tin cô đọng, riêng biệt và dễ bảo trì. Người nhận hoàn toàn độc lập về giao thức, đồng thời nó cũng rõ ràng và nhanh.

Ví dụ này dựa trên một số đoạn mã thực của một giao thức mạng thương mại. Khi tác giả nhận ra rằng hướng tiếp cận này có thể thực hiện được, vài ngàn hàng mã nguồn lặp đi lặp lại, dễ bị sai sót đã thu gọn thành vài trăm hàng dễ dàng bảo trì. Việc ký hiệu hoá (các ký hiệu) đã giảm rất nhiều những sự lộn xộn.

Bài tập 9-1. Hãy sửa hàm `pack` và `unpack` để chuyển các giá trị có dấu một cách đúng đắn, giữa các máy có kích thước `short` và `long` khác nhau. Bạn sẽ sửa các chuỗi định dạng như thế nào để đặc tả một mục dữ liệu có dấu? Làm thế nào để kiểm tra đoạn mã để xác định rằng nó chuyển đúng đắn `a-1` từ một máy tính kiểu long 32-bit sang một máy kiểu long 64-bit?

Bài tập 9-2. Hãy mở rộng hàm `pack` và `unpack` để kiểm soát dữ liệu kiểu chuỗi; một khả năng là bao gồm cả chiều dài của chuỗi trong chuỗi

định dạng. Hãy mở rộng chúng để kiểm soát các mục lặp đi lặp lại với một biến đếm. Làm thế nào để hàm này tương tác với mã hoá của các chuỗi?

Bài tập 9-3. Bảng các con trỏ hàm trong chương trình C nêu trên là lõi của các hàm ảo trong C++. Hãy viết lại hàm `pack` và `unpack` và `receive` trong C++ để ứng dụng ưu điểm của phương pháp ký hiệu vượt trội này.

Bài tập 9-4. Hãy viết một phiên bản dòng lệnh (command-line) của hàm `printf` cho phép in các tham số thứ 2 và các tham số tiếp theo của nó theo định dạng đưa ra bởi tham số thứ nhất. Một số shells đã cung cấp điều này như là một phần dựng sẵn.

Bài tập 9-5. Hãy viết một hàm cài đặt các đặc tả định dạng thường thấy trong các chương trình bảng tính hay trong lớp `DecimalFormat` của Java, cho phép hiển thị các con số theo những mẫu chỉ định số nào là bắt buộc hoặc tuỳ chọn, vị trí của dấu chấm thập phân và dấu phẩy, v.v... Ví dụ, chuỗi định dạng:

```
##,##0.00
```

chỉ ra rằng một số có 2 số lẻ, ít nhất một con số bên trái của dấu chấm thập phân, một dấu phẩy sau con số hàng ngàn, và một điền vào khoảng trắng cho đến hàng chục ngàn. Nó sẽ biểu diễn số 12345.67 thành `12,345.67` và .4 thành `_____0.40` (sử dụng dấu gạch dưới để đại diện cho khoảng trắng). Xem thêm định nghĩa của `DecimalFormat` hoặc các chương trình bảng tính để biết thêm chi tiết về đặc tả.

9.2. Các biểu thức có quy tắc

Các chỉ dẫn định dạng của hàm `pack` và `unpack` là những ký hiệu rất đơn giản dùng để định nghĩa cách thức trình bày của các gói tin. Chủ đề tiếp theo của chúng ta phức tạp hơn một chút nhưng là những ký hiệu dễ hiểu, *các biểu thức có quy tắc*, cho phép xác định mẫu của đoạn văn bản. Chúng tôi thường sử dụng các biểu thức có quy tắc trong suốt cuốn sách mà không cần phải giải thích nhiều. Mặc dù các biểu thức có quy tắc phổ biến trong môi trường lập trình Unix nhưng chúng không được sử dụng rộng rãi trong

các hệ thống khác, vì vậy phần này chúng tôi chỉ minh họa một số điểm mạnh của chúng. Trong trường hợp bạn không có sẵn thư viện biểu thức có quy tắc trong tay thì chúng tôi cũng sẽ đưa ra một cài đặt thô sơ.

Có một số đặc trưng của các biểu thức có quy tắc, nhưng nhìn chung thì chúng giống nhau, một cách mô tả các mẫu gồm các ký tự cơ bản, cùng với việc lặp lại, thay thế và viết tắt cho các lớp của các ký tự như số và chữ. Một ví dụ tương tự được gọi là "ký tự đại diện" (wildcard) sử dụng trong các trình xử lý dòng lệnh để tìm các mẫu của các tập tin. Thường thì `*` có nghĩa là "bất ký chuỗi ký tự nào", do đó ví dụ như một dòng lệnh

```
C:\> del *.exe
```

dùng một mẫu khớp với tất cả các tập tin có tên là bất kỳ chuỗi ký tự nào kết thúc bằng "`.exe`". Thông thường thì các chi tiết khác nhau trong các hệ thống khác nhau, và thậm chí trong các chương trình khác nhau cũng khác nhau.

Mặc dù những tinh thất thường của các chương trình khác nhau có thể gợi ý rằng các biểu thức có quy tắc là một kỹ thuật phức tạp, nhưng thực tế chúng là một ngôn ngữ có ngữ pháp hình thức và ngữ nghĩa chính xác nhờ cách phát âm trong ngôn ngữ này. Hơn nữa, những cài đặt đúng sẽ chạy rất nhanh; một sự kết hợp giữa lý thuyết và kiến trúc thực tế sẽ tạo ra nhiều sự khác biệt, một ví dụ của ưu điểm của các thuật toán chuyên biệt mà chúng tôi đã đề cập đến trong Chương 2.

Một biểu thức có quy tắc là chuỗi các ký tự định nghĩa một tập hợp các chuỗi dùng cho việc tìm kiếm. Hầu hết các ký tự chỉ tương đương với chúng mà thôi, vì vậy biểu thức có quy tắc `abc` sẽ tương đương với chuỗi các ký tự xuất hiện bất cứ ở đâu. Thêm vào đó, một số siêu ký tự (*metacharacter*) chỉ định sự lặp lại, nhóm hay vị trí. Trong quy ước biểu thức có quy tắc trên Unix, `^` đại diện cho việc bắt đầu một chuỗi và `$` kết thúc chuỗi, do đó `^x` tương đương với một chuỗi chỉ có `x` ở đầu, `x$` tương đương với một chuỗi chỉ có 1 `x` ở cuối, `^x$` tương đương với chuỗi chỉ có `x` nếu nó là ký tự duy nhất trong chuỗi và `^$` tương đương với 1 chuỗi rỗng.

Ký tự "." tương đương với bất kỳ ký tự nào, do đó `x.y` tương đương `xay`, `x2y`, v.v..., nhưng không phải là `xy` hay `xaby`, và `^.$` tương đương với một chuỗi chỉ có một ký tự bất kỳ.

Một tập hợp các ký tự trong dấu `[]` tương đương với bất kỳ một ký tự nào trong tập hợp đó, do đó `[0123456789]` tương đương với một số; nó có thể được viết tắt là `[0-9]`.

Những khối này được phối hợp với dấu ngoặc đơn để biểu diễn nhóm, | để biểu diễn thay thế, * để biểu diễn không hoặc nhiều thể hiện, + để biểu diễn một hoặc nhiều thể hiện, và ? để biểu diễn không hoặc một thể hiện. Cuối cùng, \ được dùng như một tiếp đầu ngữ để khai báo một siêu ký tự và tắt ý nghĩa đặc biệt của nó; `\*` là một từ khóa * và `\\` là một từ khóa backslash.

Công cụ biểu thức có quy tắc nổi tiếng nhất là chương trình `grep` mà đã được đề cập đến một vài lần. Chương trình này là một ví dụ tuyệt vời về giá trị của ký hiệu. Nó áp dụng một biểu thức có quy tắc cho mỗi dòng trong tập tin đầu vào và in những hàng này ra chứa các chuỗi tương ứng. Đặc tả đơn giản này, cùng với những điểm mạnh của các biểu thức có quy tắc, cho phép ta giải quyết nhiều công việc phải làm từ ngày này sang ngày khác. Trong ví dụ sau, hãy lưu ý là cú pháp của biểu thức có quy tắc được dùng trong tham số của `grep` khác với các ký tự đại diện dùng để chỉ định một tập hợp các tập tin; sự khác biệt này phản ánh những cách dùng khác nhau.

Tập tin nguồn nào sử dụng lớp `Regexp`?

```
% grep Regexp *.java
```

Cái gì cài đặt nó?

```
% grep 'class.*Regexp' *.java
```

Tôi đã lưu thư của Bob ở đâu?

```
% grep '^From:.*bob@' mail/*
```

Có bao nhiêu dòng mã nguồn khác rỗng trong chương trình này?

```
% grep '.' *.c++ | wc
```

Với các cờ để in thứ tự dòng của các dòng tìm thấy, số lượng kết quả tìm thấy, thực hiện tìm kiếm có phân biệt chữ hoa chữ thường, đảo ngược kết quả tìm kiếm được (chọn các dòng không khớp chuỗi cần tìm) và thực hiện các phép biến đổi khác dựa trên ý tưởng cơ bản, `grep` được dùng phổ biến đến nỗi nó đã trở thành một ví dụ cổ điển của việc lập trình dựa trên các công cụ.

Thật không may, không phải hệ thống nào cũng có `grep` hay chương trình tương tự như vậy đi kèm. Một số hệ thống có một thư viện các biểu thức có quy tắc, thường được gọi là `regex` hay `regexp`, cho phép bạn dùng để viết một phiên bản của `grep`. Nếu cả hai thứ đó đều không có thì cũng không khó khăn gì để cài đặt một tập con vừa phải của ngôn ngữ biểu thức có quy tắc hoàn chỉnh. Ở đây, chúng tôi giới thiệu một cài đặt của các biểu thức có quy tắc và `grep` đi cùng; để đơn giản chỉ có các siêu ký tự là `^$.` và `*`, trong đó `*` chỉ sự lặp đi lặp lại của một dấu chấm trước đó hay ký tự cơ bản. Tập con này cung cấp rất nhiều điểm mạnh với chỉ một ít độ phức tạp của lập trình nhờ vào các biểu thức tổng quát.

Hãy bắt đầu với chính hàm tìm kiếm chuỗi ký tự. Đó là xác định xem một chuỗi ký tự có chứa một biểu thức cần tìm hay không:

```
/* match:  tìm kiếm regexp ở bất kỳ vị trí nào
trong văn bản */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* phải xét cả trường hợp chuỗi
rỗng */
        if (matchhere(regexp, text))
```

```
            return 1;
        } while (*text++ != '\0');
        return 0;
    }
```

Nếu biểu thức cần tìm bắt đầu với ^ thì chuỗi ký tự phải bắt đầu với một tương ứng của những phần còn lại trong biểu thức. Ngược lại, chúng ta sẽ duyệt tiếp chuỗi ký tự, sử dụng hàm `matchhere` để kiểm tra xem chuỗi ký tự có được tìm thấy ở bất kỳ vị trí nào hay không. Khi chúng ta tìm thấy một kết quả, chúng ta đã hoàn tất. Hãy lưu ý cách dùng vòng lặp `do-while`: các biểu thức có thể phù hợp với chuỗi rỗng (ví dụ như `$` sánh với với chuỗi rỗng ở cuối dòng và `.*` tương ứng với bất kỳ số ký tự nào, kể cả không có ký tự nào), do đó chúng ta phải gọi hàm `matchhere` ngay cả khi chuỗi rỗng.

