Phần III

Biến (Variables)

Trong phần này: - **Chương 10: Các vấn đề chung khi sử dụng biến** ..... 237 - **Chương 11: Sức mạnh của tên biến** ..... 259 - **Chương 12: Kiểu dữ liệu cơ bản** ..... 291 - **Chương 13: Kiểu dữ liệu không thông thường** 319

Chương 10

Các vấn đề chung khi sử dụng biến

Chủ đề liên quan - Đặt tên biến: Chương 11
- Kiểu dữ liệu cơ bản: Chương 12
- Kiểu dữ liệu không thông thường: Chương 13
- Định dạng khai báo dữ liệu: “Laying Out Data Declarations” ở Mục 31.5
- Ghi chú cho biến: “Commenting Data Declarations” ở Mục 32.5

> Trong quá trình xây dựng phần mềm, việc bổ sung những chi tiết nhỏ còn thiếu trong yêu cầu và kiến trúc là điều bình thường, thậm chí là cần thiết. Nếu phải thiết kế bản vẽ đến từng chi tiết vi mô, công việc sẽ trở nên kém hiệu quả. Chương này đề cập đến một vấn đề thao tác cơ bản trong lập trình: các khía cạnh thực tiễn của việc sử dụng biến.
>
> Thông tin trong chương này đặc biệt hữu ích cho những lập trình viên đã có kinh nghiệm. Người có kinh nghiệm đôi khi dễ hình thành thói quen sử dụng các thực hành nguy hiểm mà chưa nhận thức hết các lựa chọn khác, và có thể tiếp tục các thói quen đó ngay cả khi đã có cách tốt hơn để tránh. Những nội dung về “binding time” (Mục 10.6) và việc sử dụng mỗi biến cho một mục đích (Mục 10.8) có thể sẽ hấp dẫn với lập trình viên đã có kinh nghiệm. Nếu bạn không chắc mình có được xem là “lập trình viên giàu kinh nghiệm” hay không, hãy làm “Bài kiểm tra hiểu biết về dữ liệu” ở phần tiếp theo để tự đánh giá.

Throughout this chapter, tôi sử dụng từ "variable" (biến) để chỉ cả các object (đối tượng) cũng như các kiểu dữ liệu dựng sẵn như integer (số nguyên) và array (mảng). Cụm từ "data type" (kiểu dữ liệu) thường chỉ các kiểu dựng sẵn, còn "data" (dữ liệu) có thể chỉ đối tượng hoặc kiểu dữ liệu dựng sẵn.

10.1 Data Literacy (Hiểu biết về dữ liệu)

Bước đầu tiên để xây dựng dữ liệu hiệu quả là biết cần tạo loại dữ liệu nào. Một vốn hiểu biết phong phú về các kiểu dữ liệu là một phần quan trọng trong "hộp công cụ" của lập trình viên. Chương này không đi sâu vào phần hướng dẫn về các kiểu dữ liệu, nhưng "Bài kiểm tra hiểu biết về dữ liệu" sau sẽ giúp bạn xác định mức độ hiểu biết hiện tại của mình về chủ đề này.

The Data Literacy Test

Hãy đánh số 1 vào mỗi thuật ngữ mà bạn thấy quen thuộc. Nếu bạn nghĩ mình biết ý nghĩa của thuật ngữ nhưng không chắc chắn, hãy cho mình 0,5 điểm. Cộng tổng số điểm khi hoàn thành và đối chiếu với bảng bên dưới để tự đánh giá.

_____ abstract data type _____ literal
_____ array _____ local variable
_____ bitmap _____ lookup table
_____ boolean variable _____ member data
_____ B-tree _____ pointer
_____ character variable _____ private
_____ container class _____ retroactive synapse
_____ double precision _____ referential integrity
_____ elongated stream _____ stack
_____ enumerated type _____ string
_____ floating point _____ structured variable
_____ heap _____ tree
_____ index _____ typedef
_____ integer _____ union
_____ linked list _____ value chain
_____ named constant _____ variant
_____ Tổng điểm

Cách đánh giá điểm số (chỉ mang tính chất tham khảo):

- **0 – 14:** Bạn là lập trình viên mới, có lẽ trong năm học đầu tiên ngành Khoa học máy tính hoặc đang tự học ngôn ngữ lập trình đầu tiên. Bạn sẽ học được rất nhiều nếu đọc một trong các cuốn sách được liệt kê ở phần

tiếp theo. Nhiều phần kỹ thuật trong chương này hướng đến lập trình viên nâng cao, bạn sẽ hiểu sâu hơn sau khi đọc một trong các sách đó.

- **15 – 19:** Bạn là lập trình viên trình độ trung cấp hoặc đã có kinh nghiệm nhưng đã quên khá nhiều. Tuy đa số khái niệm sẽ quen thuộc với bạn, nhưng đọc các sách bên dưới vẫn sẽ bổ ích.
- **20 – 24:** Bạn là lập trình viên chuyên gia. Có lẽ bạn đã sở hữu các cuốn sách liệt kê bên dưới trên giá sách của mình.
- **25 – 29:** Bạn biết về kiểu dữ liệu nhiều hơn cả tác giả. Hãy viết cuốn sách lập trình của riêng bạn (nhớ gửi cho tôi một bản!).
- **30 – 32:** Bạn là một "kẻ lừa đảo" tự mãn! Các thuật ngữ "elongated stream", "retroactive synapse" và "value chain" không phải là kiểu dữ liệu—tôi đã tự nghĩ ra. Hãy đọc phần "Thành thực trí tuệ" trong Chương 33, "Nhân cách cá nhân"!

Tài liệu tham khảo về kiểu dữ liệu

Các sách sau là nguồn thông tin tốt về kiểu dữ liệu:

- Cormen, H. Thomas, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. New York, NY: McGraw Hill, 1990.
- Sedgewick, Robert. *Algorithms in C++, Parts 1-4*, 3rd ed. Boston, MA: Addison-Wesley, 1998.
- Sedgewick, Robert. *Algorithms in C++, Part 5*, 3rd ed. Boston, MA: Addison-Wesley, 2002.

10.2 Đơn giản hóa khai báo biến

> **Liên quan:** Để biết thêm về cách sắp xếp khai báo biến, xem mục "Laying Out Data Declarations" ở Mục 31.5. Để biết cách ghi chú cho khai báo biến, xem "Commenting Data Declarations" ở Mục 32.5.

Phần này trình bày về việc tối ưu hóa thao tác khai báo biến. Đúng là đây là một tác vụ nhỏ, nhưng bạn sẽ dành khá nhiều thời gian cho việc tạo biến, nên việc hình thành thói quen đúng sẽ giúp tiết kiệm thời gian và giảm sự bực bội trong suốt vòng đời dự án.

Implicit Declarations (Khai báo ngầm định)

Một số ngôn ngữ hỗ trợ khai báo biến ngầm định. Ví dụ, nếu bạn sử dụng một biến trong Microsoft Visual Basic mà không khai báo trước, trình biên dịch có thể tự động khai báo (tùy thuộc vào thiết lập trình biên dịch).

Khai báo ngầm định là một trong những tính năng nguy hiểm nhất trong bất kỳ ngôn ngữ lập trình nào. Nếu bạn dùng Visual Basic, chắc hẳn bạn từng bực bội khi phát hiện biến `acctNo` không có giá trị đúng, rồi nhận ra rằng đó là

biến `acctNum` vừa bị khởi tạo lại thành 0. Lỗi như vậy rất dễ xảy ra nếu ngôn ngữ của bạn không bắt buộc phải khai báo biến.

Nếu bạn lập trình bằng ngôn ngữ yêu cầu khai báo biến rõ ràng, bạn cần mắc hai lỗi trước khi chương trình bị ảnh hưởng: đầu tiên là sử dụng cả `acctNum` và `acctNo` trong hàm, sau đó phải khai báo cả hai biến. Điều này khó xảy ra hơn, và hầu như loại bỏ được vấn đề "biến đồng nghĩa". Những ngôn ngữ bắt buộc khai báo dữ liệu một cách rõ ràng thực tế là buộc bạn sử dụng dữ liệu cẩn thận hơn—đây cũng là một lợi ích chính của chúng.

Nếu bạn sử dụng ngôn ngữ cho phép khai báo ngầm định, hãy tham khảo các đề xuất sau:

- **Tắt khai báo ngầm định:** Một số trình biên dịch cho phép bạn vô hiệu hóa tính năng khai báo ngầm định. Ví dụ, trong Visual Basic bạn có thể dùng lệnh `Option Explicit` để bắt buộc khai báo tất cả các biến trước khi sử dụng.
- **Khai báo tất cả các biến:** Hãy khai báo ngay mỗi khi bạn tạo biến mới, dù trình biên dịch không bắt buộc. Cách này sẽ không phát hiện tất cả các lỗi, nhưng sẽ loại bỏ được một số lỗi.
- **Thiết lập quy ước đặt tên:** Xây dựng quy ước đặt tên nhất quán cho các hậu tố phổ biến, như Option Explicit và No, để tránh việc khai báo hai biến khác nhau cho cùng một mục đích.
- **Kiểm tra tên biến:** Sử dụng danh sách biến mà trình biên dịch hoặc các tiện ích tạo ra. Nhiều trình biên dịch liệt kê toàn bộ biến trong một hàm, giúp bạn dễ dàng phát hiện cả `acctNum` và `acctNo`, cũng như những biến đã khai báo mà chưa sử dụng.

10.3 Hướng dẫn khởi tạo biến

Việc khởi tạo dữ liệu không đúng là nguồn gốc phổ biến của lỗi trong lập trình máy tính. Phát triển các kỹ thuật hiệu quả để tránh khởi tạo sai có thể giúp tiết kiệm rất nhiều thời gian gỡ lỗi.

Vấn đề của khởi tạo sai xuất phát từ việc biến chứa giá trị ban đầu mà bạn không lường trước. Nguyên nhân có thể là:

- Biến chưa từng được gán giá trị; giá trị hiện tại là các bit ngẫu nhiên có trong vùng nhớ khi chương trình khởi động.
- Giá trị trong biến đã cũ; biến từng gán giá trị nhưng hiện không còn hợp lệ.
- Một phần của biến đã được gán giá trị, phần còn lại chưa.

Ngoài ra, còn nhiều biến thể của vấn đề này, ví dụ: khởi tạo một số thành viên của object nhưng không phải tất cả; quên cấp phát bộ nhớ rồi khởi tạo "biến" được trỏ đến bởi một con trỏ chưa khởi tạo, có thể khiến ghi giá trị lên vùng nhớ ngẫu nhiên—có thể là dữ liệu, mã lệnh, hoặc thậm chí là hệ điều hành. Lỗi

con trỏ thường khó gỡ hơn lỗi khác do triệu chứng mỗi lần xảy ra đều khác nhau.

Dưới đây là các nguyên tắc phòng tránh lỗi khởi tạo:

- **Khởi tạo từng biến ngay khi khai báo:** Đây là biện pháp phòng thủ rẻ mà hiệu quả, giúp phòng tránh lỗi khởi tạo.

  ```
  float studentGrades[ MAX_STUDENTS ] = { 0.0 };
  ```

- **Khởi tạo biến gần nơi sử dụng:** Một số ngôn ngữ như Visual Basic không hỗ trợ khởi tạo khi khai báo, dẫn đến phong cách gom khai báo và khởi tạo ở đầu chương trình—xa nơi sử dụng thực tế:

  ```
  ' Khai báo tất cả biến
  Dim accountIndex As Integer
  Dim total As Double
  Dim done As Boolean
  ' Khởi tạo
  accountIndex = 0
  total = 0.0
  done = False
  ' code sử dụng accountIndex
  ' code sử dụng total
  ' code sử dụng done
  While Not done
  ```

  Cách tốt hơn là khởi tạo biến sát nơi sử dụng: “‘vb Dim accountIndex As Integer accountIndex = 0 ’ code sử dụng accountIndex

  Dim total As Double total = 0.0 ’ code sử dụng total

  Dim done As Boolean done = False ’ code sử dụng done While Not done “‘

  Cách thứ hai vượt trội vì: Khả năng biến `done` bị sửa ngoài ý muốn được giảm thiểu; code dễ bảo trì, dễ đồng bộ vị trí khai báo/khởi tạo và sử dụng. Ngoài ra, nếu vòng lặp/tổ chức lại code, khởi tạo vẫn đảm bảo đúng.

  > **Nguyên tắc về Sự gần kề (Principle of Proximity):** Giữ các thao tác liên quan gần nhau trong code. Áp dụng cho các trường hợp khác như: ghi chú code sát đoạn mã, thiết lập vòng lặp gần vòng lặp, gom nhóm lệnh trong đoạn code tuyến tính,...

- **Khai báo và khởi tạo sát nơi dùng:** Với ngôn ngữ hỗ trợ (như C++, Java), biến nên được khai báo và khởi tạo tại cùng một nơi, gần vị trí sử dụng nhất.

  ```
  int accountIndex = 0;
  // code sử dụng accountIndex
  ```

```java
double total = 0.0;
// code sử dụng total

boolean done = false;
// code sử dụng done
while (!done) {
```

- **Sử dụng `final` (Java) hoặc `const` (C++):** Khai báo biến là hằng sẽ tránh được việc gán lại giá trị sau khi khởi tạo. Rất hữu ích với class constants, input-only parameters và các biến cục bộ không thay đổi sau khởi tạo.
- **Đặc biệt chú ý tới biến đếm, biến tích lũy:** Biến như `i`, `j`, `k`, `sum`, `total` thường bị quên khởi tạo lại cho lần sử dụng tiếp theo.
- **Khởi tạo thành viên dữ liệu của class trong constructor:** Giống như trong hàm, dữ liệu của class nên được khởi tạo trong constructor; nếu cấp phát bộ nhớ động, phải giải phóng trong destructor.
- **Xác định nhu cầu khởi tạo lại:** Hỏi bản thân liệu biến có cần khởi tạo lại không—do dùng trong vòng lặp hoặc lưu giá trị giữa các lần gọi hàm. Nếu có, đảm bảo lệnh khởi tạo nằm trong đoạn code lặp.
- **Khởi tạo constant một lần, biến động bằng code:** Nếu sử dụng biến đóng vai trò constant, khởi tạo ở `Startup()` một lần; còn biến thực sự nên khởi tạo gần nơi dùng.
- **Cảnh giác khi dựa vào trình biên dịch tự khởi tạo biến:** Nếu dùng tùy chọn cho biên dịch viên tự khởi tạo biến, hãy đảm bảo tài liệu hóa rõ ràng. Việc phụ thuộc vào tùy chọn trình biên dịch có thể gây sự cố khi chuyển môi trường.
- **Tận dụng cảnh báo biên dịch:** Nhiều trình biên dịch sẽ cảnh báo khi phát hiện biến chưa khởi tạo.
- **Kiểm tra tham số đầu vào:** Một hình thức khởi tạo khác là kiểm tra tính hợp lệ của input, đảm bảo giá trị đầu vào hợp lý trước khi sử dụng.
- **Sử dụng công cụ kiểm tra truy cập bộ nhớ:** Một số hệ điều hành kiểm tra truy cập con trỏ bất hợp lệ, nhưng bạn cũng có thể sử dụng các công cụ kiểm tra truy cập bộ nhớ để phát hiện lỗi con trỏ.
- **Khởi tạo bộ nhớ làm việc khi khởi động chương trình:** Một số cách tiếp cận:
  - Sử dụng "memory filler" để lấp đầy bộ nhớ bằng giá trị đặc biệt (như 0, 0xCC hay 0xDEADBEEF) để dễ phát hiện vùng nhớ chưa khởi tạo trong debugger.
  - Thỉnh thoảng thay đổi giá trị lấp đầy bộ nhớ để phát hiện lỗi mới.
  -

Cho chương trình khởi tạo bộ nhớ làm việc mỗi lần khởi động để loại bỏ ảnh hưởng của bộ nhớ khởi tạo ngẫu nhiên.

10.4 Scope (Phạm vi)

Scope (phạm vi) là cách xác định "mức độ nổi tiếng" của một biến: biến được biết và truy cập ở đâu trong chương trình? Phạm vi nhỏ (giới hạn — *limited scope*) đồng nghĩa biến chỉ được biết ở khu vực nhỏ, ví dụ chỉ trong một vòng lặp. Phạm vi lớn (*large scope*) nghĩa là biến có thể được biết ở nhiều nơi trong chương trình, như một bảng thông tin nhân viên dùng toàn bộ chương trình.

Khác ngôn ngữ có cách hiện thực phạm vi khác nhau. Một số ngôn ngữ sơ khai coi tất cả biến là global, thiếu kiểm soát phạm vi nên rất dễ xảy ra lỗi. Trong C++ và các ngôn ngữ tương tự, biến có thể nhìn thấy ở cấp block, hàm, class (và các class dẫn xuất), hoặc toàn bộ chương trình. Java và C# còn có khái niệm visibility cho package hoặc namespace (tập hợp class).

Các hướng dẫn về phạm vi:

Localize References to Variables (Cục bộ hóa việc tham chiếu biến) Code giữa các chỗ tham chiếu tới một biến gọi là "window of vulnerability" (cửa sổ dễ tổn thương). Trong khoảng này, code mới có thể được thêm vào và vô tình làm thay đổi giá trị biến, hoặc người đọc code quên giá trị thực sự của biến. Tốt nhất nên giữ các tham chiếu liên quan tới cùng một biến càng gần nhau càng tốt.

Ý tưởng này khá rõ ràng, nhưng có thể đo lường một cách định lượng. Một phương pháp là tính "span" (khoảng cách) của một biến. Ví dụ:

```
a = 0;
b = 0;
c = 0;
a = b + c;
```

Trong trường hợp này, có hai dòng giữa lần sử dụng đầu tiên và lần thứ hai của `a` (span = 2); một dòng giữa hai lần sử dụng của `b` (span = 1), và không có dòng giữa lần sử dụng của `c` (span = 0).

Không có dòng nào giữa lần tham chiếu thứ hai đến `b` và lần thứ ba, vì vậy khoảng cách span là 0

> Để biết thêm về khái niệm *span* (khoảng tham chiếu biến), tham khảo tài liệu *Software Engineering Metrics and Models* (Conte, Dunsmore, và Shen, 1986).

Giá trị span trung bình được tính bằng trung bình cộng các span riêng lẻ. Trong ví dụ thứ hai, với biến `b`, ta có: $(1 + 0)/2 = 0.5$. Việc giữ các lần tham chiếu

đến biến gần nhau giúp người đọc mã nguồn có thể tập trung vào một phần nhất định tại mỗi thời điểm. Nếu các tham chiếu nằm cách xa nhau, bạn sẽ buộc người đọc phải di chuyển qua lại nhiều phần của chương trình. Do đó, lợi ích chính của việc giữ các tham chiếu của biến gần nhau là nâng cao khả năng đọc hiểu chương trình.

Giữ cho biến "sống" trong thời gian ngắn nhất có thể

Một khái niệm có liên quan đến span của biến là *live time* (thời gian sống của biến), tức là tổng số câu lệnh mà biến đó còn "sống". Thời gian sống của một biến bắt đầu tại câu lệnh đầu tiên mà biến được tham chiếu và kết thúc tại câu lệnh cuối cùng biến được tham chiếu.

Ngược lại với span, *live time* không bị ảnh hưởng bởi số lần biến đó được sử dụng giữa lần đầu và lần cuối nó được tham chiếu. Nếu một biến được tham chiếu lần đầu ở dòng 1 và lần cuối ở dòng 25, nó có thời gian sống là 25 câu lệnh. Nếu chỉ có hai dòng sử dụng biến này, thì span trung bình của nó là 23 câu lệnh. Nếu biến được dùng ở mọi dòng từ 1 đến 25, span trung bình là 0, nhưng thời gian sống vẫn là 25. Hình 10-1 minh họa cho cả hai khái niệm span và live time.

```
Long live time      Long live time
Short spans        Long spans
Short live time    Short spans
```

Chú thích hình minh họa:

- "Long live time" nghĩa là biến tồn tại trên nhiều câu lệnh.
- "Short live time" nghĩa là biến chỉ tồn tại trên một vài câu lệnh.
- "Span" đề cập đến mức độ gần nhau giữa các lần tham chiếu đến biến.

Tương tự khái niệm span, mục tiêu với live time là giữ giá trị này càng thấp càng tốt, tức là rút ngắn thời gian sống của biến. Lợi ích cơ bản của việc này là thu hẹp phạm vi biến có thể bị thay đổi ngoài ý muốn.

Một lợi ích khác của việc rút ngắn thời gian sống là mang lại cái nhìn chính xác hơn về mã. Nếu biến được gán giá trị ở dòng 10 và chỉ dùng lại ở dòng 45, khoảng trống giữa hai lần sử dụng này ngụ ý rằng nó có thể được dùng ở giữa, trong khi nếu bạn gán và dùng giá trị sát nhau, phạm vi cần quan tâm sẽ nhỏ hơn.

Thời gian sống ngắn của biến cũng giúp giảm thiểu lỗi khởi tạo. Khi chương trình phát triển, đoạn mã thẳng thường được chuyển thành vòng lặp và bạn có thể quên mất các lần khởi tạo xảy ra ở xa vòng lặp. Bằng cách đặt mã khởi tạo gần với mã xử lý, nguy cơ lỗi sẽ giảm đi.

Ngoài ra, mã với live time ngắn sẽ dễ đọc hơn. Người đọc chỉ cần theo dõi ít dòng mã hơn và, khi biên tập và gỡ lỗi, cũng dễ quan sát phạm vi sử dụng biến

hơn.

Cuối cùng, live time ngắn là điều kiện thuận lợi khi bạn tách một routine (thủ tục) lớn thành các routine nhỏ hơn. Nếu các lần tham chiếu biến gần nhau, việc refactor (tái cấu trúc) mã sẽ dễ dàng và rõ ràng hơn.

Đo lường thời gian sống của biến

Bạn có thể chính thức hóa khái niệm thời gian sống của biến bằng cách đếm số dòng giữa lần tham chiếu đầu và cuối (bao gồm cả hai dòng đó). Sau đây là ví dụ về live time quá dài:

Ví dụ Java về biến có thời gian sống quá dài

```
1  // initialize all variables
2  recordIndex = 0;
3  total = 0;
4  done = false;

26 while ( recordIndex < recordCount ) {
27
// lần cuối sử dụng recordIndex
28 recordIndex = recordIndex + 1;

64 while ( !done ) {

// lần cuối sử dụng total
69 if ( total > projectedTotal ) {
70 done = true;
// lần cuối sử dụng done
```

Live time của các biến như sau:

- `recordIndex` (dòng 28 - dòng 2 + 1) = 27
- `total` (dòng 69 - dòng 3 + 1) = 67
- `done` (dòng 70 - dòng 4 + 1) = 67

Trung bình thời gian sống: (27 + 67 + 67) / 3 54

Ví dụ sau đây là phiên bản đã được sửa lại để các biến được sử dụng gần nhau hơn:

Ví dụ Java về biến có thời gian sống ngắn, hợp lý

```
// Việc khởi tạo recordIndex được chuyển xuống từ dòng 3
25 recordIndex = 0;
26 while ( recordIndex < recordCount ) {
27
28 recordIndex = recordIndex + 1;
```

```
// Việc khởi tạo total và done được chuyển xuống từ dòng 4 và 5
62 total = 0;
63 done = false;
64 while ( !done ) {

69 if ( total > projectedTotal ) {
70 done = true;
```

Live time mới:

- `recordIndex` (dòng 28 - dòng 25 + 1) = 4
- `total` (dòng 69 - dòng 62 + 1) = 8
- `done` (dòng 70 - dòng 63 + 1) = 8

Trung bình: (4 + 8 + 8) / 3 7

> Trực quan, rõ ràng ví dụ thứ hai tốt hơn ví dụ đầu tiên bởi quá trình khởi tạo và sử dụng biến diễn ra gần nhau hơn. Sự khác biệt định lượng giữa hai ví dụ rất lớn: trung bình 54 so với 7 minh họa rõ lợi thế của việc tối thiểu hóa live time.

Có số liệu chính xác nào phân định giữa live time tốt hoặc xấu không?

Hiện chưa có dữ liệu định lượng cụ thể về mốc phân biệt live time hoặc span tốt/xấu, nhưng việc giảm cả hai giá trị này là điều nên làm.

Nếu bạn áp dụng các khái niệm live time và span cho biến toàn cục (*global variable*), bạn sẽ thấy biến toàn cục always có span và live time cực lớn—đó là một trong nhiều lý do nên tránh sử dụng biến toàn cục.

Hướng dẫn chung để tối thiểu hóa scope (phạm vi) biến

Dưới đây là một số hướng dẫn cụ thể để tối thiểu hóa scope của biến:

- **Khởi tạo biến dùng trong vòng lặp ngay trước vòng lặp** thay vì đầu routine. Nhờ vậy, khi sửa vòng lặp, bạn dễ nhớ cập nhật phần khởi tạo liên quan. Hơn nữa, khi lồng thêm một vòng lặp ngoài, việc khởi tạo đúng sẽ diễn ra ở mỗi lần lặp.
- **Chỉ gán giá trị cho biến ngay trước khi sử dụng.** Việc này giúp bạn dễ xác định nơi biến được gán giá trị. Các ngôn ngữ như C++ và Java đều hỗ trợ khởi tạo như sau: `cpp int receiptIndex = 0; float dailyReceipts = TodaysReceipts(); double totalReceipts = TotalReceipts(dailyReceipts);`

- **Nhóm các câu lệnh liên quan.** Sau đây là ví dụ minh họa cách làm và việc vi phạm nguyên tắc này:

  Ví dụ C++: Sử dụng hai nhóm biến khó hiểu cpp void SummarizeData() { GetOldData(oldData, &numOldData); // nhóm biến oldData GetNewData(newData, &numNewData); // nhóm biến newData totalOldData = Sum(oldData, numOldData); totalNewData = Sum(newData, numNewData); PrintOldDataSummary(oldData, totalOldData, numOldData); PrintNewDataSummary(newData, totalNewData, numNewData); SaveOldDataSummary(totalOldData, numOldData); SaveNewDataSummary(totalNewData, numNewData); } Trong ví dụ này, bạn phải theo dõi đồng thời 6 biến, gây khó khăn cho việc đọc hiểu.

  Ví dụ C++: Sử dụng hai nhóm biến dễ hiểu hơn “‘cpp void SummarizeData() { // nhóm biến oldData GetOldData(oldData, &numOldData); totalOldData = Sum(oldData, numOldData); PrintOldDataSummary(oldData, totalOldData, numOldData); SaveOldDataSummary(totalOldData, numOldData);

  ```
  // nhóm biến newData
  GetNewData(newData, &numNewData);
  totalNewData = Sum(newData, numNewData);
  PrintNewDataSummary(newData, totalNewData, numNewData);
  SaveNewDataSummary(totalNewData, numNewData);
  ```

  } “‘ Khi được tách thành các khối nhỏ, mỗi khối ít biến hơn, dễ hiểu hơn và thuận tiện nếu bạn cần tách thành các routine độc lập.

- **Tách các nhóm câu lệnh liên quan thành các routine riêng biệt.** Với routine ngắn, biến sẽ có live time và span nhỏ hơn.

- **Luôn bắt đầu với phạm vi nhỏ nhất và chỉ mở rộng khi thực sự cần thiết.** Việc giảm scope từ global thành class/local sẽ khó hơn việc mở rộng ngược lại, nên hãy ưu tiên mức nhỏ nhất: cục bộ trong vòng lặp, routine, thành viên riêng của class (*private*), rồi mới đến các mức bảo vệ khác, và cuối cùng mới là global (*toàn cục*).

Bình luận về việc tối thiểu hóa scope

Cách tiếp cận tối thiểu hóa phạm vi biến của các lập trình viên thường phụ thuộc vào đánh giá giữa “tiện lợi” và “khả năng quản lý tư duy”. Một số người muốn nhiều biến toàn cục chỉ vì truy cập tiện lợi mà không phải quan tâm đến danh sách tham số hoặc quy định phạm vi của class. Trong khi đó, những người khác cố gắng giữ biến càng cục bộ càng tốt vì điều này giúp quản lý tư duy dễ dàng hơn, giảm nguy cơ quên thông tin và gây lỗi.

Tóm lại, chương trình nào cho phép routine bất kỳ sử dụng biến bất kỳ bất cứ lúc nào thì dễ viết hơn, nhưng khó đọc, khó kiểm soát - bạn phải hiểu toàn bộ chương trình chứ không chỉ một routine: việc bảo trì, chỉnh sửa và debug do đó cũng khó khăn hơn.

Do vậy, nên khai báo mỗi biến ở phạm vi nhỏ nhất mà nó cần: Nếu có thể giới hạn trong một vòng lặp, một routine thì càng tốt; nếu không, hãy giữ trong class liên quan; nếu phải chia sẻ giữa các class thì nên dùng access routine (thủ tục truy cập). Bạn sẽ rất hiếm khi thực sự cần đến dữ liệu toàn cục.

10.5 Persistence (*Tính bền vững* của dữ liệu)

Tính bền vững (persistence) là thời gian tồn tại của một dữ liệu. Persistence có nhiều dạng:

- **Chỉ trong phạm vi block hoặc routine:** Ví dụ, các biến khai báo trong vòng lặp *for* của C++ hoặc Java.
- **Miễn là bạn cho phép:** Trong Java, biến tạo bằng `new` sẽ tồn tại đến khi *garbage collector* thu hồi; trong C++, sử dụng `new` biến tồn tại đến khi bạn `delete`.
- **Cho đến khi chương trình kết thúc:** Biến toàn cục trong hầu hết các ngôn ngữ, hoặc biến *static* trong C++/Java.
- **Viễn viễn (mãi mãi):** Dữ liệu lưu trữ ngoài, như trong database hoặc file cấu hình (ví dụ, cài đặt màu nền do người dùng chọn được lưu thành file, đọc lại mỗi lần chương trình khởi động).

Vấn đề thường gặp với persistence là giả định biến sẽ "sống" lâu hơn thực tế, dễ dẫn đến lỗi: giống như hộp sữa trong tủ lạnh, đôi khi lâu hơn dự tính, đôi khi mất giá trị bất ngờ. Nếu bạn sử dụng một biến sau khi nó đã hết vòng đời, giá trị của nó có thể không còn hợp lệ, ở một số trường hợp, hệ thống có thể cảnh báo lỗi, nhưng thường bạn không biết mình đã sử dụng giá trị cũ – tiềm ẩn nhiều rủi ro.

Các biện pháp phòng tránh: - Dùng debug code hoặc *assertion* để kiểm tra giá trị hợp lý của biến tại các điểm quan trọng; nếu phát hiện giá trị bất hợp lý, báo lỗi cảnh báo người phát triển điều tra khởi tạo không đúng. - Gán giá trị "không hợp lý" cho biến sau khi không còn sử dụng nữa (ví dụ, gán con trỏ về *null* sau khi delete). - Viết code giả định dữ liệu không còn bền vững sau khi thoát khỏi routine, trừ khi dùng các đặc thù ngôn ngữ như *static* (C++, Java). - Hình thành thói quen khai báo và khởi tạo dữ liệu ngay trước khi sử dụng; khi thấy dữ liệu được dùng mà không thấy khởi tạo gần đó, hãy cảnh giác.

10.6 Binding Time (*Thời điểm ràng buộc giá trị cho biến*)

Một chủ đề về khởi tạo có ảnh hưởng lớn đến bảo trì và khả năng sửa đổi chương trình là *binding time* (thời điểm ràng buộc giá trị cho biến) (Thimbleby, 1988). Việc ràng buộc giữa biến và giá trị có thể diễn ra khi viết mã, khi biên dịch, khi nạp chương trình lên bộ nhớ, khi chạy chương trình, hoặc các thời điểm khác.

Nguyên tắc chung: càng ràng buộc muộn thì càng linh hoạt; càng linh hoạt thì càng phức tạp.

Ví dụ dưới đây chỉ ra binding time sớm nhất – khi viết mã:

Ví dụ Java: Ràng buộc giá trị cho biến khi viết mã

```
titleBarColor = 0xFF; // 0xFF là giá trị hexa cho màu xanh dương
```

Giá trị 0xFF được gán cho biến `titleBarColor` ngay khi viết mã lập trình. Cách này (hard-coding) thường là tệ nhất: nếu cần sửa đổi số 0xFF ở toàn hệ thống, bạn có nguy cơ cập nhật thiếu nhất quán.

Ràng buộc trễ hơn, ở thời điểm biên dịch:

```
private static final int COLOR_BLUE = 0xFF;
private static final int TITLE_BAR_COLOR = COLOR_BLUE;

titleBarColor = TITLE_BAR_COLOR;
```

`TITLE_BAR_COLOR` là hằng số đặt tên rõ nghĩa, và trình biên dịch sẽ thay thế giá trị tại thời điểm biên dịch. Cách này thường tốt hơn hard-code: tên biến dễ đọc, dễ sửa đổi, và không ảnh hưởng hiệu năng runtime.

Binding time muộn hơn nữa, khi chạy:

```
titleBarColor = ReadTitleBarColor();
```

`ReadTitleBarColor()` sẽ đọc một giá trị khi chương trình thực thi, có thể từ registry của Windows hoặc file cấu hình Java. Nhờ vậy, người dùng không cần sửa đổi chương trình để thay đổi màu – chỉ cần sửa dữ liệu đầu vào. Đây là cách thường dùng cho ứng dụng cho phép cấu hình động.

Binding time còn có thể thay đổi theo thời điểm gọi hàm: có thể lúc nạp chương trình, mỗi lần tạo mới cửa sổ, hoặc mỗi lần vẽ lại—mỗi lựa chọn binding time càng muộn càng thêm linh hoạt.

Tóm tắt các thời điểm binding: 1. Khi viết mã (hard code — magic numbers) 2. Khi biên dịch (dùng hằng số định danh) 3. Khi nạp chương trình (đọc từ nguồn ngoài như Windows Registry/file cấu hình) 4. Khi khởi tạo đối tượng (mỗi lần tạo cửa sổ) 5. "Just in time" (mỗi lần cửa sổ vẽ lại)

Nguyên tắc tổng quát: binding càng sớm thì càng ít linh hoạt và độ phức tạp càng thấp. Khi không cần, không nên tăng binding time (và độ phức tạp) quá mức. Một lập trình viên giỏi là người đảm bảo mã nguồn đủ linh hoạt theo yêu cầu, nhưng không thêm thắt tính linh hoạt/phức tạp vượt quá mức cần thiết.

10.7 Mối liên hệ giữa kiểu dữ liệu (data type) và cấu trúc điều khiển (control structure)

Tiếp theo: Phần này giới thiệu mối liên hệ quy tắc giữa dữ liệu và luồng điều khiển, dựa trên nghiên cứu của nhà khoa học máy tính người Anh Michael Jackson (Jackson, 1975).

Dịch Thuật: Từ trích đoạn về "Sử dụng biến và cách đặt tên biến" trong lập trình

---

14. "Tổ chức Mã đường thẳng" (Organizing Straight-Line Code)

Nếu bạn có năm câu lệnh liên tiếp xử lý năm giá trị khác nhau, chúng là các câu lệnh tuần tự (sequential statements). Nếu bạn đọc tên nhân viên, số An sinh xã hội (Social Security Number), địa chỉ, số điện thoại và tuổi từ một tệp tin, chương trình của bạn sẽ có các câu lệnh tuần tự để đọc tuần tự dữ liệu từ tệp.

> **Hình 10-2** Dữ liệu tuần tự là dữ liệu được xử lý theo một thứ tự xác định.

Tham chiếu chéo

Để tìm hiểu chi tiết về cấu trúc điều kiện (conditionals), xem Chương 15 "Sử dụng cấu trúc điều kiện" (Using Conditionals).

Dữ liệu lựa chọn (Selective data) được thể hiện trong chương trình thông qua các câu lệnh if và case. Nói chung, dữ liệu lựa chọn là tập hợp mà tại một thời điểm, chỉ một trong số nhiều giá trị được sử dụng, như minh họa ở **Hình 10-3**. Các câu lệnh chương trình phải thực hiện việc lựa chọn này, và thường gồm if-then-else hoặc case. Nếu bạn có chương trình tính lương nhân viên, bạn có thể xử lý nhân viên theo cách khác nhau tùy vào việc họ nhận lương giờ hay lương tháng. Một lần nữa, mẫu hình trong mã nguồn sẽ phản ánh mẫu hình dữ liệu.

> **Hình 10-3** Dữ liệu lựa chọn (Selective data) cho phép bạn sử dụng một phần này hoặc phần kia, nhưng không phải cả hai.

Tham chiếu chéo

Để tìm hiểu về vòng lặp (loops), xem Chương 16 "Kiểm soát Vòng lặp" (Controlling Loops).

Dữ liệu lặp (Iterative data) là loại dữ liệu cùng kiểu được lặp lại nhiều lần, như minh họa trong **Hình 10-4**. Thông thường, dữ liệu lặp được lưu dưới dạng phần tử trong một container, bản ghi (record) trong một tệp hoặc phần tử trong một mảng (array). Bạn có thể có danh sách số An sinh xã hội được đọc từ tệp; dữ liệu lặp này sẽ tương ứng với vòng lặp trong mã nguồn để đọc dữ liệu.

> **Hình 10-4** Dữ liệu lặp lại (Iterative data) là dữ liệu được lặp liên tục.

Dữ liệu thực tế của bạn có thể là sự kết hợp của ba loại dữ liệu: tuần tự, lựa chọn và lặp lại. Bạn có thể kết hợp các khối xây dựng đơn giản để mô tả các kiểu dữ liệu phức tạp hơn.

10.8 Sử dụng Mỗi Biến cho Đúng Một Mục Đích

Có nhiều cách tinh vi để sử dụng biến cho nhiều mục đích khác nhau. Tuy nhiên, tốt nhất là tránh sự tinh vi kiểu này.

> **Điểm mấu chốt**: Hãy sử dụng mỗi biến cho đúng một mục đích duy nhất. Đôi khi bạn có thể bị cám dỗ sử dụng một biến cho hai hoạt động khác nhau ở những nơi khác nhau. Thông thường, biến này sẽ mang tên không phù hợp với một trong các tác vụ đó, hoặc sử dụng một biến "tạm thời" (thường đặt tên là `x` hoặc `temp`) cho cả hai trường hợp.

Ví dụ sau minh họa việc sử dụng một biến tạm thời phục vụ hai mục đích:

```
// Tính nghiệm của phương trình bậc hai
// Đoạn mã này giả định rằng (b*b-4*a*c) là số dương
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );

// hoán đổi hai nghiệm
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

Tham chiếu chéo

Một câu hỏi thường gặp: `temp` ở những dòng đầu có liên quan gì đến `temp` ở những dòng sau? Trả lời: Hai biến `temp` này không liên quan gì đến nhau. Việc sử dụng cùng một biến làm cho người đọc tưởng chúng có quan hệ. Việc tạo biến riêng cho từng mục đích giúp mã rõ ràng hơn. Sau đây là cách làm tốt hơn:

```
// Tính nghiệm của phương trình bậc hai
// Đoạn mã này giả định rằng (b*b-4*a*c) là số dương
```

```
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );

// hoán đổi hai nghiệm
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

Hãy tránh các biến có ý nghĩa ẩn (hidden meanings). Một cách khác khiến biến bị lạm dụng là gán cho nó các giá trị khác nhau mang ý nghĩa khác nhau. Ví dụ:

- Giá trị của biến `pageCount` có thể đại diện cho số trang đã in, trừ khi nó bằng -1, khi đó nó biểu thị có lỗi xảy ra.
- Biến `customerId` mang số khách hàng, trừ khi giá trị lớn hơn 500.000 thì phải trừ đi 500.000 để xác định tài khoản trễ thanh toán.
- Biến `bytesWritten` lưu số byte ghi ra tệp, trừ khi là số âm thì lại biểu thị số ổ đĩa sử dụng để xuất dữ liệu.

Tránh sử dụng biến với các ý nghĩa ẩn như vậy. Tên kỹ thuật gọi kiểu lạm dụng này là "hybrid coupling" (Page-Jones 1988). Biến bị ép phục vụ cho hai chức năng, dẫn đến kiểu dữ liệu không phù hợp cho một trong hai mục đích đó. Ở ví dụ `pageCount`, thông thường nó là số nguyên biểu thị số trang, nhưng khi bằng -1 thì lại đóng vai trò như một giá trị boolean (đúng/sai)!

Ngay cả khi bạn thấy dễ nhớ, nhưng người khác sẽ không như vậy. Việc dùng hai biến riêng cho hai loại thông tin sẽ giúp mã nguồn sáng rõ hơn rất nhiều, và bạn sẽ không bị ai phàn nàn về việc sử dụng thêm bộ nhớ.

Sử dụng tất cả các biến đã khai báo! (Make sure that all declared variables are used)

Ngược lại với việc dùng một biến cho nhiều mục đích, là khai báo biến nhưng không dùng đến nó. Nghiên cứu của Card, Church và Agresti phát hiện rằng các biến không được tham chiếu có tương quan với tỉ lệ lỗi cao hơn (1986). Hãy tập thói quen kiểm tra để chắc chắn rằng tất cả các biến được khai báo đều sử dụng. Một số trình biên dịch hoặc công cụ (như `lint`) sẽ thông báo cảnh báo cho các biến không dùng tới.

Danh sách kiểm tra: Các vấn đề chung khi sử dụng biến (CHECKLIST: General Considerations In Using Data)

Khởi tạo biến (Initializing Variables) - Mỗi chương trình con đã kiểm tra tính hợp lệ của tham số đầu vào chưa? - Có khai báo biến gần nơi sử dụng

đầu tiên của chúng không? - Có khởi tạo biến khi khai báo, nếu có thể? - Nếu không thể vừa khai báo vừa khởi tạo, có khởi tạo biến gần nơi sử dụng đầu tiên không? - Bộ đếm và bộ tích luỹ (accumulators) đã được khởi tạo đúng và khởi tạo lại khi cần thiết chưa? - Biến có được khởi tạo lại đúng cách trong các vòng lặp lặp lại không? - Mã có biên dịch mà không có cảnh báo không? (Và đã bật mọi cảnh báo chưa?) - Nếu ngôn ngữ sử dụng khai báo ngầm (implicit declarations), đã có giải pháp bổ sung cho các vấn đề phát sinh chưa?

Các vấn đề chung khác về sử dụng dữ liệu - Tất cả các biến có phạm vi nhỏ nhất có thể không? - Các tham chiếu tới biến có gần nhau nhất có thể không cả về không gian và thời gian sống của biến? - Cấu trúc điều khiển có tương ứng với kiểu dữ liệu? - Tất cả biến khai báo đều có được sử dụng? - Thời điểm ràng buộc (binding) của biến có phù hợp không – cân bằng giữa sự linh hoạt và độ phức tạp? - Mỗi biến có đúng một mục đích duy nhất không? - Ý nghĩa của mỗi biến có rõ ràng, không có ý nghĩa ẩn?

Điểm mấu chốt (Key Points)

- Khởi tạo dữ liệu dễ phát sinh lỗi, vì thế hãy sử dụng các kỹ thuật khởi tạo được trình bày để tránh các giá trị ban đầu bất ngờ.
- Tối thiểu hóa phạm vi của mỗi biến; giữ các tham chiếu tới biến gần nhau; ưu tiên giữ biến ở phạm vi nội bộ của chương trình con hoặc lớp; tránh sử dụng dữ liệu toàn cục.
- Giữ các câu lệnh thao tác trên cùng biến gần nhau nhất có thể.
- Gán giá trị sớm (early binding) giúp giảm độ phức tạp nhưng làm giảm linh hoạt; gán giá trị muộn (late binding) tăng tính linh hoạt nhưng phức tạp hơn.
- Mỗi biến chỉ nên phục vụ một mục đích duy nhất.

Chương 11: Sức mạnh của tên biến (The Power of Variable Names)

11.1 Lưu ý khi chọn tên biến (Considerations in Choosing Good Names)

Bạn không thể đặt tên biến như cách đặt tên cho chó vì nó "dễ thương" hoặc phát âm vui tai. Khác với con chó và cái tên là hai vật thể tách biệt nhau, biến và tên của nó trên thực tế là một. Vì thế, chất lượng của một biến phụ thuộc phần lớn vào tên của nó. Hãy chọn tên biến thật cẩn trọng.

Ví dụ: Tên biến tồi (Java Example of Poor Variable Names)

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

Bạn có hiểu đoạn mã này làm gì không? `x1`, `xx`, `xxx` nghĩa là gì? `fido` là gì? Nếu người khác nói với bạn rằng đoạn mã này tính tổng hóa đơn khách hàng dựa trên số dư nợ và khoản mua mới thì bạn sẽ dùng biến nào để in hóa đơn cho phần mua mới?

Phiên bản dùng tên biến tốt hơn

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

So sánh hai đoạn mã, một tên biến tốt là tên dễ đọc, dễ nhớ và phù hợp. Có thể sử dụng các quy tắc sau để đạt được điều đó:

Yếu tố quan trọng nhất Yếu tố cốt lõi nhất khi đặt tên biến là tên biến phải mô tả đầy đủ và chính xác thực thể mà nó đại diện. Một kỹ thuật hiệu quả là hãy diễn đạt bằng lời câu hỏi: "Biến này đại diện cho cái gì?" Nhiều khi câu trả lời đó sẽ là một tên biến tốt.

> **Điểm mấu chốt**: Tên biến không dùng ký hiệu khó hiểu, không mơ hồ, mô tả đầy đủ ý niệm đối tượng, giúp tránh nhầm lẫn và dễ nhớ nhờ sự tương ứng với khái niệm thực tế.

Ví dụ:

- Biến đại diện cho số người trong đội Olympic Mỹ có thể đặt là `numberOfPeopleOnTheUsOlympicTeam`.
- Số ghế trong sân vận động: `numberOfSeatsInTheStadium`.
- Số điểm cao nhất mà đội của một nước đạt được trong Olympics hiện đại: `maximumNumberOfPointsInModernOlympics`.
- Lãi suất hiện tại nên đặt là `interestRate` thay vì `r` hoặc `x`.

Các đặc điểm điển hình của tên biến tốt

- Dễ hiểu, không cần phải giải mã.
- Không quá dài; vấn đề chiều dài tên sẽ phân tích ở phần sau.

Ví dụ tên biến tốt/xấu

Chức năng biến	Tên tốt	Tên xấu
Tổng số séc đã viết đến hiện tại	runningTotal, checkTotalWritten	ct, checks, CHKTTL, x
Vận tốc viên đạn, tàu	velocity, trainVelocity, velocityInMph	velt, v, tv, x
Ngày hiện tại	currentDate, todaysDate	cd, c, date, x
Số dòng mỗi trang	linesPerPage	lpp, lines, l, x

"Tên biến như `currentDate` và `todaysDate` là tốt vì mô tả rõ nghĩa 'ngày hiện tại'. Tên biến như `cd`, `c`, `date`, `x`… là không tốt, vì hoặc quá ngắn, không rõ ràng, hoặc không mô tả đầy đủ bản chất dữ liệu."

Đặt tên theo *vấn đề* chứ không phải theo *giải pháp* (Problem Orientation)

Một tên tốt thường thể hiện *cái gì* (what) thay vì *cách làm* (how). Nên ưu tiên tên liên quan đến lĩnh vực vấn đề thay vì các khái niệm máy tính.