Hàm đệ quy `matchhere` thực hiện hầu hết các công việc:

```
    /* matchhere:  tìm kiếm regexp từ vị trí đầu tiên
của văn bản */
    int matchhere(char *regexp, char *text)
    {
        if (regexp[0] == '\0')
            return 1 ;
        if (regexp[1] == '*')
            return matchstar(regexp[0], regexp+2,
text);
        if (regexp[0] == '$' && regexp[1] == '\0')
            return *text  == '\0';
        if   (*text!='\0'   &&   (regexp[0]=='.'   ||
regexp[0]==*text))
```

```
            return matchhere(regexp+1, text+1);
        return 0;
    }
```

Nếu biểu thức `regexp` có giá trị rỗng thì việc tìm kiếm sẽ kết thúc và do đó chúng ta đã tìm được kết quả. Nếu biểu thức kết thúc với ký tự `$` thì kết quả sẽ chỉ được tìm thấy ở cuối văn bản. Nếu biểu thức bắt đầu bằng dấu chấm thì sẽ tìm kiếm bất kỳ ký tự nào. Ngược lại, biểu thức bắt đầu với ký tự thông thường sẽ thực hiện tìm kiếm chính nó trong phần văn bản đó. Ký tự `^` hoặc `$` nếu xuất hiện ở giữa biểu thức tìm kiếm sẽ được xem như một ký tự thông thường khác, chứ không được xem như ký tự đặc biệt.

Nên chú ý rằng các lời gọi `matchhere` sẽ gọi lại chính bản thân chúng sau khi tìm được một ký tự của mẫu và chuỗi, vì vậy số lần thực hiện của phép đệ quy sẽ tương ứng với chiều dài của mẫu.

Việc tìm kiếm sẽ gặp khó khăn nếu biểu thức bắt đầu với ký tự thay thế với dấu sao, chẳng hạn `x*`. Ta gọi hàm `matchstar` với đối số đầu tiên là toán hạng của ký tự thay thế dấu sao `(x)`, và những đối số kế tiếp là mẫu đứng sau dấu sao và phần văn bản.

```
    /* matchstar: tìm kiếm c*regexp từ vị trí đầu
tiên của văn bản */
    int matchstar(int c, char *regexp, char *text)
    {
        do { /* dấu * không hoặc thay thế nhiều ký
tự */
            if (matchhere(regexp, text))
                return 1;
        } while (*text != '\0' && (*text++ == c ||
c == '.'));
```

```
        return 0;
    }
```

Một vòng lặp `do-while` khác một lần nữa được kích hoạt với yêu cầu là biểu thức tìm kiếm `x*` có thể tương ứng với ký tự kết thúc chuỗi. Vòng lặp sẽ thực hiện kiểm tra xem văn bản có tương ứng với phần biểu thức còn lại hay không. thực hiện kiểm tra tại mỗi vị trí của phần văn bản miễn là ký tự đầu tiên tương ứng với toán hạng của dấu sao.

Phải thừa nhận đây là việc cài đặt không phức tạp nhưng vẫn hoạt động tốt, và với ít hơn 30 dòng mã nguồn, nó đã cho thấy các biểu thức tìm kiếm không cần phải sử dụng những kỹ thuật tiên tiến.

Chúng tôi sẽ bàn đến việc mở rộng chương trình này sau. Bây giờ, chúng ta sẽ viết một phiên bản của `grep` có sử dụng hàm `match`. Và đây là thủ tục chính:

```
/* grep main: tìm kiếm regexp trong các tập tin */
    int main(int argc, char *argv[])
    {
        int i, nmatch;
        FILE *f;

        setprogname("grep");
        if (argc < 2)
            eprintf("usage: grep regexp [file ...]");
        nmatch = 0;
        if (argc == 2) {
            if (grep(argv[1], stdin, NULL))
            nmatch++;
```

```
        } else {
        for (i = 2; i < argc; i++) {
            f = fopen(argv[i], "r");
            if (f  == NULL) {
                weprintf ("can't open %s:", argv[i]);
                    continue;
                }
                 if  (grep(argv[1],  f,  argc>3  ?
argv[i] : NULL) >0)
                    nmatch ;
                fclose(f);
            }
        }
        return nmatch  == 0;
    }
```

Thông thường, các chương trình C trả về giá trị 0 nếu thành công và trả về giá trị khác không cho các loại lỗi khác nhau. Chương trình `grep` này sẽ cho biết quá trình tìm kiếm có thành công hay không trong quá trình tìm kiếm. Vì vậy, sẽ trả về giá trị 0 nếu không tìm thấy kết quả cần tìm, trả về 1 nếu không tìm thấy, và trả về giá trị 2 (thông qua hàm `eprintf`) nếu có lỗi xảy ra. Các chương trình khác có thể kiểm tra các giá trị trạng thái này.

Hàm `grep` duyệt qua từng dòng và thực hiện hàm `match` trên mỗi dòng:

```
/* grep: tìm kiếm regexp trong tập tin */
int grep(char *regexp, FILE *f , char *name)
```

```
    int n, nmatch;
    char buf[BUFSIZ];
    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n>0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)){
            nmatch++;
            if (name != NULL)
                printf ("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

Thủ tục chính sẽ không kết thúc được nếu như không mở được tập tin. Cách thiết kế này được lựa chọn vì ta thường dùng lệnh như sau:

```
% grep herpolhode *.*
```

và nhận ra rằng một trong số các tập tin thuộc thư mục đó không đọc được. Tốt hơn hết là nên thiết kế hàm `grep` sao cho nó vẫn tiếp tục thực hiện công việc sau khi thông báo lỗi xảy ra. Hàm này không nên kết thúc giữa chừng và buộc người sử dụng gõ vào cả một danh sách tập tin một cách thủ công

chỉ vì muốn tránh tập tin gây ra lỗi. Tương tự, cần lưu ý rằng thủ tục `grep` in ra tên tập tin và dòng văn bản tìm kiếm nếu nó đang đọc từ dữ liệu nhập chuẩn hoặc một tập tin. Cách thiết kế này có vẻ hơi lạ nhưng nó phản ánh phong cách sử dụng thường thấy dựa trên kinh nghiệm. Khi chỉ cho biết một dữ liệu đầu vào duy nhất, nhiệm vụ của `grep` thường là phải chọn lựa và tên các tập tin sẽ làm lộn xộn dữ liệu xuất. Nhưng nếu như yêu cầu `grep` tìm kiếm trên nhiều tập tin, công việc thường gặp nhất là tìm kiếm tất cả những lần xuất hiện với thông tin đầy đủ về điều cần tìm.

Hãy so sánh:

```
% strings markov.exe | grep 'DOS mode'
```

với

```
% grep grammer chapter*.txt
```

Đây chính là lý do vì sao `grep` lại trở nên thông dụng, và cho thấy rằng ký hiệu phải đi kèm với tính tự nhiên để có thể xây dựng được một công cụ hiệu quả.

Hàm `match` sẽ kết thúc và trả về giá trị ngay khi tìm thấy kết quả. Nhưng để cài đặt chức năng thay thế (tìm kiếm và thay thế) trong một trình soạn thảo văn bản, kết quả tìm kiếm *dài nhất tận cùng bên trái* sẽ càng thích hợp hơn. Chẳng hạn như cho trước đoạn văn bản `"aaaaa"`, mẫu `a*` sẽ tương xứng với chuỗi rỗng ngay từ vị trí đầu tiên, nhưng sẽ tự nhiên hơn nếu xem như tương ứng với cả năm chữ `a`. Để hàm `match` tìm được chuỗi ký tự dài nhất tận cùng bên trái, ta phải viết lại hàm `matchstar` để thực hiện được nhiều hơn: thay vì chỉ tìm tại mỗi ký tự từ trái sang phải, nó cần bỏ qua chuỗi dài nhất tương ứng với toán hạng của dấu sao, sau đó duyệt ngược trở lại nếu phần còn lại của chuỗi không tương ứng với phần còn lại của mẫu. Nói cách khác là sẽ thực hiện duyệt từ phải sang trái. Sau đây là một phiên bản của hàm `matchstar` thực hiện việc tìm kiếm dài nhất tận cùng bên trái:

```
/* matchstar: tìm kiếm dài nhất tận cùng bên trái cho
c*regexp */
```

```
int matchstar(int c, char *regexp, char *text)
{
        char *t;

        for (t = text; *t != '\0' && (*t == c || c
== '.'); t++)                ;
        do {  /* dấu * không hoặc thay thế nhiều ký
tự */
            if (matchhere(regexp, t))
                return 1;
        }while (t-- > text);
        return 0;
}
```

Không cần quan tâm đến kết quả tương xứng mà `grep` tìm được vì nó chỉ kiểm tra sự hiện diện của bất kỳ một kết quả tìm kiếm và in ra nguyên cả dòng đó. Do vậy, vì việc tìm kiếm chuỗi dài nhất tận cùng bên trái chỉ là một công việc phụ nên không cần thiết cho hàm `grep`, nhưng đối với việc thay thế chuỗi thì nó lại cần thiết.

Hàm `grep` cũng gặp khó khăn với những phiên bản được hệ thống hỗ trợ. Một số biểu thức rắc rối sẽ gây nên việc thực hiện có tính lũy thừa, chẳng hạn `a*a*a*a*a*b` khi cho dữ liệu đầu vào là `aaaaaaaaac`, nhưng việc thực hiện có tính lũy thừa này cũng xuất hiện trong một số chương trình thương mại. Một biến thể khác của `grep` hiện có sẵn trong Unix được gọi là `egrep` đã sử dụng thuật toán so sánh phức tạp hơn nhiều. Thuật toán này đảm bảo việc thực hiện mang tính chất tuyến tính bằng cách tránh việc quay lui khi việc tìm kiếm từng phần thất bại.

Vậy còn việc hàm `match` xử lý toàn bộ các biểu thức tìm kiếm thì

sao? Điều đó bao gồm các nhóm ký tự như `[a-zA-Z]` để tìm ký tự chữ cái, khả năng nhận biết ký tự đặc biệt (chẳng hạn như tìm kiếm một dấu chấm), các dấu ngoặc dùng cho việc gom nhóm, và các lựa chọn như (`abc` hoặc `def`). Bước đầu tiên là giúp hàm `match` biên dịch mẫu này thành cách biểu diễn sao cho dễ dàng phân tích hơn. Việc phân tích loại ký tự mỗi khi chúng ta so sánh chúng với một ký tự sẽ không tốt; trong khi đó một cách biểu diễn có tính toán trước dựa trên các vector theo bit sẽ làm cho các loại ký tự trở nên hiệu quả hơn. Với các biểu thức thông thường đầy đủ với các dấu ngoặc và các lựa chọn, việc cài đặt cần phải phức tạp hơn, nhưng vẫn có thể sử dụng một vài kỹ thuật mà chúng ta đã đề cập trong phần sau của chương này.

Bài tập 9-6. Hãy so sánh việc thực hiện của hàm `match` với hàm `strstr` khi thực hiện tìm kiếm trên văn bản thường.

Bài tập 9-7. Hãy viết một phiên bản hàm `matchhere` khác không đệ quy và so sánh việc thực hiện của nó với phiên bản có đệ quy.

Bài tập 9-8. Thêm một số tùy chọn cho hàm `grep`: `-v` dùng để thực hiện tìm kiếm đảo ngược, `-i` để thực hiện so sánh chữ cái không phân biệt chữ hoa và chữ thường, và `-n` để cho biết chỉ số dòng trong kết quả xuất. Cách thức các chỉ số dòng được in ra như thế nào? Có nên in chúng ra trên cùng dòng với văn bản cần tìm.

Bài tập 9-9. Thêm các toán tử `+` (một hoặc nhiều) và `?` (không hoặc một) để tìm kiếm. Mẫu `a+bb?` sẽ tìm một hoặc nhiều chữ `a` và theo sau là một hoặc hai chữ `b`.

Bài tập 9-10. Việc cài đặt hiện thời của hàm `match` sẽ làm cho ý nghĩa đặc biệt của `^` và `$` không còn nữa nếu chúng không bắt đầu hoặc kết thúc biểu thức, và của dấu `*` nếu nó không đi theo ngay sau một ký tự thông thường hoặc một dấu chấm. Một cách thiết kế thông thường là đánh dấu các ký tự đặc biệt bằng cách thêm vào trước nó dấu `\`. Sửa lại hàm `match` để xử lý các dấu `\` theo cách này.

Bài tập 9-11. Hãy thêm các loại ký tự vào hàm `match`. Các loại ký

tự xác định một sự tương ứng cho bất kỳ ký tự nào xuất hiện trong ngoặc. Việc thêm vào phạm vi xem xét sẽ giúp chương trình trở nên tiện lợi hơn, chẳng hạn `[a-z]` dùng để xét các ký tự chữ thường, và thay đổi ý nghĩa, chẳng hạn `[^0-9]` dùng để thực hiện tìm bất kỳ ký tự nào *ngoại trừ* chữ số.

Bài tập 9-12. Thay đổi hàm `match` để sử dụng phiên bản `matchstar` để tìm chuỗi dài nhất tận cùng bên trái, và sửa đổi chúng để có thể trả về các vị trí ký tự đầu tiên và cuối cùng của chuỗi tìm được. Dùng chương trình này để xây dựng một chương trình `gres` tương tự với `grep` nhưng in tất cả các dòng dữ liệu đầu vào sau khi thay thế chuỗi cần tìm trong mẫu bằng chuỗi mới, như sau:

```
% gres 'homoiousian' 'homoousian' mission.stmt
```

Bài tập 9-13. Hãy chỉnh đổi `match` và `grep` để chúng có thể thực thi được với các chuỗi UTF-8 chứa các ký tự Unicode. Vì UTF-8 và Unicode là tập con của bảng mã ASCII, sự thay đổi này mang tính tương thích tiến. Các biểu thức cần tìm cũng như văn bản gốc đều cần thực thi tốt với UTF-8. Vậy thì các loại ký tự nên được cài đặt như thế nào?

Bài tập 9-14. Hãy viết chương trình kiểm chứng tự động cho các biểu thức cần tìm. Chương trình này sẽ phát sinh các biểu thức kiểm chứng và các chuỗi kiểm chứng để tìm kiếm. Nếu có thể, bạn hãy dùng các thư viện có sẵn để tham khảo cách thức cài đặt; cũng có thể bạn sẽ tìm được các lỗi trong những thư viện đó.

9.3. Một số công cụ lập trình

Nhiều công cụ được cấu trúc dựa vào một *ngôn ngữ dành cho mục đích đặc biệt*. Chương trình `grep` chỉ là một trong số các họ công cụ sử dụng các biểu thức thông thường hoặc sử dụng các ngôn ngữ khác để giải quyết các bài toán lập trình.

Một trong những ví dụ đầu tiên là trình thông dịch lệnh hay ngôn ngữ điều khiển công việc. Trình tự các câu lệnh có thể được ghi vào một tập tin đã được đưa vào áp dụng rất sớm, và một phiên bản của trình thông dịch

hay *shell* có thể thi hành các lệnh trong tập tin đó. Từ đó, chỉ cần thêm các tham số, điều kiện, vòng lặp, biến, và tất cả các trường hợp khác trong một ngôn ngữ lập trình truyền thống. Điểm khác nhau chủ yếu là ở đây chỉ có một kiểu dữ liệu - đó là `string` - và các phép toán trong các chương trình *shell* có khuynh hướng là các chương trình hoàn chỉnh thực hiện các tính toán thú vị. Mặc dù lập trình *shell* không còn được ưa chuộng, thường thì chúng được chuyển sang các lựa chọn khác như Perl trong các môi trường lập trình lệnh và các nút nhấn trong các giao diện người dùng dạng đồ họa, nhưng nó vẫn là một cách hữu hiệu để xây dựng các điều khiển phức tạp.

Awk là một công cụ khả lập trình khác, nó là một ngôn ngữ nhỏ, chuyên về các thao tác mẫu (specialized pattern-action); nó tập trung vào việc lọc và biến đổi một luồng dữ liệu nhập. Như chúng ta đã thấy trong Chương 3, Awk thực hiện tự động việc đọc các tập tin dữ liệu nhập và cắt mỗi dòng ra thành các trường được đặt tên là `$1` tới `$NF`, trong đó `NF` là số trường trên một dòng. Bằng cách hỗ trợ hành vi mặc định (default behavior) cho nhiều tác vụ thông thường, chúng ta có thể tạo ra được các chương trình chỉ có một dòng. Ví dụ, sau đây là một chương trình Awk hoàn chỉnh:

```
        # split.awk: chia dữ liệu nhập thành mỗi từ trên
một dòng

        { for (i = 1; i <= NF; i++) print $i }
```

chương trình trên in ra các từ của mỗi dòng dữ liệu nhập, một từ trên một dòng. Để chuyển sang vấn đề khác, chúng ta hãy xem xét một cài đặt của hàm `fmt` sau đây, lấp mỗi dòng dữ liệu xuất bằng các từ, tối đa 60 ký tự; dòng trắng là nơi ngắt đoạn:

```
        # fmt.awk: format into 60-character lines

        /./     { for (i = 1; i <=NF; i++) addword($i) }
#nonblank line

        /^$/   { printline(); print "" }                    #
blank line
```

```
END    { printline() }

function addword(w) {
        if (length(line) + 1 + length(w) > 60)
                printline()
        if (length(line) == 0)
                line = w
        else
                line = line " " w
}
function printline() {
        if (length(line) > 0) {
                print line
                line = ""
        }
}
```

Chúng ta thường dùng hàm `fmt` để chia đoạn lại các dòng tin trong email và các tài liệu ngắn khác, chúng ta cũng dùng nó để định dạng dữ liệu xuất của các chương trình *Markov* trong Chương 3.

Các công cụ lập trình thường có nguồn gốc từ các ngôn ngữ nhỏ được thiết kế cho việc diễn đạt tự nhiên các giải pháp cho các vấn đề tồn tại trong một phạm vi hẹp. Một ví dụ dễ thấy là công cụ `eqn` của Unix, nó thiết lập công thức toán học. Ngôn ngữ dữ liệu nhập của nó gần với cách đọc các phép toán trong các công thức, ví dụ `π/2` được viết là "`pi over 2`". TEX cũng theo cách tiếp cận này; ký hiệu của nó cho công thức này là `\pi\over 2`. Nếu có các ký hiệu tự nhiên hay quen thuộc cho vấn đề bạn đang giải

quyết thì hãy dùng nó và hiệu chỉnh nó cho thích hợp chứ đừng thực hiện từ đầu.

Awk bị thuyết phục bởi một chương trình sử dụng các biểu thức có quy tắc để chỉ ra các dữ liệu khác thường trong các hồ sơ theo dõi điện thoại, tuy nhiên Awk thêm vào các biến, diễn đạt, vòng lặp,... để làm cho nó trở thành một ngôn ngữ lập trình thực thụ. Perl và Tcl ngay từ đầu đã được xây dựng cho việc kết hợp sự tiện lợi và ấn tượng của các ngôn ngữ nhỏ với sức mạnh của các ngôn ngữ lớn. Chúng thật sự là các ngôn ngữ cho mục đích tổng quát, mặc dù chúng rất hay được dùng cho việc xử lý văn bản.

Thuật ngữ dùng chung cho các công cụ như thế là *ngôn ngữ kịch bản* (scripting language), bởi vì chúng được phát triển từ các trình thông dịch lệnh đầu tiên có khả năng hỗ trợ lập trình được giới hạn trong khuôn khổ chạy các "kịch bản" được gắn vào các chương trình. Các ngôn ngữ kịch bản cho phép cách dùng sáng tạo các biểu thức có quy tắc, không những cho việc xác định các mẫu hợp lệ - nhận biết sự xuất hiện của một mẫu nào đó - mà còn cho việc định vị các vùng văn bản sẽ được chuyển đổi dạng thức. Điều này xuất hiện trong hai lệnh `regsub` (REGular expression SUBstitution) trong chương trình Tcl dưới đây. Chương trình này là một sự tổng quát hoá không đáng kể của chương trình mà chúng ta đã đưa ra trong Chương 4 được dùng để lấy các thành phần trong các dấu nháy; chương trình này được dùng để lấy URL được lưu trong đối số đầu tiên của nó. Sự thay thế đầu tiên sẽ loại bỏ chuỗi `http://` nếu có; sự thay thế thứ hai sẽ thay thế dấu `/` đầu tiên bằng một khoảng trắng, kết quả là chia đối số ra thành hai trường. Lệnh `lindex` lấy các trường từ một chuỗi (bắt đầu với chỉ số 0). Văn bản bên trong `[]` được thi hành như một lệnh Tcl và được thay thế bởi văn bản kết quả; `$x` được thay thế bởi giá trị của biến `x`.