Ví dụ, bản ghi dữ liệu nhân viên nên đặt là `employeeData` thay vì `inputRec` (vì `inputRec` là từ ngữ thiên về máy tính). Tương tự, `printerReady` thì tốt hơn `bitFlag`. Trong ứng dụng kế toán, `sum` thì tốt hơn `calcVal`.

Độ dài tối ưu của tên biến (Optimum Name Length)

Tên quá ngắn sẽ không mang đủ ý nghĩa; quá dài lại khó gõ và gây rối mắt. Nghiên cứu cho thấy, thời gian sửa lỗi tối thiểu khi tên biến có chiều dài trung bình từ 10 đến 16 ký tự, và vẫn hiệu quả nếu từ 8 đến 20 ký tự. Không có nghĩa là mọi biến đều phải từ 9 tới 16 ký tự, nhưng nếu bạn thấy nhiều tên biến ngắn hơn, hãy xét lại.

Bảng ví dụ:

Quá dài	Quá ngắn	Vừa phải
numberOfPeopleOnTheUsOlympicTeam	n, np	numTeamMembers
numberOfSeatsInTheStadium	ns, nsisd	numSeatsInStadium
maximumNumberOfPointsInModernOlympics	mp, mpio	pointsRecord

Phạm vi ảnh hưởng của tên (Effect of Scope on Variable Names)

Tên biến ngắn không phải lúc nào cũng xấu. Đặt tên ngắn như `i` thường ngầm hiểu đây là biến tạm, phạm vi nhỏ (ví dụ, biến đếm trong vòng lặp). Nghiên cứu cũng chỉ ra rằng, biến ít dùng hoặc biến toàn cục nên dùng tên dài, biến cục bộ hoặc biến vòng lặp dùng tên ngắn.

Lưu ý bảo vệ: Tuy nhiên tên ngắn vẫn có nhiều bất lợi, nên một số lập trình viên chọn hạn chế dùng tên ngắn để tăng tính an toàn phòng lỗi.

Biệt hiệu/phân vùng cho tên toàn cục (Qualifiers on Global Namespace)

Với biến, class, hằng số toàn cục, hãy cân nhắc quy ước để phân vùng và tránh xung đột tên. Với C++/C#, có thể dùng từ khóa `namespace`. Ví dụ:

```
namespace UserInterfaceSubsystem {
    // nhiều khai báo
}
namespace DatabaseSubsystem {
    // nhiều khai báo
}
```

Nếu có hai class cùng tên `Employee` ở hai namespace trên, bạn xác định rõ bằng `UserInterfaceSubsystem::Employee` hoặc `DatabaseSubsystem::Employee`. Java hỗ trợ tính năng này qua package.

Với ngôn ngữ không hỗ trợ namespace hoặc package, dùng tiền tố để phân biệt. Ví dụ, class nhân viên giao diện nên là `uiEmployee`, class nhân viên của database nên là `dbEmployee`, giảm thiểu xung đột tên.

Quy ước định danh cho biến giá trị tính toán (Computed-Value Qualifiers in Variable Names)

Nên đặt phần mô tả chính ở đầu tên biến, thành tố toán học (Total, Sum, Average, Max…) ở cuối. Ví dụ: `revenueTotal`, `expenseAverage`. Lưu ý tạo sự nhất quán giúp dễ đọc, dễ bảo trì.

Ngoại lệ: Từ `Num` thường ở đầu tên biến nếu chỉ tổng số (vd: `numCustomers` = tổng số khách hàng), và ở cuối nếu chỉ vị trí/index (vd: `customerNum` = vị trí khách hàng hiện tại). Tuy nhiên, tốt nhất nên dùng `Count` cho tổng số (`customerCount`) và `Index` cho index (`customerIndex`) để tránh nhầm lẫn.

Tóm tắt

- Chỉ sử dụng biến cho một mục đích duy nhất.
- Đặt tên biến rõ ràng, biểu thị ý nghĩa thật sự, tránh các tên ngắn hoặc đa nghĩa.
- Tên biến nên ưu tiên thể hiện bản chất doanh nghiệp/vấn đề hơn là khía cạnh kỹ thuật.
- Quan tâm độ dài tên biến phù hợp với phạm vi sử dụng.
- Giữ miền ảnh hưởng của biến nhỏ nhất có thể.
- Đảm bảo sự nhất quán và có quy ước rõ ràng khi đặt tên để giúp mã nguồn dễ đọc, dễ bảo trì và giảm thiểu lỗi.*

Nếu có lỗi đánh máy hoặc định dạng ở bản gốc, đã được sửa trong bản dịch để đảm bảo rõ ràng, mạch lạc.

Sử dụng quy ước đặt tên cho các cặp đối lập giúp tăng tính nhất quán

Việc sử dụng quy ước đặt tên cho các cặp đối lập giúp duy trì tính nhất quán, qua đó cải thiện khả năng đọc mã nguồn. Những cặp tên như *begin/end* rất dễ hiểu và dễ nhớ. Tuy nhiên, các cặp tên không tương ứng với ngôn ngữ tự nhiên thường khó nhớ và dễ gây nhầm lẫn. Dưới đây là một số cặp đối lập thường dùng:

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

11.2 Đặt tên cho các kiểu dữ liệu cụ thể

Bên cạnh các nguyên tắc chung trong việc đặt tên dữ liệu, khi đặt tên cho các loại dữ liệu cụ thể thì cần có các lưu ý riêng. Phần này mô tả những lưu ý đặc biệt đối với: biến lặp (loop variables), biến trạng thái (status variables), biến tạm thời (temporary variables), biến boolean (boolean variables), kiểu liệt kê (enumerated types), và hằng số được đặt tên (named constants).

Đặt tên chỉ số lặp (Naming Loop Indexes)

> Tham khảo chi tiết hơn về vòng lặp tại Chương 16, "Controlling Loops".

Trong lập trình, các biến *i*, *j*, *k* thường được dùng làm chỉ số lặp:

Ví dụ Java - Đặt tên biến chỉ số lặp đơn giản

```
for (i = firstItem; i < lastItem; i++) {
    data[i] = 0;
}
```

Nếu một biến sẽ được sử dụng ngoài phạm vi vòng lặp, hãy đặt một tên mô tả ý nghĩa thay vì *i*, *j* hoặc *k*. Ví dụ, khi đọc các bản ghi từ tệp tin và cần lưu số lượng bản ghi đã đọc, một tên như *recordCount* sẽ phù hợp:

Ví dụ Java - Đặt tên biến mô tả trong vòng lặp

```
recordCount = 0;
while (moreScores()) {
    score[recordCount] = GetNextScore();
    recordCount++;
}
// các dòng sử dụng recordCount
```

Nếu đoạn vòng lặp dài hơn vài dòng, rất dễ quên ý nghĩa của *i*, do đó nên đặt tên rõ ràng hơn cho chỉ số lặp. Vì mã nguồn thường xuyên được thay đổi, mở rộng, và sao chép, nhiều lập trình viên giàu kinh nghiệm tránh sử dụng *i*.

Vòng lặp thường mở rộng thành vòng lặp lồng nhau (nested loops). Khi đó, nên đặt tên dài hơn cho các biến chỉ số để tăng khả năng đọc hiểu:

Ví dụ Java - Đặt tên biến chỉ số lặp trong vòng lặp lồng nhau

```
for (teamIndex = 0; teamIndex < teamCount; teamIndex++) {
    for (eventIndex = 0; eventIndex < eventCount[teamIndex]; eventIndex++) {
        score[teamIndex][eventIndex] = 0;
    }
}
```

Đặt tên biến chỉ số lặp cẩn thận giúp tránh lỗi phổ biến khi nhầm lẫn giữa các biến chỉ số, và cũng làm cho việc truy cập mảng rõ ràng hơn: *score[teamIndex][eventIndex]* rõ ràng hơn *score[i][j]*.

Nếu bắt buộc phải dùng *i*, *j*, *k*, chỉ nên dùng chúng cho chỉ số vòng lặp của các vòng đơn giản – quy ước này đã được thiết lập rộng rãi và sử dụng chúng cho mục đích khác sẽ gây nhầm lẫn. Cách đơn giản để tránh các vấn đề này là suy nghĩ và đặt tên mô tả hơn thay vì chỉ sử dụng *i*, *j*, *k*.

Đặt tên biến trạng thái (Naming Status Variables)

Biến trạng thái dùng để mô tả trạng thái của chương trình. Nguyên tắc đặt tên là:

> Hãy nghĩ ra tên tốt hơn thay vì chỉ dùng "flag" cho biến trạng thái. Đặt tên không nên chứa "flag" vì điều đó không cho biết chức năng thật sự của biến.

Để đảm bảo rõ nghĩa, các biến flag cần gán giá trị cụ thể và kiểm tra bằng kiểu liệt kê (enumerated types), hằng số được đặt tên (named constants), hoặc biến

toàn cục đóng vai trò như hằng số. Sau đây là ví dụ về những tên flag kém rõ ràng:

Ví dụ C++ - Các biến flag khó hiểu

```
if (flag)
if (statusFlag & 0x0F)
if (printFlag == 16)
if (computeFlag == 0)
flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

Những dòng như *statusFlag = 0x80* không cho biết ý nghĩa thực sự nếu không có tài liệu giải thích. Sau đây là ví dụ rõ ràng hơn:

Ví dụ C++ - Sử dụng biến trạng thái dễ hiểu

```
if (dataReady)
if (characterType & PRINTABLE_CHAR)
if (reportType == ReportType_Annual)
if (recalcNeeded == True)
dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Rõ ràng, *characterType = CONTROL_CHARACTER* có ý nghĩa hơn *statusFlag = 0x80*. Tương tự, điều kiện *if (reportType == ReportType_Annual)* rõ ràng hơn *if (printFlag == 16)*. Cách này cũng được sử dụng với kiểu liệt kê và hằng số đặt tên.

Khai báo biến trạng thái trong C++

```
// Giá trị cho CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
const int PRINTABLE_CHAR = (LETTER | DIGIT | PUNCTUATION | LINE_DRAW);
const int CONTROL_CHARACTER = 0x80;

// Giá trị cho ReportType
enum ReportType {
    ReportType_Daily,
    ReportType_Monthly,
    ReportType_Quarterly,
    ReportType_Annual,
```

```
    ReportType_All
};
```

Khi phải "đoán ý" về một đoạn mã, nên cân nhắc việc đổi tên biến cho rõ ràng. Lập trình viên nên có thể đọc được mã thay vì phải giải mã nó.

Đặt tên biến tạm thời (Naming Temporary Variables)

Biến tạm thời được dùng để lưu trữ giá trị tạm thời khi tính toán, hoặc dùng cho mục đích quản lý nội bộ (housekeeping). Thông thường, chúng được đặt tên như *temp*, *x* hoặc các tên không rõ ràng khác. Thực tế, việc sử dụng nhiều biến tạm thời là dấu hiệu cho thấy người lập trình chưa hiểu rõ vấn đề cần giải quyết. Việc coi thường các biến này cũng dễ dẫn đến lỗi.

> Hãy cẩn trọng khi sử dụng biến "tạm thời". Hầu hết các biến trong chương trình đều có tính chất tạm thời, nên nếu gọi riêng một số biến là "tạm thời" có thể cho thấy bạn chưa xác định được mục đích thực sự của chúng.

Ví dụ C++ - Đặt tên biến tạm thời không rõ ràng

```
// Tính nghiệm phương trình bậc hai
// Giả sử (b^2-4*a*c) là số dương
temp = sqrt(b^2 - 4*a*c);
root[0] = (-b + temp) / (2 * a);
root[1] = (-b - temp) / (2 * a);
```

Việc lưu giá trị của *sqrt(b^2 - 4ac)* vào một biến là hợp lý vì sẽ dùng lặp lại, nhưng tên *temp* không nói rõ vai trò. Dưới đây là một cách tốt hơn:

Ví dụ C++ - Đặt tên biến tạm thời mô tả rõ ràng

```
// Tính nghiệm phương trình bậc hai
// Giả sử (b^2-4*a*c) là số dương
discriminant = sqrt(b^2 - 4*a*c);
root[0] = (-b + discriminant) / (2 * a);
root[1] = (-b - discriminant) / (2 * a);
```

Mã tương tự, nhưng tên gọi rõ ràng, mô tả đầy đủ vai trò của biến.

Đặt tên biến boolean (Naming Boolean Variables)

Một số hướng dẫn khi đặt tên biến boolean như sau:

- **done**: Dùng để kiểm tra trạng thái hoàn thành. Đặt *done* là *false* trước khi hoàn thành, *true* khi đã hoàn tất.
- **error**: Dùng để nhận biết lỗi. Đặt *error* là *false* khi không có lỗi, *true* khi có lỗi.

- **found**: Dùng để xác định đã tìm thấy giá trị hay chưa. Dùng *found* khi tìm kiếm giá trị trong mảng, tệp, danh sách, v.v.
- **success** hoặc **ok**: Xác định một thao tác có thành công hay không. Đặt giá trị *false* khi thất bại, *true* khi thành công. Nếu có thể, nên dùng tên mô tả cụ thể hơn như *processingComplete* hoặc *found* tuỳ ngữ cảnh.

Đặt tên gợi ý trạng thái đúng/sai Các tên như *done*, *success* phù hợp cho biến boolean vì trạng thái rõ ràng (true/false). Các tên như *status* hoặc *sourceFile* khó hiểu khi kiểm tra giá trị boolean vì không rõ ý nghĩa trạng thái.

Ví dụ: *status* là true nghĩa là gì? Có thể là mọi thứ đều ổn, hoặc không có gì sai. Nên thay thế các tên mơ hồ như vậy bằng tên như *error*, *statusOK*, *sourceFileAvailable*, *sourceFileFound*, v.v.

Nhiều lập trình viên thêm tiền tố *Is* ở đầu tên boolean (ví dụ: *isDone*, *isError*, *isFound*, *isProcessingComplete*), biến tên thành một câu hỏi có thể trả lời true/false. Cách này không phù hợp với tên mơ hồ (*isStatus* không mang ý nghĩa). Tuy nhiên, nó có thể làm cho biểu thức logic bớt trực quan, ví dụ: *if (isFound)* kém rõ ràng hơn *if (found)* một chút.

Sử dụng tên boolean dạng khẳng định Tránh dùng tên phủ định như *notFound*, *notDone*, *notSuccessful* vì khi phủ định sẽ khó đọc, ví dụ:

> if not notFound

Thay vào đó, chuyển sang dạng khẳng định (*found*, *done*, *processingComplete*) và sử dụng phép toán phủ định khi cần.

Đặt tên cho kiểu liệt kê (Naming Enumerated Types)

> Xem chi tiết về *enumerated types* ở mục 12.6, "Enumerated Types".

Khi sử dụng *enumerated type* (kiểu liệt kê), nên đảm bảo các thành viên thuộc về cùng một nhóm, bằng cách sử dụng tiền tố nhóm như *Color_*, *Planet_*, hoặc *Month_*:

Ví dụ Visual Basic - Quy tắc đặt tên cho các thành viên của kiểu liệt kê

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum

Public Enum Planet
    Planet_Earth
    Planet_Mars
```

```
    Planet_Venus
End Enum

Public Enum Month
    Month_January
    Month_February
    '...
    Month_December
End Enum
```

Tên kiểu liệt kê (như *Color*, *Planet*, *Month*) cũng có thể được đặt bằng toàn bộ chữ in hoa hoặc thêm tiền tố (*e_Color*, *e_Planet*, *e_Month*). Một số ý kiến cho rằng kiểu liệt kê là kiểu do người dùng định nghĩa và nên đặt tên tương tự như class; ý kiến khác cho rằng nên đặt như constant (hằng số). Sách này sử dụng chữ hoa - thường xen kẽ (mixed case) cho tên kiểu liệt kê.

Ở một số ngôn ngữ, kiểu liệt kê được xử lý giống như class, và các thành viên của chúng phải có tiền tố tên lớp, do vậy có thể rút gọn tên thành *Color Red* thay vì *Color_Color_Red*.

Đặt tên hằng số (Naming Constants)

Xem chi tiết về hằng số đặt tên tại mục 12.7, “Named Constants”.

Khi đặt tên hằng số, nên đặt theo thực thể trừu tượng mà nó đại diện, không phải giá trị số. Ví dụ, *FIVE* là tên không phù hợp cho hằng số (dù giá trị của nó có là 5.0 hay không). *CYCLES_NEEDED* là tên phù hợp, vì giá trị có thể là 5.0 hoặc 6.0. Việc đặt *FIVE = 6.0* là vô nghĩa. Tương tự, tên hằng *BAKERS_DOZEN* không phù hợp; *DONUTS_MAX* là tên tốt hơn.

11.3 Sức mạnh của quy ước đặt tên

Một số lập trình viên ngần ngại với các quy ước và tiêu chuẩn, và điều này hoàn toàn có lý do. Một số tiêu chuẩn quá cứng nhắc, thiếu hiệu quả, làm giảm tính sáng tạo và chất lượng chương trình. Tuy vậy, các tiêu chuẩn hiệu quả là một trong những công cụ mạnh mẽ nhất mà bạn có thể sử dụng. Phần này giải thích lý do, thời điểm và cách thức nên tự xây dựng quy ước đặt tên biến.

Lý do cần có quy ước

Quy ước đặt tên mang lại một số lợi ích cụ thể:

- **Giảm số lượng quyết định nhỏ nhặt**: Quyết định toàn cục một lần tốt hơn nhiều quyết định nhỏ lẻ, giúp bạn tập trung hơn vào những vấn đề quan trọng.

- **Chia sẻ kiến thức giữa các dự án**: Tương đồng trong tên biến giúp hiểu nhanh ý nghĩa của các biến không quen thuộc.
- **Học mã nhanh hơn ở dự án mới**: Quy ước nhất quán giúp làm quen nhanh với mã nguồn, không bị rối bởi các phong cách đặt tên khác nhau của từng lập trình viên.
- **Giảm sự đa dạng không cần thiết trong tên biến**: Nếu không có quy ước, bạn dễ đặt nhiều tên khác nhau cho cùng một khái niệm, điều này gây khó khăn cho người đọc mã về sau.
- **Khắc phục các điểm yếu của ngôn ngữ lập trình**: Dùng quy ước để mô phỏng các hằng số đặt tên, kiểu liệt kê, phân biệt dữ liệu cục bộ, class, global; bổ sung thông tin kiểu cho loại dữ liệu mà trình biên dịch không hỗ trợ.
- **Nhấn mạnh mối quan hệ giữa các biến liên quan**: Nếu không có hướng đối tượng, quy ước đặt tên giúp nhóm các biến thuộc cùng một thực thể rõ ràng, ví dụ: *employeeAddress*, *employeePhone*, *employeeName* giúp nhận biết đây đều là thuộc tính của nhân viên.

  Điểm then chốt là: **Có quy ước nào đó luôn tốt hơn không có quy ước**. Quy ước có thể tùy ý, nhưng tính tổ chức giúp giảm nỗi lo cho người lập trình.

Khi nào nên xây dựng quy ước đặt tên

Không có quy tắc bắt buộc, nhưng có thể cân nhắc:

- Khi nhiều lập trình viên cùng thực hiện một dự án
- Khi dự định chuyển giao mã cho người khác bảo trì (thường xuyên xảy ra)
- Khi mã nguồn được các lập trình viên khác xem xét
- Khi dự án quá lớn để bạn có thể nắm toàn bộ trong đầu mà phải chia nhỏ để xử lý
- Khi chương trình sẽ được duy trì lâu dài, có thể bạn phải tạm bỏ nó rồi quay lại sau vài tuần/tháng
- Khi dự án có nhiều thuật ngữ chuyên ngành, cần tiêu chuẩn/viết tắt chung cho mã nguồn

Bạn luôn nhận được lợi ích từ quy ước đặt tên ở mức độ nhất định. Các lưu ý phía trên giúp xác định mức độ quy ước cần thiết cho dự án.

Cấp độ trang trọng của quy ước

> Tham khảo chi tiết về sự khác biệt trong quy ước giữa dự án nhỏ và lớn tại Chương 27, "How Program Size Affects Construction".

Mỗi quy ước có độ trang trọng khác nhau. Quy ước không chính thức có thể chỉ đơn giản là "đặt tên có ý nghĩa". Mức độ trang trọng phụ thuộc vào số lượng thành viên, kích thước dự án, và thời gian dự kiến sử dụng chương trình. Các dự án nhỏ, ngắn hạn không cần quy ước chặt chẽ; nhưng với dự án lớn, nhiều

người tham gia, quy ước chính thức là yếu tố không thể thiếu để đảm bảo khả năng đọc hiểu và bảo trì.

11.4 Quy ước đặt tên không chính thức (Informal Naming Conventions)

Hầu hết các dự án sử dụng quy ước đặt tên ở mức độ không quá nghiêm ngặt. Dưới đây là một số nguyên tắc tham khảo:

Nguyên tắc tạo quy ước độc lập ngôn ngữ

- **Phân biệt tên biến và tên hàm**: Quy ước trong sách này là tên biến và tên đối tượng bắt đầu bằng chữ thường, tên hàm thủ tục bắt đầu bằng chữ hoa, ví dụ: *variableName* so với *RoutineName()*
- **Phân biệt class và object**: Quan hệ giữa tên class và tên object (hoặc giữa type và biến kiểu đó) có thể phức tạp. Sau đây là một số cách phổ biến:

Lựa chọn 1: Dùng chữ hoa/c thường ở ký tự đầu

```
Widget widget;
LongerWidget longerWidget;
```

Lựa chọn 2: Dùng toàn bộ chữ hoa cho type

```
WIDGET widget;
LONGERWIDGET longerWidget;
```

Lựa chọn 3: Thêm tiền tố "t_" cho type

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

Lựa chọn 4: Thêm tiền tố "a" cho biến

```
Widget aWidget;
LongerWidget aLongerWidget;
```

Lựa chọn 5: Dùng tên biến cụ thể hơn

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

Mỗi lựa chọn đều có ưu nhược riêng. Lựa chọn 1 phổ biến trong ngôn ngữ phân biệt hoa-thường như C++ và Java, nhưng một số người không thích chỉ khác biệt ở ký tự đầu tiên vì không đủ khác biệt về mặt tâm lý cũng như trực quan.

Lựa chọn 1 cũng không áp dụng được trong môi trường đa ngôn ngữ mà có ngôn ngữ không phân biệt hoa-thường, ví dụ Visual Basic coi *widget* và *Widget* là một.

Lựa chọn 2 tạo khác biệt rõ hơn, nhưng trong C++/Java, tên constant thường toàn bộ chữ hoa, và nó cũng chịu nhược điểm về môi trường không phân biệt hoa-thường như trên.

Lựa chọn 3 phù hợp với mọi ngôn ngữ, nhưng một số người không thích thêm tiền tố.

Lựa chọn 4 cũng có thể áp dụng, nhưng phải đổi tên tất cả instance thay chỉ đổi một tên class.

Lựa chọn 5, buộc nghĩ đến tên cụ thể cho biến, giúp mã dễ đọc hơn. Tuy nhiên, đôi khi biến thực sự chỉ là widget chung chung, nếu phải nghĩ ra tên như *genericWidget* sẽ giảm khả năng hiểu.

Tóm lại, mỗi lựa chọn đều có đánh đổi. Sách này chọn lựa chọn 5 vì nó dễ hiểu nhất trong trường hợp người đọc mã không quen với các quy ước kém trực quan.

- **Phân biệt biến toàn cục**: Một vấn đề thường gặp là sử dụng sai biến toàn cục. Nếu tất cả biến toàn cục bắt đầu với tiền tố *g_*, ví dụ *g_RunningTotal*, lập trình viên sẽ biết đây là biến toàn cục và xử lý cẩn thận hơn.
- **Phân biệt biến thành viên class (member variables)**: (Đoạn mẫu dừng ở đây, phần tiếp theo sẽ hướng dẫn cách đặt tên biến thành viên của class.)

Ví dụ về Quy ước đặt tên biến và thực thể trong lập trình

Nhận diện biến thành viên lớp (class member variables)

Ví dụ, bạn có thể nhận diện biến thành viên của lớp bằng cách thêm tiền tố `m_` để chỉ ra đó là dữ liệu thành viên (member data).

Nhận diện định nghĩa kiểu dữ liệu (type definitions)

Quy ước đặt tên cho kiểu dữ liệu (type) phục vụ hai mục đích: - Rõ ràng xác định một tên là tên kiểu dữ liệu. - Tránh bị trùng lặp tên với các biến.

Để đáp ứng hai yêu cầu này, sử dụng tiền tố (prefix) hoặc hậu tố (suffix) là một phương pháp tốt. Trong C++, cách làm phổ biến là sử dụng toàn bộ chữ hoa cho tên kiểu—for example, `COLOR` và `MENU`. (Quy ước này áp dụng cho typedefs và structs, không áp dụng cho tên lớp).

Tuy nhiên, điều này có thể gây nhầm lẫn với các hằng số được định nghĩa bởi preprocessor. Để tránh nhầm lẫn, bạn có thể thêm tiền tố `t_` cho tên kiểu, chẳng hạn như `t_Color` và `t_Menu`.

Nhận diện hằng số đã đặt tên (named constants)

Hằng số đã đặt tên cần được nhận diện rõ ràng để bạn có thể phân biệt việc gán giá trị cho một biến thông qua một biến khác (giá trị có thể thay đổi) hoặc thông qua một hằng số đã đặt tên. Trong Visual Basic, còn có khả năng giá trị được lấy từ một hàm. Visual Basic không yêu cầu tên hàm phải có dấu ngoặc đơn, trong khi C++ yêu cầu kể cả với hàm không tham số.

Một phương pháp đặt tên hằng số là sử dụng tiền tố như `c_` cho tên hằng số. Điều này sẽ tạo ra các tên như `c_RecsMax` hoặc `c_LinesPerPageMax`. Trong C++ và Java, quy ước là dùng toàn bộ chữ hoa, có thể phân tách từ bằng dấu gạch dưới (_), ví dụ: `RECSMAX` hoặc `RECS_MAX`, `LINESPERPAGEMAX` hoặc `LINES_PER_PAGE_MAX`.

Nhận diện các phần tử của kiểu liệt kê (enumerated types)

Các phần tử của kiểu liệt kê cũng cần được nhận diện, tương tự như hằng số đã đặt tên, nhằm giúp phân biệt giữa tên kiểu liệt kê với biến, hằng số hoặc hàm. Phương pháp tiêu chuẩn là sử dụng toàn bộ chữ hoa, hoặc thêm tiền tố kiểu `e_` hoặc `E_` cho bản thân tên kiểu, và sử dụng tiền tố dựa trên kiểu cụ thể như `Color_` hoặc `Planet_` cho các thành viên.

Nhận diện tham số chỉ nhập (input-only parameters) trong các ngôn ngữ không bắt buộc quy định rõ

Đôi khi các tham số đầu vào (input parameters) vô tình bị thay đổi. Trong các ngôn ngữ như C++ và Visual Basic, bạn cần chỉ rõ liệu bạn muốn trả về giá trị đã thay đổi cho chương trình gọi hàm hay không, được xác định bởi các từ khóa như `*`, `&`, và `const` trong C++ hoặc `ByRef` và `ByVal` trong Visual Basic.

Ở các ngôn ngữ khác, nếu bạn sửa đổi biến đầu vào, nó sẽ được trả về dù bạn có muốn hay không, đặc biệt đúng với việc truyền đối tượng. Ví dụ, trong Java, mọi đối tượng đều được truyền "by value", nên khi bạn truyền một đối tượng đến một hàm, nội dung của đối tượng có thể bị thay đổi trong hàm đó (Arnold, Gosling, Holmes 2000).

> **Tham khảo bổ sung:** Việc thiết lập quy ước đặt tên để chỉ rõ tham số chỉ nhập (input-only parameter), ví dụ bằng tiền tố `const` (hoặc `final`, `nonmodifiable`, hoặc tương tự), giúp bạn phát hiện lỗi nếu thấy bất cứ biến nào có tiền tố `const` nằm ở phía bên trái dấu bằng. Nếu xuất hiện `constMax SetNewMax()`, bạn sẽ biết rằng đây là sai sót vì tiền tố `const` chỉ ra biến không được phép sửa đổi.

Định dạng tên để tăng khả năng đọc

Hai kỹ thuật phổ biến để tăng khả năng đọc là sử dụng chữ hoa thường hợp lý và ký tự phân tách từ (chẳng hạn dấu gạch dưới _). Ví dụ: `GYMNASTICSPOINTTOTAL` ít dễ đọc hơn so với `gymnasticsPointTotal` hoặc `gymnastics_point_total`. Các ngôn ngữ như C++, Java, Visual Basic đều hỗ trợ việc sử dụng chữ hoa lẫn chữ thường và dấu gạch dưới.

Không nên trộn lẫn nhiều kỹ thuật này, vì điều đó sẽ làm mã nguồn khó đọc. Tuy nhiên, nỗ lực sử dụng nhất quán bất kỳ kỹ thuật nào sẽ cải thiện mã nguồn của bạn. Nhiều lập trình viên tranh luận về chi tiết như liệu ký tự đầu tiên trong tên nên viết hoa (`TotalPoints`) hay viết thường (`totalPoints`), nhưng miễn là bạn và nhóm sử dụng nhất quán thì điều đó không ảnh hưởng nhiều. Sách này sử dụng chữ thường ở đầu tiên, do quy ước thực hành mạnh mẽ từ Java và để đồng nhất với nhiều ngôn ngữ khác.

Hướng dẫn quy ước đặt tên theo từng ngôn ngữ

Quy ước của ngôn ngữ C

Một số quy ước đặt tên dành riêng cho ngôn ngữ lập trình C:

- `c` và `ch` dùng cho biến ký tự (character variables).
- `i` và `j` là chỉ số nguyên (integer indexes).
- `n` là số lượng.
- `p` là con trỏ (pointer).
- `s` là chuỗi (string).
- Các macro tiền xử lý viết IN HOA TOÀN BỘ (ALL_CAPS). Quy ước này thường mở rộng cho typedefs.
- Tên biến và hàm viết chữ thường toàn bộ.
- Sử dụng dấu gạch dưới (_) để phân tách: `letters_in_lowercase` dễ đọc hơn `lettersinlowercase`.

Các quy ước này phổ biến với phong cách lập trình UN*X và Linux, tuy nhiên ở Windows và Macintosh có thể áp dụng quy tắc khác.

Quy ước của ngôn ngữ C++

Các quy ước tiêu biểu trong lập trình C++ bao gồm:

- `i` và `j` dùng cho chỉ số nguyên (integer indexes).
- `p` là con trỏ (pointer).
- Các hằng số, typedefs và macro tiền xử lý viết IN HOA TOÀN BỘ (ALL_CAPS).
- Tên lớp (class) và kiểu (type) dùng viết hoa thường lẫn nhau, chữ cái đầu mỗi từ viết hoa.

- Tên biến và hàm bắt đầu bằng viết thường, từ tiếp theo viết hoa chữ cái đầu (ví dụ: `variableOrRoutineName`).
- Dấu gạch dưới không được dùng trong tên trừ khi là tên viết hoa toàn bộ hoặc một số dạng tiền tố đặc biệt.

Quy ước của ngôn ngữ Java

Trái ngược với C và C++, quy ước của Java đã được chuẩn hóa từ đầu:

- `i` và `j` là chỉ số nguyên.
- Hằng số viết IN HOA TOÀN BỘ, phân tách các từ bằng dấu gạch dưới.
- Tên lớp (class) và giao diện (interface) viết hoa chữ cái đầu mỗi từ, kể cả từ đầu tiên, ví dụ: `ClassOrInterfaceName`.
- Tên biến và phương thức (method) dùng chữ thường ở từ đầu tiên, các từ sau viết hoa chữ cái đầu, ví dụ: `variableOrRoutineName`.
- Không dùng dấu gạch dưới để phân tách từ trong tên, trừ trường hợp tên viết hoa toàn bộ.
- `get` và `set` được dùng làm tiền tố cho các phương thức truy xuất (accessor methods).

Quy ước của Visual Basic

Visual Basic không có quy ước đặt tên chính thức, nhưng có thể tham khảo bảng hướng dẫn sau:

Thực thể	Mô tả
C_ClassName	Tên lớp (class), viết hoa thường lẫn nhau, tiền tố `C_`.
T_TypeName	Định nghĩa kiểu (type definition), kể cả kiểu liệt kê và typedefs, tiền tố `T_`.
T_EnumeratedTypes	Tên kiểu liệt kê, luôn dùng dạng số nhiều.
localVariable	Biến cục bộ, viết hoa thường lẫn nhau, chữ cái đầu thường.
routineParameter	Tham số hàm, giống quy tắc biến cục bộ.

Thực thể	Mô tả
RoutineName()	Tên hàm, viết hoa thường lẫn nhau.
m_ClassVariable	Biến thành viên lớp, tiền tố `m_`.
g_GlobalVariable	Biến toàn cục, tiền tố `g_`.
CONSTANT	Hằng số, IN HOA TOÀN BỘ.
Base_EnumeratedType	Thành viên kiểu liệt kê, tiền tố là dạng viết tắt ý nghĩa của kiểu ở dạng số ít, ví dụ: `Color_Red`.

Cân nhắc khi lập trình đa ngôn ngữ

Khi lập trình trong môi trường đa ngôn ngữ, quy ước đặt tên (cũng như quy tắc định dạng, tài liệu…) nên tối ưu cho tính nhất quán và dễ đọc tổng thể, kể cả khi điều đó có thể khác với quy ước riêng của từng ngôn ngữ. Ví dụ, trong cuốn sách này, tên biến luôn bắt đầu bằng chữ thường (theo quy tắc phổ biến ở Java), và tên hàm bắt đầu bằng chữ hoa (theo quy tắc của C++), để đảm bảo phong cách nhất quán xuyên suốt.

Quy ước đặt tên mẫu

Dưới đây là các ví dụ quy ước đặt tên cho C, C++, Java và Visual Basic, đã được rút gọn từ các hướng dẫn trước:

Bảng 11-3: Quy ước đặt tên mẫu cho C++ và Java

Thực thể	Mô tả
ClassName	Tên lớp, viết hoa thường lẫn nhau, chữ cái đầu hoa.

Thực thể	Mô tả
TypeName	Định nghĩa kiểu (including enum, typedefs), viết hoa thường lẫn nhau, chữ cái đầu hoa.
EnumeratedTypes	Tên kiểu liệt kê, dạng số nhiều.
localVariable	Biến cục bộ, chữ cái đầu thường, viết hoa chữ cái đầu từ tiếp theo.
routineParameter	Tham số hàm, giống biến cục bộ.
RoutineName()	Tên hàm, viết hoa thường lẫn nhau.
m_ClassVariable	Biến thành viên lớp, tiền tố `m_`.
g_GlobalVariable	Biến toàn cục, tiền tố `g_`.
CONSTANT	Hằng số, IN HOA TOÀN BỘ.
MACRO	Macro, IN HOA TOÀN BỘ.
Base_EnumeratedType	Thành viên kiểu liệt kê, tiền tố dạng viết tắt của kiểu, số ít, ví dụ `Color_Red`.

Bảng 11-4: Quy ước đặt tên mẫu cho C

Thực thể	Mô tả
TypeName	Định nghĩa kiểu, viết hoa chữ cái đầu.
GlobalRoutineName()	Hàm công khai, viết hoa thường lẫn nhau.
f_FileRoutineName()	Hàm nội bộ, tiền tố `f_`.

Thực thể	Mô tả
LocalVariable	Biến cục bộ, viết hoa thường lẫn nhau.
RoutineParameter	Tham số hàm, giống biến cục bộ.
f_FileStaticVariable	Biến cục bộ module, tiền tố `f_`.
G_GLOBAL_GlobalVariable	Biến toàn cục, tiền tố `G_` và từ viết tắt module.
LOCAL_CONSTANT	Hằng số nội bộ, IN HOA TOÀN BỘ.
G_GLOBALCONSTANT	Hằng số toàn cục, tiền tố `G_` và từ viết tắt module.
LOCALMACRO()	Macro nội bộ, IN HOA TOÀN BỘ.
G_GLOBAL_MACRO()	Macro toàn cục, tiền tố `G_` và từ viết tắt module.

Tiền tố chuẩn hóa (Standardized Prefixes)

Việc chuẩn hóa tiền tố mang lại một phương pháp đặt tên ngắn gọn nhưng nhất quán và dễ đọc cho dữ liệu. Quy ước nổi tiếng nhất là Hungarian naming convention được dùng rộng rãi một thời trong lập trình Microsoft Windows. Mặc dù không còn phổ biến, ý tưởng sử dụng các viết tắt chính xác vẫn còn giá trị.

Tiền tố chuẩn hóa gồm hai phần: - Viết tắt kiểu do người dùng định nghĩa (UDT - User-Defined Type). - Tiền tố ý nghĩa (semantic prefix).

Viết tắt kiểu do người dùng định nghĩa (User-Defined Type Abbreviations)

Viết tắt UDT giúp nhận diện kiểu dữ liệu của đối tượng hoặc biến. Thông thường, UDT không chỉ định các kiểu dữ liệu có sẵn mà mô tả các thực thể trong chương trình của bạn (ví dụ: cửa sổ, vùng màn hình, văn bản…). Ví dụ:

Viết tắt	Ý nghĩa
ch	Ký tự (character)
doc	Tài liệu (document)
pa	Đoạn văn (paragraph)
scr	Vùng màn hình (screen region)
sel	Phần chọn (selection)
wn	Cửa sổ (window)

Ví dụ khai báo:

```
CH chCursorPosition;
SCR scrUserWorkspace;
DOC docActive;
PA firstPaActiveDocument;
PA lastPaActiveDocument;
WN wnMain;
```

Tiền tố ý nghĩa (Semantic Prefixes)

Tiền tố ý nghĩa mô tả thêm cách sử dụng của biến hoặc đối tượng. Khác với UDTs, tiền tố ý nghĩa có tính thống nhất hơn giữa các dự án. Ví dụ:

Tiền tố	Ý nghĩa
c	Đếm số lượng (count)
first	Phần tử đầu tiên cần xử lý (first element)
g	Biến toàn cục (global variable)
i	Chỉ số (index)
last	Phần tử cuối cùng cần xử lý (last element)
lim	Giới hạn trên (upper limit, không bao gồm phần tử này)
m	Biến thành viên lớp (class-level variable)
max	Phần tử cuối cùng thực sự của mảng (absolute last element)
min	Phần tử đầu tiên thực sự của mảng (absolute first element)
p	Con trỏ (pointer)

Khi kết hợp với UDT: ví dụ, `firstPa` là đoạn văn đầu tiên, `iPa` là chỉ số, `cPa` là số lượng đoạn văn, … Điều này giúp biểu diễn ngắn gọn và chính xác ý nghĩa của biến.

Lưu ý: Điểm hạn chế của phương pháp này là nhiều khi lập trình viên chỉ sử dụng mỗi tiền tố mà quên đặt tên mô tả đầy đủ cho biến, ví dụ `ipa` không nói rõ ràng như `ipaActiveDocument`. Để mã nguồn dễ đọc, hãy dùng tên mô tả đầy đủ.

Tạo tên ngắn nhưng vẫn dễ đọc

Việc dùng tên biến ngắn là tàn dư của các ngôn ngữ lập trình cũ với giới hạn độ dài tên. Ngày nay, các ngôn ngữ hiện đại như C++, Java, Visual Basic cho phép đặt tên dài, bạn hầu như không có lý do để viết tắt tên biến một cách không rõ nghĩa.

Nếu vẫn phải đặt tên ngắn, hãy loại bỏ từ không cần thiết, sử dụng từ đồng nghĩa ngắn, hoặc áp dụng các phương pháp viết tắt hợp lý, đảm bảo tên biến vẫn thu hút và dễ hiểu.

> **Lưu ý**: Toàn bộ bản dịch đã được kiểm tra để đảm bảo tính nhất quán và chính xác về mặt thuật ngữ và văn phong học thuật, trang trọng. Code được giữ nguyên và trình bày đúng theo yêu cầu.

Dịch Thuật: Các Quy tắc Đặt Tên Biến Ngắn Gọn và Dễ Đọc

Các nguyên tắc viết tắt tên biến

- **Sử dụng các dạng viết tắt tiêu chuẩn** (những dạng viết tắt được sử dụng phổ biến và có trong từ điển).
- **Loại bỏ tất cả các nguyên âm không đứng đầu** (ví dụ: "computer" thành "cmptr", "screen" thành "scrn", "apple" thành "appl", "integer" thành "intgr").
- **Loại bỏ các mạo từ**: and, or, the, v.v.
- **Sử dụng chữ cái đầu hoặc một vài chữ cái đầu của mỗi từ**.
- **Cắt ngắn đồng nhất sau chữ cái thứ nhất, thứ hai, hoặc thứ ba của mỗi từ (tùy trường hợp phù hợp)**.
- **Giữ lại chữ cái đầu và cuối của mỗi từ**.
- **Sử dụng tất cả các từ quan trọng trong tên, tối đa là ba từ**.
- **Loại bỏ các hậu tố không cần thiết**—ing, ed, v.v.
- **Giữ lại âm nổi bật nhất trong mỗi âm tiết**.
- **Bảo đảm không làm thay đổi ý nghĩa của biến**.
- **Lặp lại các kỹ thuật trên cho đến khi viết tắt mỗi tên biến còn từ 8 đến 20 ký tự, hoặc theo giới hạn số ký tự của ngôn ngữ lập trình bạn sử dụng**.

Viết Tắt Theo Âm Đọc (Phonetic Abbreviations)

Một số người đề xuất viết tắt dựa trên âm đọc thay vì chính tả. Ví dụ, "skating" thành "sk8ing", "highlight" thành "hilite", "before" thành "b4", "execute" thành "xqt", v.v. Tuy nhiên, phương pháp này tương tự như việc giải mã biển số xe cá nhân hóa và **không được khuyến nghị sử dụng**.

Bài tập: Hãy thử đoán ý nghĩa của các tên sau: ILV2SK8, XME-QWK, S2DTM8O, NXTC, TRMN8R

Nhận Xét Về Việc Viết Tắt

Bạn có thể gặp một số rắc rối khi tạo các tên viết tắt. Sau đây là các quy tắc để tránh các bẫy phổ biến:

- **Không rút gọn bằng cách chỉ loại bỏ một ký tự khỏi từ**. Việc gõ thêm một ký tự không tốn nhiều công sức, nhưng việc loại bỏ một ký tự lại làm giảm đáng kể khả năng đọc hiểu. Ví dụ, viết "Jun" thay vì "June" không tiết kiệm được gì đáng kể. Hãy loại bỏ nhiều hơn một ký tự, hoặc giữ nguyên từ.
- **Viết tắt một cách nhất quán**. Luôn sử dụng cùng một dạng viết tắt cho một từ. Ví dụ, chọn "Num" hoặc "No" cho toàn bộ dự án, không dùng cả hai. Tương tự, đừng viết tắt từ trong một số tên mà giữ nguyên ở chỗ khác.
- **Tạo các tên có thể phát âm được**. Sử dụng "xPos" thay vì "xPstn", hoặc "needsComp" thay vì "ndsCmptg". Áp dụng "kiểm tra qua điện thoại"—nếu bạn không thể đọc tên biến qua điện thoại cho người khác hiểu, hãy đổi tên biến cho dễ hiểu hơn (theo Kernighan và Plauger, 1978).
- **Tránh các tổ hợp dễ gây hiểu nhầm khi đọc hoặc phát âm**. Ví dụ, nên dùng "ENDB" thay vì "BEND" để chỉ kết thúc của B. Nếu bạn sử dụng kỹ thuật phân tách tốt ("B-END", "BEnd" hoặc "b_end"), bạn sẽ tránh được lỗi này.
- **Sử dụng từ đồng nghĩa để tránh trùng tên viết tắt**. Khi viết tắt các tên, đôi khi bạn có thể gặp xung đột (ví dụ: "fired" và "full revenue disbursal" đều thành "frd" nếu giới hạn 3 ký tự). Có thể sử dụng từ thay thế, như "dismissed" thay cho "fired", hoặc "complete revenue disbursal" thay cho "full revenue disbursal" thành "dsm" và "crd".

Ví dụ về bảng dịch tên biến chuẩn trong Fortran