```
# geturl.tcl: retrieve document from URL
# input has form [http://]abc.def.com[/whatever...]
regsub "http://" $argv "" argv      ;#      remove
```

```
http:// if present

      regsub "/" $argv " " argv            ;#     replace
leading / with blank


      set  so  [socket  [lindex  $argv  0]  80]  ;#  make
network connection

      set q "/[lindex $argv 1]"


      puts $so "GET $q HTTP/1.0\n\n"       ;#send
request

      flush $so

      while {[gets $so line] >= 0 && $line != ""} {}
;#skip header

      puts [read $so]                      ;#read     and
print entire reply
```

Script này tạo kết quả đầu ra rất dài, phần lớn là các tag của HTML được đóng trong hai dấu < và >. Perl là một ngôn ngữ rất mạnh trong việc thay thế chuỗi, do đó công cụ tiếp theo của chúng ta là một script của Perl sử dụng các biểu thức cần tìm và các sự thay thế để loại bỏ các tag này:

```
      # unhtml.pl: delete HTML tags


      while (<>) {              # collect all input into
single string
            $str .= $_;         # by concatenating input
lines

      }
```

```
$str =~ s/<[^>]*>//g;    # delete <...>
$str =~ s/ / /g;    # replace   by blank
$str =~ s/\s+/\n/g;      # compress white space
print $str;
```

Ví dụ này sẽ khó hiểu đối với những người không đọc được chương trình Perl. Cấu trúc

```
$str = ~s/regexp/repl/g
```

có chức năng thay văn bản hợp với (có đầu bên trái dài nhất) các biểu thức có quy tắc `regexp` trong chuỗi `str` bằng chuỗi `repl`; chữ `g` ở cuối là viết tắt của "global", nghĩa là thực hiện thay thế đối với tất cả các văn bản phù hợp trong chuỗi chứ không phải chỉ cho văn bản đầu tiên. Ký tự đặc biệt `\s` là dạng viết tắt cho một loại ký tự trắng (khoảng trắng, tab, sang dòng,...); `\n` là ký tự sang dòng. Chuỗi "` `" là một ký tự HTML, tương tự như các ký tự trong Chương 2, nó định nghĩa một ký tự trắng không thể tách ra (non-breakable space character).

Sắp xếp tất cả chúng lại với nhau, ta được một trình duyệt web theo hướng chức năng được cài đặt như một script shell một dòng, mặc dù không mạnh:

```
# web: retrieve web page and format its text,
ignoring HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

Lệnh này lấy về trang web, loại bỏ tất cả thông tin định dạng và điều khiển, và định dạng lại văn bản theo quy tắc của nó. Đây là một cách nhanh chóng để lấy một trang văn bản từ web.

Hãy chú ý sự khác nhau của các ngôn ngữ đã được thảo luận, mỗi ngôn ngữ thích hợp cho một tác vụ nào đó: Tcl, Perl, Awk, và các ngôn ngữ biểu thức có quy tắc. Sức mạnh của chúng được thể hiện khác nhau trong

từng ngôn ngữ. Tcl đặc biệt tốt cho việc lấy về văn bản thông qua mạng; Perl và Awk tốt cho hiệu chỉnh và định dạng văn bản; và dĩ nhiên các biểu thức có quy tắc tốt cho xác định văn bản cho việc tìm kiếm và sửa chữa. Kết hợp các ngôn ngữ này với nhau sẽ mạnh hơn bất kỳ một ngôn ngữ riêng lẻ nào trong chúng. Một việc rất đáng được thực hiện là chia nhỏ công việc ra nếu điều này cho phép chúng ta tận dụng được thế mạnh từ hệ thống các ký hiệu của các ngôn ngữ này.

9.4. Các trình thông dịch, trình biên dịch và máy ảo

Một chương trình sẽ thực hiện như thế nào để chuyển từ dạng mã nguồn của nó sang dạng thực thi? Trong một ngôn ngữ đủ đơn giản, như trong `printf` hay các biểu thức có quy tắc đơn giản nhất của chúng ta chẳng hạn, chúng ta có thể chạy trực tiếp từ nguồn. Điều này dễ làm và khởi động rất nhanh.

Có một sự cân đối giữa thời gian cài đặt và tốc độ thi hành. Trong một ngôn ngữ phức tạp, người ta thích chuyển mã nguồn sang dạng thể hiện bên trong có hiệu quả và tiện lợi cho việc thi hành. Nó tốn thời gian để xử lý nguồn ban đầu nhưng bù lại là thi hành nhanh hơn. Các chương trình kết hợp việc đối thoại và thi hành vào trong một chương trình duy nhất có khả năng đọc mã nguồn, chuyển đổi nó, và chạy nó được gọi là các trình thông dịch. Awk và Perl thông dịch giống như nhiều *ngôn ngữ mục đích đặc biệt* và kịch bản khác.

Khả năng thứ ba là phát sinh các chỉ dẫn cho một loại máy tính đặc biệt mà chương trình sẽ chạy trên đó, giống như các trình biên dịch thực hiện. Điều này đòi hỏi có nhiều cố gắng và thời gian nhất trong giai đoạn đầu nhưng nó cho ra kết quả thi hành nhanh nhất.

Ngoài ra còn có các kết hợp khác. Một kết hợp mà chúng ta sẽ học trong phần này là biên dịch một chương trình thành các chỉ dẫn cho một máy tính giả lập (một máy ảo) có thể được giả lập trên bất kỳ một máy tính thực tế nào. Một máy ảo có khả năng kết hợp nhiều thế mạnh của các thông dịch và biên dịch truyền thống.

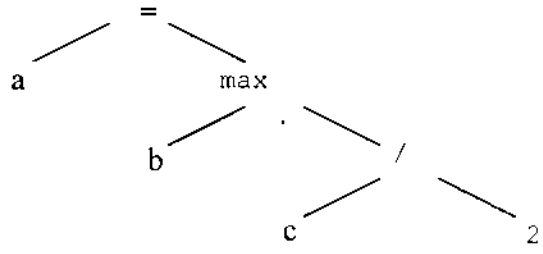
Trong một ngôn ngữ đơn giản thì không cần phải xử lý nhiều để hiểu được cấu trúc chương trình và chuyển đổi nó sang dạng thức bên trong. Tuy nhiên, trong một ngôn ngữ có một số phức tạp - các khai báo, các cấu trúc lồng nhau, các phát biểu hay các diễn đạt được định nghĩa đệ quy, các toán tử với thứ tự ưu tiên,... - thì sẽ phức tạp hơn trong việc phân tích dữ liệu nhập và xác định cấu trúc.

Các trình phân tích (parser) thường được viết với mục đích tạo ra một trình phát sinh phân tích tự động (automatic parser generator), còn được gọi là trình biên dịch của trình biên dịch (compiler – compiler), ví dụ như `yacc` hay `bison`. Các chương trình này dịch một mô tả của ngôn ngữ – được gọi là ngữ pháp của nó, sang (thường là) một chương trình C hay C++, và khi chương trình C/C++ này được biên dịch, nó sẽ dịch các phát biểu trong ngôn ngữ này sang một dạng thể hiện bên trong. Dĩ nhiên, việc phát sinh một trình phân tích trực tiếp từ một cấu trúc ngữ pháp là một minh họa khác về sức mạnh của hệ thống các ký hiệu tốt.

Dạng được tạo ra bởi một trình phân tích thường là một cây, với các nút trong chứa các toán tử (hay các chỉ dẫn điều khiển) và các nút lá chứa các toán hạng (hay các đối số của điều khiển). Phát biểu sau đây:

```
a = max(b, c/2);
```

có thể tạo ra cây phân tích (parse tree) hay còn gọi là cây cú pháp như sau:



Nhiều thuật toán cây được mô tả trong Chương 2 có thể được áp dụng để xây dựng và xử lý các cây phân tích.

Một khi một cây được xây dựng, có nhiều cách khác nhau để xử lý nó. Cách trực tiếp nhất, được dùng trong Awk, là duyệt cây một cách trực tiếp, tính số nút khi đi qua. Một phiên bản được đơn giản hoá của một tác vụ tính toán như thế trong một ngôn ngữ diễn đạt dựa trên số nguyên liên quan đến cách duyệt theo hậu thứ tự như sau:

```
typedef struct Symbol Symbol;
typedef struct Tree Tree;

struct Symbol {
        int    value;
        char   *name;
};

struct Tree {
        int         op;         /* mã số thao tác
tính toán */
        int         value;      /* chứa giá trị nếu
là số */
        Symbol      *symbol;    /* kiểu Symbol nếu
là biến */
        Tree        *left;
        Tree        *right;
};
```

```
/* eval: phiên bản 1: định giá biểu thức cây */
int eval(Tree *t)
{
        int left, right;

        switch (t->op) {
        case NUMBER:
                return t->value;
        case VARIABLE:
                return t->symbol->value;
        case ADD:
                return   eval(t->left)   +   eval(t-
>right);
        case DIVIDE:
                left = eval(t->left);
                right = eval(t->right);
                if (right == 0)
                        eprintf("divide  %d  by  zero",
left);
                return left / right;
        case MAX:
                left = eval(t->left);
                right = eval(t->right);
```

```
                return left>right ? left : right;
            case ASSIGN:
                t->left->symbol->value = eval(t-
>right);
                return t->left->symbol->value;
            /* ... */
            }
        }
```

Các trường hợp đầu tiên đánh giá các diễn đạt được đơn giản như các hằng số và giá trị; các trường hợp tiếp theo đánh giá các diễn đạt số học, và các trường hợp khác có thể thực hiện các xử lý đặc biệt, điều kiện, và vòng lặp. Để có thể cài đặt được các cấu trúc điều khiển, cây cần phải có thêm thông tin mô tả luồng điều khiển; vấn đề này không được trình bày ở đây.

Như trong hàm `pack` và `unpack`, chúng ta có thể thay thế một `switch` được mô tả rõ ràng bằng một bảng các con trỏ hàm. Các toán tử (hay chỉ dẫn điều khiển) riêng thì rất giống như trong phát biểu `switch` này:

```
/* addop: trả về tổng của hai biểu thức cây */
int addop(Tree *t)
{
    return eval(t->left) + eval(t->right);
}
```

Bảng các con trỏ hàm liên kết các chỉ dẫn điều khiển đến các hàm trình diễn các điều khiển:

```
enum { /* các mã số thao tác tính toán */
        NUMBER,
        VARIABLE,
        ADD,
        DIVIDE,
        /* ... */
};
/* optab: bảng chứa các chức năng thao tác tính
toán */
int (*optab[])(Tree *) = {
        pushop,          /* NUMBER */
        pushsymop,       /* VARIABLE */
        addop,           /* ADD */
        divop,           /* DIVIDE */
        /* ... */
};
```

Sau đây là đánh giá bằng cách sử dụng chỉ dẫn điều khiển để định vị bên trong bảng các con trỏ hàm để gọi hàm tương ứng: phiên bản này sẽ kích hoạt các hàm khác một cách đệ quy.

```
/* eval: phiên bản 2: định giá cây từ bảng thao
tác tính toán*/
int eval(Tree *t)
{
        return (*optab[t->op])(t);
}
```

Cả hai phiên bản của hàm `eval` đều đệ quy. Chúng ta cũng có các cách khử đệ quy, bao gồm một kỹ thuật thông minh được gọi là *đoạn mã nguồn theo thread*; nó hoàn toàn không sử dụng ngăn xếp. Một phương pháp đơn giản nhưng hiệu quả nhất là khử hoàn toàn đệ quy bằng cách lưu các hàm vào một mảng được dùng cho việc duyệt tuần tự để chạy chương trình. Mảng này trở thành một thứ tự liên tiếp các chỉ dẫn được một máy nhỏ phục vụ cho mục đích đặc biệt thi hành.