```
C ****************************************************************
C Translation Table
C
C Variable   Meaning
C --------   -------
C XPOS       x-Coordinate Position (in meters)
C YPOS       Y-Coordinate Position (in meters)
C NDSCMP     Needs Computing (=0 if no computation is needed;
C                     =1 if computation is needed)
C PTGTTL     Point Grand Total
C PTVLMX     Point Value Maximum
C PSCRMX     Possible Score Maximum
C ****************************************************************
```

Bạn có thể nghĩ rằng phương pháp này đã lỗi thời, nhưng cho đến những năm 2003, tôi vẫn làm việc với khách hàng sử dụng RPG với giới hạn tên biến chỉ 6 ký tự. Vấn đề này vẫn tồn tại cho đến ngày nay.

Tài liệu hóa các tên viết tắt

Lưu ý: Việc viết tắt trong mã nguồn gây ra hai rủi ro:

- Người đọc có thể không hiểu ý nghĩa viết tắt.
- Các lập trình viên khác có thể sử dụng nhiều cách viết tắt khác nhau cho cùng một từ, gây khó hiểu.

Để giải quyết cả hai vấn đề trên, nên tạo một tài liệu "Standard Abbreviations" (Danh sách viết tắt chuẩn) ở cấp độ dự án. Tên tài liệu có thể là tài liệu văn bản, bảng tính, hoặc thậm chí là cơ sở dữ liệu đối với các dự án lớn và luôn được quản lý trên hệ thống kiểm soát phiên bản.

Tài liệu này giúp:

- **Ghi nhận tất cả các dạng viết tắt đang sử dụng**—bất kỳ ai muốn tạo viết tắt mới phải cập nhật tại đây, giảm khả năng xuất hiện các dạng viết tắt trùng lặp.
- **Tăng tính thống nhất**—lập trình viên sẽ kiểm tra tài liệu trước khi tạo viết tắt mới và sử dụng dạng đã được chuẩn hóa nếu có.

Bạn cần cân nhắc giữa sự thuận tiện khi viết và dễ đọc khi đọc mã nguồn. Thực tế, các lập trình viên dành nhiều thời gian đọc mã hơn là viết mã, vì vậy hãy ưu tiên sự thuận tiện khi đọc.

Các loại tên nên tránh

- **Tránh những tên gây hiểu nhầm hoặc viết tắt dễ nhầm lẫn**. Ví dụ, sử dụng "FALSE" làm viết tắt cho "Fig and Almond Season" sẽ gây hiểu nhầm vì thường "FALSE" là đối lập với "TRUE".
- **Tránh các tên có ý nghĩa gần giống nhau**. Nếu bạn có thể hoán đổi tên hai biến mà không làm ảnh hưởng đến chương trình, hãy đổi lại tên các biến đó cho rõ ràng hơn. Ví dụ: "input" và "inputValue", "recordNum" và "numRecords", "fileNumber" và "fileIndex".
- **Tránh các tên gần giống nhau về mặt ký tự, nhưng ý nghĩa khác biệt**. Ví dụ: "clientRecs" (client records) và "clientReps" (client representatives)—chỉ khác một ký tự và khó nhận diện. Nên sử dụng các tên như "clientRecords" và "clientReports" để rõ ràng hơn.
- **Tránh các tên dễ gây nhầm lẫn về phát âm**. Ví dụ: "wrap" và "rap" có thể nhầm khi nói chuyện về mã nguồn. Hoặc "Goal Donor" và "Gold Owner" rất dễ bị lẫn khi giao tiếp.
- **Tránh đặt số trong tên biến**. Nếu giá trị số thật sự quan trọng, hãy dùng array, nếu không thì hạn chế dùng tên như "file1", "file2", "total1",

"total2". Trong một số trường hợp thực tế (ví dụ: Route 66), việc dùng số là hợp lý, nhưng nên xem xét các giải pháp đặt tên thay thế trước.

- **Tránh các tên có lỗi chính tả**. Việc yêu cầu những người khác nhớ "cách viết sai chính tả đúng" là quá khó—ví dụ, "hilite" thay vì "highlight" dễ khiến người đọc không nhớ viết tắt gốc là gì.
- **Tránh sử dụng các từ tiếng Anh phổ biến dễ sai chính tả**. Ví dụ: absense, acummulate, acsend, calender, concieve, defferred, … (xem các danh sách trong sách tiếng Anh chuẩn).
- **Đừng phân biệt tên biến chỉ dựa vào chữ hoa/thường** trong ngôn ngữ phân biệt hoa-thường như C++. Dùng "frd" cho "fired", "FRD" cho "final review duty", "Frd" cho "full revenue disbursal" sẽ gây nhầm lẫn.
- **Tránh sử dụng nhiều ngôn ngữ tự nhiên trong cùng một dự án**. Áp dụng một loại tiếng Anh duy nhất (nếu dự án đa quốc gia) để tránh lẫn lộn "color"/"colour", "check"/"cheque"…
- **Tránh sử dụng tên của kiểu dữ liệu chuẩn, biến chuẩn, hàm chuẩn của ngôn ngữ**. Đọc kỹ tài liệu hướng dẫn của ngôn ngữ để tránh trùng tên.
- **Tránh tên không liên quan đến ý nghĩa biến**. Đừng đặt tên như "margaret" hay "pookie" vào các vị trí không liên quan.
- **Tránh sử dụng các ký tự khó nhận biết**. Một số ký tự như (1 và l), (1 và I), (, và .), (0 và O), (2 và Z), (; và :), (S và 5), (G và 6) có thể dễ gây nhầm lẫn khi quan sát mã nguồn. > Ví dụ: eyeChartl, eyeChartI, eyeChart1; TTLCONFUSION, TTLC0NFUSION; hard2Read, hardZRead; GRANDTOTAL, 6RANDTOTAL; ttl5, ttlS.

Lưu ý: Một lỗi nhỏ như thế có thể gây hậu quả lớn (như thêm dấu phẩy thay vì dấu chấm trong FORMAT của Fortran từng làm mất một tàu vũ trụ trị giá 1,6 tỷ đô la).

Checklist: Đặt Tên Biến

Các yếu tố cần cân nhắc chung

- Tên có mô tả đầy đủ và chính xác ý nghĩa của biến không?
- Tên có liên hệ với bài toán thực tế hay giải pháp của ngôn ngữ lập trình?
- Độ dài tên đủ để không gây băn khoăn khi đọc?
- Các "qualifier" giá trị tính toán (nếu có) đặt ở cuối tên?
- Nên dùng hậu tố "Count" hoặc "Index" thay cho "Num"?

Đặt tên cho từng loại dữ liệu

- Tên index của vòng lặp có ý nghĩa rõ ràng (không chỉ đơn thuần là "i", "j", "k" với các vòng lặp dài hoặc lồng nhau)?
- Biến "tạm" đã được đặt lại tên cho có ý nghĩa hơn chưa?
- Các biến boolean có tên rõ nghĩa khi giá trị true?
- Tên kiểu liệt kê (enumerated type) có chứa prefix/suffix chỉ loại không? (vd: "Color_Red", "Color_Green", "Color_Blue")

- Hằng số được đặt tên theo bản chất, không phải theo giá trị?

Quy ước đặt tên

- Quy ước có phân biệt biến cục bộ, biến lớp, biến toàn cục không?
- Có phân biệt giữa kiểu dữ liệu, hằng số, kiểu liệt kê, biến không?
- Cho biết rõ tham số chỉ vào (input-only) ở các ngôn ngữ không bắt buộc khai báo?
- Tối đa tuân thủ các quy tắc chuẩn của ngôn ngữ?
- Định dạng tên biến rõ ràng?

Các tên ngắn

- Có sử dụng tên đủ dài trừ khi bắt buộc dùng tên ngắn?
- Tránh các dạng viết tắt chỉ tiết kiệm một ký tự?
- Viết tắt các từ một cách nhất quán?
- Tên phát âm được?
- Tránh tên ngắn dễ gây hiểu nhầm?
- Có lập bảng dịch tên ngắn (translation table) cho tên khó hiểu?

Vấn đề đặt tên thường gặp: Đã Tránh

- tên gây hiểu nhầm?
- tên có ý nghĩa gần giống?
- tên chỉ khác nhau 1-2 ký tự?
- tên phát âm gần giống nhau?
- tên có chứa số?
- tên cố tình viết sai chính tả cho ngắn?
- tên là từ trong tiếng Anh thường sai chính tả?
- tên trùng với tên hàm hoặc biến chuẩn của thư viện/ngôn ngữ?
- tên hoàn toàn ngẫu nhiên?
- ký tự khó đọc hoặc dễ nhầm lẫn?

Các ý chính (Key Points)

- **Tên biến tốt là yếu tố then chốt tạo nên khả năng đọc hiểu mã nguồn**. Đặc biệt với các loại biến như biến index của vòng lặp và biến trạng thái.
- **Tên nên càng cụ thể càng tốt**. Tên mơ hồ/thường dùng cho nhiều mục đích là tên tồi.
- **Có quy ước đặt tên phân biệt rõ các biến cục bộ, lớp, toàn cục, kiểu, hằng số, kiểu liệt kê**.
- **Dù dự án ở quy mô nào, hãy xây dựng quy ước đặt tên cho biến**. Quy ước tùy thuộc vào quy mô chương trình và số lượng thành viên dự án.

- **Hầu như không cần viết tắt trong ngôn ngữ lập trình hiện đại**. Nếu cần, hãy ghi lại viết tắt trong từ điển dự án hoặc sử dụng tiền tố chuẩn hóa.
- **Mã nguồn được đọc nhiều hơn viết**, nâng cao tính dễ đọc (read-time convenience), quan trọng hơn dễ viết (write-time convenience).

Chương 12: Kiểu Dữ Liệu Cơ Bản (Fundamental Data Types)

Mục Lục

Chủ đề liên quan

- Đặt tên dữ liệu: Chương 11
- Kiểu dữ liệu không phổ biến: Chương 13
- Vấn đề chung về biến: Chương 10
- Định dạng khai báo dữ liệu: “Laying Out Data Declarations” tại Mục 31.5
- Ghi chú biến: “Commenting Data Declarations” tại Mục 32.5
- Xây dựng lớp: Chương 6

Kiểu dữ liệu cơ bản là nền tảng cho tất cả các kiểu dữ liệu khác. Chương này cung cấp các lưu ý khi sử dụng số (nói chung), số nguyên, số thực dấu chấm động, ký tự và chuỗi, biến boolean, kiểu liệt kê, hằng số có tên và mảng. Phần cuối cùng mô tả cách tạo kiểu dữ liệu riêng.

Nếu bạn đã nắm vững các kiểu dữ liệu cơ bản, hãy xem lại checklist các vấn đề cần tránh ở cuối chương, sau đó nghiên cứu thêm về các kiểu dữ liệu không phổ biến ở Chương 13.

12.1 Sử Dụng Số Học Nói Chung (Numbers in General)

Dưới đây là một số hướng dẫn để giảm lỗi khi sử dụng số trong lập trình:

Tham khảo chi tiết: Xem thêm Mục 12.7 (“Named Constants”) về việc sử dụng hằng số có tên thay cho magic number.

- **Tránh “magic number”**: Magic number là số cụ thể (ví dụ: 100, 47524) xuất hiện trực tiếp trong mã mà không có giải thích. Nếu ngôn ngữ hỗ trợ hằng số có tên (named constant), hãy sử dụng chúng thay cho magic number. Nếu không thể, hãy xem xét dùng biến toàn cục (global variable) khi có thể.
- **Lợi ích khi tránh magic number**:
  - **Dễ dàng thay đổi**: Khi dùng hằng số có tên, bạn sẽ không bỏ sót bất kỳ trường hợp nào hoặc nhầm lẫn các số giống nhau khác mục đích.
  - **Dễ dàng bảo trì**: Nếu giá trị tối đa thay đổi từ 100 lên 200, với magic number, bạn phải tìm và thay tất cả các số 100, kể cả phép tính như 100 + 1, 100 - 1 (bạn sẽ còn phải tìm cả 101, 99).
  - **Mã dễ đọc hơn**: Ví dụ:

    ```
    for i = 0 to 99 do
    ```

    Bạn có thể đoán 99 là số mục tối đa, nhưng nếu viết:

    ```
    for i = 0 to MAX_ENTRIES-1 do
    ```

    thì không còn nghi ngờ gì nữa. Ngay cả khi bạn chắc chắn giá trị này không thay đổi, dùng hằng số vẫn giúp mã dễ đọc hơn.
- **Có thể viết trực tiếp 0 và 1 nếu cần thiết**: Các giá trị 0 và 1 thường dùng để tăng/giảm, hoặc làm chỉ số bắt đầu mảng. Ví dụ:

  ```
  for i = 0 to CONSTANT do
  ```

  là hoàn toàn chấp nhận được, hay

  ```
  total = total + 1
  ```

  cũng vậy. Quy tắc là: giá trị literal duy nhất nên xuất hiện trực tiếp trong mã là 0 hoặc 1.

(Bản dịch đảm bảo tính nhất quán thuật ngữ, giữ nguyên code, bổ sung cấu trúc và chú thích theo đúng hướng dẫn.)

Dự đoán lỗi chia cho số 0

Mỗi khi sử dụng ký hiệu chia (`/` trong hầu hết các ngôn ngữ lập trình), cần cân nhắc khả năng mẫu số của biểu thức có thể bằng 0 hay không. Nếu khả năng này tồn tại, hãy viết code nhằm ngăn chặn lỗi chia cho số 0.

Chuyển đổi kiểu dữ liệu một cách rõ ràng

Hãy đảm bảo rằng bất kỳ ai đọc mã nguồn cũng nhận biết được khi nào xảy ra chuyển đổi giữa các kiểu dữ liệu. Ví dụ, trong C++ có thể sử dụng:

```
y = x + (float) i;
```

Còn trong Microsoft Visual Basic có thể viết:

```
y = x + CSng(i)
```

Thực hành này cũng giúp đảm bảo rằng quá trình chuyển đổi là đúng ý bạn, bởi vì các trình biên dịch khác nhau sẽ thực hiện chuyển đổi theo những cách khác nhau; nếu không, bạn sẽ phải chấp nhận rủi ro về kết quả.

Tránh so sánh kiểu hỗn hợp

Nếu `x` là một số thực (floating-point number) và `i` là số nguyên (integer), phép so sánh sau gần như chắc chắn không hoạt động đúng như mong đợi:

```
if ( i = x ) then
```

Trình biên dịch sẽ phải xác định kiểu muốn sử dụng để so sánh, chuyển đổi một trong hai kiểu sang kiểu còn lại, làm tròn giá trị, và cuối cùng quyết định kết quả. Bạn sẽ rất may mắn nếu chương trình chạy đúng. Thay vào đó, hãy tự chuyển đổi kiểu dữ liệu để trình biên dịch chỉ phải so sánh hai số cùng kiểu và bạn biết chính xác điều gì đang được so sánh.

Lưu ý các cảnh báo của trình biên dịch

Nhiều trình biên dịch hiện đại sẽ cảnh báo bạn khi phát hiện các kiểu số khác nhau trong cùng một biểu thức. Hãy chú ý! Mỗi lập trình viên đều từng được nhờ giúp tìm lỗi nào đó, rồi phát hiện ra trình biên dịch đã cảnh báo về lỗi đó từ trước. Những lập trình viên giỏi sẽ sửa code để loại bỏ toàn bộ cảnh báo của trình biên dịch. Hãy tận dụng sức mạnh của trình biên dịch – nó sẽ tiết kiệm công sức cho bạn.

12.2 Integers

Khi sử dụng số nguyên, cần lưu ý các điểm sau:

Kiểm tra phép chia số nguyên

Khi dùng số nguyên, phép chia `7/10` không ra `0.7`, mà thường ra `0`, hoặc âm vô cực, hoặc số nguyên gần nhất – tuỳ thuộc vào ngôn ngữ lập trình. Điều này cũng đúng với kết quả trung gian của biểu thức. Ở ngoài đời, `10 * (7/10) = (10*7)/10 = 7`. Tuy nhiên, trong toán học số nguyên, `10 * (7/10)` lại bằng `0` vì `(7/10)` là `0`. Cách khắc phục đơn giản nhất là sắp xếp lại biểu thức sao cho phép chia được thực hiện sau cùng: `(10*7)/10`.

Kiểm tra tràn số nguyên (integer overflow)

Khi thực hiện phép nhân hay cộng số nguyên, cần chú ý đến giá trị tối đa mà kiểu số nguyên có thể lưu trữ. Ví dụ, số nguyên không dấu (unsigned integer) lớn nhất thường là `2^32-1` hoặc đôi khi là `2^16-1` tức 65,535. Khi bạn nhân hai số mà kết quả vượt quá giá trị tối đa này sẽ gây tràn số nguyên (integer overflow). Ví dụ, `250 * 300 = 75,000`; nhưng nếu giá trị tối đa là 65,535 thì kết quả bạn nhận được chỉ là 9,464 do tràn số nguyên (`75,000 - 65,536 = 9,464`).

Bảng 12-1: Phạm vi của các kiểu số nguyên phổ biến

Loại số nguyên	Phạm vi
Signed 8-bit	-128 đến 127
Unsigned 8-bit	0 đến 255
Signed 16-bit	-32,768 đến 32,767
Unsigned 16-bit	0 đến 65,535
Signed 32-bit	-2,147,483,648 đến 2,147,483,647
Unsigned 32-bit	0 đến 4,294,967,295
Signed 64-bit	-9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807
Unsigned 64-bit	0 đến 18,446,744,073,709,551,615

Cách đơn giản nhất để tránh tràn số nguyên là suy xét trước các giá trị lớn nhất mà từng toán hạng trong biểu thức số học có thể đạt tới. Ví dụ, với biểu thức số nguyên `m = j * k`, nếu giá trị lớn nhất của `j` là 200 và của `k` là 25 thì `m` tối đa là 5,000, an toàn cho số nguyên 32-bit. Nhưng nếu `j` lên tới 200,000 và `k` là 100,000 thì `m` có thể là 20,000,000,000 – lớn hơn 2,147,483,647, nên bạn cần dùng số nguyên 64-bit hoặc floating-point number (số thực).

Ngoài ra, hãy cân nhắc đến việc mở rộng chương trình trong tương lai: nếu `m` không bao giờ lớn hơn 5,000 thì ổn, nhưng nếu dự đoán sẽ tăng trong nhiều năm, cũng cần tính đến điều này.

Kiểm tra tràn số nguyên ở kết quả trung gian

Số cuối cùng của biểu thức không phải thứ duy nhất cần quan tâm. Xét ví dụ sau trong Java:

```
int termA = 1000000;
int termB = 1000000;
int product = termA * termB / 1000000;
System.out.println("( " + termA + " * " + termB + " ) / 1000000 = " + product );
```

Nếu nghĩ phép toán gán cho `product` giống `(1,000,000 * 1,000,000) / 1,000,000`, bạn kỳ vọng kết quả là 1,000,000. Nhưng trên thực tế, trình biên dịch sẽ tính `termA * termB` trước, tạo kết quả trung gian là 1,000,000,000,000 – vượt xa giá trị tối đa của số nguyên 32-bit, dẫn đến tràn số nguyên. Kết quả thực tế sẽ là:

```
( 1000000 * 1000000 ) / 1000000 = -727
```

Bạn có thể xử lý tràn số nguyên ở kết quả trung gian như cách xử lý tràn số nguyên thông thường: chuyển sang sử dụng số nguyên 64-bit (long integer) hoặc floating-point number.

12.3 Floating-Point Numbers (Số thực dấu phẩy động)

Điều cần lưu ý khi sử dụng số thực là nhiều số thập phân phân số không thể biểu diễn chính xác bằng các bit 1 và 0 của máy tính số. Những số thập phân vô hạn như `1/3` hoặc `1/7` thường chỉ có thể được biểu diễn chính xác tới 7 hoặc 15 chữ số. Trong Microsoft Visual Basic, biểu diễn floating-point 32-bit của `1/3` là `0.33333330`, chính xác tới 7 chữ số. Điều này đủ chính xác với hầu hết mục đích sử dụng, nhưng đôi khi có thể gây nhầm lẫn.

Sau đây là một số hướng dẫn cụ thể khi sử dụng floating-point number:

Tránh phép cộng và trừ các số có độ lớn rất khác nhau

Với biến floating-point 32-bit, `1,000,000.00 + 0.1` có thể vẫn ra kết quả là `1,000,000.00` vì 32 bit không đủ số chữ số có nghĩa để "chứa" khoảng cách giữa 1,000,000 và 0.1. Tương tự, `5,000,000.02 - 5,000,000.01` có thể cho ra 0.0.

Giải pháp Nếu phải cộng dãy số có sự chênh lệch lớn như vậy, hãy sắp xếp số trước và cộng bắt đầu từ số nhỏ nhất. Với các chuỗi vô hạn, hãy cộng từ số hạng nhỏ nhất – thực chất là cộng ngược lại. Cách này không loại bỏ hoàn toàn sai số làm tròn, nhưng sẽ giảm thiểu chúng. Nhiều sách chuyên về thuật toán cũng có hướng dẫn về vấn đề này.

Tránh so sánh bằng (equality comparison) với số thực

Các số floating-point khi về lý thuyết nên bằng nhau thì thực tế lại không. Lý do là hai con đường tính toán khác nhau dẫn tới cùng một kết quả thường không cho cùng một số. Ví dụ, cộng `0.1` mười lần hiếm khi ra chính xác `1.0`.

Ví dụ Java về phép so sánh không tốt đối với floating-point number

```
double nominal = 1.0;
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
if (nominal == sum) {
    System.out.println("Numbers are the same");
} else {
    System.out.println("Numbers are different");
}
```

Đầu ra sẽ là:

```
Numbers are different
```

Giá trị của `sum` trong vòng lặp lần lượt là:

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

Vì vậy, nên tìm giải pháp thay thế cho việc so sánh bằng giữa các số thực. Một phương pháp hiệu quả là xác định một ngưỡng độ chính xác chấp nhận được và sử dụng hàm boolean để kiểm tra xem hai giá trị xấp xỉ bằng nhau không. Ví dụ trong Java:

```
final double ACCEPTABLE_DELTA = 0.00001;
boolean Equals(double Term1, double Term2) {
    if (Math.abs(Term1 - Term2) < ACCEPTABLE_DELTA) {
        return true;
    } else {
        return false;
    }
}
```

Nếu áp dụng hàm này cho đoạn code so sánh trước đó, kết quả sẽ là:

```
Numbers are the same
```

Tùy vào yêu cầu của ứng dụng, bạn có thể cần tính toán `ACCEPTABLE_DELTA` dựa trên độ lớn của hai số được so sánh thay vì dùng giá trị cố định.

Dự đoán lỗi làm tròn số

Vấn đề về lỗi làm tròn (rounding error) cũng là một dạng của phép toán với các số có độ lớn rất khác nhau. Một số giải pháp để xử lý lỗi này bao gồm:

- **Chuyển sang kiểu biến có độ chính xác cao hơn:** Nếu đang dùng floating-point đơn (single-precision), hãy nâng lên double-precision.
- **Chuyển sang sử dụng BCD (Binary Coded Decimal):** Cách này thường tốn bộ nhớ và chậm hơn, nhưng ngăn ngừa nhiều lỗi làm tròn, đặc biệt khi lưu trữ các đại lượng như tiền tệ.
- **Chuyển từ floating-point sang integer:** Bạn có thể tự quản lý phần số nguyên và phần thập phân bằng integer, ví dụ lưu trữ tổng số cent thay vì đô-la và cent, sử dụng 64-bit integer khi cần độ chính xác cao. Nên xây dựng một lớp, ví dụ `DollarsAndCents`, ẩn việc thao tác với integer và hỗ trợ các phép toán số học cần thiết.
- **Kiểm tra hỗ trợ sẵn của ngôn ngữ hoặc thư viện:** Một số ngôn ngữ như Visual Basic có kiểu dữ liệu Currency hỗ trợ riêng cho dữ liệu nhạy cảm với lỗi làm tròn. Nếu ngôn ngữ của bạn có kiểu dữ liệu thích hợp, hãy sử dụng nó.

12.4 Characters and Strings

Dưới đây là một số lưu ý khi làm việc với chuỗi ký tự (string):

Tránh sử dụng "magic characters" và "magic strings"

Magic character là ký tự hằng xuất hiện lặp lại (ví dụ, `'A'`), *magic string* là chuỗi ký tự hằng (ví dụ, `"Gigamatic Accounting Program"`). Nếu ngôn ngữ bạn hỗ trợ hằng số có tên (named constant), hãy sử dụng chúng, hoặc sử dụng biến toàn cục thay vì đưa trực tiếp giá trị vào mã nguồn.

Lý do:

- Những chuỗi xuất hiện nhiều lần như tên chương trình, lệnh, tiêu đề báo cáo… có thể cần thay đổi nội dung, ví dụ `"Gigamatic Accounting`

`Program"` thành `"New and Improved! Gigamatic Accounting Program"` phiên bản mới.

- Việc hỗ trợ quốc tế hóa (internationalization) thuận lợi hơn nếu chuỗi tập trung ở file tài nguyên, thay vì rải rác trong chương trình.
- Chuỗi hằng quá nhiều sẽ tốn bộ nhớ và khó quản lý, đặc biệt trong các hệ thống nhúng hoặc môi trường giới hạn bộ nhớ.
- "Magic characters/strings" thường không rõ ý nghĩa; sử dụng hằng số có tên hoặc chú thích sẽ giúp tăng khả năng đọc hiểu mã nguồn.

Ví dụ trong C++:

```
// Không nên
if (input_char == 0x1B)

// Nên
if (input_char == ESCAPE)
```

Cảnh giác với lỗi lệch chỉ số (off-by-one error)

Bởi vì chuỗi có thể truy cập như mảng, dễ xảy ra lỗi vượt khỏi biên (đọc hoặc ghi ngoài giới hạn chuỗi).

Biết khả năng hỗ trợ Unicode của ngôn ngữ và môi trường

Một số ngôn ngữ như Java hỗ trợ Unicode cho tất cả các chuỗi, nhưng các ngôn ngữ như C hoặc C++ cần sử dụng hàm hoặc thư viện riêng. Việc chuyển đổi giữa Unicode và các bộ ký tự khác thường cần thiết khi giao tiếp với các thư viện chuẩn hoặc thư viện bên thứ ba.

Nếu chương trình không sử dụng Unicode, hãy quyết định sớm có dùng Unicode hay không, và nếu có, xác định rõ nơi và thời điểm sử dụng.

Xác định chiến lược quốc tế hóa (internationalization) và bản địa hóa (localization) ngay từ đầu

Điều này rất quan trọng: quyết định có lưu trữ toàn bộ chuỗi trong file tài nguyên bên ngoài hay không, xây dựng riêng từng bản cho mỗi ngôn ngữ hay xác định ngôn ngữ khi chạy chương trình.

Nếu chỉ hỗ trợ một ngôn ngữ bảng chữ cái (alphabetic language), hãy cân nhắc sử dụng bộ ký tự ISO 8859 (extended ASCII).

Nếu cần hỗ trợ đa ngôn ngữ, hãy sử dụng Unicode để đảm bảo tương thích.

Cần xây dựng chiến lược chuyển đổi chuỗi nhất quán giữa các kiểu string. Một cách làm phổ biến là lưu mọi chuỗi ở một định dạng trong chương trình, chỉ chuyển đổi sang các định dạng khác khi nhập/xuất.

Làm việc với chuỗi trong C

C++ cung cấp lớp string trong thư viện chuẩn (STL) giúp khắc phục hầu hết các vấn đề với chuỗi trong C. Tuy nhiên, nếu bạn vẫn phải làm việc trực tiếp với chuỗi kiểu C, hãy chú ý:

Phân biệt chỉ số con trỏ chuỗi và mảng ký tự

- Hãy cảnh giác với các biểu thức gán (`=`) liên quan đến chuỗi; các thao tác trên chuỗi trong C thường dùng các hàm như `strcmp()`, `strcpy()`, `strlen()`. Dấu `=` thường ám chỉ lỗi con trỏ.
- Sử dụng quy ước đặt tên để phân biệt con trỏ chuỗi và mảng ký tự, ví dụ prefix `ps` cho con trỏ chuỗi, `ach` cho mảng ký tự.

Khai báo độ dài chuỗi là hằng số +1

Lỗi lệch chỉ số với chuỗi kiểu C rất phổ biến vì dễ quên chừa chỗ cho ký tự null kết thúc (byte giá trị 0). Cách phòng tránh là sử dụng hằng số tên rõ ràng cho mọi khai báo chuỗi.

```
char name[NAME_LENGTH + 1] = { 0 }; // Chuỗi độ dài NAME_LENGTH + 1
for (i = 0; i < NAME_LENGTH; i++)
    name[i] = 'A';

strncpy(name, some_other_name, NAME_LENGTH);
```

Không có quy ước rõ ràng, dễ nhầm lẫn và phát sinh lỗi khó kiểm soát. Hãy thống nhất một cách sử dụng để giảm tải tâm lý và hạn chế lỗi.

Khởi tạo chuỗi về null để tránh "endless string"

C xác định kết thúc chuỗi bằng ký tự null (0). Nếu quên thiết lập ký tự null ở cuối, các thao tác chuỗi sẽ sai lệch, có thể truy xuất vượt quá vùng nhớ dự kiến.

Cách phòng tránh

- Khởi tạo mảng ký tự với giá trị 0 khi khai báo: `c char EventName[MAX_NAME_LENGTH + 1] = { 0 };`
- Khi cấp phát động, dùng `calloc()` thay vì `malloc()` để tự động khởi tạo về 0.

Kết luận

Hãy áp dụng các quy tắc trên để giảm thiểu lỗi liên quan đến kiểu dữ liệu cơ bản trong lập trình, nâng cao chất lượng mã nguồn, đảm bảo tính nhất quán, khả năng bảo trì, mở rộng và quốc tế hóa phần mềm của bạn.

12.5 Biến Boolean

> **Tham chiếu chéo:** Để biết thêm về mảng, hãy đọc mục 12.8 "Mảng" ở phần sau của chương này.

Sử dụng mảng ký tự thay cho con trỏ trong C

Nếu bộ nhớ không phải là một rào cản—và điều này thường đúng—hãy khai báo tất cả biến chuỗi của bạn dưới dạng mảng ký tự. Cách làm này giúp tránh các vấn đề về con trỏ, và trình biên dịch sẽ cảnh báo cho bạn khi xảy ra sai sót.

Sử dụng `strncpy()` thay cho `strcpy()` để tránh chuỗi vô tận

Các hàm xử lý chuỗi trong C có cả phiên bản an toàn và phiên bản nguy hiểm. Các hàm nguy hiểm như `strcpy()` và `strcmp()` sẽ tiếp tục xử lý cho tới khi gặp ký tự kết thúc chuỗi (null terminator). Các hàm an toàn hơn như `strncpy()` và `strncmp()` nhận thêm một tham số về độ dài tối đa, do đó kể cả khi chuỗi dài vô tận thì lời gọi hàm của bạn cũng không bị ảnh hưởng.

Biến Boolean

Sử dụng hợp lý các biến *logical* hoặc *boolean* giúp chương trình của bạn rõ ràng và sạch sẽ hơn, đồng thời chúng cũng khó bị sử dụng sai.

> **Tham chiếu chéo:** Để tìm hiểu thêm về việc dùng chú thích (comment) để ghi tài liệu cho chương trình, xem Chương 32: "Tự mô tả mã nguồn".

Sử dụng biến boolean để làm rõ mục đích kiểm tra điều kiện

Thay vì chỉ kiểm tra một biểu thức boolean, bạn có thể gán biểu thức này cho một biến có tên mô tả ý nghĩa, khiến mục đích kiểm tra trở nên rõ ràng. Ví dụ, trong đoạn mã sau, không rõ việc kiểm tra trong lệnh if là nhằm mục đích hoàn thành, kiểm tra lỗi, hay điều gì khác:

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||
     ( elementIndex == lastElementIndex ) ) {
    // ...
}
```

> **Tham chiếu chéo:** Để xem ví dụ về việc sử dụng hàm boolean để ghi tài liệu cho chương trình, xem mục "Đơn giản hóa biểu thức phức tạp" trong phần 19.1.

Trong ví dụ dưới đây, việc sử dụng biến boolean làm cho mục đích của lệnh if trở nên rõ hơn:

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );
repeatedEntry = ( elementIndex == lastElementIndex );
if ( finished || repeatedEntry ) {
    // ...
}
```

Sử dụng biến boolean để đơn giản hóa kiểm tra phức tạp

Thường khi bạn phải viết một điều kiện phức tạp, bạn có thể mất nhiều lần để làm đúng. Khi sửa đổi sau này, việc hiểu được mục đích ban đầu của kiểm tra đó cũng sẽ khó khăn. Việc dùng biến logical sẽ giúp kiểm tra dễ đọc, ít lỗi và dễ chỉnh sửa.

Ví dụ về một kiểm tra phức tạp trong Visual Basic:

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _
   ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _
   ( Not ErrorProcessing() ) Then
   ' thực hiện thao tác nào đó
End If
```

Bài kiểm tra này khá phức tạp và đặt gánh nặng lớn lên người đọc mã nguồn. Có lẽ bạn sẽ bỏ qua và bảo "Tôi sẽ tìm hiểu nó sau khi thực sự cần". Những người khác khi đọc code của bạn cũng sẽ phản ứng tương tự nếu bạn viết những kiểm tra như thế này.

Dưới đây là phiên bản đã đơn giản hóa nhờ biến boolean:

```
allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )
legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )
If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() ) Then
   ' thực hiện thao tác nào đó
End If
```

Dạng này đọc dễ dàng và nhanh chóng hơn nhiều.

Tạo kiểu dữ liệu boolean riêng nếu cần thiết

Một số ngôn ngữ như C++, Java và Visual Basic có kiểu dữ liệu boolean được định nghĩa sẵn. Một số khác như C thì không. Trong C, bạn có thể định nghĩa kiểu boolean của riêng mình như sau:

```
typedef int BOOLEAN;
```

Hoặc có cách như sau, với lợi ích là định nghĩa luôn giá trị true và false:

```
enum Boolean {
    True = 1,
    False = (!True)
};
```

Việc khai báo biến với kiểu BOOLEAN thay vì int giúp ý định sử dụng của biến rõ ràng hơn và chương trình của bạn cũng trở nên tự mô tả hơn.

12.6 Kiểu liệt kê (Enumerated Types)

Kiểu *enumerated* là một kiểu dữ liệu cho phép mô tả các thành viên của một tập hợp đối tượng bằng từ tiếng Anh. Kiểu liệt kê có sẵn trong C++ và Visual Basic, và thường dùng khi bạn biết tất cả các giá trị có thể của một biến và muốn thể hiện chúng bằng từ ngữ thay vì số.

Ví dụ về kiểu liệt kê trong Visual Basic:

```
Public Enum Color
    Color_Red
    Color_Green
    Color_Blue
End Enum

Public Enum Country
    Country_China
    Country_England
    Country_France
    Country_Germany
    Country_India
    Country_Japan
    Country_Usa
End Enum

Public Enum Output
    Output_Screen
    Output_Printer
    Output_File
End Enum
```

Kiểu liệt kê là lựa chọn mạnh mẽ thay cho các biểu diễn cũ kiểu "1 là đỏ, 2 là xanh lá, 3 là xanh dương". Một số hướng dẫn khi dùng kiểu liệt kê:

- **Dùng kiểu liệt kê để tăng tính dễ đọc:** Thay vì viết dòng lệnh như:

```
if chosenColor = 1
```

bạn có thể viết rõ ràng hơn: `vb    if chosenColor = Color_Red`

Bất cứ khi nào gặp số nguyên dùng như mã trạng thái, hãy tự hỏi liệu có nên thay thế bằng kiểu liệt kê hay không.

- **Kiểu liệt kê đặc biệt hữu dụng trong việc định nghĩa tham số cho hàm:** Ví dụ so sánh hai lời gọi hàm trong C++:

```
int result = RetrievePayrollData(data, true, false, false, true);
```

(Khó hiểu!)

So với:

```
int result = RetrievePayrollData(
    data,
    EmploymentStatus_CurrentEmployee,
    PayrollType_Salaried,
    SavingsPlan_NoDeduction,
    MedicalCoverage_IncludeDependents
);
```

(Dễ hiểu hơn nhiều!)

- **Dùng kiểu liệt kê để tăng độ tin cậy:** Một vài ngôn ngữ (như Ada) cho phép trình biên dịch kiểm tra kiểu kỹ hơn khi dùng kiểu liệt kê thay vì số nguyên hoặc hằng số. Khi dùng hằng số đặt tên, trình biên dịch không biết giá trị hợp lệ là gì, nhưng nếu một biến được khai báo kiểu Color, trình biên dịch chỉ cho phép các giá trị Color_Red, Color_Green và Color_Blue.
- **Dùng kiểu liệt kê để tăng khả năng chỉnh sửa code:** Nếu phát hiện sơ đồ ánh xạ số - màu sắc (1 = đỏ,...) bị sai, bạn chỉ cần sửa định nghĩa kiểu liệt kê mà không cần lục soát và thay thế toàn bộ chương trình với các số 1, 2, 3, ...
- **Dùng kiểu liệt kê thay cho boolean khi cần:** Nếu muốn phân biệt nhiều trạng thái hơn chỉ "thành công/không thành công", bạn có thể mở rộng với các giá trị như Status_Success, Status_Warning, Status_FatalError trong một kiểu liệt kê.
- **Kiểm tra giá trị không hợp lệ:** Khi kiểm tra kiểu liệt kê trong lệnh if hoặc case, hãy đảm bảo luôn xử lý trường hợp giá trị không hợp lệ:

```
Select Case screenColor
    Case Color_Red
    Case Color_Blue
    Case Color_Green
    Case Else
```

```
        DisplayInternalError(False, "Internal Error 752: Invalid color")
End Select
```

- **Định nghĩa cụ thể phần tử đầu/cuối cho giới hạn vòng lặp:** Việc định nghĩa rõ ràng để dùng trong vòng lặp qua các phần tử của kiểu liệt kê:

```
Public Enum Country
    Country_First = 0
    Country_China = 0
    Country_England = 1
    Country_France = 2
    Country_Germany = 3
    Country_India = 4
    Country_Japan = 5
    Country_Usa = 6
    Country_Last = 6
End Enum

Dim iCountry As Country
For iCountry = Country_First To Country_Last
    usaCurrencyConversionRate(iCountry) = ConversionRate(Country_Usa, iCountry)
Next
```

- **Dành phần tử đầu như giá trị không hợp lệ:** Nhiều trình biên dịch gán phần tử đầu của kiểu liệt kê là 0. Việc định nghĩa giá trị đầu là không hợp lệ giúp dễ phát hiện các biến chưa được khởi tạo.

```
Public Enum Country
    Country_InvalidFirst = 0
    Country_First = 1
    ...
    Country_Last = 7
End Enum
```

Đảm bảo quy chuẩn dự án quy định rõ về cách sử dụng các giá trị Invalid-First, First, Last để tránh nhầm lẫn.

- **Cẩn trọng khi gán giá trị tường minh:** Nếu bạn gán các giá trị tường minh cho kiểu liệt kê (như trong ví dụ dưới đây), hãy thận trọng khi lặp qua các giá trị:

```
enum Color {
    Color_InvalidFirst = 0,
    Color_First = 1,
    Color_Red = 1,
    Color_Green = 2,
    Color_Blue = 4,
    Color_Black = 8,
```

```
    Color_Last = 8
};
```

Khi đó nếu lặp từ 1 đến 8 sẽ gặp cả giá trị không hợp lệ.

Nếu ngôn ngữ của bạn không hỗ trợ kiểu liệt kê

Bạn có thể mô phỏng chúng với biến toàn cục hoặc class. Ví dụ trong Java (phiên bản chưa có enum):

```
// thiết lập kiểu Country
class Country {
    private Country() {}
    public static final Country China = new Country();
    public static final Country England = new Country();
    public static final Country France = new Country();
    public static final Country Germany = new Country();
    public static final Country India = new Country();
    public static final Country Japan = new Country();
}

// thiết lập kiểu Output
class Output {
    private Output() {}
    public static final Output Screen = new Output();
    public static final Output Printer = new Output();
    public static final Output File = new Output();
}
```

Những loại kiểu liệt kê này giúp chương trình dễ đọc hơn vì bạn có thể dùng các thành viên lớp như `Country.England` và `Output.Screen` thay vì hằng số có tên. Đặc biệt, nó cũng đảm bảo *type safety* (an toàn kiểu dữ liệu); ví dụ, trình biên dịch sẽ phát hiện lỗi nếu gán giá trị kiểu Output cho biến kiểu Country.

Nếu ngôn ngữ không hỗ trợ class, bạn có thể đạt hiệu ứng tương tự bằng cách sử dụng các biến toàn cục với quản lý có kỷ luật.

12.7 Hằng có tên (Named Constants)

Hằng có tên (named constant) giống như một biến, nhưng giá trị của nó không thể thay đổi sau khi gán. Sử dụng hằng có tên cho phép bạn tham chiếu đến các giá trị cố định bằng tên thay vì số—for example, dùng MAXIMUM_EMPLOYEES thay vì 1000.

Việc sử dụng hằng có tên là cách để "tham số hóa" chương trình—chuyển các khía cạnh của chương trình có khả năng thay đổi thành tham số, để khi cần sửa đổi, chỉ cần thay đổi tại một nơi duy nhất thay vì phải chỉnh sửa khắp mọi nơi trong chương trình. Nếu bạn từng khai báo một mảng lớn mà sau đó vẫn bị thiếu dung lượng, bạn sẽ thấy giá trị của hằng có tên.

Khi kích thước mảng thay đổi, bạn chỉ cần thay đổi định nghĩa của hằng ở nơi khai báo, điều này giúp kiểm soát tập trung, và làm cho phần mềm linh hoạt, dễ bảo trì hơn.

Dùng hằng có tên trong khai báo dữ liệu

Sử dụng hằng có tên giúp mã nguồn dễ đọc, dễ bảo trì trong cả khai báo dữ liệu lẫn các câu lệnh thao tác có liên quan đến kích thước. Ví dụ, thay vì dùng số 7, hãy dùng LOCAL_NUMBER_LENGTH:

```
Const AREA_CODE_LENGTH = 3
Const LOCAL_NUMBER_LENGTH = 7
Type PHONE_NUMBER
    areaCode(AREA_CODE_LENGTH) As String
    localNumber(LOCAL_NUMBER_LENGTH) As String
End Type

' kiểm tra tất cả ký tự trong số điện thoại có phải là chữ số
For iDigit = 1 To LOCAL_NUMBER_LENGTH
    If (phoneNumber.localNumber(iDigit) < "0") Or _
       ("9" < phoneNumber.localNumber(iDigit)) Then
        ' xử lý lỗi
    End If
Next
```

Sau này nếu cần đổi độ dài số điện thoại, chỉ việc sửa `LOCAL_NUMBER_LENGTH` mà thôi.

> **Tham khảo thêm:** Giá trị của việc tập trung quyền kiểm soát tại một điểm đã được chứng minh là rất hữu ích cho việc bảo trì phần mềm. (Xem thêm Glass 1991).

Tránh dùng số 'cứng' (literal), ngay cả với giá trị "an toàn"

Trong đoạn sau, 12 đại diện cho cái gì?

```
For i = 1 To 12
    profit(i) = revenue(i) - expense(i)
Next
```

Có vẻ nó lặp qua 12 tháng, nhưng điều đó có chắc chắn không? Thay vì để người khác phải suy đoán, hãy làm rõ ý định với hằng có tên:

```
For i = 1 To NUM_MONTHS_IN_YEAR
    profit(i) = revenue(i) - expense(i)
Next
```

Hoặc rõ ràng hơn nữa bằng cách đặt tên biến chỉ mục:

```
For month = 1 To NUM_MONTHS_IN_YEAR
    profit(month) = revenue(month) - expense(month)
Next
```

Cách tốt nhất là dùng luôn kiểu liệt kê cho tháng:

```
For month = Month_January To Month_December
    profit(month) = revenue(month) - expense(month)
Next
```

Kể cả khi nghĩ rằng một literal là "an toàn", hãy ưu tiên dùng hằng có tên.

> **Tham chiếu chéo:** Để biết cách mô phỏng hằng có tên với biến hoặc class phù hợp phạm vi, xem phần về mô phỏng kiểu liệt kê ở trên.

Sử dụng hằng có tên nhất quán

Việc dùng hằng có tên ở một nơi và literal ở nơi khác cho cùng một thực thể là rất nguy hiểm. Nếu cần thay đổi giá trị, bạn sẽ vô tình bỏ sót các literal mã hóa cứng, dẫn đến lỗi khó phát hiện và sửa chữa.

12.8 Mảng (Arrays)

Mảng là dạng cấu trúc dữ liệu đơn giản và phổ biến nhất. Ở một số ngôn ngữ, đó là kiểu cấu trúc duy nhất. Một mảng chứa các phần tử cùng kiểu dữ liệu và được truy cập trực tiếp thông qua chỉ số mảng.

Một số lưu ý khi sử dụng mảng:

- **Đảm bảo chỉ số mảng nằm trong phạm vi:** Lỗi phổ biến nhất là truy cập ngoài phạm vi mảng. Một số ngôn ngữ sẽ báo lỗi, một số khác sẽ gây ra hành vi bất định, không lường trước được.
- **Xem xét dùng container thay cho mảng, hoặc dùng mảng như cấu trúc tuần tự:** Một số nhà khoa học máy tính nổi bật cho rằng nên sử dụng mảng chỉ theo cách tuần tự thay vì truy cập ngẫu nhiên (Mills và Linger, 1986). Việc truy cập ngẫu nhiên được xem là không có kỷ luật, dễ gây lỗi và khó chứng minh tính đúng đắn chương trình.

Ghi chú: Một số lỗi đánh máy và định dạng đã được chỉnh lại cho mạch lạc và dễ hiểu hơn; nội dung giữ nguyên ý nghĩa gốc.

3. Dữ kiện thực nghiệm

Trong một thử nghiệm nhỏ, Mills và Linger nhận thấy rằng các thiết kế được tạo ra theo cách này dẫn đến việc sử dụng ít biến hơn và số lần tham chiếu biến cũng ít hơn. Những thiết kế này tương đối hiệu quả và mang lại phần mềm có độ tin cậy cao.

SỬ DỤNG DỮ LIỆU CỨNG

Hãy xem xét sử dụng các lớp vùng chứa (container class) mà bạn có thể truy cập tuần tự — chẳng hạn như set, stack, queue, v.v. — như những lựa chọn thay thế trước khi bạn mặc định chọn array (mảng).

Tham khảo chéo

Các vấn đề về sử dụng array (mảng) và loop (vòng lặp) có nhiều điểm tương tự, liên quan. Để biết thêm thông tin về loop, tham khảo Chương 16, "Controlling Loops".

> Hãy kiểm tra các điểm đầu, điểm cuối của array. Tương tự như việc bạn nên cân nhắc điểm bắt đầu và kết thúc trong cấu trúc loop, bạn có thể phát hiện nhiều lỗi bằng cách kiểm tra các điểm đầu và cuối của array. Hãy tự hỏi: Code có truy cập đúng phần tử đầu tiên của array hay vô tình truy cập phần tử trước/sau phần tử đầu tiên không? Phần tử cuối cùng thì sao? Liệu code có gặp lỗi off-by-one không? Cuối cùng, kiểm tra xem code có truy cập đúng các phần tử ở giữa của array không.

Nếu một array là đa chiều (multidimensional), hãy đảm bảo rằng các subscript (chỉ số) được sử dụng đúng thứ tự. Rất dễ xảy ra nhầm lẫn giữa `Array[i][j]` và `Array[j][i]`, vì vậy hãy dành thời gian kiểm tra kỹ thứ tự các chỉ số. Cân nhắc sử dụng những tên biến có ý nghĩa hơn thay vì chỉ dùng `i` và `j` nếu vai trò của chúng không rõ ràng.

Lưu ý về "index cross-talk" (nhầm lẫn chỉ số)

Nếu bạn dùng loop lồng nhau (nested loop), rất dễ viết `Array[j]` thay vì `Array[i]`. Sự nhầm lẫn giữa chỉ số vòng lặp được gọi là "index cross-talk". Hãy kiểm tra kỹ lỗi này. Tốt hơn nữa, hãy đặt tên chỉ số có ý nghĩa để giảm nguy cơ mắc lỗi cross-talk ngay từ đầu.

Trong C, sử dụng macro `ARRAY_LENGTH()` để làm việc với array

Bạn có thể thêm tính linh hoạt khi làm việc với array bằng cách định nghĩa macro `ARRAY_LENGTH()` như sau:

```
#define ARRAY_LENGTH( x ) (sizeof(x)/sizeof(x[0]))
```

Khi thao tác với array, thay vì sử dụng một hằng số đặt tên để xác định kích thước tối đa, hãy dùng macro `ARRAY_LENGTH()`. Ví dụ:

```
ConsistencyRatios[] =
{ 0.0, 0.0, 0.58, 0.90, 1.12,
  1.24, 1.32, 1.41, 1.45, 1.49,
  1.51, 1.48, 1.56, 1.57, 1.59 };
for ( ratioIdx = 0; ratioIdx < ARRAY_LENGTH( ConsistencyRatios ); ratioIdx++ );
```

Kỹ thuật này đặc biệt hữu ích với các array không có kích thước cố định như ví dụ trên. Nếu thêm hoặc bớt phần tử, bạn không phải chỉnh sửa hằng số kích thước array. Phương pháp này cũng áp dụng được cho array có kích thước xác định, và nhờ vậy bạn không cần đặt thêm hằng số ngoài cho kích thước array khi khai báo.

12.9 Tạo kiểu dữ liệu riêng (Type Aliasing)

Kiểu dữ liệu do lập trình viên định nghĩa (user-defined data type) là một trong những tính năng mạnh mẽ nhất của ngôn ngữ lập trình để giúp bạn hiểu rõ hơn về chương trình. Những kiểu này giúp bảo vệ chương trình trước những thay đổi bất ngờ và khiến code dễ đọc hơn mà không cần thiết kế, xây dựng hoặc kiểm thử các class mới. Nếu bạn sử dụng C, C++ hoặc ngôn ngữ cho phép định nghĩa kiểu dữ liệu, hãy tận dụng chúng!

> Đôi khi tốt hơn nên tạo class thay vì kiểu dữ liệu đơn giản; chi tiết xem tại Chương 6, "Working Classes".

Để hiểu được sức mạnh của việc tạo kiểu dữ liệu, giả sử bạn viết chương trình chuyển đổi tọa độ hệ trục *x, y, z* sang vĩ độ, kinh độ và cao độ. Bạn nghĩ có thể cần dùng biến kiểu double-precision floating-point, nhưng ưu tiên viết chương trình với single-precision floating-point đến khi chắc chắn cần thay đổi. Bạn có thể tạo kiểu riêng cho tọa độ bằng lệnh `typedef` trong C hoặc C++. Ví dụ:

```
typedef float Coordinate; // cho biến tọa độ
```

Khai báo này tạo kiểu mới tên là `Coordinate`, về mặt chức năng tương đương với float. Sử dụng như kiểu dữ liệu có sẵn:

```
Routine1() {
    Coordinate latitude;   // vĩ độ
    Coordinate longitude;  // kinh độ
```

```
    Coordinate elevation;  // cao độ (từ tâm Trái Đất)
}
Routine2() {
    Coordinate x;
    Coordinate y;
    Coordinate z;
}
```

Giả sử về sau, bạn cần sử dụng biến double-precision. Do đã tách riêng kiểu cho tọa độ, bạn chỉ cần thay đổi duy nhất ở khai báo kiểu:

```
typedef double Coordinate; // đổi từ float sang double
```

Ví dụ thứ hai, trong Pascal: Giả sử bạn xây dựng hệ thống bảng lương, tên nhân viên tối đa 30 ký tự. Thay vì hard-code số 30 nhiều nơi, phương pháp tốt hơn là định nghĩa kiểu:

```
Type
    employeeName = array[ 1..30 ] of char;
```

Đối với string hoặc array, nên định nghĩa hằng chứa độ dài rồi sử dụng trong khai báo kiểu:

```
Const
    NAME_LENGTH = 30;
Type
    employeeName = array[ 1..NAME_LENGTH ] of char;
```

Ví dụ mạnh mẽ hơn là kết hợp việc tạo kiểu dữ liệu và ẩn thông tin (information hiding). Có trường hợp bạn cần che giấu thông tin về kiểu dữ liệu. Đoạn ví dụ C++ sử dụng `Coordinate` đã che giấu được phần nào kiểu thực sự của dữ liệu, bởi tất cả đều dùng `Coordinate` thay vì float hay double. C++ cung cấp khả năng ẩn thông tin tượng trưng; muốn ẩn tuyệt đối, bạn hoặc người dùng tiếp theo cần tuân thủ không tìm kiểu của `Coordinate`.

Ở ngôn ngữ như Ada, việc ẩn thông tin là bắt buộc:

```
package Transformation is
    type Coordinate is private;
end Transformation;
```

Khi sử dụng kiểu này ở package khác:

```
with Transformation;
procedure Routine1 is
    latitude: Coordinate;
    longitude: Coordinate;
begin
    -- sử dụng latitude, longitude
end Routine1;
```

Khi khai báo là private trong package, chỉ phần nội bộ mới biết chi tiết kiểu Coordinate. Phân phối cho lập trình viên khác chỉ bản mô tả package, không có thông tin kiểu thật phía sau, giúp ẩn dữ liệu tốt hơn rất nhiều so với C++ (bắt buộc phải chia sẻ định nghĩa kiểu trong header file).

Một số lý do nên tạo kiểu dữ liệu riêng:

- **Dễ sửa đổi:** Việc tạo kiểu mới không tốn nhiều công sức nhưng giúp linh hoạt trước thay đổi.
- **Tránh phân tán thông tin:** Kiểu dữ liệu "cứng" khiến chi tiết khai báo loại dữ liệu lan tỏa khắp chương trình, thay vì tập trung một nơi. Điều này vi phạm nguyên tắc information hiding (tập trung hóa).
- **Tăng độ tin cậy:** Trong Ada, bạn có thể định nghĩa phạm vi giá trị cho kiểu, ví dụ `type Age is range 0..99;` giúp compiler kiểm tra mọi trường hợp sử dụng có đúng phạm vi không.
- **Khắc phục thiếu hụt của ngôn ngữ:** Nếu ngôn ngữ không có sẵn kiểu bạn cần (ví dụ boolean ở C), bạn có thể tạo mới: `c typedef int Boolean;`

Vì sao ví dụ dùng Pascal và Ada? Pascal và Ada ít phổ biến hiện nay, tuy nhiên về khía cạnh định nghĩa type đơn giản, các ngôn ngữ hiện đại như C++, Java, Visual Basic có một số bất lợi. Ví dụ:

```
currentTemperature: INTEGER range 0..212;
```

Chứa thông tin ngữ nghĩa quan trọng hơn so với:

```
int temperature;
```

Với Ada, khai báo type như sau:

```
type Temperature is range 0..212;
currentTemperature: Temperature;
```

Giúp compiler bảo đảm các phép gán cho `currentTemperature` chỉ sử dụng biến cùng kiểu, tăng mức an toàn, không cần nhiều code bổ sung.

Dĩ nhiên, lập trình viên có thể xây dựng một class `Temperature` để ràng buộc ngữ nghĩa tương tự, nhưng việc tạo class có thể gây ngần ngại do phức tạp hơn khai báo type một dòng.

Hướng dẫn khi tạo kiểu dữ liệu riêng

> Tham khảo chéo: mỗi trường hợp nên cân nhắc liệu tạo class có phù hợp hơn kiểu dữ liệu đơn giản không; xem chi tiết tại Chương 6, "Working Classes".

- **Đặt tên kiểu theo chức năng thực tế:** Nên tránh dùng tên phản ánh dạng dữ liệu máy tính mà hãy liên hệ với vấn đề thực tế mà kiểu dữ liệu biểu diễn (vd: "Coordinate", "employeeName").
- **Tránh kiểu tên gắn với kiểu có sẵn:** Những cái tên như `BigInteger`, `LongString` chỉ gợi ý về dạng dữ liệu chứ không liên hệ tới bài toán.
- **Tránh dùng kiểu định sẵn ở nơi khác ngoài khai báo type:** Nếu khả năng cần sửa đổi kiểu, đừng dùng trực tiếp kiểu định sẵn ngoại trừ trong phần `typedef` hay khai báo type tương tự.
- **Không định nghĩa lại kiểu sẵn có:** Nếu ngôn ngữ đã có kiểu `Integer`, tránh khai báo lại kiểu cùng tên. Việc này dễ gây nhầm lẫn khi đọc code.
- **Định nghĩa loại thay thế cho tính di động:** Bạn có thể định nghĩa `INT32`, `LONG64`,..., tương ứng với từng nền tảng phần cứng.
- **Tránh các kiểu dễ bị nhầm lẫn với kiểu hệ thống:** Nên tạo sự khác biệt rõ ràng (như `INT32` thay vì `INT`).
- **Xem xét tạo class thay vì đơn giản sử dụng typedef:** Đôi khi nên lập trình class để tăng mức kiểm soát, bảo vệ thông tin, thay cho khai báo type đơn thuần.

---

cc2e.com/1206 CHECKLIST: Kiểm tra dữ liệu nền tảng

Numbers in General

> Tham khảo danh sách kiểm tra áp dụng cho dữ liệu nói chung tại trang 257, Chương 10, "General Issues in Using Variables". Danh sách đặt tên biến xem tại trang 288, Chương 11, "The Power of Variable Names".

- Code có tránh sử dụng magic number không?
- Code có xử lý tình huống chia cho 0 không?
- Chuyển đổi kiểu có rõ ràng không?
- Khi dùng hai kiểu khác nhau trong một biểu thức, kết quả có đúng ý định không?
- Code có tránh so sánh hỗn hợp kiểu không?
- Chương trình biên dịch không có cảnh báo chứ?

Integers

- Các biểu thức với chia số nguyên hoạt động đúng như kỳ vọng?
- Biểu thức integer tránh lỗi tràn không?

Floating-Point Numbers

- Code có tránh cộng, trừ các số khác biệt về độ lớn không?
- Đã kiểm soát lỗi làm tròn chưa?
- Code có tránh so sánh floating-point bằng dấu bằng không?

Characters and Strings

- Code có tránh magic character và magic string không?
- Truy xuất chuỗi có tránh lỗi off-by-one không?
- Code C tránh nhầm lẫn giữa string pointer và character array không?
- Code C có khai báo chuỗi dài bằng hằng số +1?
- Code C dùng mảng ký tự đúng lúc không?
- Code C khởi tạo chuỗi với NULL tránh lỗi string không kết thúc không?
- Code C dùng `strncpy()` thay vì `strcpy()`, `strncat()` thay vì `strcat()`, `strncmp()` thay vì `strcmp()` không?

Boolean Variables

- Có sử dụng biến boolean bổ sung để giải thích điều kiện không?
- Có dùng biến boolean bổ sung để làm đơn giản điều kiện không?

Enumerated Types

- Program có dùng enumerated type thay vì hằng số đặt tên để tăng khả năng đọc, tin cậy và sửa đổi dễ dàng?
- Khi một biến không chỉ là true/false, có sử dụng enumerated type thay vì boolean không?
- Có kiểm tra giá trị không hợp lệ với enumerated type chưa?
- Giá trị đầu tiên của enumerated type có dành cho "invalid" không?

Named Constants

- Dùng hằng số đặt tên cho khai báo dữ liệu, giới hạn loop, thay vì magic number chưa?
- Sử dụng hằng số đặt tên một cách nhất quán? Không dùng ở chỗ này, literal ở chỗ khác?

Arrays

- Tất cả index truy cập array nằm trong phạm vi array không?
- Tham chiếu array có tránh lỗi off-by-one không?
- Tất cả subscript của array đa chiều đúng thứ tự chưa?
- Nếu có nested loop, subscript array sử dụng đúng biến chưa, tránh lẫn index giữa các vòng lặp?

Tạo kiểu dữ liệu riêng

- Chương trình có dùng kiểu khác biệt cho từng loại dữ liệu có khả năng thay đổi không?
- Tên kiểu dữ liệu hướng tới thực thể thực tế hay mô hình ngôn ngữ lập trình?
- Tên kiểu dữ liệu có đủ rõ nghĩa để hỗ trợ tài liệu hóa code không?
- Có tránh định nghĩa đè lên kiểu hệ thống không?

- Đã cân nhắc tạo class thay vì chỉ định nghĩa lại type chưa?

Key Points

- Làm việc với từng kiểu dữ liệu yêu cầu bạn ghi nhớ nhiều quy tắc chi tiết. Sử dụng danh sách kiểm tra ở chương này giúp bạn kiểm soát các vấn đề phổ biến.
- Việc tạo kiểu dữ liệu riêng giúp chương trình dễ sửa đổi, tự tài liệu hóa tốt hơn nếu ngôn ngữ hỗ trợ tính năng này.
- Khi tạo kiểu dữ liệu đơn giản với `typedef` hoặc tương đương, hãy cân nhắc liệu có nên tạo một class mới thay vì kiểu đơn giản đó.

Chương 13: Những kiểu dữ liệu không thông thường (Unusual Data Types)

13.1 Cấu trúc (Structure)

Thuật ngữ "structure" ám chỉ dữ liệu được xây dựng từ các kiểu dữ liệu khác. Do array (mảng) là trường hợp đặc biệt, chúng được bàn riêng ở Chương 12. Phần này hướng dẫn về dữ liệu cấu trúc do người dùng tạo (struct trong C, C++ hoặc Structure trong Microsoft Visual Basic). Trong Java và C++, đôi khi class cũng được sử dụng như structure khi class chỉ bao gồm dữ liệu public mà không có phương thức public.

Thông thường, bạn nên tạo class thay vì structure để tận dụng các tính năng bảo mật và chức năng mà class mang lại, ngoài dữ liệu public của structure. Tuy nhiên, đôi khi thao tác trực tiếp với block dữ liệu vẫn hữu ích. Sau đây là một số lý do sử dụng structure:

- Sử dụng structure để làm rõ mối quan hệ dữ liệu. Structure gom nhóm các thông tin liên quan lại. Khó khăn lớn nhất khi đọc một chương trình thường là xác định thông tin nào liên quan đến thông tin nào – như thể đến một thị trấn nhỏ hỏi ai là họ hàng với ai: mọi người đều hơi liên quan đến nhau nhưng không thực sự rõ ràng.

Nếu dữ liệu đã được cấu trúc đúng, việc nhận biết dữ liệu nào liên quan trở nên dễ dàng hơn.

Ví dụ về dữ liệu khai báo không có cấu trúc (trích dẫn code Visual Basic):

```
name = inputName
address = inputAddress
phone = inputPhone
title = inputTitle
```

```
department = inputDepartment
bonus = inputBonus
```

Do dữ liệu này không có cấu trúc, trông như tất cả các lệnh gán đều liên quan tới nhau.

(Nếu bạn cần bản dịch tiếp phần cuối hoặc có yêu cầu cụ thể về mục nào, vui lòng chỉ định tiếp nhé.)

Sử dụng Kiểu Dữ Liệu Cấu Trúc (Structures) và Con Trỏ (Pointers) trong Lập Trình

Minh họa về Việc Sử dụng Cấu Trúc để Làm Rõ Quan Hệ Dữ Liệu

Đoạn mã dưới đây không cung cấp bất kỳ gợi ý nào cho thấy có hai loại dữ liệu đang được xử lý. Trong đoạn mã phía dưới, việc sử dụng *structures* (cấu trúc) giúp làm rõ hơn các mối quan hệ:

Ví dụ Visual Basic về Biến Cấu Trúc Rõ Nghĩa

```
employee name = inputName
employee address = inputAddress
employee phone = inputPhone
supervisor title = inputTitle
supervisor department = inputDepartment
supervisor bonus = inputBonus
```

Trong đoạn mã sử dụng các biến cấu trúc, có thể thấy rõ rằng một số dữ liệu liên kết với employee (nhân viên), những dữ liệu khác liên quan đến supervisor (người giám sát).

Sử dụng Structures để Đơn Giản Hóa Thao Tác trên Khối Dữ Liệu

Bạn có thể kết hợp các thành phần có liên quan thành một *structure* và thực hiện thao tác trên cấu trúc đó. Thao tác trên một *structure* đơn giản hơn so với thao tác trên từng phần tử riêng lẻ, đồng thời cũng đáng tin cậy hơn và tiết kiệm nhiều dòng mã hơn.

Giả sử bạn có một nhóm các phần tử dữ liệu có mối liên hệ chặt chẽ – ví dụ, thông tin về một employee trong một cơ sở dữ liệu nhân sự. Nếu không gộp chúng thành *structure*, chỉ riêng việc sao chép nhóm dữ liệu này cũng cần rất nhiều câu lệnh. Dưới đây là ví dụ sử dụng Visual Basic:

Ví dụ Copy Dữ Liệu của Nhóm Không Sử Dụng Structure

```
newName = oldName
newAddress = oldAddress
newPhone = oldPhone
newSsn = oldSsn
newGender = oldGender
newSalary = oldSalary
```

Mỗi lần cần chuyển thông tin về một *employee*, bạn phải viết toàn bộ nhóm các câu lệnh này. Nếu bổ sung một trường mới, ví dụ numWithholdings, bạn cần phải tìm và chỉnh sửa ở tất cả các nơi có khối lệnh gán này.

Việc hoán đổi dữ liệu giữa hai employee theo cách này sẽ rất phức tạp:

Ví dụ Hoán Đổi Hai Nhóm Dữ Liệu Theo Cách Thủ Công

```
' swap new and old employee data
previousOldName = oldName
previousOldAddress = oldAddress
previousOldPhone = oldPhone
previousOldSsn = oldSsn
previousOldGender = oldGender
previousOldSalary = oldSalary
oldName = newName
oldAddress = newAddress
oldPhone = newPhone
oldSsn = newSsn
oldGender = newGender
oldSalary = newSalary
newName = previousOldName
newAddress = previousOldAddress
newPhone = previousOldPhone
newSsn = previousOldSsn
newGender = previousOldGender
newSalary = previousOldSalary
```

Một cách tiếp cận dễ dàng hơn là khai báo một biến cấu trúc:

Ví dụ Khai Báo Structure trong Visual Basic

```
Structure Employee
    name As String
    address As String
    phone As String
    ssn As String
    gender As String
```

```
    salary As Long
End Structure

Dim newEmployee As Employee
Dim oldEmployee As Employee
Dim previousOldEmployee As Employee
```

Bây giờ, bạn chỉ cần ba câu lệnh để hoán đổi toàn bộ dữ liệu:

Ví dụ Đơn Giản Hóa Việc Hoán Đổi Dữ Liệu Giữa Hai Structures

```
previousOldEmployee = oldEmployee
oldEmployee = newEmployee
newEmployee = previousOldEmployee
```

Nếu cần thêm trường mới như numWithholdings, chỉ việc bổ sung vào khai báo *Structure*. Các câu lệnh trên không cần thay đổi. Các ngôn ngữ khác như C++ cũng hỗ trợ khả năng tương tự.

Sử Dụng Structures Để Đơn Giản Hoá Danh Sách Tham Số

Bạn có thể đơn giản hóa danh sách tham số bằng cách dùng biến cấu trúc. Thay vì truyền từng thành phần riêng lẻ, hãy nhóm chúng vào một *structure* và truyền toàn bộ. Ví dụ:

Ví dụ Truyền Tham Số Một Cách Thủ Công

```
HardWayRoutine( name, address, phone, ssn, gender, salary )
```

Ví dụ Truyền Tham Số Bằng Structure

```
EasyWayRoutine( employee )
```

Nếu bạn cần thêm numWithholdings vào loại gọi thủ công, phải cập nhật tất cả các lời gọi đến HardWayRoutine(). Ngược lại, chỉ cần thêm trường này vào *Employee*, các lời gọi đến EasyWayRoutine() không cần sửa đổi.

Lưu ý: Tuy nhiên, không nên đưa tất cả biến trong chương trình thành một *structure* lớn rồi truyền bất cứ đâu. Lập trình viên cẩn trọng chỉ nên nhóm dữ liệu khi thực sự hợp lý và tránh truyền *structure* nếu chỉ sử dụng một vài trường bên trong – khi ấy, nên truyền trường cụ thể. Đây là biểu hiện của *information hiding* (che giấu thông tin): một số thông tin được ẩn trong các routine, một số bị ẩn khỏi routine khác, và việc truyền dữ liệu chỉ nên dựa trên nguyên tắc "cần biết".

Sử Dụng Structures để Giảm Chi Phí Bảo Trì Mã

Khi bạn nhóm dữ liệu liên quan nhờ *structures*, việc thay đổi cấu trúc này sẽ yêu cầu ít thay đổi hơn trong toàn bộ chương trình, đặc biệt đối với những đoạn mã không liên quan trực tiếp đến thay đổi này. Vì mỗi thay đổi đều tiềm ẩn khả năng lỗi, số lần chỉnh sửa ít đi đồng nghĩa với ít lỗi hơn. Nếu *Employee* có trường title và bạn quyết định loại bỏ nó, bạn chỉ cần sửa những nơi liên quan tới thao tác với trường này. Những phần mã khác gọi hoặc gán vào toàn bộ *structure* không ảnh hưởng.

Lợi ích lớn nhất của dữ liệu cấu trúc là những đoạn mã liên quan đến nhóm dữ liệu mà không quan tâm đến thành phần cụ thể như title trường hợp này sẽ không sai sót khi bạn thay đổi.

13.2 Con Trỏ (Pointers)

Việc sử dụng *pointers* (con trỏ) là một trong những khu vực dễ phạm lỗi nhất trong lập trình hiện đại. Đến mức nhiều ngôn ngữ hiện đại như Java, C#, và Visual Basic không hỗ trợ kiểu dữ liệu pointer. Sử dụng pointer tự thân đã rất phức tạp và yêu cầu phải hiểu cặn kẽ cơ chế quản lý bộ nhớ của trình biên dịch. Nhiều vấn đề bảo mật phổ biến, đặc biệt là buffer overrun (tràn bộ đệm), đều xuất phát từ lỗi dùng con trỏ (Howard và LeBlanc 2003).

Ngay cả khi ngôn ngữ của bạn không yêu cầu sử dụng con trỏ, việc hiểu rõ về pointer cũng giúp bạn thấu hiểu sâu hơn cách ngôn ngữ vận hành. Áp dụng nguyên tắc *defensive programming* (lập trình phòng vệ) càng cho hiệu quả tốt hơn.

Mô hình Khái Niệm về Pointer

Về mặt khái niệm, mỗi pointer gồm hai phần: vị trí trong bộ nhớ và cách hiểu nội dung tại vị trí đó.

Vị trí trong Bộ nhớ

Vị trí này là một *address* (địa chỉ), thường là một giá trị hệ thập lục phân. Trên vi xử lý 32-bit, địa chỉ sẽ là giá trị 32-bit, ví dụ 0x0001EA40. Bản thân con trỏ chỉ chứa địa chỉ. Muốn sử dụng dữ liệu mà con trỏ trỏ tới, bạn phải đến địa chỉ này và diễn giải nội dung bộ nhớ ở đó. Nếu chỉ nhìn bộ nhớ ở địa chỉ đó, bạn sẽ chỉ thấy một tập hợp các bit – phải có cách diễn giải thì nó mới có ý nghĩa.

Hiểu Biết Cách Diễn Giải Dữ Liệu

Cách diễn giải nội dung bộ nhớ được xác định qua *base type* (kiểu cơ sở) của pointer. Nếu một pointer trỏ tới một integer, có nghĩa là trình biên dịch sẽ hiểu vùng nhớ tại vị trí con trỏ như một số nguyên. Bạn có thể có nhiều con trỏ trỏ cùng một địa chỉ bộ nhớ như integer pointer, string pointer, và floating-point pointer nhưng chỉ có một pointer đọc đúng kiểu tại vị trí đó.

Hãy nhớ rằng bộ nhớ không tự nó mang ý nghĩa – ý nghĩa chỉ xuất hiện khi dùng đúng loại pointer. Ví dụ minh họa:

> **Hình 13-1** bên dưới hiển thị nhiều cách nhìn cùng một vùng bộ nhớ, mỗi cách diễn giải đều ra kết quả khác nhau:

```
0A 61 62 63 64 65 66 67 68 69 6A

- Dưới dạng raw memory (bộ nhớ thô): không có ý nghĩa nếu thiếu pointer
- Dưới dạng String[10] (Visual Basic, có byte độ dài ở đầu): kết quả "abcdefghij"
- Dưới dạng số nguyên 2 bytes: kết quả 24842
- Dưới dạng số thực 4 bytes: kết quả 4.17595656202980E+0021
- Dưới dạng số nguyên 4 bytes: kết quả 1667391754
- Dưới dạng ký tự char: kết quả ký tự linefeed (ASCII hex 0A hoặc thập phân 10)
```

Số byte được sử dụng phụ thuộc vào cách diễn giải vùng nhớ cũng như loại nền của pointer.

Một Số Lưu Ý Khi Sử Dụng Con Trỏ

Với nhiều loại lỗi, xác định vị trí lỗi là phần đơn giản nhất, việc sửa lỗi mới khó. Nhưng với lỗi con trỏ, nguyên nhân chủ yếu là con trỏ trỏ nhầm địa chỉ dẫn đến việc ghi dữ liệu vào vùng nhớ không đúng (gọi là *memory corruption* – hỏng bộ nhớ). Hậu quả có thể là chương trình sụp đổ, tính toán sai, thực thi sai hoặc thậm chí không thấy lỗi gì – những trường hợp này là "quả bom hẹn giờ" chờ bùng nổ khi quan trọng nhất. Triệu chứng và nguyên nhân lỗi pointer thường không liên quan trực tiếp.

Phần lớn thời gian khắc phục lỗi con trỏ nằm ở việc xác định nguyên nhân.

Chiến Lược Làm Việc Thành Công với Con Trỏ

Mục tiêu: - Tránh tạo ra lỗi pointer ngay từ đầu vì khó tìm. - Phát hiện lỗi pointer càng sớm càng tốt, vì triệu chứng rất khó đoán.

Một số phương pháp cụ thể:

1. Tách Biệt Thao Tác Con Trỏ Trong Routine hoặc Class

Giả sử bạn dùng linked list (danh sách liên kết) ở nhiều nơi, hãy xây dựng các routine như NextLink(), PreviousLink(), InsertLink(), và DeleteLink() để thao tác danh sách, thay vì truy cập pointer trực tiếp. Như vậy, phạm vi thao tác pointer thu hẹp, giảm nguy cơ lỗi và phù hợp để tái sử dụng code.

2. Khai Báo và Khởi Tạo Pointer Đồng Thời

Nên gán giá trị khởi tạo cho pointer gần vị trí khai báo:

Ví dụ C++ Sai:

```
Employee *employeePtr;
// rất nhiều dòng code
employeePtr = new Employee;
```

Lưu ý: Nếu có người dùng employeePtr trước khi khởi tạo thì sẽ phát sinh lỗi tiềm ẩn. (Đã sửa lỗi đánh máy trong bản gốc).

Cách đúng:

```
// các dòng code cần thiết
Employee *employeePtr = new Employee;
```

3. Xoá Pointer Cùng Phạm Vi Khi Được Cấp Phát

Phải đảm bảo new và delete diễn ra cùng phạm vi. Nếu cấp phát con trỏ trong constructor, hãy xóa trong destructor. Nếu không, bạn tạo ra sự bất đồng bộ, khó quản lý và dễ lỗi.

4. Kiểm Tra Pointer Trước Khi Sử Dụng

Trước lúc thao tác trên pointer, cần kiểm tra giá trị mà pointer trỏ tới có hợp lệ không (ví dụ, đảm bảo ở trong phạm vi StartData đến EndData).

5. Kiểm Tra Giá Trị Biến Mà Pointer Trỏ Đến

Đôi khi bạn có thể kiểm tra hợp lý giá trị đó, ví dụ pointer trỏ tới một số nguyên giới hạn từ 0-1000, hoặc một chuỗi không vượt quá 100 ký tự. Cách này có thể tự động hóa khi truy cập bằng routine thay vì trực tiếp.

6. Sử Dụng "Dog-tag Fields" để Kiểm Tra Lỗi Bộ Nhớ

Dog-tag field là trường thêm vào structure để kiểm tra lỗi. Khi cấp phát biến, gán giá trị đặc biệt vào tag này; khi giải phóng hay sử dụng, kiểm tra lại tag, nếu khác giá trị gốc thì đã có lỗi bộ nhớ. Sau khi delete, đổi giá trị tag để ngăn dùng lại vùng nhớ đã giải phóng.

Quy trình: 1. Cấp phát nhiều hơn số byte cần (ví dụ: new 104 bytes thay vì 100 bytes). 2. Đặt giá trị dog-tag vào 4 byte đầu, trả về pointer trỏ sau dog-tag.

3. Khi giải phóng, kiểm tra tag đầu, rồi set tag và dữ liệu về giá trị không hợp lệ. 4. Giải phóng toàn bộ block.

Đặt tag ở đầu khối bộ nhớ giúp phát hiện giải phóng lặp, ở cuối block giúp kiểm tra ghi tràn bộ nhớ. Bạn có thể kết hợp kiểm tra pointer nằm trong danh sách cấp phát.

7. Thêm Tính Dư Thừa Rõ Ràng

Một lựa chọn khác là lưu trường dữ liệu hai lần (redundant field). Nếu hai bản không trùng nhau, chắc chắn bộ nhớ đã bị hỏng. Tuy nhiên cách này gây tốn bộ nhớ, nên chỉ nên dùng khi pointer được truy cập qua routine.

8. Dùng Nhiều Biến Pointer Để Đảm Bảo Rõ Ràng

Không dùng một pointer cho nhiều mục đích. Ví dụ, khi thao tác linked list, nên đặt tên cho từng node liên quan giúp code dễ hiểu.

Ví dụ C++ Chèn Node Kiểu Truyền Thống Khó Hiểu:

```
void InsertLink(Node *currentNode, Node *insertNode) {
    // insert "insertNode" after "currentNode"
    insertNode->next = currentNode->next;
    insertNode->previous = currentNode;
    if ( currentNode->next != NULL ) {
        currentNode->next->previous = insertNode;
    }
    currentNode->next = insertNode;
}
```

Đoạn này chỉ rõ hai object, khiến người đọc phải tự suy ra mối liên hệ.

Phiên Bản Dễ Hiểu Hơn:

```
void InsertLink(Node *startNode, Node *newMiddleNode) {
    // insert "newMiddleNode" between "startNode" and "followingNode"
    Node *followingNode = startNode->next;
    newMiddleNode->next = followingNode;
    newMiddleNode->previous = startNode;
    if ( followingNode != NULL ) {
        followingNode->previous = newMiddleNode;
    }
    startNode->next = newMiddleNode;
}
```

Mặc dù thêm một dòng, đoạn code này dễ đọc hơn.

9. Đơn Giản Hóa Biểu Thức Pointer Phức Tạp

Biểu thức pointer lồng nhau quá nhiều rất khó theo dõi và bảo trì.

Ví dụ C++ Coding Horror - Biểu Thức Pointer Khó Đọc:

```
for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
    netRate[ rateIndex ] = baseRate[ rateIndex ] * rates->discounts->factors->net;
}
```

Những biểu thức kiểu này buộc lập trình viên phải cố hiểu thay vì chỉ đọc chương trình.

Tóm lại:
- Nên sử dụng *structures* để nhóm dữ liệu có liên quan và giảm trùng lặp mã, đơn giản hóa tham số truyền và giảm lỗi bảo trì. - Khi sử dụng *pointer*, cần thận trọng tối đa, cô lập thao tác pointer, kiểm tra hợp lệ liên tục, bổ sung cơ chế kiểm tra lỗi và ưu tiên viết code rõ ràng, dễ đọc.

Chú thích thuật ngữ: - API (Application Programming Interface): Giao diện Lập trình Ứng dụng - Structure: Cấu trúc - Pointer: Con trỏ - Linked list: Danh sách liên kết - Defensive programming: Lập trình phòng vệ - Information hiding: Che giấu thông tin

---

Ghi chú về lỗi: - Đã sửa lỗi đánh máy và sai định dạng minor trong các đoạn mã, đảm bảo tính nhất quán, dễ hiểu.

Phiên bản cải tiến của ví dụ

Ví dụ C++: Đơn giản hóa một biểu thức con trỏ phức tạp

```
quantityDiscount = rates->discounts->factors->net;
for ( rateIndex = 0; rateIndex < numRates; rateIndex++ ) {
    netRate[ rateIndex ] = baseRate[ rateIndex ] * quantityDiscount;
}
```

Bằng cách đơn giản hóa như trên, không chỉ nâng cao khả năng đọc hiểu của mã nguồn, mà còn có thể tăng hiệu suất nhờ việc đơn giản hóa thao tác với con trỏ trong vòng lặp. Tuy nhiên, như thường lệ, bạn cần đo đạc lợi ích về mặt hiệu suất trước khi đặt niềm tin tuyệt đối vào điều đó.

> **Ghi chú:** Đoạn mã gốc dạng vòng lặp phức tạp được refactor thành dạng tối ưu, dễ đọc, và tiềm năng hiệu quả hơn.

Vẽ sơ đồ mô tả con trỏ

Việc mô tả con trỏ bằng đoạn mã dễ gây nhầm lẫn. Thông thường, việc vẽ một sơ đồ sẽ giúp ích. Ví dụ, sơ đồ cho bài toán chèn node vào danh sách liên kết

(linked-list insertion) có thể như trong **Hình 13-2** bên dưới.

Sơ đồ Tham chiếu Chéo (Cross-Reference Diagrams)

Sơ đồ liên kết ban đầu có thể trở thành một phần trong tài liệu bên ngoài của chương trình. Để biết thêm về thực tiễn tài liệu hóa tốt, tham khảo Chương 32, "Tự tài liệu hóa mã nguồn" ("Self-Documenting Code").

Hình 13-2: Ví dụ sơ đồ hỗ trợ tư duy qua các bước nối lại con trỏ.

```
Initial Linkage
startNode->next
startNode
followingNode->previous
followingNode

Desired Linkage
startNode->next
startNode
newMiddleNode->previous  newMiddleNode->next
newMiddleNode
followingNode->previous
followingNode
```

Xóa con trỏ trong danh sách liên kết theo đúng thứ tự

Một vấn đề phổ biến khi thao tác với danh sách liên kết cấp phát động là giải phóng (free) con trỏ đầu tiên khi chưa lưu con trỏ tiếp theo, dẫn đến không thể truy cập các phần tử còn lại. Để tránh điều này, hãy chắc chắn rằng bạn đã giữ lại một con trỏ trỏ tới phần tử tiếp theo trước khi giải phóng phần tử hiện tại.

Cấp phát bộ nhớ dự phòng (reserve parachute)

Khi chương trình sử dụng bộ nhớ động (dynamic memory), cần tránh nguy cơ hết bộ nhớ đột ngột, khiến mất dữ liệu hoặc trạng thái chương trình. Một giải pháp là **cấp phát trước một lượng bộ nhớ dự phòng** ngay khi khởi động chương trình, chỉ sử dụng khi cạn kiệt bộ nhớ, giúp làm sạch và thoát chương trình một cách an toàn.

Đọc thêm tăng độ an toàn cho bộ nhớ

Để tìm hiểu thêm về các phương pháp an toàn xử lý con trỏ trong C, tham khảo **Writing Solid Code** (Maguire 1993).

Hủy dữ liệu rác (Shred your garbage)

Lỗi con trỏ rất khó debug do thời điểm vùng bộ nhớ mà con trỏ trỏ đến trở nên không hợp lệ là không xác định. Đôi khi bộ nhớ vẫn có vẻ hợp lệ sau khi giải

phóng, đôi khi lại bị ghi đè ngay lập tức.

Để nhất quán hơn, bạn nên **ghi đè lên vùng nhớ với dữ liệu rác ngay trước khi giải phóng**, ví dụ:

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
```

Kỹ thuật này đòi hỏi bạn duy trì một danh sách kích thước các con trỏ (lấy bằng hàm `MemoryBlockSize()`).

Đặt con trỏ về NULL sau khi giải phóng

Một lỗi phổ biến là "con trỏ treo" (dangling pointer): sử dụng một con trỏ đã bị xóa (`delete`) hoặc giải phóng (`free`). Khi đặt con trỏ về `NULL` sau khi giải phóng, đọc qua con trỏ treo thì cơ bản không khác biệt, nhưng ghi sẽ gây lỗi rõ rệt — nhờ đó giúp bạn dễ phát hiện hơn.

Ví dụ:

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
pointer = NULL;
```

Kiểm tra con trỏ trước khi xóa

Đừng xóa hoặc giải phóng một con trỏ đã bị xóa/giải phóng trước đó. Hầu hết các ngôn ngữ lập trình không tự động phát hiện vấn đề này. Đặt con trỏ về `NULL` sau khi giải phóng còn giúp bạn kiểm tra trước khi sử dụng/xóa tiếp:

```
ASSERT( pointer != NULL, "Attempting to delete null pointer " );
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
delete pointer;
pointer = NULL;
```

Theo dõi danh sách phân bổ con trỏ

Hãy duy trì một danh sách các con trỏ đã cấp phát; điều này giúp kiểm tra hợp lệ khi xóa:

```
ASSERT( pointer != NULL, "Attempting to delete null pointer " );
if ( IsPointerInList( pointer ) ) {
    memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );
    RemovePointerFromList( pointer );
    delete pointer;
    pointer = NULL;
}
else {
    ASSERT( FALSE, "Attempting to delete unallocated pointer " );
}
```

Viết hàm bao (cover routines) quản lý con trỏ tập trung

Để giảm lặp lại mã, giảm lỗi, và tránh đa dạng chiến lược xử lý con trỏ gây mâu thuẫn, hãy bọc các thao tác cấp phát, giải phóng con trỏ trong các *cover routine* (hàm bao). Ví dụ:

- **SAFE_NEW**: Gọi `new`, thêm con trỏ vào danh sách đã cấp phát, trả về con trỏ mới cho hàm gọi, đồng thời kiểm tra lỗi hết bộ nhớ ở một điểm duy nhất.
- **SAFE_DELETE**: Kiểm tra xem con trỏ có nằm trong danh sách đã cấp phát không, ghi đè vùng nhớ bằng dữ liệu rác, xóa con trỏ khỏi danh sách, gọi `delete`, và đặt con trỏ về `NULL`. Nếu không hợp lệ, ghi log hoặc báo lỗi.

Macro thực hiện có thể như sau:

```
#define SAFE_DELETE( pointer ) { \
    ASSERT( pointer != NULL, "Attempting to delete null pointer "); \
    if ( IsPointerInList( pointer ) ) { \
        memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) ); \
        RemovePointerFromList( pointer ); \
        delete pointer; \
        pointer = NULL; \
    } \
    else { \
        ASSERT( FALSE, "Attempting to delete unallocated pointer " ); \
    } \
}
```

> **Lưu ý:** Trong C++ cần bổ sung thêm phiên bản tương tự để xóa mảng (`SAFE_DELETE_ARRAY`).

Bằng cách quản lý bộ nhớ tập trung trong các hàm này, bạn cũng có thể thay đổi hành vi phù hợp với chế độ debug hoặc production.

Bạn hoàn toàn có thể áp dụng chiến lược này với `calloc` và `free` trong C, hoặc bất kỳ ngôn ngữ nào dùng con trỏ.

Ưu tiên giải pháp không dùng con trỏ

Con trỏ về bản chất khó hiểu, nhiều lỗi và thiếu tính di động. Nếu bạn có thể lựa chọn một giải pháp không dùng con trỏ thay vì, hãy sử dụng nó để tiết kiệm nhiều rắc rối.

Pointers trong C++

Đọc thêm

Tham khảo: **Effective C++, 2nd edition** (Meyers 1998) và **More Effective C++** (Meyers 1996).

Hiểu sự khác biệt giữa pointer và reference

Trong C++, cả pointer (`*`) và reference (`&`) đều gián tiếp tham chiếu đến đối tượng, nhưng:

- Reference **phải** luôn trỏ đến một đối tượng và không thể thay đổi đích sau khi khởi tạo.
- Pointer có thể trỏ tới `NULL` và trỏ tới đối tượng khác sau khi gán lại.

Sử dụng pointer cho tham số truyền bằng tham chiếu ("pass by reference"), và const reference cho tham số truyền bằng giá trị ("pass by value")

C++ mặc định truyền tham số theo giá trị, dẫn đến việc tạo bản sao đối tượng — tốn thời gian và tài nguyên với đối tượng lớn. Để tránh lãng phí này:

- Truyền bằng con trỏ khi muốn thay đổi thông tin đối tượng ("pass by reference").
- Truyền bằng reference hằng (`const reference`) nếu chỉ cần đọc mà không sửa ("pass by value").

Ví dụ:

```
void SomeRoutine(
    const LARGE_OBJECT &nonmodifiableObject,
    LARGE_OBJECT *modifiableObject
);
```

Lợi ích: Trong hàm, cú pháp gọi trực tiếp cho đối tượng bất biến (`object.member`) khác biệt với đối tượng có thể sửa đổi (`object->member`), giúp mã nguồn rõ ràng hơn.

Hạn chế của reference hằng

Nếu bạn kiểm soát toàn bộ mã nguồn, nên sử dụng `const` bất cứ khi nào có thể. Nhưng với thư viện hoặc đoạn mã bạn không kiểm soát, có thể gặp vấn đề với tham số hằng trong hàm. Cách dự phòng là truyền tham số dưới dạng reference nhưng không khai báo `const`.

Sử dụng auto_ptr

Nếu chưa quen dùng `auto_ptr`, hãy tập cho thói quen đó! `auto_ptr` tự động thu hồi bộ nhớ khi ra khỏi phạm vi, giúp giảm rò rỉ bộ nhớ so với con trỏ thông thường.

Tham khảo thảo luận trong **More Effective C++**, mục số 9.

Tìm hiểu về smart pointer

Smart pointer là giải pháp thay thế "con trỏ ngu" (dumb pointer). Chúng kiểm soát tốt hơn việc quản lý tài nguyên, hoạt động sao chép, gán, xây dựng thái hủy v.v. Hãy tham khảo **More Effective C++**, mục số 28, để xem chi tiết.

Pointers trong C

Sử dụng kiểu con trỏ rõ ràng thay cho kiểu mặc định

C cho phép dùng con trỏ dạng `char` hoặc `void` cho bất kỳ biến kiểu nào, nhưng như vậy làm mất kiểm tra kiểu của trình biên dịch. Luôn sử dụng kiểu con trỏ cụ thể để trình biên dịch cảnh báo khi dùng không đúng kiểu.

Ví dụ:

```
NodePtr = (NODE_PTR) calloc( 1, sizeof( NODE ) );
```

Hạn chế ép kiểu (type casting)

Ép kiểu tắt cảnh báo kiểm tra kiểu của trình biên dịch, tạo lỗ hổng cho lỗi lập trình phòng thủ. Nếu chương trình đòi hỏi ép kiểu nhiều, có thể cần xem lại thiết kế. Nếu không thể, hãy hạn chế ép kiểu tối đa.

Quy tắc dấu * cho truyền tham số (the asterisk rule)

Một tham số chỉ có thể trả về giá trị cho hàm gọi nếu bạn dùng dấu * trong câu lệnh gán. Điều này giúp ghi nhớ: cần có ít nhất một dấu * trong gán nếu muốn trả giá trị về. Ví dụ sau sẽ không trả về giá trị:

```
void TryToPassBackAValue( int *parameter ) {
    parameter = SOME_VALUE;
}
```

Đoạn sau sẽ trả về giá trị vì có dấu * trước tham số:

```
void TryToPassBackAValue( int *parameter ) {
    *parameter = SOME_VALUE;
}
```

Dùng sizeof() để xác định kích thước khi cấp phát

Dùng `sizeof()` thuận tiện và tránh lỗi khi tra cứu kích thước thủ công, đặc biệt với struct tự định nghĩa. `sizeof()` xác định kích thước lúc biên dịch, không gây ảnh hưởng hiệu suất, dễ bảo trì và di động.

13.3 Dữ liệu toàn cục (Global Data)

Khái niệm

Biến toàn cục (global variable) có thể truy cập ở bất cứ đâu trong chương trình. Đôi khi khái niệm này được mở rộng sai cho các biến có phạm vi lớn hơn biến cục bộ, ví dụ biến cấp lớp (class variables). Tuy nhiên, biến sở hữu bởi một lớp không đồng nghĩa là biến toàn cục.

Phần lớn lập trình viên dày dạn nhận ra việc sử dụng biến toàn cục rủi ro hơn biến cục bộ, nhưng cũng nhận thấy việc nhiều hàm có thể truy cập chung dữ liệu là hữu ích.

Dù biến toàn cục không phải lúc nào cũng gây lỗi, nhưng hầu như **không bao giờ là cách tốt nhất để lập trình**. Các chủ đề về ẩn thông tin (information hiding), tính module hóa (modularity), và sử dụng lớp được thiết kế tốt sẽ giúp chương trình lớn dễ hiểu và bảo trì hơn. Khi hiểu được giá trị này, bạn sẽ muốn viết các hàm và lớp với càng ít liên hệ đến biến toàn cục càng tốt.

Các vấn đề thường gặp với dữ liệu toàn cục

Những vấn đề về biến toàn cục thực chất quy tụ về một số ít nguyên nhân chủ yếu:

1. Thay đổi ngoài ý muốn (side effect)

Bạn có thể thay đổi giá trị biến toàn cục tại một nơi và nghĩ rằng giá trị đó không bị thay đổi ở chỗ khác—tạo ra hiệu ứng phụ ngoài ý muốn.

Ví dụ trong Visual Basic:

```
theAnswer = GetTheAnswer()
otherAnswer = GetOtherAnswer()
averageAnswer = (theAnswer + otherAnswer) / 2
```

Ở đây, nếu `GetOtherAnswer()` thay đổi giá trị `theAnswer`, giá trị trung bình sẽ sai.

2. Aliasing (đặt nhiều tên cho cùng một biến)

Aliasing xảy ra khi cùng một vùng nhớ được gọi bởi hai tên khác nhau—thường xuất hiện khi truyền biến toàn cục vào hàm như tham số, vừa dùng dưới dạng biến toàn cục vừa dùng dưới dạng tham số.

Ví dụ (Visual Basic):

```
Sub WriteGlobal( ByRef inputVar As Integer )
    inputVar = 0
    globalVar = inputVar + 5
    MsgBox( "Input Variable: " & Str( inputVar ) )
```

```
    MsgBox( "Global Variable: " & Str( globalVar ) )
End Sub

WriteGlobal( globalVar )
```

Ở đây, `globalVar` và `inputVar` thực chất là cùng một biến, nên kết quả in ra giống nhau, trái với mong đợi.

3. Vấn đề tái nhập (re-entrant) với dữ liệu toàn cục

Khi mã có thể được truy cập đồng thời bởi nhiều luồng (multithreaded code), dữ liệu toàn cục có thể bị chia sẻ giữa nhiều bản sao của chương trình hoặc luồng, dễ dẫn đến lỗi nếu không bảo vệ đồng bộ hóa.

4. Cản trở tái sử dụng mã

Để tái sử dụng mã từ chương trình này qua chương trình khác, bạn cần tách biệt code. Biến toàn cục làm phức tạp việc này vì phải phụ thuộc — lý tưởng là mỗi hàm hoặc lớp nên độc lập, dễ tái sử dụng.

> **Kết luận:** Việc lạm dụng biến toàn cục sẽ làm giảm khả năng hiểu, bảo trì, kiểm thử, tái sử dụng mã, đồng thời tăng nguy cơ lỗi phức tạp khó phát hiện.

Ghi chú về lỗi: - Bản gốc có một số chỗ thiếu định dạng rõ ràng, lặp lại, hoặc nối từ chưa chuẩn; bản dịch đã biên tập lại để tăng tính mạch lạc, không làm mất ý nghĩa gốc.

Sử Dụng Dữ Liệu Toàn Cục: Lý Do, Rủi Ro và Giải Pháp

Khả Năng Tương Thích Giữa Chương Trình Mới và Lớp Cũ

Bạn sẽ phải chỉnh sửa chương trình mới hoặc lớp cũ để chúng tương thích với nhau. Nếu lựa chọn phương án tốt, bạn sẽ chỉnh sửa lớp cũ để nó không sử dụng dữ liệu toàn cục (global data). Nếu làm vậy, lần tiếp theo khi bạn cần tái sử dụng lớp này, bạn có thể tích hợp nó vào mà không gặp khó khăn thêm. Nếu chọn cách đơn giản, bạn sẽ chỉnh sửa chương trình mới để tạo ra dữ liệu toàn cục mà lớp cũ cần sử dụng. Điều này như một loại virus; không chỉ làm ảnh hưởng đến chương trình gốc, mà còn lan sang các chương trình mới sử dụng bất kỳ lớp nào của chương trình cũ.

Vấn Đề Thứ Tự Khởi Tạo Không Xác Định Với Dữ Liệu Toàn Cục

Trật tự khởi tạo dữ liệu giữa các "translation unit" (đơn vị biên dịch, tức các tệp) trong một số ngôn ngữ, đặc biệt là C++, là không xác định. Nếu việc khởi tạo một biến toàn cục trong một tệp sử dụng biến toàn cục đã được khởi tạo ở tệp khác, giá trị của biến thứ hai sẽ không được đảm bảo, trừ khi bạn thực hiện các biện pháp rõ ràng để đảm bảo chúng được khởi tạo theo đúng thứ tự.