Chúng ta vẫn cần một ngăn xếp để lưu một phần các giá trị được đánh giá trong các tính toán, do đó hình thức của các hàm sẽ thay đổi, nhưng sự thay đổi này thì dễ thấy. Kết quả là, chúng ta tạo ra một *máy ngăn xếp* (stack machine) trong đó các chỉ dẫn là các hàm nhỏ và các toán hạng được lưu trong một ngăn xếp riêng biệt chứa các toán hạng. Đây không phải là một máy thật nhưng chúng ta có thể lập trình nó như thật, và chúng ta có thể cài đặt nó một cách dễ dàng như một trình thông dịch.

Chúng ta duyệt cây để phát sinh mảng các hàm để chạy chương trình thay vì duyệt để đánh giá nó. Mảng cũng chứa các giá trị dữ liệu mà các hướng dẫn dùng, như là các hằng số và các biến số (các biểu tượng), do đó các phần tử của mảng nên có kiểu là union:

```
typedef union Code Code;
union Code {
        void    (*op)(void);    /* hàm nếu là thao
tác tính toán */
        int     value;          /* giá trị nếu là
số */
        Symbol  *symbol;        /* kiểu Symbol nếu
là biến */
};
```

Sau đây là tác vụ phát sinh các con trỏ hàm và đưa chúng vào một mảng chứa các phần tử này, gọi là mảng code. Giá trị trả về của hàm generate thì không phải là giá trị của các diễn đạt sẽ được tính khi code phát sinh được thi hành, mà là chỉ số trong mảng code của điều khiển kế tiếp sẽ được phát sinh.

```
/* generate: phát sinh các chỉ thị bằng cách duyệt cây
*/
      int generate(int codep, Tree *t)
      {

            switch (t->op) {
            case NUMBER:
                  code[codep++].op = pushop;
                  code[codep++].value = t->value;
                  return codep;
            case VARIABLE:
                  code[codep++].op = pushsymop;
                  code[codep++].symbol = t->symbol;
                  return codep;
            case ADD:
                  codep = generate(codep, t->left);
                  codep = generate(codep, t->right);
                  code[codep++].op = addop;
                  return codep;
```

```
        case DIVIDE:
                codep = generate(codep, t->left);
                codep = generate(codep, t->right);
                code[codep++].op = divop;
                return codep;
        case MAX:
                /* ... */
        }
}
```

Với phát biểu `a = max(b, c/2)`, mảng `code` được tổng hợp sẽ là:

```
pushsymop
b
pushsymop
c
pushop
2
divop
maxop
storesymop
a
```

Các hàm toán tử sẽ thao tác trên ngăn xếp, lấy các toán hạng ra khỏi ngăn xếp (pop) và đưa các kết quả vào ngăn xếp (push).

Trình thông dịch là một vòng lặp duyệt mảng các con trỏ hàm thông qua một biến đếm chương trình:

```
Code code[NCODE];

int stack[NSTACK];

int stackp;

int pc; /* biến đếm chương trình */


/* eval: phiên bản 3: định giá biểu thức từ mã
nguồn đã phát sinh */

int eval(Tree *t)

{

      pc = generate(0, t);

      code[pc].op = NULL;


      stackp = 0;

      pc = 0;

      while (code[pc].op != NULL)

            (*code[pc++].op)();

      return stack[0];

}
```

Vòng lặp này chạy mô phỏng trong phần mềm trên máy ngăn xếp giả lập của chúng ta, giống như là những gì xảy ra trong phần cứng trên một máy thật sự. Sau đây là một số hàm toán tử tiêu biểu:

```
/* pushop: chèn số: giá trị là từ kế tiếp trong luồng
mã số */

void pushop(void)

{
```

```
        stack[stackp++] = code[pc++].value;
}

/* divop: tính toán tỷ số của hai biểu thức */
void divop(void)
{
        int left, right;

        right = stack[--stackp];
        left = stack[--stackp];
        if (right == 0)
            eprintf("divide %d by zero\n", left);
        stack[stackp++] = left / right;
}
```

Chú ý rằng kiểm tra chia cho `0` xuất hiện trong hàm `divop`, chứ không phải trong hàm `generate`.

Các lệnh có điều kiện, các rẽ nhánh, và các vòng lặp vận hành bằng cách sửa biến đếm chương trình trong một hàm toán tử, đưa một nhánh tới một điểm khác trong mảng các hàm. Ví dụ, một điều khiển `goto` luôn gán giá trị của biến `pc`, trong khi một nhánh điều kiện chỉ gán `pc` khi điều kiện là đúng.

Mảng `code` dĩ nhiên là phần bên trong của trình thông dịch, nhưng hãy tưởng tượng xem trong trường hợp chúng ta muốn lưu chương trình được phát sinh vào một tập tin. Nếu chúng ta nêu ra các địa chỉ hàm thì kết quả sẽ không linh hoạt và khó sử dụng. Tuy nhiên, thay vào đó chúng ta sẽ

nêu ra các hằng số đại diện cho các hàm, ví dụ `1000` cho `addop`, `1001` cho `pushop`,..., và dịch ngược chúng sang các con trỏ hàm khi chúng ta đọc chương trình vào cho việc thông dịch.

Nếu chúng ta quan sát một tập tin do thủ tục này tạo ra, nó sẽ trông giống như một luồng chỉ dẫn cho một máy ảo mà ở đó các chỉ dẫn của nó tương ứng với các điều khiển cơ bản của ngôn ngữ nhỏ của chúng ta, và hàm `generate` thì thật sự là một trình biên dịch; nó dịch ngôn ngữ này sang dạng máy ảo. Các máy ảo là một ý tưởng cũ tuyệt vời, gần đây được phổ biến trở lại bởi Java và máy ảo Java (Java Virtual Machine – JVM); chúng hỗ trợ một cách dễ dàng để tạo ra các thể hiện hiệu quả, linh hoạt của các chương trình được viết trong một ngôn ngữ bậc cao.

9.5. Các chương trình để viết các chương trình khác

Điểm nổi bật nhất của hàm `generate` có lẽ là nó là một chương trình để viết một chương trình: kết quả của nó là một loạt chỉ dẫn có thể chạy được trên một máy (ảo) khác. Các trình thông dịch luôn làm việc này, dịch mã nguồn sang các chỉ dẫn máy, do đó ý tưởng này chắc chắn không có gì mới. Trong thực tế, các chương trình để viết các chương trình khác xuất hiện dưới nhiều dạng.

Một ví dụ thường gặp là việc phát sinh động HTML cho các trang web. HTML là một ngôn ngữ hạn chế, và nó cũng có thể chứa mã JavaScript. Các trang web thường được phát sinh trực tiếp trong khi chạy bởi các chương trình Perl hay C, với các nội dung đặc thù (ví dụ các kết quả tìm kiếm và các dịch vụ quảng cáo) được xác định bởi các yêu cầu đang được thực hiện. Chúng ta đã sử dụng các ngôn ngữ chuyên dụng cho các đồ thị, hình ảnh, bảng biểu, các biểu thức toán học, các chỉ mục trong quyển sách này. Lấy một ví dụ khác, PostScript là một ngôn ngữ lập trình được phát sinh từ các trình xử lý văn bản, các chương trình vẽ, và nhiều nguồn khác; ở giai đoạn cuối của quá trình xử lý, toàn bộ quyển sách được thể hiện như một chương trình PostScript với 57000 dòng.

Một tài liệu là một chương trình tĩnh, nhưng ý tưởng sử dụng một

ngôn ngữ lập trình như hệ thống ký hiệu để giải quyết bất kỳ một bài toán nào thì cực kỳ hay. Nhiều năm qua, các lập trình viên ao ước có được các máy tính viết tất cả các chương trình cho họ. Có lẽ điều này vẫn chỉ là một ước mơ, tuy nhiên các máy tính ngày nay vẫn đều đặn viết các chương trình cho chúng ta, thường để thể hiện những việc mà trước kia chúng ta đã không quan tâm đến trong các chương trình.

Chương trình viết ra chương trình thông dụng nhất là một trình biên dịch; nó dịch các ngôn ngữ cấp cao sang mã máy. Thật sự hữu ích nếu có thể dịch mã nguồn sang một ngôn ngữ lập trình. Trong phần trước, chúng ta đã đề cập đến việc các bộ phát sinh đã chuyển một định nghĩa cấu trúc ngữ pháp của một ngôn ngữ sang một chương trình C, trong đó ngôn ngữ C có khả năng phân tích ngôn ngữ. C thường được dùng theo cách này và được sánh như một loại "ngôn ngữ hợp ngữ (assembly) bậc cao". Modula-3 và C++ là hai trong số các ngôn ngữ được dùng cho mục đích tổng quát với các trình biên dịch đầu tiên tạo ra mã nguồn C, sau đó mã nguồn C này được một trình biên dịch C chuẩn biên dịch. Cách tiếp cận này có nhiều thuận lợi, bao gồm: hiệu quả – bởi vì theo nguyên tắc thì các chương trình có thể chạy nhanh như các chương trình C – và khả chuyển – bởi vì các trình biên dịch có thể được mang đến bất kỳ một hệ thống nào có một trình biên dịch C. Điều này mang lại nhiều lợi ích trong giai đoạn đầu phổ biến ngôn ngữ này.

Lấy một ví dụ khác, giao diện đồ họa của Visual Basic phát sinh các lệnh gán để khởi tạo các đối tượng; người dùng chọn các lệnh gán này từ các danh sách và dùng chuột đặt lên màn hình. Nhiều ngôn ngữ khác cũng có hệ thống phát triển phần mềm "trực quan" và sử dụng "wizard" để tạo ra mã nguồn giao diện chỉ thông qua các thao tác nhắp chuột.

Mặc dù đã có nhiều bộ phát sinh chương trình tiện lợi và có sẵn nhiều ví dụ hay nhưng các ký hiệu vẫn chưa được đánh giá cao và thường được các lập trình viên đơn lẻ sử dụng. Tuy nhiên, chúng ta cũng có thể hưởng một số ích lợi từ việc phát sinh mã nguồn bằng một chương trình. Sau đây là một số ví dụ phát sinh mã C hay C++.