Vấn đề này có thể giải quyết bằng một thủ thuật mà Scott Meyers đã mô tả trong *Effective C++*, Item #47 (Meyers 1998). Tuy nhiên, sự phức tạp của giải pháp này cho thấy việc sử dụng dữ liệu toàn cục thực chất làm tăng sự phức tạp của chương trình.

Modularization và Khả Năng Quản Lý Trí Tuệ bị Tổn Hại Bởi Dữ Liệu Toàn Cục

Bản chất của việc tạo ra chương trình có quy mô lớn hơn vài trăm dòng là quản lý sự phức tạp. Cách duy nhất bạn có thể quản lý trí tuệ một chương trình lớn là phân nhỏ nó ra để chỉ phải suy nghĩ về từng phần một. Modularization (phân mảnh thành module) là công cụ mạnh mẽ nhất giúp bạn chia nhỏ chương trình.

Dữ liệu toàn cục phá vỡ năng lực phân mảnh của bạn. Nếu sử dụng dữ liệu toàn cục, liệu bạn có tập trung vào từng routine (thủ tục) một lúc? Không. Bạn phải tập trung cả vào đoạn routine đó **và** tất cả các routine khác cùng sử dụng dữ liệu toàn cục đó. Dù dữ liệu toàn cục không hoàn toàn phá hủy tính phân mảnh chương trình, nó làm suy yếu modularity, và đó là lý do đủ để tìm kiếm giải pháp khác.

Lý Do Nên Sử Dụng Dữ Liệu Toàn Cục

Một số người theo chủ nghĩa "thuần túy dữ liệu" cho rằng lập trình viên tuyệt đối không nên sử dụng dữ liệu toàn cục. Tuy nhiên, hầu hết các chương trình vẫn sử dụng "dữ liệu toàn cục" nếu bạn hiểu thuật ngữ này một cách rộng rãi. Dữ liệu trong cơ sở dữ liệu là dữ liệu toàn cục, cũng giống như dữ liệu trong tệp cấu hình (chẳng hạn Windows registry). Ngoài ra, hằng số đặt tên (named constants) cũng là dữ liệu toàn cục, chỉ là không phải biến toàn cục.

Khi sử dụng một cách kỷ luật, biến toàn cục có thể hữu ích trong một số tình huống:

Bảo Toàn Giá Trị Toàn Cục

Đôi khi bạn có dữ liệu áp dụng cho toàn bộ chương trình. Điều này có thể là một biến phản ánh trạng thái của chương trình — ví dụ: mode tương tác

(interactive) vs mode dòng lệnh (command-line), hoặc trạng thái bình thường vs trạng thái phục hồi lỗi (error-recovery). Hoặc có thể là thông tin cần thiết xuyên suốt chương trình, chẳng hạn một bảng dữ liệu mà mọi routine đều sử dụng.

Mô Phỏng Hằng Số Đặt Tên

Mặc dù C++, Java, Visual Basic và hầu hết các ngôn ngữ hiện đại đều hỗ trợ named constants, một số ngôn ngữ như Python, Perl, Awk và UNIX shell script thì không. Bạn có thể sử dụng biến toàn cục thay cho named constants nếu ngôn ngữ không hỗ trợ. Ví dụ, thay vì dùng giá trị 1 và 0, bạn có thể định nghĩa biến toàn cục `TRUE = 1` và `FALSE = 0`, hoặc thay giá trị 66 bằng `LINES_PER_PAGE = 66`. Cách này giúp dễ sửa code về sau và code cũng dễ đọc hơn.

Mô Phỏng Kiểu Liệt Kê (Enumerated Type)

Bạn cũng có thể dùng biến toàn cục để mô phỏng enumerated types trong những ngôn ngữ không hỗ trợ trực tiếp, như Python.

Đơn Giản Hoá Việc Sử Dụng Dữ Liệu Thường Xuyên

Đôi khi bạn có một biến xuất hiện ở mọi danh sách tham số của routine. Thay vì thêm vào tất cả các danh sách tham số, bạn có thể biến nó thành biến toàn cục. Tuy vậy, trong thực tế, biến này thường chỉ được truy cập bởi một nhóm routine nhất định, và bạn nên đóng gói chúng vào một class (lớp).

Loại Bỏ Tramp Data

Đôi khi bạn truyền dữ liệu cho routine hoặc class chỉ để nó lại truyền tiếp cho routine hoặc class tiếp theo. Ví dụ, bạn có thể có một đối tượng xử lý lỗi được dùng ở mỗi routine. Nếu routine nằm giữa chuỗi gọi mà không dùng tới đối tượng này, nó được gọi là "tramp data". Biến toàn cục có thể loại bỏ tramp data này.

Chỉ Sử Dụng Dữ Liệu Toàn Cục Như Là Phương Án Cuối Cùng

Trước khi dùng dữ liệu toàn cục, hãy cân nhắc một số phương án thay thế:

- Hãy khởi đầu bằng việc khai báo mọi biến là biến cục bộ, chỉ mở rộng thành biến toàn cục khi thực sự cần thiết.
- Phân biệt rõ giữa biến toàn cục thực sự và biến dữ liệu của class. Nếu biến chỉ dùng trong một nhóm routine, hãy để nó trong class, cung cấp giá trị cho các routine ngoài class qua access routine (routine truy cập), đừng truy cập trực tiếp như biến toàn cục.

- Sử dụng access routines là biện pháp chủ lực để tránh các vấn đề của dữ liệu toàn cục.

Sử Dụng Access Routines Thay Cho Dữ Liệu Toàn Cục

Bất cứ điều gì bạn có thể làm với dữ liệu toàn cục, bạn đều có thể làm tốt hơn với access routine (routine truy cập). Việc sử dụng access routine là kỹ thuật cốt lõi để hiện thực hoá abstract data type (kiểu dữ liệu trừu tượng) và đạt được information hiding (giấu thông tin).

Cho dù bạn không muốn sử dụng kiểu dữ liệu trừu tượng hoàn chỉnh, bạn vẫn có thể dùng access routine để tập trung kiểm soát dữ liệu và tự bảo vệ khỏi các thay đổi.

Lợi Ích Của Access Routine

- **Tập trung kiểm soát dữ liệu**: Nếu về sau bạn thay đổi cách implement (cài đặt) cấu trúc dữ liệu, bạn chỉ phải chỉnh trong access routine, thay vì phải sửa ở mọi nơi dữ liệu được truy cập.
- **Đảm bảo mọi tác động lên biến được kiểm soát**: Ví dụ, khi dùng access routine như `PushStack(newElement)`, bạn có thể tích hợp kiểm tra tràn stack vào routine này, đảm bảo luôn được kiểm tra mỗi lần gọi và không phải lặp lại ở từng chỗ gọi.
- **Tự động nhận được lợi ích của information hiding**: Bạn có thể thay đổi cấu trúc nội bộ của access routine mà không cần đổi giao diện với các phần còn lại của chương trình.
- **Tạo được một level of abstraction (mức trừu tượng)**: Ví dụ, thay vì dùng `if lineCount > MAX_LINES`, bạn dùng access routine `if PageFull()`, giúp mục đích code rõ ràng hơn, tăng readability (tính dễ đọc).

Cách Sử Dụng Access Routine

Hãy giấu dữ liệu trong một class. Khai báo dữ liệu với từ khóa `static` (hoặc tương đương) để đảm bảo chỉ có một instance duy nhất. Viết các routine cho phép xem hoặc chỉnh dữ liệu. Yêu cầu mọi đoạn code bên ngoài phải truy cập dữ liệu thông qua các routine này.

Ví dụ, với biến toàn cục `g_globalStatus` mô tả trạng thái tổng thể của chương trình, bạn tạo hai access routine: `globalStatusGet()` và `globalStatusSet()`, mỗi routine sẽ truy cập biến bị ẩn bên trong class.

Nếu ngôn ngữ không hỗ trợ class, bạn vẫn có thể tạo access routine để thao tác dữ liệu toàn cục, nhưng bạn cần tuân thủ quy ước code để kiểm soát truy cập.

Một số hướng dẫn chi tiết:

- **Bắt buộc mọi code phải thông qua access routine**: Đặt quy ước tất cả biến toàn cục bắt đầu bằng tiền tố `g_`, và chỉ các access routine của biến này mới được phép truy cập trực tiếp. Mọi code khác phải truy cập qua routine.
- **Không đưa tất cả dữ liệu toàn cục vào một nơi**: Khi viết access routine, hãy cân nhắc biến toàn cục đó thuộc về class nào, nhóm nó vào class cùng với các routine liên quan.
- **Sử dụng locking để kiểm soát truy cập**: Trước khi thao tác giá trị biến toàn cục, phải "check out" (khóa) rồi "check in" (mở khóa) sau khi dùng xong. Nếu đoạn khác cố truy cập trong lúc đã khóa, routine khóa/mở khóa sẽ báo lỗi hoặc assert.
- **Xây dựng access routine ở mức trừu tượng theo domain problem (bài toán thực tế)**: Access routine ở mức trừu tượng làm code dễ đọc và dễ thay đổi khi implement thay đổi.

Ví dụ so sánh:

Truy cập trực tiếp dữ liệu toàn cục	**Thông qua access routine**
node = node next	account = NextAccount(account)
node = node next	employee = NextEmployee(employee)
node = node next	rateLevel = NextRateLevel(rateLevel)
event = eventQueue[queueFront]	event = HighestPriorityEvent()
event = eventQueue[queueBack]	event = LowestPriorityEvent()

Các access routine như `NextAccount()`, `NextEmployee()` vừa ẩn chi tiết dữ liệu, vừa thể hiện ý đồ rõ ràng hơn so với thao tác trực tiếp trên data structure.

- **Giữ nhất quán mức trừu tượng**: Nếu dùng access routine để đọc dữ liệu, hãy dùng access routine cả khi ghi dữ liệu. Nếu bạn gọi `InitStack()`, `PushStack()`, cũng nên dùng `PopStack()` thay vì thao tác mảng trực tiếp.

Giảm Thiểu Rủi Ro Của Việc Dùng Dữ Liệu Toàn Cục

Thông thường, dữ liệu toàn cục là dữ liệu class của một class chưa được thiết kế hoặc cài đặt đúng. Trong một số trường hợp thật sự cần thiết, hãy dùng access routine để giảm thiểu rủi ro. Ở số ít trường hợp bắt buộc phải dùng dữ liệu toàn cục, hãy tuân thủ theo các quy tắc sau để hạn chế nguy hại:

- **Đặt quy ước tên cho biến toàn cục dễ nhận biết**: Sử dụng tiền tố (như `g_`), phân biệt rõ giữa mục đích khác nhau như biến và hằng số.

- **Tạo danh sách đầy đủ, chú thích rõ mọi biến toàn cục**: Giúp người khác dễ dàng tìm hiểu và kiểm soát code.
- **Không dùng biến toàn cục cho kết quả trung gian**: Hãy gán giá trị cuối cùng sau khi tính xong, tránh lưu trữ các trạng thái trung gian vào biến toàn cục.
- **Đừng ngụy trang việc sử dụng biến toàn cục bằng cách đóng gói tất cả vào đối tượng lớn rồi truyền đi khắp nơi**: Cách làm này chỉ làm tăng gánh nặng mà không mang lại lợi ích thực sự của encapsulation (đóng gói). Nếu đã dùng dữ liệu toàn cục, hãy làm một cách công khai, đừng che giấu bằng object "quá khổ".

Tài Liệu Tham Khảo Bổ Sung

- Steve Maguire, *Writing Solid Code*, Microsoft Press, 1993. Chương 3 thảo luận về rủi ro khi dùng pointer và nhiều mẹo thực tiễn để tránh lỗi với pointer.
- Scott Meyers, *Effective C++* (2nd Edition), Addison-Wesley, 1998 & *More Effective C++*, Addison-Wesley, 1996. Hai cuốn này cung cấp nhiều hướng dẫn, trong đó có đề cập chi tiết về quản lý bộ nhớ và sử dụng pointer an toàn, hiệu quả trong C++.

Dịch sang tiếng Việt (Phong cách học thuật, trang trọng)

Khuyến nghị về cấu trúc, biến toàn cục, và con trỏ

- **Mỗi khi cân nhắc sử dụng structure (cấu trúc), hãy xem xét liệu class (lớp) có phù hợp hơn không.**
- **Con trỏ (pointer) thường dễ gây lỗi.** Hãy bảo vệ bạn thân bằng cách sử dụng access routine (thủ tục truy cập) hoặc class cùng với các thực hành lập trình phòng thủ (defensive-programming practices).
- **Tránh sử dụng biến toàn cục (global variable), không chỉ vì chúng nguy hiểm, mà còn vì bạn có thể thay thế chúng bằng giải pháp tốt hơn.**
- **Nếu bạn không thể tránh biến toàn cục, hãy thao tác với chúng thông qua access routine.** Access routine cung cấp cho bạn mọi thứ mà biến toàn cục mang lại, thậm chí còn nhiều hơn.

Phần IV: Câu lệnh (Statements)

Nội dung phần này

- **Chương 14:** Tổ chức mã tuyến tính (Organizing Straight-Line Code)
- **Chương 15:** Sử dụng câu điều kiện (Using Conditionals)
- **Chương 16:** Kiểm soát vòng lặp (Controlling Loops)
- **Chương 17:** Cấu trúc điều khiển bất thường (Unusual Control Structures)
- **Chương 18:** Phương pháp dựa trên bảng (Table-Driven Methods)
- **Chương 19:** Các vấn đề tổng quát về điều khiển (General Control Issues)

Chương 14: Tổ chức mã tuyến tính

cc2e com/1465 Nội dung

- **14.1 Những câu lệnh cần theo thứ tự cụ thể**: trang 347
- **14.2 Những câu lệnh không phụ thuộc thứ tự**: trang 351

Chủ đề liên quan

- Chủ đề điều khiển tổng quát: Chương 19
- Mã với câu điều kiện: Chương 15
- Mã với vòng lặp: Chương 16
- Phạm vi biến và đối tượng: Mục 10.4, “Scope”

Chuyển đổi từ góc nhìn tập trung vào dữ liệu sang tập trung vào câu lệnh

Chương này chuyển từ quan điểm tập trung vào dữ liệu trong lập trình sang quan điểm tập trung vào câu lệnh. Nó giới thiệu loại luồng điều khiển (control flow) đơn giản nhất: sắp xếp các câu lệnh và khối câu lệnh theo thứ tự tuần tự.

Dù việc tổ chức mã tuyến tính (straight-line code) là nhiệm vụ khá đơn giản, vẫn tồn tại các yếu tố tinh tế về mặt tổ chức có thể ảnh hưởng tới chất lượng, độ chính xác, khả năng đọc và bảo trì của mã nguồn.

14.1 Các câu lệnh cần theo thứ tự xác định

Những câu lệnh dễ dàng nhất để sắp xếp là các câu lệnh mà thứ tự thực sự quan trọng. Ví dụ:

Ví dụ Java: Câu lệnh cần giữ đúng thứ tự

```
data = ReadData();
results = CalculateResultsFromData(data);
PrintResults(results);
```

Nếu không có điều gì bí ẩn xảy ra với đoạn mã này, các câu lệnh phải được thực thi theo đúng thứ tự đã trình bày. Dữ liệu phải được đọc trước khi có thể tính toán kết quả, và kết quả cần được tính trước khi in ra.

Khái niệm cơ bản ở đây là dependency (phụ thuộc).
- Câu lệnh thứ ba phụ thuộc vào câu lệnh thứ hai. - Câu lệnh thứ hai phụ thuộc vào câu lệnh đầu tiên.

Trong ví dụ trên, phụ thuộc đó rõ ràng thông qua tên thủ tục. Tuy nhiên, có những trường hợp phụ thuộc không hiển thị rõ ràng như sau:

Ví dụ Java: Câu lệnh cần giữ đúng thứ tự, nhưng không rõ ràng

```
revenue ComputeMonthly();
revenue ComputeQuarterly();
revenue ComputeAnnual();
```

Ở đây, việc tính toán doanh thu theo quý giả định rằng doanh thu từng tháng đã được tính trước đó. Kiến thức về kế toán hoặc thậm chí là lôgic thông thường sẽ cho bạn biết rằng cần tính doanh thu theo quý trước khi tính doanh thu thường niên.

Có sự phụ thuộc, nhưng không rõ ràng ngay từ việc đọc mã. Trong trường hợp khác, phụ thuộc còn bị "ẩn" hoàn toàn:

Ví dụ Visual Basic: Câu lệnh có phụ thuộc thứ tự bị ẩn

```
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

Giả sử `ComputeMarketingExpense()` khởi tạo các biến thành viên class mà tất cả các routine khác sử dụng để điền dữ liệu vào. Như vậy, nó cần được gọi trước các routine khác. Làm sao có thể biết được điều này chỉ bằng cách đọc mã? Bởi vì các routine này không có tham số, bạn chỉ có thể phỏng đoán rằng mỗi routine tác động tới dữ liệu thành viên class, nhưng không thể chắc chắn.

Hướng dẫn: Làm rõ dependency

> **TỔNG QUAN QUAN TRỌNG:**
> Tổ chức mã sao cho các dependency trở nên rõ ràng. Trong ví dụ Microsoft Visual Basic trên, `ComputeMarketingExpense()` không nên khởi tạo biến thành viên class. Tên của routine gợi ý rằng nó chỉ tương tự như `ComputeSalesExpense()`, `ComputeTravelExpense()`,... nhưng chỉ khác là xử lý dữ liệu marketing thay vì sales hay travel. Việc để `ComputeMarketingExpense()` khởi tạo biến thành viên là một thực hành tùy ý và nên tránh: Không có lý do hợp lý nào để thao tác khởi tạo lại phải nằm trong routine đó mà không nằm ở các routine khác. Nếu không đưa ra được lý do thỏa đáng, bạn nên viết routine khác, chẳng hạn như `InitializeExpenseData()`, để khởi tạo biến thành viên.
> **Tên routine này là một chỉ báo rõ ràng rằng nó phải được gọi trước các routine khác.**

Đặt tên routine để làm rõ dependency

Ví dụ trong Visual Basic, nếu bạn không muốn tạo một routine riêng để khởi tạo dữ liệu, ít nhất phải đặt tên routine chính xác với chức năng của nó, ví dụ: `ComputeMarketingExpenseAndInitializeMemberData()`.

Bạn có thể thấy cái tên này dường như quá dài, nhưng thực chất nó mô tả chính xác chức năng routine thực hiện và vì thế không hề tệ.
Trong thực tế, vấn đề ở chính routine, không phải ở cái tên!

Sử dụng tham số để làm rõ dependency

Nếu không có dữ liệu truyền qua lại giữa các routine, bạn không thể biết các routine có dùng chung dữ liệu không. Nếu thiết kế lại code để dữ liệu truyền giữa các routine, bạn sẽ hiểu rõ hơn thứ tự thực thi là quan trọng. Ví dụ:

```
InitializeExpenseData(expenseData)
ComputeMarketingExpense(expenseData)
ComputeSalesExpense(expenseData)
ComputeTravelExpense(expenseData)
ComputePersonnelExpense(expenseData)
DisplayExpenseSummary(expenseData)
```

Tất cả các routine cùng sử dụng `expenseData`, điều này gợi ý rằng chúng đang thao tác trên cùng dữ liệu và thứ tự thực thi có thể quan trọng.

Thậm chí, một cách tiếp cận tốt hơn là để các routine là function lấy `expenseData` làm tham số đầu vào, trả lại `expenseData` đã cập nhật:

```
expenseData = InitializeExpenseData(expenseData)
expenseData = ComputeMarketingExpense(expenseData)
expenseData = ComputeSalesExpense(expenseData)
```

```
expenseData = ComputeTravelExpense(expenseData)
expenseData = ComputePersonnelExpense(expenseData)
DisplayExpenseSummary(expenseData)
```

Ngược lại, nếu dữ liệu khác nhau không có liên hệ, thứ tự thực thi không còn quan trọng nữa:

```
ComputeMarketingExpense(marketingData)
ComputeSalesExpense(salesData)
ComputeTravelExpense(travelData)
ComputePersonnelExpense(personnelData)
DisplayExpenseSummary(marketingData, salesData, travelData, personnelData)
```

Bốn routine đầu không dùng chung dữ liệu, nên thứ tự gọi không quan trọng. Routine ở dòng thứ năm sử dụng dữ liệu tổng hợp từ bốn routine đầu, nên chỉ cần gọi sau chúng.

Ghi chú dependency không rõ ràng bằng comment

> **Hãy thử viết code không tạo ra dependency về thứ tự thực thi. Nếu điều đó là không thể, hãy ít nhất làm cho dependency trở nên rõ ràng. Nếu bạn vẫn lo lắng dependency chưa đủ explicit, hãy bổ sung comment vào code.**
>
> **Ghi chú các dependency không rõ ràng là một phần của việc ghi chú giả định mã nguồn (coding assumptions), một yếu tố quan trọng để đảm bảo mã nguồn dễ bảo trì và sửa đổi.**

Ví dụ:

```
' Tính toán các khoản chi phí. Mỗi routine truy cập dữ liệu thành viên expenseData.
' DisplayExpenseSummary phải được gọi cuối cùng vì nó phụ thuộc vào dữ liệu tính bởi các rou
InitializeExpenseData
ComputeMarketingExpense
ComputeSalesExpense
ComputeTravelExpense
ComputePersonnelExpense
DisplayExpenseSummary
```

Mặc dù cách này không làm dependency trở nên rõ ràng bằng mặt cú pháp code, nhưng vẫn tốt hơn không làm gì cả khi bạn không thể cải thiện code.

Kiểm tra sự phụ thuộc bằng assertion hoặc xử lý lỗi

Trong trường hợp code đặc biệt quan trọng, bạn có thể sử dụng biến trạng thái và mã xử lý lỗi hoặc assertion để kiểm soát các dependency tuyến tính:
- Trong constructor của class, khởi tạo biến thành viên `isExpenseDataInitialized` là false. - Trong `InitializeExpenseData()`, đặt biến này thành true. - Các

function phụ thuộc vào việc khởi tạo `expenseData` sẽ kiểm tra biến này trước khi thực thi tiếp.

Tùy vào mức độ phức tạp, bạn có thể cần các biến như `isMarketingExpenseComputed`, `isSalesExpenseComputed`,...

Lưu ý:
Kỹ thuật này tạo ra biến mới, mã khởi tạo mới và thêm code kiểm tra lỗi, làm tăng độ phức tạp cũng như khả năng phát sinh lỗi phụ. Do đó, cần cân nhắc lợi ích đối với chi phí về độ phức tạp và nguy cơ lỗi đi kèm.

14.2 Những câu lệnh không phụ thuộc thứ tự

Ở một số trường hợp, có vẻ như thứ tự của một vài câu lệnh hoặc khối code không thực sự quan trọng. Tuy nhiên, việc sắp xếp có thể ảnh hưởng đến khả năng đọc, hiệu năng, và khả năng bảo trì.

Nguyên tắc chung: *Principle of Proximity* – Giữ các hành động liên quan gần nhau.

Tổ chức mã theo hướng đọc từ trên xuống dưới

Quy tắc tổng quát:
Làm cho chương trình có thể đọc từ trên xuống dưới. Các chuyên gia đồng ý rằng mã lệnh được sắp xếp theo thứ tự này sẽ tăng đáng kể tính dễ đọc.

Chỉ đảm bảo dòng lệnh thực thi từ trên xuống dưới là chưa đủ. Nếu ai đó phải lục tìm toàn bộ chương trình để tìm thông tin cần thiết, bạn nên tái tổ chức lại mã.

Ví dụ:

C++: Ví dụ về sắp xếp mã kém

```
MarketingData marketingData;
SalesData salesData;
TravelData travelData;
travelData ComputeQuarterly();
salesData ComputeQuarterly();
marketingData ComputeQuarterly();
salesData ComputeAnnual();
marketingData ComputeAnnual();
travelData ComputeAnnual();
salesData Print();
travelData Print();
marketingData Print();
```

Nếu bạn muốn biết marketingData được tính toán như thế nào, bạn phải bắt đầu từ dòng cuối cùng và lần theo tất cả chỗ có liên quan đến marketingData lên tới dòng đầu.
Các biến được "sử dụng" rải rác, làm khó khăn cho việc theo dõi quá trình xử lý.

C++: Ví dụ về mã tuần tự, dễ đọc

```
MarketingData marketingData;
marketingData ComputeQuarterly();
marketingData ComputeAnnual();
marketingData Print();

SalesData salesData;
salesData ComputeQuarterly();
salesData ComputeAnnual();
salesData Print();

TravelData travelData;
travelData ComputeQuarterly();
travelData ComputeAnnual();
travelData Print();
```

Ở đây, mỗi đối tượng được tham chiếu gần nhau ("localize"), thời gian sống (live) của mỗi đối tượng bị giới hạn trong ít dòng code. Điều này cũng giúp bạn hình dung dễ dàng việc tách riêng thành các routine cho marketing, sales, và travel.

Nhóm các câu lệnh liên quan

Cần nhóm các câu lệnh liên quan lại với nhau – có thể vì chúng thao tác trên cùng dữ liệu, thực hiện tác vụ tương tự, hoặc phụ thuộc vào nhau về mặt thứ tự thực thi.

Một cách kiểm tra đơn giản:
In mã nguồn ra, kẻ khung quanh các nhóm câu lệnh liên quan. Nếu các khung không chồng lấn nhau, tức là bạn tổ chức các nhóm tốt. Nếu chồng lấn, bạn nên tổ chức lại.

Nếu bạn nhận thấy các câu lệnh nhóm rất chặt với nhau và không liên quan tới phần trước/sau, hãy refactor thành routine riêng.

---

cc2e com/1472 Checklist: Tổ chức mã tuyến tính

- Code đã làm rõ các dependency giữa các câu lệnh chưa?
- Tên của routine đã làm rõ dependency chưa?

- Tham số truyền vào routine đã làm rõ dependency chưa?
- Comment có mô tả các dependency chưa rõ không?
- Có sử dụng biến kiểm soát (housekeeping variables) để kiểm tra dependency tuần tự ở đoạn code quan trọng chưa?
- Code có được đọc theo kiểu từ trên xuống dưới không?
- Các câu lệnh liên quan đã được nhóm sát nhau chưa?
- Nhóm câu lệnh khá độc lập đã chuyển thành routine riêng chưa?

Tóm tắt các điểm chính

- Nguyên tắc mạnh mẽ nhất khi tổ chức code tuyến tính là dựa vào thứ tự dependency.
- Dependency phải được làm rõ thông qua tên routine, tham số, comment, và nếu code quan trọng, các biến kiểm soát.
- Nếu không có dependency về thứ tự, hãy giữ các câu lệnh liên quan càng gần nhau càng tốt.

Chương 15: Sử dụng câu điều kiện (Conditionals)

cc2e com/1538 Nội dung

- **15.1 if Statements**: trang 355
- **15.2 case Statements**: trang 361

Chủ đề liên quan

- Làm chủ nesting sâu: Mục 19.4
- Các vấn đề điều khiển tổng quát: Chương 19
- Mã với vòng lặp: Chương 16
- Mã tuyến tính: Chương 14
- Quan hệ giữa kiểu dữ liệu và cấu trúc điều khiển: Mục 10.7

Câu điều kiện (Conditional)

Một conditional (câu điều kiện) là câu lệnh kiểm soát việc thực thi các câu lệnh khác; việc thực thi các câu lệnh này được "điều kiện hóa" dựa trên các câu lệnh như if, else, case, và switch.

Mặc dù về mặt logic, các cấu trúc điều khiển vòng lặp như while và for cũng có thể coi là conditional, nhưng theo thông lệ, chúng được xét riêng. Chương 16 sẽ đề cập đến while và for.

15.1 Câu lệnh if (if Statements)

Dựa vào ngôn ngữ lập trình, bạn có thể sử dụng một trong các kiểu if sau đây: Đơn giản nhất là if hoặc if-then. Phức tạp hơn là if-then-else, và cao nhất là chuỗi if-then-else-if.

If-then đơn giản

Hướng dẫn khi viết if:

- **Viết trước lối đi "bình thường" (nominal path) qua mã, sau đó là các trường hợp bất thường (unusual case).**
  -> Đảm bảo đường đi thông thường của chương trình (nominal path) là rõ ràng, tránh để các trường hợp hiếm gặp che khuất luồng thực thi bình thường. Điều này quan trọng cho cả khả năng đọc và hiệu năng.
- **Đảm bảo so sánh đúng về dấu bằng.**
  -> Sử dụng > thay vì >= hay < thay vì <= cũng giống như sai lệch một đơn vị (off-by-one error) khi truy cập mảng hoặc tính chỉ mục vòng lặp. Hãy cân nhắc kỹ trường hợp bằng (=) để không mắc lỗi này.
- **Đặt trường hợp thông thường sau if, không phải sau else.**
  -> Trường hợp bạn kỳ vọng xử lý nhất hãy đặt đầu tiên, càng sát câu lệnh if càng tốt.

Ví dụ về mã xử lý lỗi không hệ thống:

```
OpenFile( inputFile, status )
If ( status = Status_Error ) Then
    ' Trường hợp lỗi
    errorType = FileOpenError
Else
    ' Trường hợp bình thường
    ReadFile( inputFile, fileData, status )
    If ( status = Status_Success ) Then
        ' Trường hợp bình thường
        SummarizeFileData( fileData, summaryData, status )
        If ( status = Status_Error ) Then
            ' Trường hợp lỗi
            errorType = ErrorType_DataSummaryError
        Else
            ' Trường hợp bình thường
            PrintSummary( summaryData )
            SaveSummaryData( summaryData, status )
            If ( status = Status_Error ) Then
                ' Trường hợp lỗi
                errorType = ErrorType_SummarySaveError
            Else
                ' Trường hợp bình thường
```

```
                UpdateAllAccounts()
                EraseUndoFile()
                errorType = ErrorType_None
            End If
        End If
    Else
        errorType = ErrorType_FileReadError
    End If
End If
```

Mã trên khó theo dõi vì lẫn lộn giữa trường hợp bình thường và trường hợp lỗi, dẫn đến đường đi thông thường bị che lấp. Hơn nữa, do lỗi được xử lý lúc thì trong if, lúc thì trong else khiến khó xác định đường đi chính xác.

Mã được viết lại hợp lý hơn như sau:

```
OpenFile( inputFile, status )
If ( status = Status_Success ) Then
    ' Trường hợp bình thường
    ReadFile( inputFile, fileData, status )
    If ( status = Status_Success ) Then
        ' Trường hợp bình thường
        SummarizeFileData( fileData, summaryData, status )
        If ( status = Status_Success ) Then
            ' Trường hợp bình thường
            PrintSummary( summaryData )
            SaveSummaryData( summaryData, status )
            If ( status = Status_Success ) Then
                ' Trường hợp bình thường
                UpdateAllAccounts()
                EraseUndoFile()
                errorType = ErrorType_None
            Else
                ' Trường hợp lỗi
                errorType = ErrorType_SummarySaveError
            End If
        Else
            ' Trường hợp lỗi
            errorType = ErrorType_DataSummaryError
        End If
    Else
        ' Trường hợp lỗi
        errorType = ErrorType_FileReadError
    End If
Else
    ' Trường hợp lỗi
    errorType = ErrorType_FileOpenError
```

```
End If
```

Trong ví dụ viết lại, đường đi chính qua các phép kiểm tra if là đường đi bình thường, còn các trường hợp lỗi được xử lý sau cùng. Đây là dấu hiệu của mã xử lý lỗi hợp lý.

> Bản dịch đã được kiểm tra lại đầy đủ về thuật ngữ, mạch lạc và logic chương trình, giữ nguyên code gốc và điều chỉnh nhỏ về diễn đạt cho phù hợp với ngữ cảnh khoa học máy tính.

Danh sách đầy đủ các phương pháp

Để có danh sách đầy đủ về các phương pháp có thể sử dụng, vui lòng xem "Summary of Techniques for Reducing Deep Nesting" (Tóm tắt các kỹ thuật giảm lồng ghép sâu) tại Mục 19.4.

Sử dụng mệnh đề if với câu lệnh có ý nghĩa

Đôi khi bạn có thể thấy đoạn mã như ví dụ sau, trong đó mệnh đề if không thực hiện gì:

Ví dụ Java về mệnh đề if rỗng

```
if ( SomeTest )
    ;
else {
    // do something
}
```

> Lỗi định dạng: "CODING HORROR" không cần thiết, đã được loại bỏ.

Các lập trình viên có kinh nghiệm thường tránh viết mã như vậy, bởi họ không muốn tốn công viết thêm dòng rỗng cho phần if và một dòng else không cần thiết. Cách viết này trông vụng về và rất dễ cải tiến bằng cách phủ định điều kiện trong mệnh đề if, chuyển mã từ phần else sang if. Kết quả sau khi sửa sẽ như sau:

Ví dụ Java sau khi chuyển mệnh đề if rỗng

```
if ( !someTest ) {
    // do something
}
```

Xem xét mệnh đề else

Nếu bạn nghĩ mình chỉ cần một câu lệnh if đơn giản, hãy tự hỏi liệu bạn có thực sự không cần mệnh đề if-then-else hay không. Một phân tích điển hình của General Motors đã phát hiện rằng từ 50% đến 80% các mệnh đề if nên có mệnh đề else (Elshoff 1976).

- Một lựa chọn là vẫn mã hóa mệnh đề else – với câu lệnh rỗng nếu cần thiết – để thể hiện rằng bạn đã cân nhắc trường hợp còn lại.
- Tuy nhiên, việc mã hóa else rỗng chỉ để chứng minh rằng bạn đã xem xét trường hợp đó có thể là quá mức cần thiết, nhưng ít nhất bạn nên cân nhắc đến trường hợp else.
- Khi bạn có một kiểm tra if không đi kèm else, trừ khi lý do rất rõ ràng, hãy bổ sung chú thích để giải thích vì sao else không cần thiết.

Ví dụ Java: Mệnh đề else được chú thích rõ ràng

```
// if color is valid
if ( COLOR_MIN <= color && color <= COLOR_MAX ) {
    // do something
}
else {
    // else color is invalid
    // screen not written to -- safely ignore command
}
```

Hãy kiểm tra tính chính xác của mệnh đề else. Khi kiểm thử mã, bạn có thể nghĩ chỉ cần kiểm thử phần if. Tuy nhiên, nếu có thể kiểm thử mệnh đề else, hãy đảm bảo bạn đã làm điều đó.

Lỗi phổ biến: Đảo ngược nhầm mệnh đề if và else (hoặc logic của điều kiện). Hãy kiểm tra lại mã để tránh lỗi này.

Chuỗi if-then-else

Trong các ngôn ngữ không hỗ trợ câu lệnh case, hoặc chỉ hỗ trợ một phần, bạn sẽ thường phải sử dụng các chuỗi if-then-else. Ví dụ, để phân loại một ký tự, bạn có thể viết mã như sau:

Ví dụ C++: Chuỗi if-then-else để phân loại ký tự

```
if ( inputCharacter < SPACE ) {
    characterType = CharacterType_ControlCharacter;
}
else if (
    inputCharacter == ' ' ||
    inputCharacter == ',' ||
```

```
    inputCharacter == ' ' ||
    inputCharacter == '!' ||
    inputCharacter == '(' ||
    inputCharacter == ')' ||
    inputCharacter == ':' ||
    inputCharacter == ';' ||
    inputCharacter == '?' ||
    inputCharacter == '-'
) {
    characterType = CharacterType_Punctuation;
}
else if ( '0' <= inputCharacter && inputCharacter <= '9' ) {
    characterType = CharacterType_Digit;
}
else if (
    ( 'a' <= inputCharacter && inputCharacter <= 'z' ) ||
    ( 'A' <= inputCharacter && inputCharacter <= 'Z' )
) {
    characterType = CharacterType_Letter;
}
```

Các hướng dẫn khi viết chuỗi if-then-else:

- **Đơn giản hóa các kiểm tra phức tạp với hàm boolean:** Việc các điều kiện phức tạp sẽ làm mã khó đọc hơn. Hãy thay thế các kiểm tra này bằng các hàm boolean rõ nghĩa.

Ví dụ C++: Chuỗi if-then-else sử dụng hàm boolean

```
if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
```

- **Đặt trường hợp phổ biến nhất lên đầu:** Việc này giảm số lượng kiểm tra phải thực hiện đối với các trường hợp thông thường, đồng thời giúp mã dễ đọc hơn.

Ví dụ C++: Kiểm tra trường hợp phổ biến nhất trước

```
if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
```

- **Đảm bảo xử lý hết các trường hợp:** Mã hóa một mệnh đề else cuối cùng với thông báo lỗi hoặc assertion để bắt các trường hợp ngoài dự kiến. Thông báo này hướng tới lập trình viên thay vì người dùng.

Ví dụ C++: Sử dụng mệnh đề else cuối cùng để kiểm tra lỗi

```
if ( IsLetter( inputCharacter ) ) {
    characterType = CharacterType_Letter;
}
else if ( IsPunctuation( inputCharacter ) ) {
    characterType = CharacterType_Punctuation;
}
else if ( IsDigit( inputCharacter ) ) {
    characterType = CharacterType_Digit;
}
else if ( IsControl( inputCharacter ) ) {
    characterType = CharacterType_ControlCharacter;
}
else {
    DisplayInternalError( "Unexpected type of character detected" );
}
```

- **Thay thế chuỗi if-then-else bằng câu lệnh khác nếu ngôn ngữ hỗ trợ:** Một số ngôn ngữ như Microsoft Visual Basic, Ada cung cấp câu lệnh case hỗ trợ chuỗi, enum và các hàm logic. Nên sử dụng chúng vì vừa dễ viết vừa dễ đọc hơn so với chuỗi if-then-else.

Ví dụ Visual Basic: Sử dụng câu lệnh case thay vì chuỗi if-then-else

```
Select Case inputCharacter
    Case "a" To "z"
        characterType = CharacterType_Letter
    Case " ", ",", " ", "!", "(", ")", ":", ";", "?", "-"
```

```
        characterType = CharacterType_Punctuation
    Case "0" To "9"
        characterType = CharacterType_Digit
    Case FIRST_CONTROL_CHARACTER To LAST_CONTROL_CHARACTER
        characterType = CharacterType_Control
    Case Else
        DisplayInternalError( "Unexpected type of character detected" )
End Select
```

15.2 Câu lệnh case (case Statements)

Câu lệnh case hoặc switch là cấu trúc kiểm soát mà các ngôn ngữ lập trình hỗ trợ ở mức độ khác nhau. C++ và Java chỉ hỗ trợ case cho các kiểu số thứ tự (ordinal types), còn Visual Basic hỗ trợ cả dãy giá trị và cung cấp các ký hiệu viết tắt mạnh mẽ. Nhiều ngôn ngữ kịch bản không hỗ trợ câu lệnh case.

Hướng dẫn sử dụng câu lệnh case hiệu quả

Chọn thứ tự sắp xếp phù hợp cho các trường hợp trong case

- **Sắp xếp theo alphabet hoặc số:** Nếu các trường hợp đều quan trọng như nhau, hãy sắp xếp theo abc hoặc số để dễ tra cứu.
- **Đặt trường hợp thông thường lên đầu:** Nếu có một trường hợp thường xuất hiện kèm nhiều trường hợp ngoại lệ, hãy đặt trường hợp thông thường đầu tiên, đồng thời chú thích đây là trường hợp bình thường.
- **Sắp xếp theo tần suất:** Đặt những trường hợp thường xuyên được thực hiện lên trên cùng để tăng hiệu quả và giúp lập trình viên dễ tìm kiếm.

Một số lưu ý khi sử dụng câu lệnh case

- **Giữ cho hành động của từng case đơn giản:** Nên giữ đoạn mã cho từng case ngắn gọn để tăng tính rõ ràng. Nếu trường hợp cần thực hiện nhiều tác vụ, hãy viết hàm xử lý riêng rồi gọi tới trong case.
- **Không tạo biến giả chỉ để dùng cho case:** Nếu dữ liệu mà bạn muốn phân loại không phù hợp dễ dàng với case, hãy dùng chuỗi if-then-else thay thế. Biến giả khiến mã dễ nhầm lẫn và khó bảo trì.

Ví dụ Java: Sử dụng biến giả — Thực tiễn không tốt

```
action = userCommand[ 0 ];
switch ( action ) {
    case 'c':
        Copy();
```

```
            break;
        case 'd':
            DeleteCharacter();
            break;
        case 'f':
            Format();
            break;
        case 'h':
            Help();
            break;
        default:
            HandleUserInputError( ErrorType_InvalidUserCommand );
}
```

Thay vì tạo biến action bằng cách lấy ký tự đầu tiên của chuỗi lệnh, hãy dùng chuỗi if-then-else-if để kiểm tra toàn bộ chuỗi:

Ví dụ Java: Sử dụng if-then-else thay vì biến giả cho case

```
if ( UserCommand.equals( COMMAND_STRING_COPY ) ) {
    Copy();
}
else if ( UserCommand.equals( COMMAND_STRING_DELETE ) ) {
    DeleteCharacter();
}
else if ( UserCommand.equals( COMMAND_STRING_FORMAT ) ) {
    Format();
}
else if ( UserCommand.equals( COMMAND_STRING_HELP ) ) {
    Help();
}
else {
    HandleUserInputError( ErrorType_InvalidCommandInput );
}
```

- **Chỉ sử dụng mệnh đề default cho các trường hợp hợp lệ:** Đôi khi, bạn chỉ còn một trường hợp và có thể thấy nên đặt vào default, nhưng như vậy sẽ mất tài liệu tự động do nhãn case cung cấp và khả năng phát hiện lỗi ở mệnh đề default. Luôn để default kiểm tra các trường hợp lỗi.

Ví dụ Java: Sử dụng default để phát hiện lỗi — Thực tiễn tốt

```
switch ( commandShortcutLetter ) {
    case 'a':
        PrintAnnualReport();
        break;
    case 'p':
```

```
        // no action required, but case was considered
        break;
    case 'q':
        PrintQuarterlyReport();
        break;
    case 's':
        PrintSummaryReport();
        break;
    default:
        DisplayInternalError( "Internal Error 905: Call customer support" );
}
```

Nếu mệnh đề default được sử dụng cho lý do khác ngoài kiểm tra lỗi, hãy chắc chắn mọi biến có thể vào case đều là hợp lệ. Nếu không, hãy chỉnh lại mã để default thực sự kiểm tra các lỗi chưa lường trước.

- **Trong C, C++ hoặc Java, tránh rơi xuống cuối mỗi case:** Các ngôn ngữ này không tự động thoát khỏi mỗi case, bạn phải sử dụng lệnh break, nếu không sẽ chuyển sang case tiếp theo—a điều nên tránh vì dễ gây lỗi.

Ví dụ C++: Lạm dụng case statement

```
switch ( InputVar ) {
    case 'A':
        if ( test ) {
            // statement 1
            // statement 2
        }
    case 'B':
        // statement 3
        // statement 4
        break;
}
```

> Chú ý: Không nên để control structure chồng lấn theo cách này.

- **Trong C++, ghi chú rõ ràng các trường hợp cố tình rơi qua case:** Nếu bạn thực sự cần trường hợp này, hãy chú thích rõ ở vị trí rơi xuống.

Ví dụ C++: Chú thích rõ ràng cho trường hợp rơi xuống

```
switch ( errorDocumentationLevel ) {
    case DocumentationLevel_Full:
        DisplayErrorDetails( errorNumber );
        // FALLTHROUGH -- Full documentation also prints summary comments
```

```
    case DocumentationLevel_Summary:
        DisplayErrorSummary( errorNumber );
        // FALLTHROUGH -- Summary documentation also prints error number
    case DocumentationLevel_NumberOnly:
        DisplayErrorNumber( errorNumber );
        break;
    default:
        DisplayInternalError( "Internal Error 905: Call customer support" );
}
```

Nhìn chung, việc để code rơi xuống case khác là điều nên tránh bởi sẽ dễ gây lỗi khi bảo trì hoặc mở rộng chức năng.

CHECKLIST: Sử dụng Conditionals

if-then Statements

- Đường đi chính của mã có rõ ràng không?
- Các kiểm tra if-then có phân nhánh đúng khi bằng nhau không?
- Mệnh đề else có xuất hiện và được chú thích không?
- Mệnh đề else có chính xác không?
- Mệnh đề if và else có được sử dụng đúng thứ tự, không bị đảo ngược khối lệnh không?
- Trường hợp thường gặp có nằm trong if thay vì else không?

if-then-else-if Chains

- Các kiểm tra phức tạp có được bao gói bằng hàm boolean không?
- Các trường hợp phổ biến nhất có được kiểm tra trước không?
- Các trường hợp có được bao phủ đầy đủ không?
- Chuỗi if-then-else-if có phải là cách tốt nhất, hay nên dùng case statement?

case Statements

- Các trường hợp có được sắp xếp hợp lý không?
- Các thao tác với từng trường hợp có ngắn gọn, gọi hàm riêng nếu cần không?
- case statement có dùng biến thực sự, không dùng biến giả không?
- Việc sử dụng default có hợp lý không?
- default có dùng để phát hiện và báo các trường hợp không mong muốn không?
- Ở C, C++ hoặc Java, cuối mỗi case có lệnh break không?

Những điểm chính

- **Đối với if-else đơn giản**, chú ý đến thứ tự của if và else, đặc biệt khi xử lý nhiều lỗi. Đảm bảo đường đi chính rõ ràng.
- **Đối với chuỗi if-then-else và case statements**, hãy chọn thứ tự sắp xếp tối ưu hóa khả năng đọc.
- **Để phát hiện lỗi**, sử dụng mệnh đề default trong case statement hoặc else cuối cùng trong if-then-else chain.
- **Không phải mọi cấu trúc điều khiển đều như nhau.** Chọn cấu trúc điều khiển phù hợp nhất cho từng đoạn mã.

Chương 16: Kiểm soát vòng lặp (Controlling Loops)

Nội dung chương

Chủ đề liên quan

- Giảm lồng ghép sâu: Mục 19.4
- Vấn đề kiểm soát chung: Chương 19
- Code với conditionals: Chương 15
- Code tuyến tính: Chương 14
- Quan hệ giữa cấu trúc điều khiển và kiểu dữ liệu: Mục 10.7

16.1 Chọn loại vòng lặp phù hợp

Loop là thuật ngữ không chính thức, chỉ bất kỳ cấu trúc kiểm soát lặp nào khiến chương trình thực thi lại một khối mã nhiều lần. Các loại vòng lặp thường gặp gồm for, while, và do-while trong C++ và Java, cùng For-Next, While-Wend, và Do-Loop-While trong Microsoft Visual Basic.

Việc sử dụng vòng lặp là một trong những phần phức tạp nhất khi lập trình; biết cách và thời điểm chọn loại vòng lặp nào là yếu tố then chốt để xây dựng phần mềm chất lượng cao.

Trong hầu hết các ngôn ngữ, bạn sẽ gặp một số loại vòng lặp sau:

- **Vòng lặp đếm (counted loop):** Được thực hiện một số lần xác định trước, ví dụ một lần cho mỗi nhân viên.
- **Vòng lặp đánh giá liên tục (continuously evaluated loop):** Không biết trước số lần lặp, kiểm tra điều kiện hoàn thành ở mỗi vòng lặp, ví dụ lặp khi còn tiền, cho tới khi người dùng chọn thoát, hoặc tới khi gặp lỗi.
- **Vòng lặp vô tận (endless loop):** Thực thi mãi mãi sau khi bắt đầu, thường thấy trong các hệ nhúng như máy trợ tim, lò vi sóng, hay điều khiển hành trình.
- **Vòng lặp theo iterator (iterator loop):** Thực hiện một lần cho từng phần tử trong container class.

Các loại vòng lặp trên khác nhau chủ yếu bởi tính linh hoạt – liệu vòng lặp thực hiện số lần xác định trước hay đánh giá điều kiện lặp ở mỗi lần lặp.

Một đặc điểm khác biệt nữa là vị trí kiểm tra điều kiện hoàn thành: Bạn có thể đặt kiểm tra này ở đầu, giữa, hoặc cuối vòng lặp. Đặc điểm này cho biết liệu thân vòng lặp có chắc chắn được thực thi ít nhất một lần hay không. Nếu kiểm tra ở đầu, thân vòng lặp có thể không thực thi. Nếu kiểm tra ở cuối, thân vòng lặp chắc chắn chạy ít nhất một lần.

Bản dịch đã được kiểm tra tính nhất quán thuật ngữ, độ mạch lạc, cũng như sửa những lỗi đánh máy/định dạng để đảm bảo dễ hiểu nhất cho người đọc chuyên ngành.

Sự linh hoạt và vị trí kiểm tra quyết định loại vòng lặp nên lựa chọn làm cấu trúc điều khiển

Sự linh hoạt và vị trí kiểm tra (test location) sẽ xác định loại vòng lặp thích hợp để sử dụng làm cấu trúc điều khiển. Bảng 16-1 liệt kê các loại vòng lặp trong một số ngôn ngữ lập trình và mô tả về mức độ linh hoạt cũng như vị trí kiểm tra của từng loại.

Bảng 16-1: Các loại vòng lặp

Ngôn ngữ	Loại vòng lặp	Mức độ linh hoạt	Vị trí kiểm tra
Visual Basic	For-Next	cứng nhắc (rigid)	Đầu (beginning)
	While-Wend	linh hoạt (flexible)	Đầu
	Do-Loop-While	linh hoạt	Đầu hoặc cuối
	For-Each	cứng nhắc	Đầu
C, C++, C#, Java	for	linh hoạt	Đầu

Ngôn ngữ	Loại vòng lặp	Mức độ linh hoạt	Vị trí kiểm tra
	while	linh hoạt	Đầu
	do-while	linh hoạt	Cuối
	foreach*	cứng nhắc	Đầu

*Chỉ có trong C# tại thời điểm biên soạn. Dự kiến sẽ có mặt ở các ngôn ngữ khác, bao gồm Java.

Khi nào sử dụng vòng lặp while

Các lập trình viên mới thường nghĩ rằng vòng lặp while được đánh giá liên tục và kết thúc ngay khi điều kiện while trở nên sai, bất kể câu lệnh nào trong vòng lặp đang được thực thi (Curtis et al., 1986). Dù không hoàn toàn linh hoạt như vậy, vòng lặp while là một lựa chọn vòng lặp linh hoạt. Nếu bạn không biết trước chính xác số lần lặp, hãy sử dụng vòng lặp while. Ngược với quan niệm của một số người mới, điều kiện thoát vòng lặp chỉ được kiểm tra mỗi lần bắt đầu một lượt lặp, và vấn đề chính là quyết định kiểm tra ở đầu hay cuối vòng lặp.

Vòng lặp kiểm tra ở đầu

Đối với vòng lặp kiểm tra ở đầu, bạn có thể dùng vòng lặp while trong C++, C#, Java, Visual Basic hoặc mô phỏng trong các ngôn ngữ khác.

Vòng lặp kiểm tra ở cuối

Đôi khi bạn cần một vòng lặp linh hoạt, nhưng vòng lặp đó cần được thực thi ít nhất một lần. Khi đó, bạn có thể sử dụng vòng lặp kiểm tra ở cuối, ví dụ: do-while trong C++, C#, Java; Do-Loop-While trong Visual Basic; hoặc mô phỏng bằng các kỹ thuật tương đương ở ngôn ngữ khác.

Khi nào sử dụng Loop-With-Exit Loop

Loop-with-exit loop (vòng lặp có điều kiện thoát ở giữa) là vòng lặp mà điều kiện thoát xuất hiện ở giữa, không phải ở đầu cũng không phải ở cuối. Loại này có sẵn trong Visual Basic, và có thể mô phỏng bằng cách sử dụng các cấu trúc như while cùng break trong C++, C, Java, hoặc với goto trong các ngôn ngữ khác.

Vòng lặp Loop-With-Exit thông thường

Một vòng lặp loop-with-exit thường gồm phần bắt đầu vòng lặp, thân vòng lặp (bao gồm điều kiện thoát), và kết thúc vòng lặp, như ví dụ sau trong Visual Basic:

```
Do
    Statements
    If ( some exit condition ) Then Exit Do
    More statements
Loop
```

Thông thường, sử dụng loop-with-exit khi kiểm tra ở đầu hoặc cuối vòng lặp dẫn đến việc phải lặp lại code (loop-and-a-half). Dưới đây là ví dụ trong C++ cho trường hợp này:

```
// Compute scores and ratings
score = 0;
// Những dòng này xuất hiện ở đây
GetNextRating( &ratingIncrement );
rating = rating + ratingIncrement;
while ( ( score < targetScore ) && ( ratingIncrement != 0 ) ) {
    GetNextScore( &scoreIncrement );
    score = score + scoreIncrement;
    // ... và được lặp lại ở đây
    GetNextRating( &ratingIncrement );
    rating = rating + ratingIncrement;
}
```

Ở ví dụ này, hai dòng code trên đầu bị lặp lại ở cuối vòng lặp while. Trong quá trình chỉnh sửa, rất dễ quên cập nhật song song hai nhóm dòng này, dẫn đến lỗi. Dưới đây là cách viết rõ ràng, bảo trì dễ hơn:

```
// Compute scores and ratings. The code uses an infinite loop
// and a break statement to emulate a loop-with-exit loop
score = 0;
while ( true ) {
    GetNextRating( &ratingIncrement );
    rating = rating + ratingIncrement;
    // Đây là điều kiện thoát vòng lặp
    if ( !( ( score < targetScore ) && ( ratingIncrement != 0 ) ) ) {
        break;
    }
    GetNextScore( &scoreIncrement );
    score = score + scoreIncrement;
}
```

Phiên bản Visual Basic tương đương:

```
' Compute scores and ratings
score = 0
Do
    GetNextRating( ratingIncrement )
    rating = rating + ratingIncrement
    If ( Not ( score < targetScore And ratingIncrement <> 0 ) ) Then Exit Do
```

```
    GetNextScore( ScoreIncrement )
    score = score + scoreIncrement
Loop
```

Lưu ý khi sử dụng loại vòng lặp này: - Đặt tất cả điều kiện thoát ở một chỗ. Phân tán các điều kiện thoát sẽ làm dễ dàng bỏ sót một điều kiện trong quá trình debug, chỉnh sửa hoặc kiểm thử. - Sử dụng chú thích (comment) để làm rõ ý định. Khi bạn sử dụng loop-with-exit trong ngôn ngữ không hỗ trợ trực tiếp, hãy chú thích rõ ràng để dễ hiểu.

Loop-with-exit loop là cấu trúc điều khiển một-lối-vào, một-lối-ra (structured control construct), được ưu tiên sử dụng nhất (Software Productivity Consortium, 1989). Các nghiên cứu cho thấy loại vòng lặp này dễ hiểu hơn so với các loại vòng lặp khác; sinh viên đạt điểm hiểu bài cao hơn 25% khi sử dụng loop-with-exit so với các loại kiểm tra ở đầu/cuối (Soloway, Bonar, and Ehrlich 1983). Kết luận là cấu trúc loop-with-exit phù hợp với cách suy nghĩ tự nhiên của con người về kiểm soát lặp.

Trên thực tế, loại vòng lặp này chưa phổ biến rộng rãi. Tuy nhiên, nó là một kỹ thuật hữu ích bạn nên biết và cân nhắc sử dụng cẩn thận trong công việc lập trình.

Loop-with-exit bất thường (Abnormal Loop-With-Exit Loops)

Một kiểu khác để tránh loop-and-a-half, nhưng là thực hành không tốt:

```
goto Start;
while ( expression ) {
    // do something
Start:
    // do something else
}
```

Ở đây, mã dùng `goto` để nhảy vào giữa vòng lặp, chỉ hợp lý trong các mô phỏng mà đoạn `// do something` không cần thực hiện ở lần lặp đầu tiên. Tuy nhiên, cách này có hai vấn đề: sử dụng `goto` (nên tránh) và khá khó hiểu.

Cách thay thế tốt hơn trong C++ mà không dùng goto như sau:

```
while ( true ) {
    // do something else
    if ( !( expression ) ) {
        break;
    }
    // do something
}
```

Nếu ngôn ngữ bạn sử dụng không hỗ trợ `break`, có thể mô phỏng bằng `goto`.

Khi nào sử dụng vòng lặp for

Vòng lặp for phù hợp khi bạn cần lập một số lần cụ thể. Bạn có thể sử dụng for trong C++, C, Java, Visual Basic và hầu hết các ngôn ngữ khác.

Sử dụng for để thực hiện các thao tác đơn giản không đòi hỏi điều khiển vòng lặp phức tạp ở bên trong, ví dụ như lặp qua các phần tử trong một container. Mục đích của vòng lặp for là khai báo điều kiện lặp gọn ở đầu vòng, sau đó "quên" về điều kiện này khi đã vào thân vòng lặp.

Nếu có điều kiện cần nhảy ra giữa chừng, hãy dùng while thay vì for. Tương tự, không nên thay đổi thủ công giá trị chỉ số (index) của vòng lặp for để dừng vòng lặp — nên dùng while thay thế cho các trường hợp phức tạp; for chỉ nên dùng cho các thao tác đơn giản.

Khi nào sử dụng vòng lặp foreach

Vòng lặp foreach hoặc tương đương (foreach trong C#, For-Each trong Visual Basic, for-in trong Python) thuận tiện cho việc thực hiện thao tác trên từng phần tử của mảng hoặc container khác. Ưu điểm là loại bỏ hoàn toàn toán học xử lý vòng lặp (loop-housekeeping arithmetic), giúp giảm rủi ro lỗi.

Ví dụ trong C#:

```
int [] fibonacciSequence = new int [] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int oddFibonacciNumbers = 0;
int evenFibonacciNumbers = 0;
// count the number of odd and even numbers in a Fibonacci sequence
foreach ( int fibonacciNumber in fibonacciSequence ) {
    if ( (fibonacciNumber % 2) == 0 ) {
        evenFibonacciNumbers++;
    }
    else {
        oddFibonacciNumbers++;
    }
}
Console.WriteLine( "Found {0} odd numbers and {1} even numbers ", oddFibonacciNumbers, evenF
```

16.2 Kiểm soát vòng lặp

Những vấn đề có thể gặp với vòng lặp bao gồm: - Khởi tạo vòng lặp sai hoặc thiếu - Không khởi tạo biến tích lũy hoặc các biến liên quan - Lồng ghép vòng lặp không đúng - Điều kiện kết thúc không đúng - Quên tăng biến điều khiển hoặc tăng sai cách - Truy xuất phần tử mảng không đúng thông qua chỉ số

Bạn có thể phòng tránh những vấn đề này bằng cách: 1. Giảm tối thiểu các yếu tố ảnh hưởng đến vòng lặp — hãy đơn giản hóa tối đa. 2. Xem phần trong vòng lặp như một "hàm con" — giữ càng nhiều điều khiển bên ngoài vòng lặp càng tốt.

Điểm chính: Hãy khai báo rõ ràng các điều kiện để thực hiện thân vòng lặp. Không nên để người đọc phải kiểm tra bên trong vòng lặp để hiểu cách điều khiển.

Xem vòng lặp như một *hộp đen* (black box): chương trình bên ngoài chỉ biết điều kiện kiểm soát, không cần biết chi tiết nội dung bên trong.

Ví dụ:

```
while ( !inputFile.EndOfFile() && moreDataAvailable ) {
    // phần thân vòng lặp
}
```

Bạn chỉ biết rằng vòng lặp kết thúc khi một trong hai điều kiện là `inputFile.EndOfFile()` trở thành true hoặc `moreDataAvailable` trở thành false.

Khi bước vào vòng lặp

Hãy tuân thủ các quy tắc sau: - **Chỉ vào vòng lặp từ một vị trí duy nhất:** Các cấu trúc vòng lặp hiện tại đủ linh hoạt để luôn cho phép bắt đầu vào vòng lặp từ đầu – không cần vào từ nhiều vị trí khác nhau. - **Đặt mã khởi tạo ngay trước vòng lặp:** Nguyên tắc Gần kề (Proximity Principle) khuyên nên hội tụ các câu lệnh liên quan lại với nhau. Nếu phân tán khắp nơi, dễ bỏ sót hoặc cập nhật sai khi sửa đổi. - **Gắn các câu lệnh khởi tạo vòng lặp sát với vòng lặp:** Nếu không làm vậy, dễ dẫn đến lỗi khi tổng quát hóa thành vòng lặp lớn hơn hoặc khi di chuyển vòng sang hàm khác mà quên chuyển khởi tạo. - **Sử dụng `while ( true )` cho vòng lặp vô hạn:** Đôi lúc bạn cần một vòng lặp chạy liên tục (như trong phần mềm nhúng cho máy tạo nhịp tim hoặc lò vi sóng). Đừng giả lập vòng lặp vô hạn bằng các giới hạn số lần lặp lớn giả tạo (ví dụ: `for i = 1 to 99999`), vì điều này làm nhầm lẫn về mục đích và dễ bị lỗi khi bảo trì. - **Ưu tiên for khi thích hợp:** Vòng lặp for giúp quy tụ toàn bộ mã kiểm soát vòng lặp vào một chỗ, thuận tiện cho việc đọc và sửa đổi. - **Không dùng for khi while thích hợp hơn:** Không nhồi nhét logic phức tạp của while vào phần header của for. Cần phân biệt các câu lệnh điều khiển vòng lặp với các câu lệnh dọn dẹp vòng lặp (housekeeping). - **Chỉ đặt các câu lệnh thực sự điều khiển vòng lặp vào header của for:** Những câu lệnh như khởi tạo, kiểm tra điều kiện kết thúc, và cập nhật tiến tới kết thúc vòng lặp. Nếu phải cập nhật biến không liên quan đến điều kiện kết thúc, hãy giữ những câu lệnh này ngoài header.

Ví dụ:

```
// Sử dụng for không hợp lý
for ( inputFile.MoveToStart(), recordCount = 0; !inputFile.EndOfFile(); recordCount++ ) {
    inputFile.GetRecord();
}

// Sử dụng for hợp lý
recordCount = 0;
for ( inputFile.MoveToStart(); !inputFile.EndOfFile(); inputFile.GetRecord() ) {
    recordCount++;
}

// while vẫn là tốt nhất cho trường hợp này
inputFile.MoveToStart();
recordCount = 0;
while ( !inputFile.EndOfFile() ) {
    inputFile.GetRecord();
    recordCount++;
}
```

Xử lý phần giữa vòng lặp

Sử dụng dấu ngoặc để bao quanh các câu lệnh trong vòng lặp

Luôn sử dụng `{ }` để bao quanh các câu lệnh trong vòng lặp. Việc này không ảnh hưởng đến tốc độ hoặc dung lượng chạy, giúp tăng khả năng đọc mã nguồn và phòng ngừa lỗi khi code bị chỉnh sửa. Đây là một thực hành lập trình phòng ngừa (defensive programming) tốt.

Tránh vòng lặp rỗng

Trong C++ và Java có thể tạo vòng lặp rỗng, tức là công việc thực hiện/hay điều kiện kết thúc diễn ra trên cùng một dòng, ví dụ:

```
while ( (inputChar = dataFile.GetChar()) != CharType_Eof ) {
    ;
}
```

Nên chuyển thành vòng lặp đầy đủ:

```
do {
    inputChar = dataFile.GetChar();
} while ( inputChar != CharType_Eof );
```

Đưa các công việc "dọn dẹp" về đầu hoặc cuối vòng lặp

Những thao tác như `i = i + 1` hoặc `j++` – tức là các biểu thức kiểm soát vòng lặp, nên thực hiện ở đầu hoặc cuối vòng lặp để mã dễ hiểu và bảo trì.

Ví dụ:

```
nameCount = 0;
totalLength = 0;
while ( !inputFile.EndOfFile() ) {
    // thực hiện công việc của vòng lặp
    inputFile >> inputString;
    names[nameCount] = inputString;

    // housekeeping (dọn dẹp)
    nameCount++;
    totalLength = totalLength + inputString.length();
}
```

Quy tắc tổng quát: các biến được khởi tạo trước vòng lặp là những biến bạn sẽ thao tác trong phần housekeeping của vòng lặp.

Mỗi vòng lặp nên chỉ làm một việc

Dù vòng lặp có thể dùng để làm nhiều việc cùng lúc, song mỗi vòng lặp nên thực hiện một chức năng duy nhất, làm thật tốt chức năng đó, giống như nguyên tắc thiết kế hàm/routine. Nếu cần hai chức năng, nên tách thành hai vòng lặp, chỉ gộp lại nếu thực sự chứng minh được cần tối ưu hiệu suất và có số liệu benchmark xác nhận khu vực code đó gây ảnh hưởng hiệu suất chương trình.

Thoát khỏi vòng lặp

Đảm bảo vòng lặp có thể kết thúc

Quan sát quá trình thực thi vòng lặp để chắc chắn rằng nó có thể kết thúc trong mọi điều kiện. Hãy xét các trường hợp thông thường, các điểm đầu/cuối, cũng như các trường hợp ngoại lệ.

Làm rõ điều kiện kết thúc vòng lặp

Nếu bạn dùng vòng lặp for mà không can thiệp vào chỉ số lặp, cũng như không dùng `goto` hay `break` để thoát đột ngột, điều kiện kết thúc sẽ rất rõ ràng. Điều này cũng đúng với while hay repeat-until khi kiểm soát hoàn toàn bằng điều kiện lặp. Hãy quy tụ mọi điều kiện kết thúc vào một chỗ.

Không "điều chỉnh" thủ công chỉ số lặp trong vòng for

Một số lập trình viên cố ý thay đổi giá trị chỉ số lặp để kết thúc vòng lặp sớm, dẫn đến mã nguồn khó đọc và dễ gây lỗi.

Ví dụ không nên làm:

```
for ( int i = 0; i < 100; i++ ) {
    // một số mã
    if ( ... ) {
        i = 100; // điều chỉnh chỉ số lặp để kết thúc vòng for sớm
    }
    // mã khác
}
```

(Các đoạn code trong văn bản đã được giữ nguyên và bao quanh bởi markdown code block. Các thuật ngữ chuyên ngành như loop, loop-with-exit, loop-and-a-half, break, goto... đã được giữ nguyên theo đúng hướng dẫn.)

Lưu ý: Văn bản gốc có một số lỗi đánh máy (ví dụ: thiếu dấu ngoặc trong code, lỗi xuống dòng), đã được chỉnh sửa trong bản dịch để đảm bảo dễ hiểu và nhất quán.

Thực hành kiểm soát vòng lặp

Tránh sử dụng biến chỉ số vòng lặp sau khi thoát vòng lặp

Hầu như tất cả các lập trình viên giỏi đều tránh thực hành này; đó là dấu hiệu của người không chuyên. Khi bạn thiết lập một vòng lặp `for`, **counter của vòng lặp là bất khả xâm phạm**. Hãy sử dụng vòng lặp `while` để kiểm soát tốt hơn điều kiện thoát vòng lặp.

Tránh viết mã phụ thuộc vào giá trị cuối cùng của chỉ số vòng lặp. Việc sử dụng giá trị này sau vòng lặp là một thói quen xấu. Giá trị cuối cùng của chỉ số phụ thuộc vào ngôn ngữ lập trình cũng như cách triển khai. Giá trị này có thể khác khi vòng lặp kết thúc bình thường hoặc bất thường. Ngay cả khi bạn nắm rõ giá trị cuối cùng là gì, người đọc tiếp theo có khả năng sẽ phải suy nghĩ về nó. Cách tốt hơn, rõ ràng và tự tài liệu hơn là bạn nên gán giá trị cuối cùng đó cho một biến riêng khi cần thiết trong vòng lặp.

Ví dụ về sử dụng sai giá trị cuối cùng của chỉ số vòng lặp trong C++

```
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    if ( entry[ recordCount ] == testValue ) {
```

```
        break;
    }
}
// nhiều dòng mã khác

if ( recordCount < MAX_RECORDS ) {
    return( true );
}
else {
    return( false );
}
```

Trong đoạn mã này, việc kiểm tra lại `recordCount < MAX_RECORDS` khiến cho người đọc hiểu nhầm rằng vòng lặp phải duyệt hết tất cả các giá trị của `entry[]` và trả về `true` nếu tìm thấy `testValue`, ngược lại trả về `false`. Tuy nhiên, rất dễ gây ra lỗi off-by-one bởi khó nhớ được khi nào chỉ số bị tăng vượt quá giới hạn sau vòng lặp.

Bạn nên viết lại mã không phụ thuộc vào giá trị cuối cùng của chỉ số, như sau:

Ví dụ viết lại hợp lý hơn

```
found = false;
for ( recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    if ( entry[ recordCount ] == testValue ) {
        found = true;
        break;
    }
}
// nhiều dòng mã khác

return( found );
```

Sử dụng biến boolean phụ trợ và hạn chế phạm vi xuất hiện của `recordCount` giúp đoạn mã rõ ràng hơn. Điều này thường làm cho kết quả dễ đọc, khi sử dụng thêm biến boolean.

Cân nhắc sử dụng Safety Counter (Bộ đếm an toàn)

Bộ đếm an toàn là biến được tăng lên mỗi lần qua vòng lặp để kiểm tra số lần lặp đã vượt quá giới hạn cho phép hay chưa. Trong các chương trình mà lỗi có thể gây hậu quả nghiêm trọng, hãy sử dụng bộ đếm an toàn để đảm bảo mọi vòng lặp đều kết thúc đúng cách.

Ví dụ vòng lặp có thể tận dụng safety counter:

```
do {
    node = node->Next;
} while ( node->Next != NULL );
```

Khi thêm safety counter:

```
safetyCounter = 0;
do {
    node = node->Next;
    safetyCounter++;
    if ( safetyCounter >= SAFETY_LIMIT ) {
        Assert( false, "Internal Error: Safety-Counter Violation " );
    }
} while ( node->Next != NULL );
```

Bộ đếm an toàn không phải giải pháp hoàn hảo; việc thêm dần các bộ đếm này thực sự có thể làm tăng độ phức tạp và sinh ra lỗi mới. Nếu áp dụng bộ đếm an toàn như một tiêu chuẩn dự án cho các vòng lặp quan trọng, chúng sẽ góp phần duy trì sự nhất quán và an toàn hơn mà không làm tăng rủi ro sinh lỗi code.

Kết thúc vòng lặp sớm (Exiting Loops Early)

Nhiều ngôn ngữ lập trình hỗ trợ cách kết thúc vòng lặp bằng các lệnh như `break` trong C++, C, Java; `Exit-Do`, `Exit-For` trong Visual Basic; hoặc các cấu trúc tương đương (bao gồm cả mô phỏng bằng `goto` nếu ngôn ngữ không hỗ trợ trực tiếp). Lệnh `break` kết thúc vòng lặp qua kênh thoát thông thường, chương trình tiếp tục thực thi tại câu lệnh tiếp theo ngoài vòng lặp.

Lệnh `continue` tương tự như `break` ở chỗ là lệnh điều khiển phụ cho vòng lặp nhưng không kết thúc vòng lặp, mà chỉ bỏ qua phần còn lại của thân vòng lặp tại lượt hiện tại và bắt đầu lặp tiếp theo. `continue` là cách viết tắt của một mệnh đề if-then lớn dùng để ngăn thực hiện nốt phần còn lại của vòng lặp.

Cân nhắc sử dụng `break` thay vì cờ boolean trong vòng lặp `while`

Trong một số trường hợp, việc thêm cờ boolean vào vòng lặp `while` để mô phỏng thoát thân vòng lặp có thể làm vòng lặp khó đọc. Sử dụng `break` thay vì nhiều mệnh đề if có thể làm giảm độ lồng nhau và đơn giản hóa kiểm soát vòng lặp.

Nên đặt nhiều điều kiện `break` thành các lệnh riêng biệt và đặt gần nơi phát sinh để giảm lồng nhau và tăng tính dễ đọc.

Cảnh báo khi có nhiều lệnh `break` trong một vòng lặp

Một vòng lặp với nhiều lệnh `break` phân tán là dấu hiệu cho thấy cấu trúc chưa rõ ràng hoặc cách sử dụng vòng lặp chưa hợp lý. Việc này dễ dẫn đến lỗi hoặc

làm cho code khó bảo trì; có thể nên xem xét chuyển thành nhiều vòng lặp nhỏ thay vì một vòng lặp lớn có nhiều lối ra.

> Một bài báo trên Software Engineering Notes đã chỉ ra rằng lỗi phần mềm khiến hệ thống điện thoại thành phố New York bị tê liệt 9 tiếng vào ngày 15/1/1990 là do một lệnh `break` dư thừa:
>
> ```
> do {
>     switch
>         if () {
>             break;
>         }
> } while (   );
> ```
>
> Lệnh break này tưởng dành cho if, nhưng lại làm break khỏi switch.

Mặc dù nhiều lệnh `break` không chắc chắn dẫn đến lỗi, nhưng nếu xuất hiện nhiều là dấu hiệu cảnh báo cần xem xét lại cấu trúc vòng lặp.

Sử dụng `continue` cho điều kiện kiểm tra ở đầu vòng lặp

Một ví dụ điển hình khi dùng `continue` là khi cần bỏ qua phần thân vòng lặp sau khi kiểm tra điều kiện ở đầu vòng lặp. Ví dụ, khi đọc các bản ghi, bạn có thể loại bỏ bản ghi không phù hợp và chỉ xử lý bản ghi phù hợp:

```
while ( not eof( file ) ) do
    read( record, file )
    if ( record Type <> targetType ) then
        continue
    -- xử lý record có loại targetType
end while
```

Cách dùng này giúp tránh phải lồng toàn bộ thân vòng lặp trong mệnh đề if. Nếu cần nhảy tiếp ở giữa hoặc cuối vòng lặp, nên dùng if thay vì continue.

Sử dụng `break` có gắn nhãn nếu ngôn ngữ hỗ trợ

Java hỗ trợ `labeled break` để tránh lỗi như sự cố điện thoại New York. Lệnh này có thể dùng để thoát khỏi bất kỳ khối nào (vòng lặp, if…), giúp luận giải đường đi rõ ràng hơn.

```
do {
    switch
        CALL_CENTER_DOWN:
            if () {
                break CALL_CENTER_DOWN;
            }
} while (   );
```

Chỉ sử dụng `break` và `continue` khi thực sự cần thiết

Sử dụng `break` sẽ khiến vòng lặp không còn là hộp đen; bắt buộc người đọc phải nhìn vào chi tiết bên trong để hiểu truyền động vòng lặp. Hãy cân nhắc các giải pháp thay thế trước khi dùng `break` hoặc `continue`. Nếu không thể bảo vệ hợp lý việc sử dụng, đừng dùng chúng.

Kiểm tra giới hạn đầu/cuối vòng lặp

Một vòng lặp thường có ba trường hợp đáng chú ý: trường hợp đầu tiên, trường hợp giữa và trường hợp cuối cùng. Khi tạo vòng lặp, hãy suy nghĩ xem liệu nó có mắc phải lỗi off-by-one ở các trường hợp này không. Nếu có các tình huống đặc biệt khác, hãy rà soát kỹ.

Nếu vòng lặp chứa các phép tính phức tạp, hãy tính tay để xác minh.

> Sự sẵn sàng kiểm tra kiểu này là điểm khác biệt then chốt giữa lập trình viên hiệu quả và không hiệu quả. Lập trình viên hiệu quả luôn tự mô phỏng về mặt tư duy, tính toán bằng tay để phát hiện lỗi và hiểu rõ chương trình.

Lập trình viên kém hiệu quả thường thử nghiệm ngẫu nhiên tới khi tìm được tổ hợp dường như hoạt động; đổi dấu `<` thành `<=`, rồi lại sửa chỉ số lên/xuống 1 cho tới khi ra kết quả đúng hoặc sinh ra một lỗi khác tinh vi hơn. Nhưng làm vậy không rèn được khả năng phân tích đúng-sai.

Việc rèn luyện này giúp giảm lỗi khi viết mã, nhanh chóng phát hiện lỗi khi gỡ rối và hiểu sâu sắc chương trình thay vì chỉ đoán kết quả.

Nguyên tắc sử dụng biến vòng lặp

Dùng kiểu số thứ tự hoặc kiểu liệt kê

Thông thường, chỉ số vòng lặp nên là kiểu integer. Giá trị kiểu dấu phẩy động (floating-point) không tăng chính xác, có thể gây lặp vô hạn.

Đặt tên biến ý nghĩa cho chỉ số vòng lặp

Với mảng một chiều, có thể sử dụng tạm i, j, k. Tuy nhiên, với mảng nhiều chiều, **nên đặt tên chỉ số có ý nghĩa** để rõ ràng vai trò biến và cấu trúc mảng.

Ví dụ kém:

```
for ( int i = 0; i < numPayCodes; i++ ) {
    for ( int j = 0; j < 12; j++ ) {
        for ( int k = 0; k < numDivisions; k++ ) {
```

```
            sum = sum + transaction[ j ][ i ][ k ];
        }
    }
}
```

Rõ ràng hơn:

```
for ( int payCodeIdx = 0; payCodeIdx < numPayCodes; payCodeIdx++ ) {
    for (int month = 0; month < 12; month++ ) {
        for ( int divisionIdx = 0; divisionIdx < numDivisions; divisionIdx++ ) {
            sum = sum + transaction[ month ][ payCodeIdx ][ divisionIdx ];
        }
    }
}
```

Tên biến như `payCodeIdx`, `month`, `divisionIdx` cho biết rõ ý nghĩa từng chiều trong mảng `transaction`, dễ đọc hơn nhiều.

Tránh trùng lặp biến chỉ số (cross-talk)

Việc dùng lặp đi lặp lại i, j, k dễ gây "cross-talk" – tức một biến chỉ số bị sử dụng cho hai mục đích khác nhau:

```
for ( i = 0; i < numPayCodes; i++ ) {
    // nhiều dòng mã
    for ( j = 0; j < 12; j++ ) {
        // nhiều dòng mã
        for ( i = 0; i < numDivisions; i++ ) {
            sum = sum + transaction[ j ][ i ][ k ];
        }
    }
}
```

Việc này gây xung đột biến `i` giữa hai vòng lặp. Chỉ cần đặt tên biến tốt hơn sẽ tránh được sai sót này.

Nhìn chung, khi vòng lặp có thân dài hoặc lồng nhau nhiều tầng, nên tránh dùng i, j, k.

Giới hạn phạm vi biến chỉ số chỉ trong vòng lặp

Chỉ số bị sử dụng ngoài phạm vi vòng lặp có thể gây lỗi nghiêm trọng. Ngôn ngữ Ada không cho phép sử dụng biến chỉ số ngoài vòng lặp, và một số ngôn ngữ (C++, Java) cho phép khai báo biến ngay trong lệnh for để hạn chế phạm vi:

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    // mã sử dụng recordCount
}
```

Một kỹ thuật như vậy giúp tái sử dụng biến `recordCount` an toàn ở nhiều vòng lặp khác nhau:

```
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    // mã 1
}
// đoạn mã khác
for ( int recordCount = 0; recordCount < MAX_RECORDS; recordCount++ ) {
    // mã 2
}
```

Tuy nhiên, **không nên tuyệt đối trông cậy vào trình biên dịch để kiểm soát phạm vi biến này**, vì việc thực thi có thể khác nhau tùy trình biên dịch.

Độ dài hợp lý của vòng lặp

Độ dài vòng lặp có thể đo bằng số dòng mã hoặc số tầng lồng nhau.

- **Nên giữ số dòng vòng lặp vừa phải**, tốt nhất chỉ dài vừa đủ nhìn thấy trên màn hình (ví dụ 50 dòng). Tuy nhiên, nên hướng đến các vòng lặp dưới 15-20 dòng.
- **Giới hạn lồng nhau không quá 3 tầng**, nhiều nghiên cứu cho thấy khả năng đọc hiểu giảm mạnh với lồng nhau sâu hơn.
- **Di chuyển phần thân vòng lặp dài vào các routine (hàm/con)** giúp mã đơn giản, dễ bảo trì.
- **Khi vòng lặp dài bắt buộc, càng phải viết rõ ràng**, nên chỉ có một điểm thoát duy nhất với điều kiện dễ hiểu.

Tạo vòng lặp đơn giản – tiếp cận từ trong ra ngoài

Nếu bạn gặp khó khăn khi mã hóa một vòng lặp phức tạp, hãy thử phương pháp sau:

1. **Bắt đầu với trường hợp đơn**: Viết code cho trường hợp cụ thể, dùng số liệu cụ thể, không quan tâm đến chỉ số vòng lặp.
2. **Thêm vòng lặp xung quanh**: Sau đó, đặt vòng lặp bao quanh đoạn mã trên, thay số liệu cụ thể bằng chỉ số phù hợp.
3. **Lặp lại nếu cần**: Có thể tiếp tục thêm vòng lặp hoặc cải tiến thêm tạo khung tổng quát.
4. **Hoàn chỉnh khởi tạo và kiểm tra**: Đảm bảo tất cả biến được khởi tạo đúng và có kiểm tra hợp lý.

Ví dụ: Tính tổng phí bảo hiểm nhân thọ (life-insurance) cho một nhóm, giả sử có bảng tra (table) phí theo tuổi (age) và giới tính (gender):

Bước 1: Viết bằng chú thích

```
-- lấy rate từ bảng
-- cộng rate vào tổng
```

Bước 2: Viết mã hóa cụ thể

```
rate = table[ ]
totalRate = totalRate + rate
```

Bước 3: Thêm chỉ số cho bảng

```
rate = table[ census Age ][ census Gender ]
totalRate = totalRate + rate
```

Ở đây, `census` là cấu trúc giữ thông tin tuổi, giới cho từng người.

Bước 4: Bọc vòng lặp cho mỗi người

```
For person = firstPerson to lastPerson
    rate = table[ census Age ][ census Gender ]
    totalRate = totalRate + rate
End For
```

Việc bắt đầu ở bên trong, xây dựng dần ra ngoài giúp giảm nhầm lẫn, đảm bảo đúng logic yêu cầu.

Lưu ý: Các thuật ngữ chuyên ngành như "API", "break", "continue", "safety counter", "off-by-one error",... được giữ nguyên theo đúng hướng dẫn dịch thuật và có chú thích hoặc giải thích rõ ràng trong ngữ cảnh khi cần thiết.

Trong trường hợp này, biến census thay đổi theo từng người, do đó cần được khái quát hóa một cách phù hợp.

16.4 Mối quan hệ giữa Vòng lặp (Loops) và Mảng (Arrays)

Bước 5: Tạo một vòng lặp từ trong ra ngoài (ví dụ giả mã - pseudocode)

```
For person = firstPerson to lastPerson
    rate = table[ census[ person ] Age, census[ person ] Gender ]
    totalRate = totalRate + rate
End For
```

Cuối cùng, hãy viết bất kỳ bước khởi tạo (initialization) nào cần thiết. Trong trường hợp này, biến totalRate cần được khởi tạo.

Bước cuối: Tạo một vòng lặp từ trong ra ngoài (ví dụ giả mã)

```
totalRate = 0
For person = firstPerson to lastPerson
    rate = table[ census[ person ] Age, census[ person ] Gender ]
    totalRate = totalRate + rate
End For
```

Nếu cần đặt một vòng lặp khác bọc ngoài vòng lặp person, bạn cũng sẽ tiến hành tương tự. Không cần phải tuân theo các bước một cách cứng nhắc; ý tưởng là bắt đầu bằng điều cụ thể, tập trung giải quyết từng vấn đề một và xây dựng vòng lặp từ các thành phần đơn giản. Hãy thực hiện từng bước nhỏ, dễ hiểu khi làm cho vòng lặp trở nên tổng quát và phức tạp hơn. Bằng cách này, bạn sẽ giảm thiểu lượng mã cần tập trung vào từng thời điểm và do đó giảm khả năng mắc lỗi.

Mối quan hệ giữa Vòng lặp và Mảng

> Tham khảo thêm: Mối quan hệ giữa các kiểu dữ liệu và cấu trúc điều khiển, xem Mục 10.7, "Relationship Between Data Types and Control Structures".

Vòng lặp và mảng thường gắn liền với nhau. Trong nhiều trường hợp, một vòng lặp được tạo ra để thao tác trên mảng, và bộ đếm vòng lặp tương ứng trực tiếp với chỉ số mảng. Ví dụ sau minh họa sự tương ứng giữa chỉ số vòng lặp trong Java và chỉ số mảng:

Ví dụ Java về nhân phần tử các mảng

```
for ( int row = 0; row < maxRows; row++ ) {
    for ( int column = 0; column < maxCols; column++ ) {
        product[ row ][ column ] = a[ row ][ column ] * b[ row ][ column ];
    }
}
```

Trong Java, một vòng lặp là cần thiết cho thao tác này trên mảng. Tuy nhiên, cần lưu ý rằng cấu trúc lặp và mảng không gắn bó chặt chẽ về bản chất. Một số ngôn ngữ, đặc biệt là APL và Fortran 90 trở về sau, cung cấp các thao tác mạnh mẽ trên mảng và loại bỏ nhu cầu sử dụng vòng lặp như trên. Sau đây là một đoạn mã APL thực hiện phép toán tương tự:

Ví dụ APL về nhân phần tử mảng

```
product <- a x b
```

APL đơn giản và ít gây lỗi hơn; nó chỉ sử dụng ba toán tử, trong khi đoạn mã Java ở trên sử dụng đến 17 toán tử. APL không có biến vòng lặp, chỉ số mảng hay cấu trúc điều khiển có thể bị mã hóa sai.

Một điểm đáng chú ý của ví dụ này là bạn lập trình để giải quyết vấn đề, đồng thời cũng lập trình để phù hợp với ngôn ngữ cụ thể. Ngôn ngữ bạn sử dụng để giải quyết vấn đề ảnh hưởng đáng kể đến giải pháp của bạn.

---

cc2e.com/1616 DANH SÁCH KIỂM TRA (CHECKLIST): Vòng lặp (Loops)

Lựa chọn và Tạo lập Vòng lặp

- Có sử dụng vòng lặp while thay vì for khi phù hợp không?
- Vòng lặp có được xây dựng từ trong ra ngoài không?

Bắt đầu vòng lặp

- Vòng lặp có bắt đầu từ trên cùng không?
- Mã khởi tạo có nằm ngay trước vòng lặp không?
- Nếu vòng lặp là vô hạn hoặc event loop, nó có được cấu trúc sạch sẽ (clean) thay vì dùng mẹo như `for i = 1 to 9999` không?
- Nếu là for-loop trong C++, C hoặc Java, phần tiêu đề vòng lặp chỉ dùng cho mã điều khiển vòng lặp?

Bên trong vòng lặp

- Vòng lặp có sử dụng `{}` hoặc tương đương để bao quanh thân vòng lặp và phòng ngừa các lỗi do thay đổi không hợp lý không?
- Thân vòng lặp có nội dung không? Có đảm bảo không rỗng?
- Các tác vụ quản lý (housekeeping) có được nhóm lại ở đầu hoặc cuối vòng lặp không?
- Vòng lặp chỉ thực hiện một và chỉ một chức năng, giống như một thủ tục (routine) được định nghĩa rõ ràng?
- Vòng lặp có đủ ngắn để có thể xem toàn bộ cùng lúc?
- Vòng lặp có lồng quá ba cấp không?
- Nội dung dài của vòng lặp đã được tách ra thành thủ tục độc lập chưa?
- Nếu vòng lặp dài, mã có đủ rõ ràng không?

Chỉ số vòng lặp (Loop Indexes)

- Nếu vòng lặp là for-loop, có tránh sửa đổi giá trị chỉ số vòng lặp bên trong không?
- Có sử dụng biến riêng để lưu các giá trị quan trọng của chỉ số vòng lặp thay vì sử dụng trực tiếp chỉ số vòng lặp sau vòng lặp không?
- Chỉ số vòng lặp có phải là kiểu số thứ tự (ordinal type) hoặc kiểu liệt kê (enumerated type), không phải là số thực (floating-point)?
- Chỉ số vòng lặp có tên ý nghĩa không?
- Vòng lặp có tránh gây trùng giá trị chỉ số với vòng lặp khác không?

Kết thúc vòng lặp

- Vòng lặp có kết thúc trong mọi trường hợp có thể không?
- Có sử dụng bộ đếm an toàn (safety counters) nếu đã đặt ra quy tắc kiểm tra an toàn không?
- Điều kiện kết thúc vòng lặp có rõ ràng không?
- Nếu dùng `break` hoặc `continue`, có đảm bảo sử dụng đúng không?

Các điểm mấu chốt

- **Vòng lặp vốn phức tạp. Giữ cho chúng đơn giản sẽ giúp người đọc mã dễ hiểu và bảo trì hơn.**
- **Các kỹ thuật giúp giữ cho vòng lặp đơn giản bao gồm tránh các dạng vòng lặp phức tạp, giảm tối đa mức lồng nhau, làm rõ điểm vào và điểm ra, và tập trung các mã quản lý tại một nơi.**
- **Chỉ số vòng lặp thường hay bị lạm dụng. Đặt tên chỉ số rõ ràng và chỉ sử dụng cho một mục đích duy nhất.**
- **Xem xét kỹ vòng lặp để xác minh rằng nó hoạt động bình thường trong mọi trường hợp và luôn kết thúc đúng điều kiện.**

Chương 17: Cấu trúc điều khiển bất thường (Unusual Control Structures)

Nội dung:

Chủ đề liên quan:

- Vấn đề điều khiển chung: Chương 19
- Mã theo trình tự thẳng: Chương 14
- Mã có cấu trúc điều kiện: Chương 15
- Mã sử dụng vòng lặp: Chương 16
- Xử lý ngoại lệ (exception handling): Mục 8.4

Có một số cấu trúc điều khiển tồn tại trong "vùng ranh giới" giữa tiến bộ và bị phê phán, thậm chí cùng lúc nằm ở cả hai vị trí! Các cấu trúc này không phải lúc nào cũng có mặt trong mọi ngôn ngữ, nhưng nếu được sử dụng cẩn trọng thì vẫn hữu ích trong những ngôn ngữ hỗ trợ chúng.

17.1 Trả về nhiều lần từ một thủ tục (Multiple Returns from a Routine)

Hầu hết các ngôn ngữ đều hỗ trợ phương pháp thoát khỏi một hàm (routine) giữa chừng. Câu lệnh `return` và `exit` là các cấu trúc điều khiển cho phép chương trình kết thúc một routine tại bất kỳ thời điểm nào, trả quyền điều khiển về routine gọi (calling routine). Ở đây, "return" được sử dụng như một từ chung cho cả lệnh `return` trong C++ và Java, `Exit Sub` và `Exit Function` trong Microsoft Visual Basic, và các cấu trúc tương tự.

Sau đây là các hướng dẫn sử dụng câu lệnh return:

- **Sử dụng return khi nó nâng cao tính dễ đọc của mã.** Trong một số routine, khi bạn đã có được kết quả, bạn muốn trả ngay về routine gọi. Nếu routine đó không cần thực hiện công việc dọn dẹp nào thêm sau khi phát hiện lỗi, việc không trả về ngay lập tức đồng nghĩa với việc bạn phải viết thêm nhiều mã thừa.

  Lưu ý: Hạn chế dùng return không cần thiết, hãy sử dụng khi thực sự làm tăng tính dễ hiểu.

Ví dụ sau đây thể hiện trường hợp dùng return từ nhiều vị trí trong hàm là hợp lý:

Ví dụ C++ về trả về nhiều lần

```
Comparison Compare( int value1, int value2 ) {
    if ( value1 < value2 ) {
        return Comparison_LessThan;
    }
    else if ( value1 > value2 ) {
        return Comparison_GreaterThan;
    }
    return Comparison_Equal;
}
```

Những trường hợp khác không rõ ràng bằng, như minh chứng ở phần tiếp theo.

Sử dụng guard clauses (tức return/exit sớm) để đơn giản hóa quá trình xử lý lỗi phức tạp.

Đoạn mã kiểm tra nhiều điều kiện lỗi trước khi thực hiện hành động chính (nominal case) có thể dẫn đến mức lồng sâu, làm che khuất logic chính, ví dụ:

Visual Basic làm che khuất trường hợp chính

```
If file validName() Then
    If file Open() Then
        If encryptionKey valid() Then
            If file Decrypt( encryptionKey ) Then
                ' code chính ở đây
            End If
        End If
    End If
End If
```

Làm như vậy sẽ khó quản lý, nhất là khi khối mã chính rất lớn. Trong trường hợp như vậy, luồng mã sẽ rõ ràng hơn nếu các trường hợp lỗi được kiểm tra trước, nhường đường cho trường hợp chính:

Visual Basic sử dụng guard clause để làm rõ trường hợp chính

```
' khởi tạo; thoát ngay khi có lỗi
If Not file validName() Then Exit Sub
If Not file Open() Then Exit Sub
If Not encryptionKey valid() Then Exit Sub
If Not file Decrypt( encryptionKey ) Then Exit Sub
' code chính ở đây
```

Mặc dù đoạn mã minh họa trên làm cho giải pháp trông gọn gàng, nhưng mã thực tế thường đòi hỏi xử lý dọn dẹp phức tạp khi phát hiện lỗi. Ví dụ thực tế hơn:

Visual Basic thực tế sử dụng guard clause

```
' khởi tạo; thoát ngay khi có lỗi
If Not file validName() Then
    errorStatus = FileError_InvalidFileName
    Exit Sub
End If
If Not file Open() Then
    errorStatus = FileError_CantOpenFile
    Exit Sub
End If
If Not encryptionKey valid() Then
    errorStatus = FileError_InvalidEncryptionKey
    Exit Sub
End If
If Not file Decrypt( encryptionKey ) Then
    errorStatus = FileError_CantDecryptFile
```

```
    Exit Sub
End If
' code chính ở đây
```

Trong mã thực tế, phương pháp Exit Sub tạo ra một lượng mã đáng kể trước khi tới logic chính, nhưng lại tránh được việc lồng sâu. Nếu đoạn mã ví dụ đầu tiên được mở rộng để thiết lập biến errorStatus, phương pháp Exit Sub sẽ giúp nhóm các câu lệnh liên quan lại với nhau tốt hơn. Khi cân nhắc kỹ, phương pháp Exit Sub vẫn có vẻ dễ đọc và dễ bảo trì hơn, dù không cách biệt rõ rệt.