Hệ điều hành Plan 9 phát sinh các thông điệp lỗi từ một tập tin tiêu

để chứa các tên lỗi và các chú thích; các chú thích được chuyển đổi một cách máy móc sang các chuỗi trong dấu nháy và được đưa vào một mảng được đánh chỉ số bằng cách liệt kê. Đoạn chương trình sau trình bày cấu trúc của tập tin tiêu đề:

```
/* errors.h: thông điệp lỗi chuẩn */


enum {
        Eperm,          /* Permission denied */
        Eio,            /* I/o error */
        Efile,          /* File does not exist */
        Emem,           /* Memory limit reached */
        Espace,         /* Out of file space */
        Egreg           /* It's all Greg's fault */
};
```

Với dữ liệu cho trước như vậy, một chương trình đơn giản có thể tạo ra các khai báo cho thông điệp lỗi như sau:

```
/* machine-generated; không được sửa */
char *err[] = {
        "Permission denied",     /* Eperm */
        "I/O error",             /* Eio */
        "File does not exist",   /* Efile */
        "Memory limit reached",  /* Emem */
        "Out of file space",     /* Espace */
        "It's all Greg's fault",/* Egreg */
```

Có một số lợi ích từ cách tiếp cận này. Trước hết, mối liên hệ giữa các giá trị `enum` và các chuỗi thông báo được đưa ra rõ ràng và dễ dàng độc lập với ngôn ngữ tự nhiên. Ngoài ra, do thông tin chỉ xuất hiện một lần, nghĩa là chỉ có "một chân lý duy nhất" mà từ đó các mã nguồn khác được phát sinh, do đó chỉ cần cập nhật thông tin ở một nơi duy nhất. Giả sử thông tin được cập nhật ở nhiều nơi thì chắc chắn đến một lúc nào đó thông tin sẽ không thống nhất. Cuối cùng, bất kỳ khi nào tập tin tiêu đề thay đổi, tập tin `.c` sẽ được tạo lại và biên dịch lại. Khi một thông điệp lỗi được thay đổi, thì chỉ cần sửa trong tập tin tiêu đề. Các thông điệp được cập nhật một cách tự động.

Trình phát sinh có thể được viết trong bất kỳ ngôn ngữ nào. Một ngôn ngữ xử lý chuỗi như Perl có thể làm điều này dễ dàng.

```
# enum.pl: phát sinh chuỗi lỗi từ các chú thích
kiểu enum

print "/* machine-generated; không được sửa.
*/\n\n";

print "char *err[] = {\n";


while (<>) {
    chop;                          #xóa dòng mới
    if (/^\s*(E[a-z0-9]+),?/) {    #từ đầu tiên
là E …
        $name = $1;                #lưu tên
        s/.*\/\* *//;              # xóa
        s/ *\*\///;
        print "\t\"$_\", /* $name */\n";
    }
```

```
}
print "};\n";
```

Các biểu thức có quy tắc lại một lần nữa được dùng đến. Các dòng có các trường đầu tiên trông giống như các dấu nhận diện được theo sau bởi một dấu phẩy là các trường được chọn ra. Thay thế thứ nhất xóa tất cả cho đến ký tự không phải là khoảng trắng đầu tiên của chú thích, trong khi thay thế thứ hai loại bỏ dấu kết thúc chú thích và bất kỳ khoảng trắng nào trước nó.

Để kiểm chứng trình biên dịch, Andy Koenig đưa ra một cách thức tiện lợi để viết mã nguồn C++ nhằm kiểm tra xem trình biên dịch có bắt được các lỗi chương trình hay không. Các đoạn mã nguồn làm cho trình biên dịch thực hiện kiểm tra được gắn vào các chú thích khác thường để mô tả các thông điệp được mong đợi. Mỗi dòng có một chú thích bắt đầu bằng /// (để phân biệt nó với các chú thích bình thường) và một biểu thức có quy tắc phù hợp với các kiểm tra trên dòng đó. Do đó, hai đoạn mã nguồn sau đây chẳng hạn sẽ phát sinh các kiểm tra:

```
int f() {}
    /// warning.* non-void function .* should
return a value

void g() {return 1;}
    /// error.* void function may not return a
value
```

Nếu chúng ta chạy hàm kiểm tra thứ hai thông qua trình biên dịch C++, thì nó sẽ in ra thông điệp như mong muốn và phù hợp với các biểu thức có quy tắc.

```
% CC x.c
"x.c", line 1: error(321): void function may not
return a value
```

Mỗi đoạn như thế được đưa đến trình biên dịch, và dữ liệu xuất được so sánh với các kết quả kiểm tra biết trước, đây là một tiến trình được quản lý bằng sự kết hợp giữa chương trình Shell và Awk. Các lỗi này cho biết kết quả do trình biên dịch thực hiện được khác với những kết quả mong muốn. Vì các chú thích là các biểu thức có quy tắc nên có một số khác biệt trong dữ liệu xuất; chúng có thể ít nhiều được thay đổi tùy ý phụ thuộc vào yêu cầu.

Ý tưởng về các chú thích có ngữ nghĩa không phải là mới. Chúng xuất hiện trong PostScript, ở đó các chú thích bình thường bắt đầu bằng `%`. Các chú thích theo truyền thống bắt đầu bằng `%%` có thể chứa thêm thông tin phụ về các số trang, các hộp có đường biên, các tên phông chữ và những thông tin tương tự khác.

```
%%PageBoundingBox: 126 307 492 768
%%Pages: 14
%%DocumentFonts: Helvetica Times-Italic Times-
Roman
                LucidaSans-Typewriter
```

Trong tất cả các ví dụ ở trên, điều quan trọng là phải nhận ra được vai trò của các ký hiệu, sự kết hợp của các ngôn ngữ, và việc sử dụng các công cụ.

Bài tập 9-15. Một trong những trường hợp thường gặp là viết một chương trình mà khi nó được thi hành sẽ tạo ra chính nó dưới dạng nguồn. Đây là một trường hợp đặc biệt có tổ chức của một chương trình để viết một chương trình khác. Hãy thử cài đặt chương trình với ngôn ngữ mà bạn thích.

9.6. Sử dụng các macro để phát sinh mã nguồn

Chuyển xuống một vài cấp thấp hơn, chúng ta có thể sử dụng các macro để viết mã lúc biên dịch. Xuyên suốt quyển sách này, việc sử dụng các macro và biên dịch có điều kiện đã được cảnh báo bởi vì chúng đưa ra một phong cách lập trình có nhiều điểm rắc rối. Nhưng chúng cũng có vị trí của nó; sự thay thế loại văn bản đôi khi chính là câu trả lời chính xác cho vấn đề đó. Một ví dụ là sử dụng bộ tiền xử lý macro C/C++ để tập hợp các phần được lặp lại nhiều lần của chương trình.

Chẳng hạn, chương trình tính tốc độ của các cấu trúc ngôn ngữ sơ cấp trong Chương 7 sử dụng bộ tiền xử lý C để tập hợp các kiểm chứng bằng cách gom chúng vào trong mã nguồn trước khi biên dịch. Trọng tâm của việc kiểm tra là đóng gói mã nguồn vào một vòng lặp để khởi tạo bộ đếm thời gian, chạy đoạn mã nguồn này nhiều lần, dừng bộ đếm thời gian, và thông báo kết quả. Tất cả các đoạn mã nguồn lặp lại được mô tả trong một số macro và đoạn mã nguồn được đặt thời gian được truyền vào như một đối số. Macro chính có dạng như sau:

```
#define LOOP(CODE) {                          \
        t0 = clock();                         \
        for (i = 0; i < n; i++) { CODE; }     \
        printf("%7d ", clock() - t0);         \
}
```

Các dấu \ cho phép thân macro ngắt ra thành nhiều dòng. Macro này được dùng trong các "phát biểu" có dạng đặc trưng như sau:

```
LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
LOOP(f1 = f2 - f3)
```

Đôi khi có các lệnh khác được dùng cho việc khởi tạo, tuy nhiên phần định thời gian cơ sở thì được thể hiện trong các phân đoạn chỉ có một

đối số này; các phân đoạn này mở rộng mã nguồn một lượng đáng kể.

Tiến trình xử lý macro cũng có thể được dùng để phát sinh mã nguồn. Bart Locanthi có lần viết một phiên bản của một trình điều khiển đồ họa hai chiều, đó là `bitblt` hay `rasterop`. Khó có thể tạo ra trình điều khiển này một cách nhanh chóng, bởi vì có nhiều đối số kết hợp với nhau theo các cách thức phức tạp. Thông qua việc xử lý cẩn thận các trường hợp có thể xảy ra, Locanthi làm giảm các sự kết hợp đối với các vòng lặp riêng lẻ, các vòng lặp này có thể được tối ưu hoá riêng lẻ. Tiếp đến, mỗi trường hợp có thể được cấu trúc bằng việc thay thế macro, tương tự như ví dụ kiểm tra sự vận hành, với tất cả các trường hợp khác nhau được giải quyết trong một lệnh switch lớn. Mã nguồn nguyên thủy chỉ có vài trăm dòng nhưng kết quả do quá trình xử lý macro tạo ra thì có tới vài ngàn dòng. Mã nguồn được mở rộng bằng macro là không tối ưu, nhưng khi xem xét khó khăn của vấn đề thì đó là cách thực tế và rất dễ để tạo ra nó. Hơn nữa, đối với mã nguồn có thể thực thi tốt thì mã nguồn này cũng mang tính khả chuyển tương ứng.

Bài tập 9-16. Bài tập 7-7 liên quan đến việc ước lượng chi phí cho các điều khiển khác nhau trong C++. Sử dụng các ý tưởng của phần này để tạo ra một phiên bản khác của chương trình.

Bài tập 9-17. Bài tập 7-8 liên quan đến việc tạo lập một mô hình chi phí cho Java, không có khả năng macro. Hãy giải quyết vấn đề bằng cách viết một chương trình khác, có thể dùng bất kỳ một ngôn ngữ nào, để viết phiên bản Java này và kích hoạt tự động các xử lý định thời gian.

9.7. Biên dịch trong khi thực thi

Trong phần trước, chúng ta đã thảo luận về các chương trình có khả năng viết các chương trình. Với mỗi ví dụ, chương trình được phát sinh là ở dạng nguồn; nó cần được biên dịch hay thông dịch để chạy. Nhưng chúng ta cũng có thể phát sinh mã nguồn có thể chạy ngay lập tức bằng cách tạo ra các chỉ dẫn máy thay vì nguồn. Điều này được biết đến như là việc biên dịch "sẵn".

Mặc dù mã nguồn đã được biên dịch cần thiết phải là không khả

chuyển - chỉ chạy trên một bộ xử lý duy nhất - nó có thể chạy rất nhanh.

Hãy xem xét câu lệnh:

```
max(b, c/2)
```

Phép tính này phải xác định giá trị của `c`, chia cho 2, so sánh kết quả với `b` và chọn số lớn hơn. Nếu chúng ta sử dụng máy ảo, được nói đến ở đầu chương, để tính giá trị câu lệnh trên thì chúng ta có thể bỏ qua việc kiểm tra chia 0 trong hàm `divop`. Vì 2 khác 0 nên kiểm tra là dư thừa. Nhưng với các vấn đề nêu ra trong các thiết kế áp dụng cho việc cài đặt máy ảo, thì chúng ta không thể bỏ qua quá trình kiểm tra; mỗi lần thực hiện lệnh chia đều phải so sánh số chia với 0.