Hạn chế số lần return trong mỗi routine. Sẽ khó hiểu hàm nếu bạn không biết rằng nó có thể đã return ở đâu đó phía trên khi đọc đến cuối hàm. Do đó, chỉ return khi điều đó thực sự nâng cao tính dễ hiểu.

17.2 Đệ quy (Recursion)

Trong đệ quy (recursion), một routine tự giải quyết một phần vấn đề nhỏ, chia nhỏ vấn đề ra và tự gọi lại chính nó để giải quyết từng phần nhỏ đó. Đệ quy thường được sử dụng khi một phần nhỏ của vấn đề dễ giải quyết và phần lớn còn lại dễ chia nhỏ tiếp tục.

Đệ quy không phải lúc nào cũng hữu ích, nhưng khi được sử dụng hợp lý sẽ tạo ra các giải pháp thanh lịch, như trong ví dụ sau về thuật toán sắp xếp sử dụng đệ quy:

Ví dụ Java về thuật toán sắp xếp sử dụng đệ quy

```
void QuickSort( int firstIndex, int lastIndex, String [] names ) {
    if ( lastIndex > firstIndex ) {
        int midPoint = Partition( firstIndex, lastIndex, names );
        // Các lời gọi đệ quy
        QuickSort( firstIndex, midPoint-1, names );
        QuickSort( midPoint+1, lastIndex, names );
    }
}
```

Trong ví dụ này, thuật toán sắp xếp chia mảng làm hai, rồi tự gọi lại chính nó để sắp xếp từng nửa. Khi gọi với một mảng con quá nhỏ để sắp xếp (lastIndex <= firstIndex), hàm sẽ dừng gọi đệ quy.

Đối với một nhóm vấn đề nhỏ, đệ quy có thể tạo ra giải pháp đơn giản và thanh nhã. Đối với nhóm bài toán lớn hơn, đệ quy có thể tạo ra giải pháp đơn giản nhưng rất khó hiểu. Đối với hầu hết các vấn đề, đệ quy có thể làm cho giải pháp trở nên quá phức tạp—khi đó, lặp tuần tự thường dễ hiểu hơn. Hãy sử dụng đệ quy một cách chọn lọc.

Ví dụ về Đệ quy Giả sử bạn có kiểu dữ liệu biểu diễn một mê cung. Một mê cung về cơ bản là một lưới (grid), và tại mỗi điểm bạn có thể rẽ trái, phải, lên hoặc xuống. Thường thì bạn có thể đi theo nhiều hướng từ một điểm.

Làm thế nào để lập trình để tìm đường ra khỏi mê cung? Nếu sử dụng đệ quy, câu trả lời khá thẳng thắn. Bắt đầu từ điểm xuất phát, thử mọi con đường đến khi tìm được đường ra. Lần đầu đến một điểm, thử đi sang trái. Nếu không đi được, thử lên hoặc xuống, nếu cũng không được thì thử sang phải. Bạn sẽ không bị lạc đường vì sẽ đánh dấu những điểm đã ghé qua, và không quay lại điểm đã thăm.

Hình 17-1: Đệ quy có thể là công cụ hữu ích khi giải quyết các vấn đề phức tạp—nếu được sử dụng đúng chỗ.

C++ Ví dụ di chuyển qua mê cung bằng đệ quy

```
bool FindPathThroughMaze( Maze maze, Point position ) {
    // nếu vị trí đã từng thử rồi thì không thử lại
    if ( AlreadyTried( maze, position ) ) {
        return false;
    }
    // nếu vị trí là lối ra thì thành công
    if ( ThisIsTheExit( maze, position ) ) {
        return true;
    }
    // ghi nhớ đã thử vị trí này
    RememberPosition( maze, position );
    // thử các hướng trái, lên, xuống, phải; nếu thành công thì dừng tìm
    if ( MoveLeft( maze, position, &newPosition ) ) {
        if ( FindPathThroughMaze( maze, newPosition ) ) {
            return true;
        }
    }
    if ( MoveUp( maze, position, &newPosition ) ) {
        if ( FindPathThroughMaze( maze, newPosition ) ) {
            return true;
        }
    }
    if ( MoveDown( maze, position, &newPosition ) ) {
        if ( FindPathThroughMaze( maze, newPosition ) ) {
            return true;
        }
    }
    if ( MoveRight( maze, position, &newPosition ) ) {
        if ( FindPathThroughMaze( maze, newPosition ) ) {
            return true;
        }
```

```
    }
    return false;
}
```

Dòng đầu tiên kiểm tra xem vị trí đã từng thử chưa—mục tiêu quan trọng khi viết hàm đệ quy là **tránh đệ quy vô tận**. Nếu không kiểm tra, bạn sẽ lặp lại ở một điểm mãi mãi.

Dòng thứ hai kiểm tra xem vị trí hiện tại có phải là lối ra khỏi mê cung—nếu đúng, hàm sẽ trả về true.

Dòng thứ ba ghi nhớ rằng điểm này đã được thăm, tránh đi vòng tròn không dứt.

Phần còn lại thử di chuyển sang các hướng—hàm dừng đệ quy nếu tìm được đường ra (trả về true).

Logic trong routine này khá trực tiếp. Nhiều người lúc đầu cảm thấy thiếu thoải mái với đệ quy vì tính tự tham chiếu, tuy nhiên việc giải bằng cách khác còn phức tạp hơn nhiều và đệ quy ở đây là hợp lý.

Lời khuyên khi sử dụng đệ quy

- **Đảm bảo đệ quy kết thúc.** Kiểm tra routine để bảo đảm tồn tại đường dẫn kết thúc không đệ quy. Thường điều này nghĩa là routine phải có điều kiện kiểm tra dừng đệ quy khi không còn cần thiết. Trong ví dụ mê cung, các kiểm tra với AlreadyTried() và ThisIsTheExit() đảm bảo điều này.
- **Sử dụng bộ đếm an toàn (safety counter) để phòng tránh đệ quy vô tận.** Nếu không thể kiểm tra điều kiện dừng rõ ràng, hãy dùng safety counter. Safety counter phải là một biến được duy trì qua nhiều lần gọi routine. Có thể dùng biến thành viên của lớp hoặc truyền qua tham số.

Visual Basic sử dụng safety counter

```
Public Sub RecursiveProc( ByRef safetyCounter As Integer )
    If ( safetyCounter > SAFETY_LIMIT ) Then
        Exit Sub
    End If
    safetyCounter = safetyCounter + 1
    RecursiveProc( safetyCounter )
End Sub
```

Nếu routine vượt quá giới hạn an toàn, nó sẽ dừng đệ quy. Nếu không muốn truyền safety counter, có thể dùng biến thành viên trong C++, Java hoặc Visual Basic.

- **Hạn chế đệ quy chỉ trong một routine.** Đệ quy vòng (A gọi B gọi C gọi lại A) rất nguy hiểm và khó nhận biết. Quản lý đệ quy trong một

routine đã đủ khó. Nếu có đệ quy vòng, hãy thiết kế lại để chỉ còn một routine, nếu không sử dụng safety counter để tăng độ an toàn.

- **Theo dõi stack.** Với đệ quy, bạn không thể đoán trước lượng bộ nhớ stack sử dụng hay hành vi lúc thực thi. Để kiểm soát, hãy:
  - Đặt giới hạn safety counter phù hợp với lượng stack bạn sẵn sàng phân bổ.
  - Tránh khai báo biến cục bộ quá lớn trong routine đệ quy; hãy dùng heap nếu đối tượng sử dụng nhiều bộ nhớ.
- **Đừng dùng đệ quy cho bài toán tính giai thừa (factorial) hoặc dãy Fibonacci.** Các sách giáo khoa thường đưa ra các ví dụ không thực tế về đệ quy với các bài toán này. Đệ quy mạnh mẽ hơn nhiều so với các ví dụ ngây thơ đó; thay vào đó, nên sử dụng lặp lại (iteration) cho các bài toán này.

Java – Một ví dụ không phù hợp: sử dụng đệ quy để tính giai thừa

```
int Factorial( int number ) {
    if ( number == 1 ) {
        return 1;
    }
    else {
        return number * Factorial( number - 1 );
    }
}
```

Ngoài việc chậm và khó kiểm soát bộ nhớ lúc thực thi, phiên bản đệ quy còn khó hiểu hơn cách lặp:

Java – Giải pháp phù hợp: sử dụng lặp để tính giai thừa

```
int Factorial( int number ) {
    int intermediateResult = 1;
    for ( int factor = 2; factor <= number; factor++ ) {
        intermediateResult = intermediateResult * factor;
    }
    return intermediateResult;
}
```

Có ba bài học từ ví dụ này:

1. Sách giáo khoa không nên lấy các ví dụ đệ quy như vậy.
2. Quan trọng hơn, đệ quy mạnh mẽ hơn rất nhiều so với các ứng dụng đơn giản như tính giai thừa hoặc Fibonacci.
3. Quan trọng nhất, luôn cân nhắc các giải pháp thay thế đệ quy trước khi sử dụng nó. Bạn có thể xử lý mọi vấn đề bằng stack và lặp như đệ quy. Đôi khi cách này tốt hơn; hãy cân nhắc cả hai.

17.3 goto

Bạn có thể nghĩ rằng tranh luận về lệnh goto đã lỗi thời, nhưng thực tế, trong các kho mã nguồn như SourceForge.net, goto vẫn còn xuất hiện không ít, thậm chí ngay trên máy chủ công ty của bạn. Hơn nữa, các tranh luận tương tự vẫn tồn tại trong các hình thức khác nhau, chẳng hạn như về multiple returns, multiple loop exits, named loop exits, xử lý lỗi và exception handling.

Lập luận chống lại goto

- **Lập luận chung chống lại goto là** mã không dùng goto thì chất lượng cao hơn.
- Bức thư nổi tiếng khởi đầu cho tranh luận này là của Edsger Dijkstra: "Go To Statement Considered Harmful" trên tạp chí Communications of the ACM, tháng 3 năm 1968. Dijkstra cho rằng chất lượng mã tỷ lệ nghịch với số lượng goto lập trình viên sử dụng. Trong các công trình tiếp theo, Dijkstra còn khẳng định mã không có goto dễ chứng minh tính đúng đắn hơn.
- **Mã có goto khó căn chỉnh (format).** Cần dùng indent để thể hiện cấu trúc logic, nhưng goto lại ảnh hưởng mạnh tới cấu trúc này, làm cho việc trình bày structure bằng indent trở nên khó hoặc bất khả thi.
- **Dùng goto làm giảm khả năng tối ưu hóa của trình biên dịch.** Một số tối ưu hóa phụ thuộc vào việc luồng điều khiển nằm trong một phạm vi hẹp. Lệnh goto vô điều kiện khiến luồng điều khiển khó phân tích, từ đó giảm khả năng tối ưu hóa mã của trình biên dịch.

(Đã sửa các lỗi đánh máy, trình bày lại để dễ đọc và giữ nguyên thuật ngữ nhất quán trong lĩnh vực khoa học máy tính.)

Những Lập Luận Xung Quanh Lệnh goto

Phản Biện Lệnh goto

Những người ủng hộ lệnh goto đôi khi lập luận rằng nó giúp mã chương trình chạy nhanh hơn hoặc nhỏ gọn hơn. Tuy nhiên, mã chứa goto hiếm khi là phương án nhanh nhất hoặc nhỏ gọn nhất. Bài báo kinh điển xuất sắc của Donald Knuth, "Structured Programming with go to Statements", cung cấp nhiều ví dụ về các trường hợp lệnh goto khiến mã chậm hơn và lớn hơn (Knuth, 1974).

Trên thực tế, việc sử dụng lệnh goto thường dẫn đến vi phạm nguyên tắc rằng luồng thực thi của mã nên đi theo hướng từ trên xuống dưới. Ngay cả khi goto không gây nhầm lẫn nếu dùng cẩn trọng, thì một khi lệnh goto được đưa vào, nó lan tỏa trong mã như mối mọt trong căn nhà mục nát. Nếu chấp nhận bất kỳ lệnh goto nào, những lệnh goto không tốt sẽ len lỏi cùng với các lệnh goto hợp lý; do đó, tốt hơn hết là không cho phép sử dụng lệnh goto.

Kinh nghiệm trong hai thập kỷ sau khi lá thư của Dijkstra được công bố đã chỉ ra sự sai lầm khi viết mã tràn ngập goto. Trong một khảo cứu tài liệu, Ben Shneiderman kết luận rằng bằng chứng ủng hộ quan điểm của Dijkstra rằng chúng ta sẽ tốt hơn khi không sử dụng goto (1980), và nhiều ngôn ngữ lập trình hiện đại, bao gồm cả Java, thậm chí không hỗ trợ lệnh goto.

Lập Luận Ủng Hộ goto

Lập luận ủng hộ lệnh goto chủ yếu nhấn mạnh việc sử dụng cẩn trọng trong các trường hợp đặc thù, thay vì sử dụng bừa bãi. Phần lớn các phản biện chống lại lệnh goto tập trung vào việc lạm dụng không kiểm soát. Cuộc tranh cãi gay gắt về goto nổ ra khi Fortran là ngôn ngữ phổ biến nhất, mà khi đó Fortran thiếu các cấu trúc vòng lặp rõ ràng. Do thiếu hướng dẫn tốt về cách lập trình vòng lặp với goto, các lập trình viên đã tạo ra nhiều mã dạng "spaghetti code". Loại mã này chắc chắn liên quan đến việc tạo ra phần mềm kém chất lượng, dù thực ra nó ít liên quan tới việc một lệnh goto được dùng cẩn trọng nhằm bù đắp cho khiếm khuyết của ngôn ngữ hiện đại.

Lệnh goto được sử dụng đúng chỗ có thể loại bỏ sự cần thiết của mã trùng lặp. Mã trùng lặp gây ra vấn đề nếu hai đoạn mã được chỉnh sửa theo cách khác nhau và làm tăng kích thước file nguồn cũng như file thực thi. Trong trường hợp này, tác động xấu của goto nhỏ hơn rủi ro do mã trùng lặp mang lại.

> Cross-Reference: Để biết chi tiết về sử dụng goto trong mã cấp phát tài nguyên, xem phần "Error Processing and gotos" trong chương này. Xem thêm thảo luận về xử lý ngoại lệ (exception handling) tại Mục 8.4, "Exceptions".

Trong một số trường hợp, lệnh goto có thể mang lại kết quả về tốc độ hoặc kích thước mã tốt hơn. Bài viết năm 1974 của Knuth đã đề cập một số trường hợp sử dụng goto mang lại lợi ích chính đáng.

Lập trình tốt không đồng nghĩa với việc loại bỏ hoàn toàn goto. Việc phân rã phương pháp, tinh chỉnh & lựa chọn cấu trúc điều khiển hợp lý gần như sẽ tự động dẫn tới mã không chứa goto. Việc đạt được mã không chứa goto là kết quả tự nhiên, chứ không phải mục tiêu. Nếu quá tập trung vào tránh goto thì cũng không giúp ích gì nhiều.

Nghiên cứu trong nhiều thập kỷ với goto không chứng minh được sự gây hại rõ ràng của nó. Trong một khảo sát tài liệu, B. A. Sheil kết luận rằng các điều kiện thử nghiệm không thực tế, phân tích dữ liệu kém và kết quả không thuyết phục đã không đủ để ủng hộ tuyên bố của Shneiderman và các tác giả khác

rằng số lỗi trong mã tỉ lệ thuận với số lượng goto (1981). Sheil không khẳng định rằng nên dùng goto, thay vào đó chỉ ra rằng bằng chứng chống lại goto không đủ thuyết phục.

> "Các thí nghiệm này hầu như không cung cấp bằng chứng về hiệu quả có lợi của bất kỳ phương pháp nào trong cấu trúc luồng điều khiển."
>
> — B. A. Sheil

Cuối cùng, lệnh goto đã được tích hợp vào nhiều ngôn ngữ hiện đại, bao gồm Visual Basic, C++, và Ada (ngôn ngữ được thiết kế công phu nhất trong lịch sử). Ada được phát triển sau khi cuộc tranh luận về goto đã phát triển đầy đủ ở cả hai phía, và các kỹ sư của Ada vẫn quyết định giữ lại goto sau khi cân nhắc toàn diện.

Tranh Luận Giả Tạo về goto

Đặc điểm nổi bật của phần lớn các tranh luận về goto là cách tiếp cận hời hợt đối với vấn đề. Người phản đối goto thường trình bày một đoạn mã nhỏ sử dụng goto và chỉ ra việc loại bỏ goto dễ dàng như thế nào, chủ yếu chứng minh rằng việc viết một đoạn mã đơn giản mà không cần goto là dễ dàng.

Ngược lại, phe "không thể sống thiếu goto" thường trình bày một trường hợp mà loại bỏ goto dẫn đến một phép so sánh phụ hoặc trùng lặp một dòng mã. Điều này chủ yếu chứng minh rằng có trường hợp dùng goto giúp giảm một phép so sánh – chẳng phải là một lợi ích lớn trên máy tính ngày nay.

Hầu hết các sách giáo khoa đều không giúp ích nhiều. Họ cung cấp các ví dụ minh họa đơn giản về việc viết lại mã không cần goto như thể thế là đã bao quát vấn đề. Dưới đây là một ví dụ theo "phong cách cải trang" từ một cuốn sách như vậy:

Ví dụ C++: Mã được cho là dễ dàng viết lại mà không cần goto

```
do {
    GetData(inputFile, data);
    if (eof(inputFile)) {
        goto LOOP_EXIT;
    }
    DoSomething(data);
} while (data != -1);
LOOP_EXIT:
```

Cuốn sách nhanh chóng thay đoạn mã trên bằng đoạn mã không dùng goto:

Ví dụ C++: Mã được viết lại mà không dùng goto

```
GetData(inputFile, data);
while ((!eof(inputFile)) && (data != -1)) {
    DoSomething(data);
    GetData(inputFile, data);
}
```

Ví dụ "đơn giản" này thực chất lại có sai sót. Trong trường hợp giá trị `data` bằng -1 ngay khi bắt đầu vòng lặp, đoạn mã dịch sai trên sẽ thoát vòng lặp trước khi thực hiện `DoSomething()`. Trong khi đó, mã gốc sẽ thực hiện `DoSomething()` trước khi phát hiện giá trị -1. Cuốn sách lập trình muốn minh họa việc loại bỏ goto dễ dàng đã dịch sai chính ví dụ của mình. Tuy nhiên, tác giả cũng không nên buồn phiền quá; nhiều cuốn sách khác cũng mắc lỗi tương tự. Ngay cả những chuyên gia cũng gặp khó khăn khi dịch mã sử dụng goto.

Dưới đây là đoạn mã tương đương thực sự, không dùng goto:

Ví dụ C++: Mã tương đương thực sự, không dùng goto

```
do {
    GetData(inputFile, data);
    if (!eof(inputFile)) {
        DoSomething(data);
    }
} while ((data != -1) && (!eof(inputFile)));
```

Ngay cả với bản dịch đúng, ví dụ này vẫn là giả tạo vì nó chỉ minh họa việc sử dụng goto trong trường hợp đơn giản. Những trường hợp như thế này không phải là lý do khiến lập trình viên cẩn trọng sử dụng goto.

Ở thời điểm hiện tại, thật khó để bổ sung nội dung đáng giá cho cuộc tranh luận lý thuyết về goto. Tuy nhiên, một điều thường bị bỏ qua là trường hợp lập trình viên, mặc dù nhận thức rõ các giải pháp thay thế không dùng goto, vẫn chọn sử dụng goto nhằm tăng cường khả năng đọc và bảo trì mã.

Các phần tiếp theo trình bày một số trường hợp mà các lập trình viên giàu kinh nghiệm đã nêu quan điểm ủng hộ việc sử dụng goto, so sánh mã dùng goto và không dùng goto, đồng thời đánh giá lợi - hại giữa các phương án.

Xử Lý Lỗi và lệnh goto

Việc viết mã có tính tương tác cao đòi hỏi quan tâm đặc biệt đến xử lý lỗi và giải phóng tài nguyên khi lỗi xảy ra. Ví dụ dưới đây thực hiện xóa một nhóm file. Trình tự là: lấy danh sách file, tìm từng file, mở file, ghi đè file và xóa file, kiểm tra lỗi tại mỗi bước.

Mã Visual Basic sử dụng goto để xử lý lỗi và thu hồi tài nguyên

```
' This routine purges a group of files
Sub PurgeFiles(ByRef errorState As Error_Code)
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList(fileList, numFilesToPurge)
    errorState = FileStatus_Success
    fileIndex = 0
    While (fileIndex < numFilesToPurge)
        fileIndex = fileIndex + 1
        If Not ( FindFile(fileList(fileIndex), fileToPurge)) Then
            errorState = FileStatus_FileFindError
            GoTo END_PROC
        End If
        If Not OpenFile(fileToPurge) Then
            errorState = FileStatus_FileOpenError
            GoTo END_PROC
        End If
        If Not OverwriteFile(fileToPurge) Then
            errorState = FileStatus_FileOverwriteError
            GoTo END_PROC
        End If
        If Not Erase(fileToPurge) Then
            errorState = FileStatus_FileEraseError
            GoTo END_PROC
        End If
    Wend
END_PROC:
    DeletePurgeFileList(fileList, numFilesToPurge)
End Sub
```

Đây là ví dụ điển hình cho trường hợp mà lập trình viên giàu kinh nghiệm quyết định sử dụng goto. Trường hợp tương tự cũng xuất hiện khi cần cấp phát và thu hồi tài nguyên như kết nối cơ sở dữ liệu, bộ nhớ, hoặc file tạm thời. Phương án thay thế cho goto trong những trường hợp này thường là nhân bản mã để dọn dẹp tài nguyên. Khi đó, lập trình viên có thể so sánh cái "xấu" của goto với sự phiền phức khi bảo trì mã trùng lặp và chọn goto như là phương án ít tệ hơn.

Bạn có thể viết lại đoạn mã trên theo vài cách để tránh dùng goto, mỗi cách đều có điểm đánh đổi riêng. Dưới đây là các phương án:

Viết lại bằng các câu lệnh if lồng nhau

Để tránh goto bằng if lồng nhau, đặt các câu lệnh if để mỗi kiểm tra chỉ thực hiện nếu kiểm tra trước đó thành công – đây là phương pháp chuẩn trong sách giáo khoa nhằm loại bỏ goto.

```
' This routine purges a group of files
Sub PurgeFiles(ByRef errorState As Error_Code)
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList(fileList, numFilesToPurge)
    errorState = FileStatus_Success
    fileIndex = 0
    While (fileIndex < numFilesToPurge And errorState = FileStatus_Success)
        fileIndex = fileIndex + 1
        If FindFile(fileList(fileIndex), fileToPurge) Then
            If OpenFile(fileToPurge) Then
                If OverwriteFile(fileToPurge) Then
                    If Not Erase(fileToPurge) Then
                        errorState = FileStatus_FileEraseError
                    End If
                Else
                    errorState = FileStatus_FileOverwriteError
                End If
            Else
                errorState = FileStatus_FileOpenError
            End If
        Else
            errorState = FileStatus_FileFindError
        End If
    Wend
    DeletePurgeFileList(fileList, numFilesToPurge)
End Sub
```

Đối với những ai đã quen lập trình không dùng goto, mã này có thể dễ đọc hơn. Tuy nhiên, nhược điểm là mức độ lồng nhau quá sâu. Khoảng cách giữa đoạn mã xử lý lỗi và câu lệnh gọi nó là khá lớn, gây khó khăn trong việc duy trì hoặc đọc mã.

Ở phiên bản sử dụng goto, không có câu lệnh nào cách điều kiện gọi nó quá bốn dòng, đồng thời bạn cũng không phải để ý tới toàn bộ cấu trúc khi đọc mã.

Viết lại bằng biến trạng thái

Một cách khác là sử dụng biến trạng thái (còn gọi là biến state). Trong ví dụ trên, biến `errorState` đã được sử dụng, nên có thể tận dụng trực tiếp.

```
' This routine purges a group of files
Sub PurgeFiles(ByRef errorState As Error_Code)
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList(fileList, numFilesToPurge)
    errorState = FileStatus_Success
    fileIndex = 0
    While (fileIndex < numFilesToPurge) And (errorState = FileStatus_Success)
        fileIndex = fileIndex + 1
        If Not FindFile(fileList(fileIndex), fileToPurge) Then
            errorState = FileStatus_FileFindError
        End If
        If (errorState = FileStatus_Success) Then
            If Not OpenFile(fileToPurge) Then
                errorState = FileStatus_FileOpenError
            End If
        End If
        If (errorState = FileStatus_Success) Then
            If Not OverwriteFile(fileToPurge) Then
                errorState = FileStatus_FileOverwriteError
            End If
        End If
        If (errorState = FileStatus_Success) Then
            If Not Erase(fileToPurge) Then
                errorState = FileStatus_FileEraseError
            End If
        End If
    Wend
    DeletePurgeFileList(fileList, numFilesToPurge)
End Sub
```

Ưu điểm của cách này là tránh được chuỗi lồng nhau phức tạp, đồng thời mã trở nên dễ hiểu hơn. Tuy nhiên, việc sử dụng biến trạng thái không phải lúc nào cũng là thông lệ phổ biến, nên cần chú thích đầy đủ cho dễ hiểu. Việc chọn tên kiểu dữ liệu dạng liệt kê rõ ràng sẽ giúp ích đáng kể.

Viết lại bằng try-finally

Một số ngôn ngữ như Visual Basic và Java hỗ trợ cấu trúc try-finally để dọn dẹp tài nguyên khi có lỗi.

```
' This routine purges a group of files. Exceptions are passed to the caller
Sub PurgeFiles()
    Dim fileIndex As Integer
    Dim fileToPurge As Data_File
```

```
    Dim fileList As File_List
    Dim numFilesToPurge As Integer
    MakePurgeFileList(fileList, numFilesToPurge)
    Try
        fileIndex = 0
        While (fileIndex < numFilesToPurge)
            fileIndex = fileIndex + 1
            FindFile(fileList(fileIndex), fileToPurge)
            OpenFile(fileToPurge)
            OverwriteFile(fileToPurge)
            Erase(fileToPurge)
        Wend
    Finally
        DeletePurgeFileList(fileList, numFilesToPurge)
    End Try
End Sub
```

Phương án này giả định tất cả các hàm gọi sẽ ném ra ngoại lệ nếu thất bại (thay vì trả về code lỗi).

Ưu điểm của try-finally là đơn giản hơn và tránh được goto cũng như lồng nhau sâu. Nhược điểm là phải nhất quán triển khai trên toàn bộ mã nguồn dự án. Nếu mã dự án còn sử dụng code lỗi cùng với ngoại lệ, thì yêu cầu xử lý ngoại lệ sẽ phức tạp như các cách khác.

So Sánh Các Phương Án

Mỗi phương án đều có ưu điểm nhất định: - Phương án goto tránh lồng nhau và kiểm tra không cần thiết, nhưng có goto. - Phương án if lồng nhau tránh goto nhưng lồng nhau sâu, khiến độ phức tạp logic bị phóng đại. - Phương án biến trạng thái tránh cả goto lẫn lồng sâu nhưng làm phát sinh các phép kiểm tra bổ sung. - Phương án try-finally tránh được cả hai vấn đề nhưng không phải ngôn ngữ nào cũng hỗ trợ.

Nếu ngôn ngữ có hỗ trợ try-finally và trong mã nguồn chưa chuẩn hóa sang phương án khác, try-finally là lựa chọn nên ưu tiên. Nếu không, phương án biến trạng thái thường dễ đọc và mô tả vấn đề tốt hơn, mặc dù không phải lúc nào cũng là lựa chọn tối ưu.

Dù lựa chọn phương án nào, chỉ cần áp dụng nhất quán cho toàn dự án, bạn sẽ đạt được mã nguồn dễ bảo trì và ổn định. Hãy cân nhắc các yếu tố và thống nhất lựa chọn trên toàn bộ dự án.

---

goto và Chia Sẻ Mã trong else

Một tình huống khó nhằn mà một số lập trình viên có thể chọn goto là trường hợp có hai điều kiện và một nhánh else, và cần thực thi một đoạn mã chung ở cả điều kiện và else.

Ví dụ C++: Dùng goto chia sẻ mã trong else

```
if (statusOk) {
    if (dataAvailable) {
        importantVariable = x;
        goto MID_LOOP;
    }
}
else {
    importantVariable = GetValue();
    MID_LOOP:
    // lots of code
}
```

Đây là ví dụ điển hình vì logic khá phức tạp – rất khó đọc và cũng khó viết lại đúng mà không dùng goto. Nếu bạn nghĩ mình có thể viết lại dễ dàng mà không cần goto, hãy nhờ người khác kiểm tra lại nhé! Đã có nhiều chuyên gia viết lại sai đoạn này.

Bạn có thể xử lý bằng nhiều cách khác như trùng lặp mã, đưa mã chung vào một hàm riêng gọi hai lần, hoặc kiểm tra lại điều kiện. Ở hầu hết các ngôn ngữ, mã được viết lại sẽ hơi lớn và chậm hơn chút, nhưng không đáng kể.

Nếu đoạn mã này không nằm trong vòng lặp hiệu năng cao, hãy viết lại mà không cần quan tâm đến hiệu suất.

Cách tốt nhất thường là tách phần `// lots of code` thành một hàm riêng, gọi ở hai vị trí thích hợp.

Ví dụ C++: Chia sẻ mã qua hàm chung

```
if (statusOk) {
    if (dataAvailable) {
        importantVariable = x;
        DoLotsOfCode(importantVariable);
    }
}
else {
    importantVariable = GetValue();
    DoLotsOfCode(importantVariable);
}
```

Thông thường, việc viết hàm mới là hướng tốt nhất. Tuy vậy, trong một số

trường hợp không tiện tách mã lặp ra hàm riêng, bạn có thể tái cấu trúc điều kiện sao cho vẫn nằm trong cùng một hàm:

Ví dụ C++: Viết lại chia sẻ mã mà không dùng goto

```
if ((statusOk && dataAvailable) || !statusOk) {
    if (statusOk && dataAvailable) {
        importantVariable = x;
    }
    else {
        importantVariable = GetValue();
    }
    // lots of code
}
```

Đây là bản dịch đúng và cơ học từ logic của phiên bản có goto. Nó kiểm tra lại các điều kiện hai lần nhưng logic giữ nguyên. Nếu không thích kiểm tra lại, lưu ý bạn có thể bỏ bớt kiểm tra lặp lại nếu thấy phù hợp.

> Cross-Reference: Một cách tiếp cận khác là sử dụng bảng quyết định (decision table). Xem chi tiết tại chương 18, "Table-Driven Methods".

Lưu ý về lỗi: Bản dịch đã bỏ qua các lỗi định dạng và lỗi đánh máy trong văn bản gốc để đảm bảo bản dịch dễ đọc và chuẩn xác. Các đoạn code được giữ nguyên theo yêu cầu.

Nguyên lý về goto trong các ngôn ngữ hiện đại

Quan điểm của tôi là trong các ngôn ngữ lập trình hiện đại, bạn có thể dễ dàng thay thế chín trên mười trường hợp sử dụng lệnh goto bằng các cấu trúc tuần tự tương đương. **Trong những trường hợp đơn giản này, bạn nên thay thế goto theo thói quen. Ở những trường hợp phức tạp hơn, bạn vẫn có thể loại bỏ goto trong chín trên mười tình huống:** bạn có thể tách mã nguồn thành các routine (thủ tục) nhỏ hơn, sử dụng `try-finally`, lồng nhiều cấu trúc `if`, kiểm tra và cập nhật biến trạng thái nhiều lần, hoặc tái cấu trúc điều kiện. Việc loại bỏ goto trong các trường hợp này có thể khó hơn, nhưng đây là một bài tập tư duy hữu ích và các kỹ thuật được thảo luận trong phần này sẽ cung cấp cho bạn công cụ để thực hiện điều đó.

Trong một trường hợp cuối cùng trên tổng số 100 trường hợp mà goto thực sự là giải pháp hợp lý, hãy tài liệu hóa rõ ràng và sử dụng nó. Nếu bạn đang đi ủng cao su, không nên đi vòng quanh khu phố chỉ để tránh một vũng bùn nhỏ. Tuy nhiên, hãy luôn cởi mở với các giải pháp không sử dụng goto mà các lập trình viên khác đề xuất—they có thể nhận ra điều mà bạn chưa thấy.

Tóm tắt các nguyên tắc khi sử dụng goto

- **Chỉ sử dụng goto để mô phỏng các cấu trúc điều khiển có cấu trúc** trong những ngôn ngữ không hỗ trợ chúng trực tiếp. Khi làm vậy, hãy mô phỏng chúng một cách chính xác, không nên lạm dụng sự linh hoạt mà goto mang lại.
- **Không sử dụng goto khi đã có cấu trúc built-in tương đương.**
- **Đo lường hiệu suất** của bất kỳ lệnh goto nào được dùng để cải thiện năng suất. Trong đa số trường hợp, bạn có thể viết lại code mà không cần goto để tăng khả năng đọc mà không làm giảm hiệu quả xử lý. Nếu trường hợp của bạn là ngoại lệ, hãy tài liệu hóa rõ ràng về sự cải thiện hiệu suất để tránh những người phản đối goto xoá nó đi khi đọc code.
- **Giới hạn tối đa một nhãn goto trên mỗi routine**, trừ khi bạn đang mô phỏng các cấu trúc có tổ chức.
- **Chỉ sử dụng goto nhảy tiến (forward)**, không nên nhảy lùi (backward), trừ khi đang mô phỏng cấu trúc có tổ chức.
- **Chắc chắn tất cả các nhãn goto đều được sử dụng**. Nhãn không sử dụng có thể là dấu hiệu cho code bị thiếu, hãy xoá nếu không cần thiết.
- **Không để goto tạo ra các đoạn code không thể truy cập được**.
- **Nếu bạn là quản lý**: đừng để một cuộc tranh cãi về một goto ảnh hưởng tới toàn bộ dự án. Nếu lập trình viên đã cân nhắc các phương án thay thế và có thể bảo vệ ý kiến của mình, goto có thể chấp nhận được.

17.4 Góc nhìn về Các cấu trúc điều khiển bất thường

Đã từng có thời điểm, người ta nghĩ rằng mỗi cấu trúc điều khiển sau đây là một ý tưởng hay:

- Sử dụng goto không hạn chế.
- Tính toán động vị trí nhảy đến của goto.
- Sử dụng goto để nhảy từ giữa một routine sang giữa routine khác.
- Gọi một routine với số dòng hoặc nhãn để bắt đầu thực thi từ giữa routine đó.
- Để chương trình sinh mã nguồn tại thời điểm chạy và thực thi chính phần mã vừa sinh ra.

Trước đây, những ý tưởng này đều từng được chấp nhận, thậm chí được xem là mong muốn, *mặc dù hiện giờ chúng trông ngớ ngẩn, lỗi thời hoặc nguy hiểm.* Ngành phát triển phần mềm đã phát triển chủ yếu thông qua việc *hạn chế những gì lập trình viên có thể làm với mã nguồn của mình.* Do đó, tôi nhìn nhận các cấu trúc điều khiển không thông thường với sự hoài nghi lớn—tin rằng hầu hết các cấu trúc trong chương này cuối cùng cũng sẽ bị loại bỏ, cùng với computed goto labels, variable routine entry points, self-modifying code và những cấu trúc khác ưu tiên sự linh hoạt và tiện lợi hơn tổ chức và khả năng quản lý độ phức tạp.

Tham khảo thêm

- **Trả về (`return`):**
  - **Fowler, Martin.** *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. Trong mô tả về refactoring "Replace Nested Conditional with Guard Clauses", Fowler đề xuất sử dụng nhiều câu lệnh return để giảm độ lồng nhau của các câu lệnh if, và cho rằng sử dụng nhiều return phù hợp để tăng tính rõ ràng mà không gây hại.
- **Goto:**
  - Chủ đề về goto là một tranh luận lâu dài. Nó bùng phát nhiều lần trong các môi trường làm việc, sách giáo khoa và tạp chí, nhưng không có gì mới so với 20 năm trước.
  - **Dijkstra, Edsger.** "Go To Statement Considered Harmful." *Communications of the ACM* 11, no. 3 (March 1968): 147–48. Đây là bức thư nổi tiếng khởi động một trong những tranh luận kéo dài nhất trong phát triển phần mềm.
  - **Wulf, W. A.** "A Case Against the GOTO." Proceedings of the 25th National ACM Conference, August 1972: 791–97. Wulf lập luận rằng nếu ngôn ngữ có đủ cấu trúc điều khiển thì goto về cơ bản không cần thiết. Sự phát triển của các ngôn ngữ như C++, Java, Visual Basic đã chứng minh điều đó.
  - **Knuth, Donald.** "Structured Programming with go to Statements," 1974. *Classics in Software Engineering*, ed. Edward Yourdon, Yourdon Press, 1979. Bài báo phân tích các ví dụ về loại bỏ hoặc thêm goto để tăng hiệu suất.
  - **Rubin, Frank.** " 'GOTO Considered Harmful' Considered Harmful." *Communications of the ACM* 30, no. 3 (March 1987): 195–96. Rubin lập luận rằng lập trình loại bỏ goto gây thiệt hại hàng trăm triệu đô la cho doanh nghiệp; ông đưa ra ví dụ code ngắn dùng goto và cho là tốt hơn so với thay thế không dùng goto. Phản hồi của độc giả về bức thư này đã tạo ra nhiều tranh luận kéo dài trong CACM.

Danh sách kiểm tra: Các cấu trúc điều khiển bất thường

return

- Mỗi routine chỉ sử dụng return khi thực sự cần thiết?
- return có làm tăng tính dễ đọc không?

Đệ quy (Recursion)

- Routine đệ quy có mã dừng đệ quy không?

- Có biến đếm đảm bảo routine kết thúc không?
- Đệ quy chỉ ở một routine?
- Độ sâu ngăn xếp đáp ứng dung lượng bộ nhớ stack của chương trình?
- Đệ quy là phương án tốt nhất cho routine này chưa? Có tốt hơn lặp đơn giản không?

goto

- goto chỉ dùng khi thật sự bất khả kháng, để code dễ đọc, bảo trì hơn?
- Nếu dùng goto cho hiệu suất, đã đo lường và tài liệu hóa hiệu quả chưa?
- Mỗi routine chỉ sử dụng tối đa một nhãn goto?
- Tất cả các goto đều nhảy tiến (forward)?
- Tất cả nhãn goto đều được sử dụng?

Các điểm chính

- **Nhiều return** có thể tăng khả năng đọc và bảo trì của routine, giúp ngăn logic lồng sâu. Tuy nhiên, cần sử dụng cẩn thận.
- **Đệ quy** cung cấp giải pháp thanh lịch cho một số vấn đề nhất định. Cũng nên sử dụng cẩn trọng.
- **Trong rất ít trường hợp, goto** là lựa chọn tốt nhất để viết code dễ đọc, dễ bảo trì. Nhưng các trường hợp này hiếm. Chỉ sử dụng goto như phương án cuối cùng.

Chương 18: Phương pháp điều khiển bằng bảng (Table-Driven Methods)

Mục lục

Chủ đề liên quan: - Che giấu thông tin (Information hiding): “Hide Secrets (Information Hiding)” ở Mục 5.3 - Thiết kế class: Chương 6 - Sử dụng bảng quyết định để thay thế các logic phức tạp: ở Mục 19.1 - Thay thế các biểu thức phức tạp bằng tra cứu bảng: ở Mục 26.1

Khái niệm tổng quát về phương pháp điều khiển bằng bảng

Phương pháp điều khiển bằng bảng là một giải pháp cho phép bạn tra cứu thông tin trong bảng thay vì sử dụng các câu lệnh logic (if, case) để xác định kết quả. Thực tế, hầu hết mọi điều bạn có thể chọn bằng logic, đều có thể thay thế bằng bảng tra cứu. *Ở các trường hợp đơn giản, câu lệnh logic sẽ dễ và trực tiếp hơn. Nhưng khi logic trở nên phức tạp, bảng tra cứu càng trở nên hấp dẫn.*

Nếu bạn đã quen thuộc với phương pháp này, có thể xem đây là phần ôn tập. Khi đó, bạn nên xem qua ví dụ tại "Flexible-Message-Format Example" ở Mục 18.2 để thấy rằng thiết kế hướng đối tượng chưa chắc đã ưu việt chỉ vì mang tính hướng đối tượng, sau đó có thể chuyển sang phần bàn luận về các vấn đề điều khiển nói chung ở Chương 19.

18.1 Xem xét tổng quát về sử dụng phương pháp điều khiển bằng bảng

Khi được sử dụng đúng hoàn cảnh, mã nguồn dựa trên bảng tra cứu đơn giản hơn logic phức tạp, dễ chỉnh sửa và hiệu quả hơn.

Ví dụ, bạn muốn phân loại ký tự thành chữ cái, dấu câu, và chữ số; bạn có thể sử dụng chuỗi logic phức tạp như sau:

Ví dụ Java: Sử dụng logic phức tạp để phân loại ký tự

```
if ( ( ( 'a' <= inputChar ) && ( inputChar <= 'z' ) ) ||
     ( ( 'A' <= inputChar ) && ( inputChar <= 'Z' ) ) ) {
    charType = CharacterType.Letter;
}
else if ( ( inputChar == ' ' ) || ( inputChar == ',' ) ||
          ( inputChar == '.' ) || ( inputChar == '!' ) || ( inputChar == '(' ) ||
          ( inputChar == ')' ) || ( inputChar == ':' ) || ( inputChar == ';' ) ||
          ( inputChar == '?' ) || ( inputChar == '-' ) ) {
    charType = CharacterType.Punctuation;
}
else if ( ( '0' <= inputChar ) && ( inputChar <= '9' ) ) {
    charType = CharacterType.Digit;
}
```

Nếu dùng bảng tra cứu, bạn chỉ cần lưu loại của từng ký tự vào một mảng và truy cập theo mã ký tự:

Ví dụ Java: Sử dụng bảng tra cứu để phân loại ký tự

```
charType = charTypeTable[ inputChar ];
```

Ví dụ này giả sử mảng `charTypeTable` đã được khởi tạo trước đó. Bạn chuyển "tri thức" của chương trình từ phần logic sang dữ liệu—cụ thể là lưu vào bảng thay vì viết các câu lệnh if.

Hai vấn đề khi sử dụng phương pháp điều khiển bằng bảng

Khi dùng phương pháp này, bạn cần giải quyết hai vấn đề:

1. **Cách tra cứu giá trị trong bảng:** Có thể dùng dữ liệu để truy cập bảng trực tiếp. Ví dụ nếu cần phân loại theo tháng, truy cập bảng tháng rất đơn giản (dùng một mảng có chỉ số từ 1 đến 12). Nhưng có dữ liệu không thuận tiện để tra cứu trực tiếp (vd: số an sinh xã hội), bạn không thể dùng số này làm chỉ số mảng nếu không muốn tạo ra 999-99-9999 phần tử. Khi đó, bạn cần giải pháp tra cứu phức tạp hơn.

   Ba kiểu truy cập phổ biến:

   - Truy cập trực tiếp (Direct access)
   - Truy cập theo chỉ số (Indexed access)
   - Truy cập kiểu bậc thang (Stair-step access)

2. **Bạn nên lưu gì trong bảng?** Đôi khi, kết quả tra cứu là dữ liệu và bạn chỉ cần lưu dữ liệu. Đôi khi, kết quả là một hành động—lúc này bạn có thể lưu mã hành động, hoặc trong một số ngôn ngữ, lưu trỏ tới routine thực hiện hành động đó. Khi đó bảng trở nên phức tạp hơn.

18.2 Bảng truy cập trực tiếp (Direct Access Tables)

Giống như hầu hết bảng tra cứu, bảng truy cập trực tiếp thay thế các cấu trúc điều khiển logic phức tạp. "Truy cập trực tiếp" vì bạn không cần thực hiện phép chuyển đổi phức tạp để tìm thông tin—bạn có thể truy xuất thẳng tới mục mong muốn.

> **Hình 18-1:** Đúng như tên gọi, bảng truy cập trực tiếp cho phép bạn truy nhập phần tử mong muốn một cách trực tiếp.