Đây chính là điểm mà mã nguồn phát sinh có thể giải quyết một cách linh động. Nếu chúng ta xây dựng mã nguồn cho câu lệnh trên một cách trực tiếp, thay vì dùng lại các hàm được định nghĩa trước đó, chúng ta có thể trách được việc kiểm tra chia 0 đối với các số chia được biết là khác 0. Thật ra, chúng ta thậm chí có thể đi xa hơn nếu toàn bộ biểu thức là hằng số. Chẳng hạn `max(3*3, 4/2)`, chúng ta có thể tính toán ngay khi phát sinh mã nguồn, và thay thế bằng một hằng có giá trị 9. Nếu biểu thức xuất hiện trong một vòng lặp, thì chúng ta có thể giảm được thời gian qua mỗi lần lặp, và nếu vòng lặp thực hiện đủ số lần lặp thì chúng ta sẽ lấy lại được chi phí phải trả cho việc phân tích biểu thức và phát sinh mã nguồn cho nó.

Ý tưởng chính là hệ thống các ký hiệu cung cấp cho chúng ta một cách tổng quát để diễn đạt vấn đề. Tuy nhiên, trình biên dịch cho hệ thống các ký hiệu có thể hiệu chỉnh mã nguồn cho các chi tiết của một tính toán đặc thù. Ví dụ, trong một máy ảo dành cho các biểu thức có quy tắc thì chúng ta có lẽ dùng riêng một hàm để kiểm tra một ký tự:

```
int matchchar(int literal, char *text)
{
    return *text == literal;
}
```

Chúng ta có thể phát sinh mã nguồn cho một mẫu cụ thể. Tuy nhiên, giá trị của đối số `literal` cho trước phải phù hợp, giả sử là '`x`', do đó thay vào đó chúng ta có thể dùng một toán tử như sau:

```
int matchx(char *text)
{
        return *text == 'x';
}
```

Và do đó, thay vì định nghĩa trước một hàm đặc biệt cho mỗi giá trị ký tự theo dạng thức được đưa ra, chúng ta có thể đơn giản hoá bằng cách phát sinh mã nguồn cho các hàm thật sự cần thiết cho biểu thức hiện hành. Mở rộng ý tưởng cho tập đầy đủ các điều khiển, chúng ta có thể viết một trình biên dịch "sẵn" để dịch một biểu thức có quy tắc hiện hành sang mã nguồn đặc biệt được tối ưu hoá cho biểu thức đó.

Ken Thompson thực hiện chính xác công việc này cho việc cài đặt các biểu thức có quy tắc trên máy IBM 7094 trong năm 1967. Phiên bản này phát sinh các khối nhỏ của các chỉ dẫn nhị phân 7094 cho các điều khiển khác nhau trong biểu thức, kết hợp chúng lại với nhau, và chạy chương trình kết quả bằng cách gọi nó, giống như một hàm bình thường. Các kỹ thuật tương tự có thể được áp dụng để tạo ra các thứ tự chỉ dẫn đặc trưng cho các công việc cập nhật màn hình trong các hệ thống đồ họa, nơi mà có nhiều trường hợp đặc biệt sao cho việc tạo lập mã nguồn linh động cho mỗi trường hợp phát sinh sẽ hiệu quả hơn là viết trước tất cả hay thêm vào các kiểm tra điều kiện trong mã nguồn tổng quát hơn.

Để minh họa các vấn đề liên quan đến việc xây dựng một trình biên dịch "sẵn" thật sự, chúng ta sẽ phải đi sâu nhiều hơn vào các chi tiết của một tập chỉ dẫn (lệnh) cụ thể, nhưng điều này đáng được bỏ ra một chút thời gian để hiểu được cách thức thực hiện của một hệ thống như thế. Phần còn

lại của phần này nhằm cung cấp các ý tưởng và hiểu được bản chất vấn đề, chứ không đi vào các chi tiết thực hiện.

Hãy nhớ lại rằng chúng ta đã đưa ra máy ảo với một cấu trúc như sau:

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* program counter */
        ...
        Tree *t;

        t = parse();
        pc = generate(0, t);
        code[pc].op = NULL;

        stackp = 0;
        pc = 0;
        while (code[pc].op != NULL)
                (*code[pc++].op)();
        return stack[0];
```

Để điều chỉnh đoạn mã nguồn này cho phù hợp với sự biên dịch "sẵn", chúng ta phải thực hiện một vài thay đổi. Trước hết, mảng `code` không còn là một mảng các con trỏ hàm nữa mà phải là một mảng các chỉ dẫn có thể thi hành được. Kiểu của các chỉ thị là `char`, `int`, hay `long` sẽ phụ thuộc vào bộ xử lý mà chúng tơ sẽ biên dịch; chúng ta sẽ giả sử là kiểu `int`. Sau khi đoạn mã nguồn được phát sinh, chúng ta sẽ gọi nó như gọi một

hàm. Sẽ không có bất kỳ một bộ đếm chương trình ảo nào bởi vì chu trình chạy của bộ xử lý sẽ đi dọc theo đoạn mã nguồn cho chúng ta; một khi phép tính được thực hiện xong, nó sẽ trả về giá trị, giống như một hàm bình thường. Cũng vậy, chúng ta có thể chọn cách duy trì một ngăn xếp riêng biệt chứa toán hạng cho máy hay sử dụng ngăn xếp riêng của bộ xử lý. Mỗi cách tiếp cận có các thuận lợi riêng, tuy nhiên, chúng ta sẽ chọn phương pháp dùng một ngăn xếp riêng và tập trung vào các chi tiết của chính đoạn mã nguồn này. Cài đặt bây giờ sẽ như sau:

```
typedef int Code;

Code code[NCODE];

int codep;

int stack[NSTACK];

int stackp;

      ...

      Tree *t;

      void (*fn)(void);

      int pc;


      t = parse();

      pc = generate(0, t);

      genreturn(pc);    /* phát sinh dãy function
return */

      stackp = 0;

      flushcaches();   /* đồng bộ bộ nhớ với bộ
vi xử lý */

fn = (void(*)(void)) code;
```

```
                                    /* ép kiểu mảng sang con trỏ
hàm */
                (*fn)();            /* thực hiện gọi hàm */
                return stack[0];
```

Sau khi hàm `generate` thực hiện xong, hàm `genreturn` đưa ra các chỉ thị yêu cầu mã nguồn được phát sinh trả lại điều khiển cho hàm `eval`.

Hàm `flushcaches` thi hành các bước cần thiết để chuẩn bị bộ xử lý cho việc chạy mã nguồn được phát sinh gần đây. Các máy hiện đại chạy nhanh một phần là vì chúng sử dụng cache cho các chỉ thị, dữ liệu, và các đường ống (pipeline) bên trong lấn một phần qua sự thi hành của các chỉ thị liên tiếp nhau. Các cache và đường ống này mong đợi luồng chỉ thị là tĩnh; nếu chúng ta phát sinh mã nguồn ngay trước khi thi hành, bộ xử lý có thể trở nên bối rối. CPU cần rút ngắn đường ống của nó và làm sạch các cache của nó trước khi nó thi hành các chỉ thị mới phát sinh. Đây là các điều khiển phụ thuộc rất nhiều vào máy; cài đặt của hàm `flushcaches` sẽ khác nhau trên các loại máy khác nhau.

Đoạn mã nguồn nguồn đáng chú ý ở đây là `(void(*)(void))`, được dùng để chuyển đổi địa chỉ của mảng chứa các chỉ thị được phát sinh sang một con trỏ có thể được dùng để gọi đoạn mã nguồn như một hàm.

Về phương diện kỹ thuật, không quá khó để phát sinh mã nguồn, mặc dù có không ít công nghệ cho phép thực hiện việc này một cách hiệu quả. Chúng ta bắt đầu với việc xây dựng các khối. Như trước đó, mảng `code` và chỉ số truy xuất mảng được duy trì trong suốt quá trình biên dịch. Để đơn giản, chúng ta sẽ khai báo chúng đều là toàn cục, như chúng ta đã làm trước đó. Tiếp đến chúng ta có thể viết một hàm để phát sinh ra các chỉ thị

```
/* emit: thêm chỉ thị vào luồng mã nguồn số */
void emit(Code inst)
{
```

```
code[codep++] = inst;
```

Các chỉ thị tự chúng có thể được định nghĩa bởi các macro phụ thuộc bộ xử lý hoặc các hàm nhỏ thiết lập các chỉ thị bằng cách điền vào các trường của từ chỉ thị. Một cách giả thuyết, chúng ta có thể có một hàm gọi là `popreg` dùng để phát sinh mã nguồn để lấy một giá trị ra khỏi ngăn xếp và lưu nó vào một thanh ghi xử lý, và một hàm khác gọi là `pushreg` dùng để phát sinh mã nguồn để lấy giá trị trong một thanh ghi và đưa nó vào ngăn xếp. Hàm `addop` được tổ chức lại của chúng ta sẽ sử dụng chúng như dưới đây, với một vài hằng số được định nghĩa trước đó mô tả các chỉ thị (như `ADDINST`) và dạng thức của chúng (các vị trí `SHIFT` khác nhau định nghĩa định dạng):

```
/* addop: phát sinh chỉ thị ADD */
void addop(void)
{
        Code inst;
        popreg(2);          /* lấy từ ngăn xếp đưa
vào thanh ghi 2 */
        popreg(1);          /* lấy từ ngăn xếp đưa
vào thanh ghi 1 */
        inst = ADDINST << INSTSHIFT;
        inst |= (R1) << OP1SHIFT;
        inst |= (R2) << OP2SHIFT;
        emit(inst);         /* thao tác ADD R1, R2 */
        pushreg(2);         /* đẩy giá trị thanh ghi
2 vào ngăn xếp */
```

Đây chỉ là điểm khởi đầu, nếu chúng ta định viết một trình biên dịch trực tiếp khi chạy chương trình thực sự thì chúng ta phải thực hiện các tối ưu hoá. Nếu chúng ta thêm một hằng số, chúng ta không cần đẩy hằng số vào ngăn xếp, lấy nó ra khỏi ngăn xếp, và thêm nó vào; chúng ta có thể thêm một cách trực tiếp. Ý tưởng tương tự có thể làm giảm bớt chi phí. Thậm chí như đã viết như trên, tuy nhiên, `addop` sẽ chạy nhanh hơn nhiều so với các phiên bản được viết trước đó bởi vì các điều khiển khác nhau không được tập hợp lại bằng các lời gọi hàm. Thay vào đó, mã nguồn dùng để chạy chúng được đặt vào bộ nhớ như là một khối các lệnh đơn, với bộ đếm chương trình của bộ xử lý thật sự làm tất cả các công việc tập hợp lại cho chúng ta.

Hàm `generate` có vẻ rất phù hợp cho việc thực hiện của máy ảo. Nhưng ở đây là các chỉ thị của máy thực sự thay vì các con trỏ tới các hàm được định nghĩa trước. Và để phát sinh mã nguồn hiệu quả, cần phải có một vài thực hiện về tìm kiếm các hằng số để loại bỏ và các tối ưu hoá khác.

Việc trình bày nhanh gọn của chúng ta về sự phát sinh mã nguồn, chỉ ra cho thấy một cách sơ lược một vài kỹ thuật được dùng bởi các trình biên dịch thực sự và còn rất nhiều vấn đề chưa được đề cập tới. Nó cũng bỏ qua nhiều vấn đề phát sinh bởi các phức tạp của các CPU hiện đại. Nhưng nó cũng minh họa cách thức mà một chương trình có thể phân tích mô tả của một vấn đề tạo ra mã nguồn với mục đích đặc biệt nhằm giải quyết nó một cách hiệu quả. Bạn có thể dùng các ý tưởng này để viết một phiên bản cực kỳ nhanh của `grep`, nhằm thay đổi ngôn ngữ nhỏ của chính bạn, để thiết kế và xây dựng một máy ảo được tối ưu hoá cho tính toán có mục đích đặc biệt, hay thậm chí, với một ít trợ giúp, để viết một trình biên dịch cho một ngôn ngữ thú vị.

Giữa một biểu thức có quy tắc và một chương trình C++ là một khoảng cách lớn, nhưng cả hai chỉ là hệ thống các ký hiệu được dùng để giải quyết các vấn đề. Với một hệ thống ký hiệu đúng, nhiều vấn đề trở nên dễ hơn. Và việc thiết kế và cài đặt hệ thống các ký hiệu có thể rất thú vị.

Bài tập 9-18. Trình biên dịch trong khi thực thi sẽ phát sinh mã nguồn thi hành nhanh hơn nếu nó có thể thay thế các biểu thức chứa các hằng số bằng giá trị của chúng. Khi nó gặp một biểu thức như thế thì nó sẽ tính giá trị của biểu thức đó như thế nào? Bạn hãy thử đề ra một cách để giải quyết.

Bài tập 9-19. Hãy đề ra một cách thức để kiểm tra trình biên dịch trong khi thực thi.

PHỤ LỤC

Phong cách

Dùng tên gợi nhớ cho biến toàn cục, tên ngắn gọn cho biến cục bộ.

Nhất quán.

Dùng tên hàm mang ý nghĩa chủ động.

Chính xác.

Canh chỉnh lề để thấy được cấu trúc.

Dùng hình thức diễn đạt tự nhiên.

Đóng và mở ngoặc để mã nguồn không bị mơ hồ.

Phân tích những biểu thức phức tạp thành những biểu thức đơn giản hơn.

Tính sáng sủa.

Cẩn thận với các hiệu ứng lề.

Sử dụng phong cách canh chỉnh lề và đấu ngoặc một cách nhất quán.

Dùng nhất quán các đặc ngữ.

Áp dụng các *else-if* cho các quyết định rẽ nhánh.

Tránh dùng các macro hàm.

Đóng và mở ngoặc phần chương trình cũng như các tham số của macro.

Đặt tên cho các số tối nghĩa.

Định nghĩa các con số dưới dạng hằng số, chứ không dùng macro.

Sử dụng các hằng ký tự, không nên dùng các số nguyên.

Dùng chính ngôn ngữ đó để tính toán kích thước của đối tượng.

Đừng tô đậm những điều hiển nhiên.

Ghi chú thích cho các hàm và biến toàn cục.

Đừng ghi chú thích cho các đoạn mã nguồn dở mà hãy viết lại nó.

Đừng phủ nhận mã nguồn.

Rõ ràng, không gây nhầm lẫn.

Các phương thức giao tiếp

Che giấu các chi tiết cài đặt.

Dùng tập các cấu trúc dữ liệu chuẩn.

Thực hiện cùng một cách đối với cùng một thể hiện ở mọi nơi.

Giải phóng tài nguyên tương ứng với cách thức cấp phát.

Dò tìm lỗi ở cấp độ thấp, xử lý chúng ở cấp độ cao.

Chỉ sử dụng ngoại lệ cho các tình huống ngoại lệ.

Gỡ rối

Tìm những mẫu tương tự.

Xem xét những thay đổi gần nhất.

Không lặp lại những lỗi đã mắc phải.

Gỡ rối ngay, không nên để lại về sau.

Cần kiểm tra lại ngăn xếp (stack).

Đọc trước khi gõ vào máy.

Giải thích mã nguồn để người khác có thể đọc hiểu.

Phải nghĩ rằng một lỗi đã phát hiện vẫn có thể xảy ra.

Áp dụng chiến lược "chia để trị".

Hiển thị kết quả để khoanh vùng tìm kiếm.

Viết các đoạn mã nguồn tự kiểm tra.

Viết một tập tin lưu vết (log file)

Vẽ hình.

Sử dụng công cụ.

Lưu giữ lại các bản ghi nhận về lỗi.

Kiểm chứng

Kiểm chứng mã nguồn tại các điều kiện biên.

Kiểm tra các điều kiện cần và đủ.

Dùng các khẳng định.

Lập trình cẩn thận.

Kiểm tra các giá trị lỗi trả về.

Thực hiện kiểm chứng ngày càng nhiều.

Kiểm tra những phần đơn giản trước.

Biết trước kết quả cần xuất ra.

Kiểm định các thuộc tính duy trì.

So sánh các phần cài đặt độc lập.

Đo lường tính bao quát của việc kiểm chứng.

Tự động hóa quá trình kiểm chứng lùi.

Tốc độ thực thi chương trình

Tự đông hóa quá trình đo lường thời gian.

Sử dụng profiler.

Tập trung vào các phần cần giải quyết tốt.

Vẽ hình.

Dùng thuật toán và cấu trúc dữ liệu tốt hơn.

Kích hoạt lựa chọn về tối ưu do trình biên dịch cung cấp.

Điều chỉnh mã nguồn.

Không cần tối ưu những phần không quan trọng.

Lựa chọn những biểu thức con.

Thay những phép toán tốn tài nguyên bằng các phép toán ít tốn hơn.

Hạn chế sử dụng vòng lặp.

Thực hiện cache những dữ liệu thường dùng.

Viết một hàm cấp phát tài nguyên cho những mục đích đặc biệt.

Cất giữ dữ liệu đầu vào và đầu ra vào vùng đệm.

Xử lý riêng lẻ các trường hợp đặc biệt.

Tính toán trước các kết quả.

Dùng các giá trị xấp xỉ.

Viết lại bằng ngôn ngữ cấp thấp hơn.

Tiết kiệm không gian bằng cách dung những kiểu dữ liệu nhỏ nhất (nếu có thể được)

Đừng lưu trữ lại những gì bạn có thể dễ dàng tính toán lại.

Tính khả chuyển

Bám sát các chuẩn.

Lập trình theo xu hướng chính.

Chú ý những điểm mà ngôn ngữ thường gây rắc rối.

Thử trên vài trình biên dịch.

Dùng các thư viện chuẩn.

Sử dụng các tính năng đã có sẵn.

Tránh dùng cách biên dịch có điều kiện.

Cục bộ hóa những phụ thuộc hệ thống bằng những tập tin riêng lẻ.

Che giấu những phụ thuộc hệ thống đằng sau các phương thức.

Dùng văn bản để trao đổi dữ liệu.

Dùng thứ tự byte cố định để trao đổi dữ liệu.

Thay đổi tên nếu thay đổi đặc tả.

Duy trì tính tương thích với hệ thống cũng như dữ liệu đang tồn tại.

Đừng giả sử đang sử dụng bảng mã ASCII.

Đừng giả sử đang dùng tiếng Anh.

Tài liệu tham khảo

[1] *Brian Kernighan and P. J. Plauger,* The Elements of Programming Style, McGraw-Hill, 1978.

[2] *Peter van der Linder,* Expert C Programming: Deep C Secrets, Prentice Hall, 1994.

[3] *Don Knuth,* The Art of Computer Programming, Volume 3, 2nd Edition, Addison Wesley, 1998.

[4] *Gerard Holzmann*, Design and Validation of Computer Protocols, Prentice Hall, 1991.

[5] *Matthew Austern*, Generic Programming and the STL, Addison Wesley, 1998.

[6] *Bjarne Stroustrup*, The C++ Programming Language, 3rd Edition, Addison Wesley, 1997.

[7] *Ken Arnold, James Gosling,* The Java Programming Language, 2nd Edition, Addison Wesley, 1998.

[8] *Larry Wall, Tom Christiansen, Randal Schwartz*, Programming Perl, 2nd Edition, O'Reilly, 1996.

[9] *Erich Gamma, Richard Helm, Ralph Johnson,* Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[10] *John Lakos,* Large-Scale C++ Software Design, Addison Wesley, 1996.

[11] *David Hanson,* C Interfaces and Implementations, Addison Wesley, 1997.

[12] *Steve McConnell,* Rapid Development, Microsoft Press, 1996.

[13] *Kevin Mullet, Darrell Sano,* Designing Visual Interfaces: Communication Oriented Techniques, Prentice Hall, 1995.

[14] *Ben Shneiderman,* Designing the User Interface: Strategies for Effective Human-Computer Interaction, 3rd Addison Wesley, 1997.

[15] *Steve Maguire*, Writing Solid Code, Microsoft Press, 1993.

[16] *Steve McConnell*, Code Complete, Microsoft Press, 1993.

[17] *Jon Bentley*, Communications of the ACM, Addison Wesley, 1986.

[18] *John Hennessy, David Patterson,* Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufman, 1997.

[19] *James Gosling, Bill Joy, Guy Steele,* The Java Language Specification, Addison Wesley, 1996.

[20] *Rich Steven,* Advanced Programming in the Unix Environment, Addison Wesley, 1992.

[21] *Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey*, The Inferno Operating System, Bell Labs Technical Journal, 2,1, Winter;-1997.

[22] *Brian Kernighan, Rob Pike,* The Unix Programming Environment, Prentice Hall, 1984.

[23] *Chris Fraser, David Hanson*, A Retargetable C Compiler: Design and Implementation, Addison Wesley, 1995.

[24] *Tim Lindholm, Frank Yellin,* The Java Virtual Machine Specification, 2nd Edition, Addison Wesley, 1999.

[25] *Brian W. Kernighan, Rob Pike,* The Practice of Programming, Addison Wesley, 1999.

KỸ NĂNG LẬP TRÌNH

Tác giả : Lê Hoài Bắc
Nguyễn Thanh Nghị

Chịu trách nhiệm xuất bản : PGS. TS. TÔ ĐĂNG HẢI

Biên tập và sửa bài : ThS. NGUYỄN HUY TIẾN

NGỌC LINH

Trình bày bìa : HƯƠNG LAN

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
70, TRẦN HƯNG ĐẠO – HÀ NỘI

In 700 cuốn, khổ 16x24cm, tại Nhà in KH&CN
Giấy phép xuất bản số: 06-287-30/12/2004
In xong và nộp lưu chiểu tháng 8 năm 2005

205189

Giá: 54.000đ