Ví dụ: Số ngày trong tháng

Bạn cần xác định số ngày trong từng tháng (không xét năm nhuận). Một cách cồng kềnh là viết một chuỗi if dài:

Ví dụ Visual Basic: Xác định số ngày trong một tháng bằng if nhiều nhánh

```
If ( month = 1 ) Then
    days = 31
ElseIf ( month = 2 ) Then
```

```
    days = 28
ElseIf ( month = 3 ) Then
    days = 31
ElseIf ( month = 4 ) Then
    days = 30
...
ElseIf ( month = 12 ) Then
    days = 31
End If
```

Một cách đơn giản hơn và dễ bảo trì là dùng một bảng dữ liệu:

Ví dụ Visual Basic: Khởi tạo bảng "Số ngày từng tháng"

```
Dim daysPerMonth() As Integer = _
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

Sau đó chỉ cần tra cứu trong mảng:

Ví dụ Visual Basic: Tra cứu số ngày từng tháng

```
days = daysPerMonth( month-1 )
```

Nếu tính đến năm nhuận, giải pháp với bảng vẫn rất đơn giản:

```
days = daysPerMonth( month-1, LeapYearIndex() )
```

Nếu dùng phiên bản if, chuỗi if sẽ còn dài và phức tạp hơn khi xét năm nhuận.

Ví dụ: Bảng tính phí bảo hiểm (Insurance Rates Example)

Giả sử bạn phải tính phí bảo hiểm y tế thay đổi theo tuổi, giới tính, tình trạng hôn nhân và tình trạng hút thuốc. Nếu dùng điều khiển logic, bạn sẽ gặp mã nguồn phức tạp như sau:

Ví dụ Java: Cách phức tạp để xác định phí bảo hiểm

```
if ( gender == Gender.Female ) {
    if ( maritalStatus == MaritalStatus.Single ) {
        if ( smokingStatus == SmokingStatus.NonSmoking ) {
            if ( age < 18 ) { rate = 200.00; }
            else if ( age == 18 ) { rate = 250.00; }
            else if ( age == 19 ) { rate = 300.00; }
            ...
            else if ( 65 < age ) { rate = 450.00; }
        }
        ...
    }
    ...
}
```

(Mã rút gọn để minh họa mức độ phức tạp có thể xảy ra.)

Rõ ràng, bạn có thể nghĩ "Tại sao không lưu các mức phí riêng cho từng tuổi vào từng mảng riêng?" Đúng vậy, như vậy là một cải tiến, nhưng tốt nhất là lưu tất cả yếu tố vào cùng một mảng đa chiều.

Ví dụ Visual Basic: Khai báo bảng dữ liệu tính phí bảo hiểm

```
Public Enum SmokingStatus
    SmokingStatus_First = 0
    SmokingStatus_Smoking = 0
    SmokingStatus_NonSmoking = 1
    SmokingStatus_Last = 1
End Enum

Public Enum Gender
    Gender_First = 0
    Gender_Male = 0
    Gender_Female = 1
    Gender_Last = 1
End Enum

Public Enum MaritalStatus
    MaritalStatus_First = 0
    MaritalStatus_Single = 0
    MaritalStatus_Married = 1
    MaritalStatus_Last = 1
End Enum

Const MAX_AGE As Integer = 125
Dim rateTable(SmokingStatus_Last, Gender_Last, MaritalStatus_Last, MAX_AGE) As Double
```

Sau khi thiết lập dữ liệu cho bảng này, khi cần tính phí bảo hiểm bạn chỉ cần một lệnh đơn giản:

Ví dụ Visual Basic: Tra cứu mức phí

```
rate = rateTable(smokingStatus, gender, maritalStatus, age)
```

Cách làm này không những tăng tính dễ hiểu của mã mà còn dễ chỉnh sửa.

Ví dụ: Flexible-Message-Format

Bạn có thể dùng bảng để mô tả logic quá động, không dễ biểu diễn bằng hard-code if. Với các ví dụ trước (phân loại ký tự, số ngày trong tháng, phí bảo hiểm), bạn có thể biểu diễn hết bằng chuỗi if rất dài. Tuy nhiên, đôi khi dữ liệu phức tạp đến mức không thể viết code với if cố định.

Giả sử bạn phải viết routine in các thông báo từ file chứa khoảng 500 message khác nhau, thuộc 20 loại message. Mỗi message xuất phát từ một buoy và bao gồm các trường như nhiệt độ nước, vị trí, v.v. Mỗi message bắt đầu bằng header chứa ID để xác định loại message. Hình minh hoạ dưới đây:

> **Hình: Dữ liệu các message được lưu trữ không theo thứ tự nhất định, mỗi message có ID xác định loại (Buoy Temperature Message, Drift Message, Location Message, ...)**
>
> **Định dạng message biến động từ khách hàng, bạn không đủ kiểm soát để ổn định định dạng này.**

(Phần sau mô tả tiếp cách áp dụng phương pháp bảng cho kiểu message động này — nếu bạn cần, tôi sẽ dịch tiếp các ví dụ hoặc nội dung còn lại.)

Phương pháp dựa trên logic

Nếu bạn sử dụng **phương pháp dựa trên logic (logic-based approach)**, bạn sẽ có xu hướng đọc từng thông điệp (message), kiểm tra ID, sau đó gọi một routine (thủ tục) được thiết kế để đọc, diễn giải và in từng loại thông điệp. Nếu bạn có 20 loại thông điệp, bạn sẽ cần 20 routine. Ngoài ra, còn có thể nhiều routine cấp thấp hơn để hỗ trợ — ví dụ, bạn sẽ có routine `PrintBuoyTemperatureMessage()` để in thông điệp nhiệt độ phao. **Phương pháp hướng đối tượng (object-oriented approach)** về cơ bản cũng không tốt hơn nhiều: bạn thường sẽ sử dụng một đối tượng message trừu tượng (abstract message object) với một subclass (lớp con) cho mỗi loại thông điệp.

Mỗi khi định dạng của bất kỳ thông điệp nào thay đổi, bạn sẽ phải thay đổi logic trong routine hoặc class phụ trách thông điệp đó. Như trong ví dụ thông điệp chi tiết trước, nếu trường *average-temperature* chuyển từ kiểu floating point sang kiểu khác, bạn sẽ phải thay đổi logic trong `PrintBuoyTemperatureMessage()` (Nếu bản thân phao đổi từ kiểu số thực (floating point) sang kiểu khác, bạn còn phải lấy… một cái phao mới!)

Ở phương pháp dựa trên logic, routine đọc thông điệp bao gồm một vòng lặp để đọc từng thông điệp, giải mã ID, rồi gọi một trong 20 routine dựa trên message ID. Sau đây là pseudocode (giả mã) cho phương pháp dựa trên logic:

```
While more messages to read
    Read a message header
    Decode the message ID from the message header
    If the message header is type 1 then
        Print a type 1 message
    Else if the message header is type 2 then
        Print a type 2 message
    ...
```

```
    Else if the message header is type 19 then
        Print a type 19 message
    Else if the message header is type 20 then
        Print a type 20 message
End While
```

Pseudocode trên đã được rút gọn vì bạn có thể hiểu ý tưởng mà không cần nhìn thấy đầy đủ 20 trường hợp.

Phương pháp hướng đối tượng

Nếu bạn sử dụng phương pháp object-oriented (hướng đối tượng) một cách máy móc, logic sẽ được ẩn trong cấu trúc kế thừa đối tượng, nhưng cấu trúc cơ bản cũng sẽ phức tạp tương tự:

```
While more messages to read
    Read a message header
    Decode the message ID from the message header
    If the message header is type 1 then
        Instantiate a type 1 message object
    Else if the message header is type 2 then
        Instantiate a type 2 message object
    ...
    Else if the message header is type 19 then
        Instantiate a type 19 message object
    Else if the message header is type 20 then
        Instantiate a type 20 message object
    End if
End While
```

Dù logic được viết trực tiếp hay được chứa trong các class chuyên biệt, mỗi loại thông điệp vẫn sẽ có routine riêng để in thông tin của nó. Dưới đây là pseudocode cho routine đọc và in thông điệp nhiệt độ của phao:

```
Print "Buoy Temperature Message"
Read a floating-point value
Print "Average Temperature"
Print the floating-point value
Read a floating-point value
Print "Temperature Range"
Print the floating-point value
Read an integer value
Print "Number of Samples"
Print the integer value
Read a character string
Print "Location"
Print the character string
```

```
Read a time of day
Print "Time of Measurement"
Print the time of day
```

Đây chỉ là code cho một loại thông điệp duy nhất. Với 19 loại thông điệp khác, bạn cũng sẽ cần các routine tương tự. Nếu có thêm một loại thông điệp mới (21 loại), bạn sẽ phải thêm routine hoặc subclass thứ 21 — cách nào đi nữa thì cũng phải thay đổi code khi có loại thông điệp mới.

Phương pháp dẫn xuất từ bảng (Table-Driven Approach)

Phương pháp table-driven (dựa trên bảng) tiết kiệm hơn các phương pháp trên. Routine đọc thông điệp gồm một vòng lặp, đọc từng message header, giải mã ID, tra cứu mô tả thông điệp trong mảng Message, rồi luôn gọi cùng một routine để giải mã thông điệp. Với table-driven approach, bạn có thể mô tả định dạng của từng thông điệp trong một bảng thay vì hard-code (mã hóa cứng) trong logic chương trình. Điều này giúp code ngắn gọn, dễ thay đổi và bảo trì hơn mà không cần sửa đổi code.

Khởi tạo bảng mô tả thông điệp

Đầu tiên, bạn liệt kê các loại thông điệp và loại trường (field). Trong C++, có thể khai báo kiểu của các field như sau:

```
enum FieldType {
    FieldType_FloatingPoint,
    FieldType_Integer,
    FieldType_String,
    FieldType_TimeOfDay,
    FieldType_Boolean,
    FieldType_BitField,
    FieldType_Last = FieldType_BitField
};
```

Thay vì hard-code routine in dữ liệu cho từng loại thông điệp, bạn chỉ cần xây dựng một vài routine in dữ liệu cho từng kiểu chính — floating point, integer, string, v.v. Nội dung của từng loại thông điệp sẽ được mô tả trong bảng (bao gồm cả tên các field), và routine sẽ giải mã từng thông điệp dựa trên mô tả này.

Ví dụ cấu trúc một phần tử bảng thông điệp:

```
Message Begin
NumFields 5
MessageName "Buoy Temperature Message"
Field 1, FloatingPoint, "Average Temperature"
Field 2, FloatingPoint, "Temperature Range"
```

```
Field 3, Integer, "Number of Samples"
Field 4, String, "Location"
Field 5, TimeOfDay, "Time of Measurement"
Message End
```

Bảng này có thể được hard-code vào chương trình, hoặc đọc từ file khi khởi động, hoặc sau đó.

Khi các mô tả message đã được đọc vào, thay vì nhúng toàn bộ logic vào code, bạn để chúng trong dữ liệu — điều này linh hoạt hơn và dễ thay đổi hơn. Nếu cần bổ sung thêm loại message mới, bạn chỉ việc thêm một phần tử vào bảng.

Vòng lặp cấp cao sử dụng bảng

Pseudocode cho vòng lặp cấp cao trong table-driven approach như sau:

```
While more messages to read
    Read a message header
    Decode the message ID from the message header
    Look up the message description in the message-description table
    Read the message fields and print them based on the message description
End While
```

Logic dưới mức này, bạn chỉ cần một routine đủ tổng quát để diễn giải mô tả thông điệp từ bảng, đọc dữ liệu và in ra thông tin—routine này thay thế cho 20 routine in message riêng lẻ, và không phức tạp hơn nhiều:

```
While more fields to print
    Get the field type from the message description
    case ( field type )
        of ( floating point )
            read a floating-point value
            print the field label
            print the floating-point value
        of ( integer )
            read an integer value
            print the field label
            print the integer value
        of ( character string )
            read a character string
            print the field label
            print the character string
        of ( time of day )
            read a time of day
            print the field label
            print the time of day
        of ( boolean )
            read a single flag
```

```
            print the field label
            print the single flag
        of ( bit field )
            read a bit field
            print the field label
            print the bit field
    End Case
End While
```

Thừa nhận, routine này với sáu trường hợp dài hơn routine in riêng cho thông điệp nhiệt độ phao — nhưng đây là routine duy nhất bạn cần; không cần thêm 19 routine khác. Routine này xử lý sáu kiểu field và lo toàn bộ các loại thông điệp.

Thiết kế đối tượng hóa cho trường dữ liệu

Thay vì sử dụng câu lệnh case, bạn có thể định nghĩa một abstract class (lớp trừu tượng) *AbstractField* và subclass cho mỗi loại field, tận dụng tính đa hình (polymorphism):

```
class AbstractField {
public:
    virtual void ReadAndPrint( string, FileStatus & ) = 0;
}
class FloatingPointField : public AbstractField {
public:
    virtual void ReadAndPrint( string, FileStatus & ) {

    }
}
class IntegerField
class StringField
```

Tiếp theo, khai báo một mảng để chứa các instance của các field object này — đây chính là lookup table:

```
AbstractField* field[ Field_Last ];
```

Sau đó khởi tạo các đối tượng này:

```
field[ Field_FloatingPoint ] = new FloatingPointField();
field[ Field_Integer ] = new IntegerField();
field[ Field_String ] = new StringField();
field[ Field_TimeOfDay ] = new TimeOfDayField();
field[ Field_Boolean ] = new BooleanField();
field[ Field_BitField ] = new BitFieldField();
```

Giờ đây, để xử lý một field trong một message, chỉ cần truy cập bảng đối tượng rồi gọi phương thức thành viên, thay vì kiểm tra từng loại đối tượng:

```
fieldIdx = 1;
while ( ( fieldIdx <= numFieldsInMessage ) and ( fileStatus == OK ) ) {
    fieldType = fieldDescription[ fieldIdx ].FieldType;
    fieldName = fieldDescription[ fieldIdx ].FieldName;
    field[ fieldType ]->ReadAndPrint( fieldName, fileStatus );
}
```

> Nếu thay câu lệnh case của phương pháp truyền thống bằng một bảng đối tượng, chỉ cần đoạn code trên là đủ để thay thế tất cả 20 routine riêng biệt của phương pháp dựa vào logic.

Bạn có thể sử dụng phương pháp này trong bất kỳ ngôn ngữ hướng đối tượng nào; vừa ít lỗi, dễ bảo trì, vừa hiệu quả hơn so với các câu lệnh if, case kéo dài, hoặc quá nhiều subclass.

Lưu ý: Việc sử dụng inheritance (kế thừa) và polymorphism (đa hình) không tự động đảm bảo một thiết kế tốt. Thiết kế hướng đối tượng kiểu máy móc ở phần trước thực tế có thể còn phức tạp hơn thiết kế hàm thông thường. Điểm mấu chốt trong trường hợp này không nằm ở hướng đối tượng hay hướng hàm, mà là việc tận dụng một bảng lookup hợp lý.

Xử lý khóa tra cứu bảng (Fudging Lookup Keys)

Trong ba ví dụ trước, bạn có thể sử dụng dữ liệu để làm khóa tra cứu vào bảng trực tiếp — chẳng hạn, sử dụng messageID, hay month trong ví dụ số ngày trong tháng, hoặc gender, maritalStatus, smokingStatus trong ví dụ bảo hiểm.

Bạn luôn muốn dùng khóa trực tiếp vào bảng vì đơn giản và nhanh. Tuy nhiên, đôi khi dữ liệu lại không thuận tiện. Ở ví dụ bảo hiểm, *age* không phù hợp hoàn toàn—một mức phí với tuổi dưới 18, mức riêng cho từng tuổi từ 18 đến 65, và một mức cho tuổi trên 65. Như vậy, tuổi 0-17 và 66 trở lên không thể trực tiếp là khóa cho bảng chỉ có một mức phí cho nhiều tuổi.

Để khắc phục, có một số cách làm như sau:

- **Lặp lại thông tin để khóa dùng được trực tiếp:** Bạn có thể lặp lại mức phí dưới 18 cho từng tuổi từ 0-17 trong bảng, và làm tương tự cho nhóm tuổi từ 66 trở lên. Lợi ích là cấu trúc và thao tác với bảng đơn giản, nhưng tốn không gian cho dữ liệu dư thừa và dễ gây lỗi khi cập nhật bảng.
- **Chuyển hóa khóa để dùng trực tiếp:** Có thể định nghĩa hàm chuyển hóa Age sang đúng khóa. Chẳng hạn, chuyển mọi tuổi 0-17 thành 17, trên 66 thành 66, có thể dùng `max(min(66, Age), 17)`. Điều này cần bạn nhận diện được quy luật trong dữ liệu và định nghĩa hàm biến đổi.
- **Cô lập việc chuyển hóa khóa vào routine riêng:** Nếu phải chuyển hóa dữ liệu để dùng làm khóa, hãy đặt sự chuyển hóa này trong một routine riêng, như *KeyFromAge()*, vừa dễ sửa đổi vừa nhất quán khi sử dụng.

Nếu ngôn ngữ của bạn có cung cấp sẵn các cấu trúc chuyển hóa khóa, hãy sử dụng. Ví dụ, Java có HashMap, cho phép ánh xạ cặp key/value.

Phương pháp bảng chỉ mục (Indexed Access Tables)

Đôi khi một phép biến đổi toán học đơn giản không đủ mạnh để biến dữ liệu như Age thành khóa bàn. Một số trường hợp sẽ phù hợp với kỹ thuật *indexed access* (truy cập qua chỉ mục).

Ở đây, bạn sử dụng dữ liệu chính để tra cứu một khóa trong bảng chỉ mục (index table), rồi dùng giá trị đó tra bảng dữ liệu chính mà bạn quan tâm.

Ví dụ, một kho chứa khoảng 100 mặt hàng, mỗi mặt hàng có số hiệu 4 chữ số từ 0000 đến 9999. Để sử dụng số hiệu làm khóa với bảng chứa thông tin chi tiết, bạn thiết lập một mảng chỉ mục 10.000 phần tử (0-9999), chỉ 100 phần tử là có giá trị (tương ứng các mặt hàng có thật), trỏ đến bảng mô tả mặt hàng nhỏ gọn hơn.

Hình minh họa: > Bảng chỉ mục (chủ yếu rỗng) Bảng lookup (chủ yếu đầy đủ)

Ưu điểm: - Nếu mỗi phần tử bảng dữ liệu chính lớn, dùng mảng chỉ mục giúp tiết kiệm rất nhiều không gian so với để trống bảng lớn. - Việc thao tác trên bảng chỉ mục thường rẻ hơn so với thao tác trên bảng chính. - Giữ cho code dễ bảo trì, vì tách việc truy cập chỉ mục thành routine riêng, dễ thay đổi hoặc chuyển sang các scheme lookup khác sau này.

Phương pháp bảng bậc thang (Stair-Step Access Tables)

Một loại truy cập bảng khác là phương pháp *stair-step* (bậc thang). Ý tưởng chung là mỗi phần tử trong bảng có giá trị cho một khoảng dữ liệu liên tục, chứ không phải cho một giá trị rời rạc.

Ví dụ, trong bài toán xếp loại điểm: - 90.0%: A - < 90.0%: B - < 75.0%: C - < 65.0%: D - < 50.0%: F

Đây là trường hợp "xấu" với truy cập bảng thông thường, vì không thể dùng hàm biến đổi đơn giản để ánh xạ sang A-F. Phép chuyển đổi sang integer thì khả thi, nhưng ví dụ giữ nguyên floating point.

Cách dùng: Bạn lưu trữ giá trị giới hạn trên của từng khoảng vào bảng rồi duyệt từng giới hạn, khi điểm số vượt qua một giá trị, biết được xếp loại tương ứng.

Ví dụ code Visual Basic:

```
' set up data for grading table
Dim rangeLimit() As Double = { 50.0, 65.0, 75.0, 90.0, 100.0 }
Dim grade() As String = { "F", "D", "C", "B", "A" }
maxGradeLevel = grade.Length - 1

' assign a grade to a student based on the student's score
gradeLevel = 0
studentGrade = "A"
While ( ( studentGrade = "A" ) and ( gradeLevel < maxGradeLevel ) )
    If ( studentScore < rangeLimit( gradeLevel ) ) Then
        studentGrade = grade( gradeLevel )
    End If
    gradeLevel = gradeLevel + 1
Wend
```

Bạn có thể mở rộng phương pháp này cho nhiều học sinh, nhiều cách chấm điểm, và thay đổi scheme chấm điểm một cách linh hoạt.

Ưu điểm: Phù hợp với dữ liệu bất quy tắc, mà các phương pháp bảng-driven khác xử lý không hiệu quả.

Kiểm tra lại: - Các thuật ngữ chuyên ngành giữ nguyên và chú thích đầy đủ (message, field, routine, object, class, polymorphism, abstraction, etc.) - Các đoạn code, pseudocode giữ nguyên, bọc bằng markdown. - Diễn đạt súc tích, nhất quán, đảm bảo mạch văn học thuật.

Nếu cần giải thích thêm hoặc mở rộng cho từng phương pháp riêng lẻ, bạn vui lòng yêu cầu cụ thể.

Phương pháp bậc thang (Stair-step approach) và các vấn đề chung về kiểm soát chương trình

Phương pháp bậc thang áp dụng cho dữ liệu không chuẩn

Phương pháp bậc thang (stair-step approach) đặc biệt phù hợp với dữ liệu không kết thúc gọn với các con số như 5 hoặc 0. Bạn có thể sử dụng phương pháp này trong các công việc thống kê, chẳng hạn như với các phân phối xác suất có các giá trị dưới đây:

Xác suất và Giá trị bồi thường bảo hiểm

Probability	Insurance Claim Amount
0.458747	$0.00
0.547651	$254.32
0.627764	$514.77
0.776883	$747.82

```
0.893211      $1,042.65

Probability   Insurance Claim Amount
0.957665      $5,887.55
0.976544      $12,836.98
0.987889      $27,234.12
```

Các giá trị không đều và khó đọc này khiến cho việc xây dựng hàm ánh xạ chúng thành khóa của bảng tra cứu trở nên không khả thi. Phương pháp bậc thang là giải pháp hợp lý.

Ưu điểm tổng quát của phương pháp bậc thang

Phương pháp này cũng mang lại những lợi ích chung của các phương pháp dựa trên bảng tra cứu (table-driven approaches): **tính linh hoạt và khả năng dễ sửa đổi**. Nếu phạm vi điểm trong ví dụ phân loại thay đổi, chương trình dễ dàng thích ứng chỉ bằng cách chỉnh sửa các phần tử của mảng `RangeLimit`. Bạn cũng có thể tổng quát hóa phần đánh giá xếp hạng để chấp nhận một bảng điểm và ngưỡng điểm tương ứng; chương trình không nhất thiết phải sử dụng phần trăm, mà hoàn toàn dùng điểm thực và không cần thay đổi nhiều.

Những lưu ý khi sử dụng kỹ thuật bậc thang

Dưới đây là một số điểm tinh tế cần lưu ý khi sử dụng kỹ thuật bậc thang:

- **Cẩn trọng với các điểm biên (endpoints)**: Đảm bảo đã xử lý trường hợp ở đầu trên của mỗi khoảng bậc thang. Thực hiện tìm kiếm sao cho các mục thuộc về bất kỳ khoảng nào ngoài khoảng cao nhất được gán đúng; phần còn lại sẽ rơi vào khoảng cao nhất. Đôi khi, bạn cần tạo giá trị ảo cho đỉnh của khoảng lớn nhất.
- **Phân biệt giữa `<` và `<=`**: Đảm bảo vòng lặp kết thúc đúng với các giá trị thuộc các khoảng lớn nhất và ranh giới khoảng được xử lý chuẩn xác.
- **Cân nhắc sử dụng tìm kiếm nhị phân (binary search) thay vì tìm kiếm tuần tự (sequential search)**: Nếu danh sách lớn, chi phí của tìm kiếm tuần tự có thể không chấp nhận được. Lúc này, bạn có thể thay thế bằng một tìm kiếm nhị phân bán phần—thay vì tìm giá trị khớp hoàn toàn, chỉ cần xác định đúng hạng mục cho giá trị.
- **Sử dụng truy cập theo chỉ số (indexed access) thay cho kỹ thuật bậc thang**: Trong một số trường hợp nhất định, sử dụng truy cập theo chỉ số, như mô tả ở Mục 18.3, là giải pháp thay thế tốt nhằm tăng tốc độ thực thi, đổi lấy việc tiêu tốn thêm bộ nhớ cho cấu trúc chỉ số phụ.
  - Tuy nhiên, không phải trường hợp nào cũng áp dụng được. Với dữ liệu xác suất như trên, không thể lập chỉ số trực tiếp với các số như 0.458747 hoặc 0.547651 được.

  Chú thích: Nếu vì các yếu tố thiết kế (design), có nhiều phương án khả dĩ, hãy tập trung chọn giải pháp tốt thay vì phải "tối ưu" tuyệt đối. Theo Butler Lampson (Kỹ sư danh tiếng của Microsoft),

> "Tốt hơn hết là có một giải pháp tốt và tránh được thảm họa, thay vì cố gắng tìm giải pháp hoàn hảo".

Nên đóng gói hàm tra cứu bảng bậc thang thành một thủ tục riêng khi cần ánh xạ một giá trị sang khóa bảng (ví dụ: từ StudentGrade sang khóa bảng).

18.5 Các ví dụ khác về Table Lookup

Bên cạnh các ví dụ vừa nêu, bảng tra cứu còn được áp dụng trong nhiều ngữ cảnh kỹ thuật khác, cụ thể như:

- **Tra cứu mức phí trong bảng bảo hiểm**: Xem Mục 16.3 "Tạo vòng lặp dễ dàng – từ trong ra ngoài".
- **Sử dụng bảng quyết định (decision table) để thay thế logic phức tạp**: Xem phần "Sử dụng bảng quyết định để thay thế các điều kiện phức tạp" trong Mục 19.1.
- **Chi phí paging bộ nhớ khi tra cứu bảng**: Xem Mục 25.3, "Các dạng lãng phí".
- **Kết hợp các giá trị boolean (A hoặc B hoặc C)**: Xem phần "Thay thế biểu thức phức tạp bằng table lookups" trong Mục 26.1.
- **Tiền tính toán giá trị trong bảng trả nợ vay**: Xem Mục 26.4, "Expressions".

CHECKLIST: Table-Driven Methods

- Đã cân nhắc các phương pháp dựa trên bảng như một giải pháp thay thế cho logic phức tạp?
- Đã cân nhắc phương pháp dùng bảng so với sử dụng cấu trúc kế thừa phức tạp?
- Đã cân nhắc lưu dữ liệu bảng bên ngoài và đọc dữ liệu khi chạy thay vì nhúng vào mã nguồn?
-

Nếu không thể truy cập bảng trực tiếp qua chỉ số mảng, đã tách phép tính khóa truy cập thành hàm riêng thay vì nhân bản logic tính chỉ số trong mã?

Những điểm then chốt

- **Bảng tra cứu** là một cách tiếp cận thay thế cho logic hoặc cấu trúc kế thừa phức tạp. Nếu bạn cảm thấy khó hiểu với logic hoặc cây kế thừa của chương trình, hãy tự hỏi liệu có thể đơn giản hóa bằng bảng tra cứu.

- Một trong những yếu tố then chốt khi sử dụng bảng là quyết định phương thức truy cập: truy cập trực tiếp, truy cập thông qua chỉ số, hay truy cập kiểu bậc thang.
- Một yếu tố khác là quyết định chính xác nội dung cần lưu trong bảng.

Chương 19: Các vấn đề điều khiển chung (General Control Issues)

Nội dung chương

Chủ đề liên quan:
- Mã tuyến tính: Chương 14
- Điều kiện: Chương 15
- Vòng lặp: Chương 16
- Cấu trúc điều khiển phi thường: Chương 17
- Độ phức tạp trong phát triển phần mềm: Xem "Yếu tố kỹ thuật then chốt của phần mềm: Quản lý độ phức tạp" trong Mục 5.2

Bất kỳ thảo luận sâu về điều khiển chương trình đều phải đề cập đến các vấn đề tổng quát phát sinh khi cân nhắc cấu trúc điều khiển. Hầu hết thông tin của chương này mang tính chi tiết, thực tiễn. Nếu bạn quan tâm tới lý thuyết hơn là chi tiết, hãy chú trọng các vấn đề về lập trình cấu trúc ở Mục 19.5 và mối quan hệ giữa các cấu trúc điều khiển ở Mục 19.6.

19.1 Biểu thức Boolean (Boolean Expressions)

Ngoại trừ cấu trúc điều khiển đơn giản nhất là thực hiện câu lệnh theo thứ tự, mọi cấu trúc điều khiển đều phụ thuộc vào việc đánh giá các biểu thức boolean (biểu thức logic đúng/sai).

Sử dụng true và false trong kiểm tra boolean

Hãy sử dụng định danh **true** và **false** trong các biểu thức boolean thay cho các giá trị như 0 và 1. Hầu hết các ngôn ngữ hiện đại đều hỗ trợ kiểu dữ liệu boolean và cung cấp các định danh sẵn. Việc này giúp ngăn ngừa các lỗi không rõ ràng và dễ đọc hơn:

Ví dụ Visual Basic dùng giá trị không rõ ràng cho biến Boolean (KHÔNG NÊN)

```
Dim printerError As Integer
Dim reportSelected As Integer
Dim summarySelected As Integer

If printerError = 0 Then InitializePrinter()
If printerError = 1 Then NotifyUserOfError()
If reportSelected = 1 Then PrintReport()
If summarySelected = 1 Then PrintSummary()
If printerError = 0 Then CleanupPrinter()
```

> **Nhận xét:** Code trên không rõ ràng vì người đọc không biết 1/0 là true/false, hay chỉ định mã khác.

Sử dụng rõ ràng true và false (Nên dùng)

```
Dim printerError As Boolean
Dim reportSelected As ReportType
Dim summarySelected As Boolean

If (printerError = False) Then InitializePrinter()
If (printerError = True) Then NotifyUserOfError()
If (reportSelected = ReportType_First) Then PrintReport()
If (summarySelected = True) Then PrintSummary()
If (printerError = False) Then CleanupPrinter()
```

So sánh boolean theo cách ngắn gọn

Các phép kiểm tra có thể viết ngắn gọn hơn mà vẫn rõ nghĩa:

```
If (Not printerError) Then InitializePrinter()
If (printerError) Then NotifyUserOfError()
If (reportSelected = ReportType_First) Then PrintReport()
If (summarySelected) Then PrintSummary()
If (Not printerError) Then CleanupPrinter()
```

> **Lưu ý:** Nếu ngôn ngữ không hỗ trợ chính thức kiểu boolean, nên định nghĩa các hằng số toàn cục hoặc macro tương đương.

Đơn giản hóa biểu thức phức tạp

1. **Tách nhỏ điều kiện** bằng cách sử dụng biến boolean trung gian thay vì kiểm tra một biểu thức lớn.
2. **Chuyển kiểm tra phức tạp vào hàm boolean** để code rõ ràng, dễ đọc hơn.

Ví dụ kiểm tra phức tạp:

```
If ( (document AtEndOfStream) And (Not inputError) ) And _
   ( (MIN_LINES <= lineCount) And (lineCount <= MAX_LINES) ) And _
   (Not ErrorProcessing()) Then
     ' Do something
End If
```

=> Đưa kiểm tra vào hàm:

```
Function DocumentIsValid( _
        ByRef documentToCheck As Document, _
        lineCount As Integer, _
        inputError As Boolean _
    ) As Boolean

    Dim allDataRead As Boolean
    Dim legalLineCount As Boolean

    allDataRead = (documentToCheck AtEndOfStream) And (Not inputError)
    legalLineCount = (MIN_LINES <= lineCount) And (lineCount <= MAX_LINES)
    DocumentIsValid = allDataRead And legalLineCount And (Not ErrorProcessing())
End Function
```

Sử dụng:

```
If (DocumentIsValid(document, lineCount, inputError)) Then
    ' Do something
End If
```

> Giải pháp này tăng khả năng đọc hiểu code và tạo ra "tài liệu tự động" trong code.

Sử dụng bảng quyết định thay cho điều kiện phức tạp

Đối với nhiều kiểm tra phức tạp, có thể sử dụng bảng quyết định thay vì if/case lồng nhau. Điều này giúp code đơn giản và dễ thay đổi khi dữ liệu thay đổi.

Đặt biểu thức boolean theo hướng dương (positively)

Biểu thức nhiều phủ định dễ gây nhầm lẫn cho lập trình viên lẫn người đọc. Thay vào đó, hãy chuyển thành các kiểm tra hướng dương và đảo vị trí code tại các nhánh if-else cho phù hợp.

Ví dụ (Java):

```
if (!statusOK) {
    // làm gì đó
} else {
    // làm gì đó khác
}
```

Nên viết lại:

```
if (statusOK) {
    // làm gì đó khác
} else {
    // làm gì đó
}
```

Hoặc chọn tên biến phản nghĩa phù hợp (`ErrorDetected`) thay vì `statusOK`.

Áp dụng Định lý DeMorgan để giản lược phủ định

Định lý DeMorgan cho phép chuyển đổi logic của các biểu thức phủ định giữa **and** và **or**.

Biểu thức ban đầu	Biểu thức tương đương
not A and not B	not (A or B)
not A or not B	not (A and B)
...	...

Ví dụ:

```
if (!displayOK || !printerOK)
```

tương đương với

```
if (!(displayOK && printerOK))
```

Sử dụng dấu ngoặc để làm rõ biểu thức boolean

Không nên phụ thuộc hoàn toàn vào thứ tự ưu tiên của toán tử ngôn ngữ; hãy dùng ngoặc để biểu thức rõ ràng hơn.

Ví dụ chưa rõ ràng:

```
if (a < b == c == d)
```

Nên viết:

```
if ((a < b) == (c == d))
```

Nếu nghi ngờ, hãy chủ động thêm dấu ngoặc để đảm bảo cả tính đúng đắn lẫn dễ đọc cho code.

Lưu ý khi kiểm tra cân bằng dấu ngoặc: Có thể dùng kỹ thuật đếm đơn giản (hoặc công cụ của trình soạn thảo) để kiểm tra dấu ngoặc trong biểu thức phức tạp.

> **Tóm lại:**
> - Sử dụng biểu thức boolean rõ ràng, ưu tiên tích cực (positively), giảm sử dụng phủ định lồng nhau. - Đưa kiểm tra phức tạp vào hàm, sử dụng bảng quyết định thay cho logic phức tạp nếu có thể. - Luôn sử dụng dấu ngoặc đầy đủ để đảm bảo ý định trong biểu thức điều kiện. - Xem xét ưu/nhược điểm của từng phương pháp kiểm tra điều kiện và chọn giải pháp tối ưu cho ngữ cảnh cụ thể.

Nếu, tại cuối biểu thức, bạn trở về 0 thì dấu ngoặc của bạn đã cân bằng

Ví dụ Java về Dấu ngoặc Cân bằng

Đọc đoạn này:

```
if ( ( ( a < b ) == ( c == d ) ) && !done )
```

```
| | | | | | |
```

Nói như sau:
0 1 2 3 2 3 2 1 0

Trong ví dụ này, bạn kết thúc với giá trị 0, vì vậy các dấu ngoặc được cân bằng. Ở ví dụ sau, dấu ngoặc không cân bằng:

Ví dụ Java về Dấu ngoặc Không cân bằng

Đọc đoạn này:

```
if ( ( a < b ) == ( c == d ) ) && !done )
```

```
| | | | | |
```

Nói như sau:
0 1 2 1 2 1 0 -1

Việc có giá trị 0 trước khi bạn đến dấu ngoặc đóng cuối cùng cho thấy có một dấu ngoặc mở bị thiếu trước đó. Bạn không nên có giá trị 0 cho đến dấu ngoặc đóng cuối cùng của biểu thức.

Hãy đặt đầy đủ dấu ngoặc cho các biểu thức logic. Dấu ngoặc không tốn kém và giúp tăng khả năng đọc hiểu. Việc đặt đầy đủ dấu ngoặc trong các biểu thức logic nên trở thành thói quen tốt.

Hiểu cách đánh giá các Biểu thức Boolean (Boolean expressions)

Nhiều ngôn ngữ có cơ chế kiểm soát ngầm xuất hiện khi đánh giá các biểu thức boolean. Trình biên dịch của một số ngôn ngữ sẽ đánh giá tất cả thành phần trong một biểu thức boolean trước khi kết hợp chúng và đánh giá toàn bộ biểu thức. Một số ngôn ngữ khác sử dụng "short-circuit" (đánh giá ngắn mạch hay lười), chỉ đánh giá những phần cần thiết.

Điều này đặc biệt quan trọng khi, tùy vào kết quả của phép kiểm tra đầu tiên, bạn có thể không muốn phép kiểm tra thứ hai được thực hiện. Ví dụ, giả sử bạn đang kiểm tra các phần tử của một mảng với cấu trúc kiểm tra sau:

Ví dụ mã giả về kiểm tra sai

```
while ( i < MAX_ELEMENTS and item[ i ] <> 0 )
```

Nếu toàn bộ biểu thức này được đánh giá, bạn sẽ gặp lỗi ở lần lặp cuối của vòng lặp: biến `i` bằng `MAX_ELEMENTS`, nên biểu thức `item[i]` tương đương với `item[MAX_ELEMENTS]`, gây ra lỗi chỉ số mảng. Bạn có thể cho rằng không quan trọng vì chỉ kiểm tra giá trị mà không thay đổi, nhưng đây là cách lập trình cẩu thả và gây khó hiểu cho người đọc mã. Ở nhiều môi trường nó cũng tạo ra lỗi thực thi hoặc vi phạm bảo vệ bộ nhớ.

Bạn có thể cấu trúc lại kiểm tra để tránh lỗi này:

Ví dụ mã giả về kiểm tra đúng

```
while ( i < MAX_ELEMENTS )
    if ( item[ i ] <> 0 ) then
```

Đây là cách đúng vì `item[i]` chỉ được đánh giá khi `i` nhỏ hơn `MAX_ELEMENTS`.

Nhiều ngôn ngữ hiện đại cung cấp cơ chế giúp ngăn lỗi này xảy ra ngay từ đầu. Ví dụ, C++ sử dụng đánh giá short-circuit: nếu toán hạng đầu của phép `and` là false, toán hạng thứ hai sẽ không được đánh giá vì toàn bộ biểu thức chắc chắn là false. Nói cách khác, trong C++ chỉ có phần

```
if ( SomethingFalse && SomeCondition )
```

mới được đánh giá là `SomethingFalse`, và việc đánh giá dừng lại ngay khi xác định được đó là false.

Việc đánh giá cũng được rút ngắn với phép `or`. Trong C++ và Java, chỉ phần

```
if ( somethingTrue || someCondition )
```

được đánh giá là `somethingTrue`. Việc đánh giá dừng lại ngay khi xác định `somethingTrue` là true, vì biểu thức luôn true nếu bất kỳ phần nào là true. Nhờ phương pháp này, đoạn mã sau là hợp lệ:

Ví dụ Java về kiểm tra hoạt động nhờ short-circuit

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) )
```

Nếu toàn bộ biểu thức này được đánh giá khi `denominator` bằng 0, phép chia trong toán hạng thứ hai sẽ phát sinh lỗi chia cho 0. Nhưng với short-circuit evaluation, toán hạng thứ hai chỉ được đánh giá nếu toán hạng thứ nhất là true, do đó không bao giờ xảy ra lỗi chia cho 0 khi `denominator` bằng 0.

Ngược lại, vì phép `&&` (and) được đánh giá từ trái sang phải, nên đoạn mã sau sẽ không được cứu bởi short-circuit evaluation:

Ví dụ Java về kiểm tra không được cứu bởi short-circuit

```
if ( ( ( item / denominator ) > MIN_VALUE ) && ( denominator != 0 ) )
```

Trong trường hợp này, biểu thức `item / denominator` được đánh giá trước `denominator != 0`. Do đó, mã này sẽ gây ra lỗi chia cho 0.

Java còn phức tạp hơn khi cung cấp các toán tử "logical" như `&` và `|`. Các toán tử này đảm bảo tất cả thành phần sẽ được đánh giá đầy đủ bất kể có cần thiết để xác định giá trị biểu thức hay không. Nói cách khác, trong Java, đoạn mã sau là an toàn:

Ví dụ Java an toàn nhờ short-circuit (conditional) evaluation

```
if ( ( denominator != 0 ) && ( ( item / denominator ) > MIN_VALUE ) )
```

Nhưng đoạn mã này là không an toàn:

Ví dụ Java không an toàn vì không đảm bảo short-circuit evaluation

```
if ( ( denominator != 0 ) & ( ( item / denominator ) > MIN_VALUE ) )
```

Mỗi ngôn ngữ có cách đánh giá khác nhau, và các nhà phát triển ngôn ngữ thường có quy định riêng trong việc đánh giá biểu thức, nên hãy kiểm tra tài liệu hướng dẫn cho phiên bản ngôn ngữ bạn đang sử dụng.

Lưu ý chính: Tốt hơn hết là bạn nên sử dụng các kiểm tra lồng nhau để làm rõ ý định thay vì phụ thuộc vào thứ tự đánh giá và short-circuit evaluation, bởi người đọc mã của bạn có thể không thành thạo như bạn.

Viết biểu thức số theo thứ tự trên trục số (number-line order)

Hãy tổ chức các phép kiểm tra số học sao cho chúng phản ánh các điểm trên trục số. Nhìn chung, hãy cấu trúc kiểm tra số như sau:

```
MIN_ELEMENTS <= i and i <= MAX_ELEMENTS
i < MIN_ELEMENTS or MAX_ELEMENTS < i
```

Ý tưởng là sắp xếp các thành phần từ trái sang phải, từ nhỏ đến lớn. Ở dòng đầu, `MIN_ELEMENTS` và `MAX_ELEMENTS` là hai điểm mốc, do đó được đặt ở hai đầu. Biến `i` cần nằm giữa, vì vậy nó ở giữa biểu thức. Ở ví dụ thứ hai, bạn kiểm tra xem `i` có nằm ngoài phạm vi không, nên `i` ở hai ngoài cùng, còn `MIN_ELEMENTS` và `MAX_ELEMENTS` ở giữa. Cách này dễ dàng liên tưởng tới hình ảnh so sánh trực quan trên trục số.

Nếu bạn so sánh `i` chỉ với `MIN_ELEMENTS`, thì vị trí của `i` phụ thuộc vào mục đích kiểm tra:

Nếu `i` cần nhỏ hơn:

```
while ( i < MIN_ELEMENTS )
```

Nếu `i` cần lớn hơn:

```
while ( MIN_ELEMENTS < i )
```

Cách này rõ ràng hơn so với kiểm tra như sau:

```
( i > MIN_ELEMENTS ) and ( i < MAX_ELEMENTS )
```

vì không giúp người đọc hình dung trực quan điều gì đang được kiểm tra.

Hướng dẫn so sánh với 0

Các ngôn ngữ lập trình sử dụng 0 cho nhiều mục đích: giá trị số, ký tự kết thúc chuỗi, giá trị con trỏ null, giá trị đầu tiên trong một liệt kê (enumeration), hoặc là false trong biểu thức logic. Vì được dùng đa dạng, bạn nên viết mã làm rõ cách sử dụng cụ thể của 0.

- **So sánh biến logic một cách ngầm định**: Như đã đề cập, bạn có thể viết: `cpp while ( !done )` Đây là phép so sánh ngầm định với 0, phù hợp với biểu thức logic.
- **So sánh số với 0 một cách tường minh**: Nên so sánh số một cách tường minh: `cpp while ( balance != 0 )` hơn là: `cpp while ( balance )`
- **So sánh ký tự với ký tự kết thúc chuỗi ('\0') một cách tường minh trong C**: Ký tự không phải là biểu thức logic, nên hãy viết: `cpp while ( *charPtr != '\0' )` thay vì: `cpp while ( *charPtr )` Mặc dù khuyến nghị này ngược với thói quen phổ biến trong C, nó giúp làm rõ rằng bạn đang làm việc với dữ liệu kiểu ký tự chứ không phải logic.
- **So sánh con trỏ với NULL một cách tường minh**: Nên viết: `cpp while ( bufferPtr != NULL )` thay vì: `cpp while ( bufferPtr )`

Việc so sánh tường minh như trên giúp tăng khả năng đọc hiểu, dù đi ngược lại một số quy ước của C.

Các vấn đề thường gặp với Biểu thức Boolean

Các biểu thức boolean còn có một số lỗi đặc thù từng ngôn ngữ:

- **Trong các ngôn ngữ xuất phát từ C, hãy đặt hằng số ở bên trái biểu thức so sánh**: Nếu bạn dễ nhầm lẫn giữa `=` và `==`, hãy đặt hằng số ở bên trái: `cpp if ( MIN_ELEMENTS = i )` Biên dịch sẽ báo lỗi vì không thể gán cho một hằng số, trong khi nếu viết: `cpp if ( i = MIN_ELEMENTS )` chỉ gây ra cảnh báo (nếu bật cảnh báo đầy đủ).

  Tuy nhiên, khuyến nghị này mâu thuẫn với nguyên tắc xếp thứ tự trên trục số. Tác giả ưa dùng cách thứ hai và để biên dịch cảnh báo về gán nhầm.
- **Trong C++, có thể định nghĩa macro thay thế cho &&, ||, và == (chỉ khi thực sự cần thiết)**: Bạn có thể định nghĩa macro như `AND` cho `&&`, `OR` cho `||` hoặc `EQUALS` cho `==` nếu thường xuyên mắc lỗi, nhưng giải pháp này được xem là kém về khả năng đọc hiểu với lập trình viên có kinh nghiệm. Hầu hết trình biên dịch đều cảnh báo với gán nhầm. Tốt nhất, chỉ nên sử dụng macro không chuẩn trong trường hợp đặc biệt.
- **Trong Java, phân biệt sự khác biệt giữa `a==b` và `a.equals(b)`**: Trong Java, `a==b` kiểm tra hai đối tượng có trỏ tới cùng một địa chỉ không, trong khi `a.equals(b)` kiểm tra giá trị logic bên trong đối tượng. Nói chung, nên sử dụng `a.equals(b)` thay vì `a==b`.

19.2 Câu lệnh ghép (Compound statements / Blocks)

Một "compound statement" (câu lệnh ghép) hay "block" là tập hợp các câu lệnh được xem như một câu lệnh duy nhất khi kiểm soát luồng thực thi chương trình. Các câu lệnh ghép được tạo bằng cặp `{` và `}` trong C++, C#, C và Java. Đôi khi các từ khóa cũng ngầm định một block, như For/Next trong Visual Basic. Các hướng dẫn sử dụng block hiệu quả như sau:

- **Viết cặp dấu ngoặc nhọn cùng lúc**: Viết `{` và `}` ngay khi bắt đầu một block, rồi điền phần thân sau.
  - Viết đầu tiên: `cpp for ( i = 0; i < maxLines; i++ )`
  - Viết tiếp: `cpp for ( i = 0; i < maxLines; i++ ) { }`
  - Viết cuối: `cpp for ( i = 0; i < maxLines; i++ ) { // nội dung ở đây }`

  Nguyên tắc này áp dụng cho mọi cấu trúc chặn, như `if`, `for`, `while` trong C++ và Java, hay `If-Then-Else`, `For-Next`, `While-Wend` trong Visual Basic.
- **Sử dụng dấu ngoặc nhọn để làm rõ điều kiện**: Luôn dùng `{}` bao quanh cả khi khối mã chỉ có một dòng, vì trong thực tế bảo trì, nó có thể phát triển và dễ gây lỗi khi quên `{}`.
- **Sử dụng block để thể hiện rõ ý định, dù mã trong block chỉ có 1 hay 20 dòng**.

19.3 Câu lệnh rỗng (Null statements)

Trong C++, có thể có câu lệnh rỗng—một dòng chỉ gồm dấu chấm phẩy, như sau:

Ví dụ truyền thống về câu lệnh rỗng trong C++

```
while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() )
    ;
```

`while` trong C++ yêu cầu một câu lệnh theo sau, có thể là câu lệnh rỗng. Chấm phẩy một mình trên dòng là câu lệnh rỗng.

Hướng dẫn với câu lệnh rỗng trong C++:

- **Làm nổi bật câu lệnh rỗng**: Đặt dấu chấm phẩy trên dòng riêng và thụt lề như một câu lệnh, để dễ nhận biết. `cpp while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) ;`

- **Hoặc dùng cặp ngoặc nhọn rỗng**: `cpp while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) {}`
- **Tạo macro hoặc hàm inline DoNothing() để làm rõ hơn ý đồ**: `cpp #define DoNothing() while ( recordArray.Read( index++ ) != recordArray.EmptyRecord() ) { DoNothing(); }`

Cách này tương tự như đánh dấu trang trắng trong tài liệu: "This page intentionally left blank".

Nếu ngôn ngữ bạn dùng không hỗ trợ macro/hàm inline, hãy tạo một hàm DoNothing() trả về ngay lập tức.

- **Xem xét liệu việc dùng thân vòng lặp rỗng có thực sự cần thiết**: Nhiều vòng lặp rỗng dựa vào hiệu ứng phụ trong phần kiểm soát vòng lặp, nên có thể nên viết lại để tăng rõ ràng:

```
RecordType record = recordArray.Read( index );
index++;
while ( record != recordArray.EmptyRecord() ) {
    record = recordArray.Read( index );
    index++;
}
```

Mặc dù mã dài hơn, nhưng rõ ràng hơn và dễ bảo trì.

19.4 Chinh phục việc lồng ghép sâu nguy hiểm (Taming Dangerously Deep Nesting)

Việc lồng ghép quá nhiều cấp (nested) đã bị chỉ trích trong tài liệu khoa học máy tính từ hơn 25 năm, và là nguyên nhân chính gây ra mã nguồn khó hiểu. Các nghiên cứu cho thấy ít người có thể hiểu được mã lồng ghép quá ba cấp. Nhiều chuyên gia khuyên tránh lồng ghép quá ba hoặc bốn cấp.

Lồng ghép sâu cản trở việc quản lý độ phức tạp—yếu tố kỹ thuật cốt lõi của phần mềm. Do vậy, hãy tránh lồng ghép sâu.

Có thể tránh lồng ghép sâu bằng cách: - Thiết kế lại biểu thức điều kiện trong `if`, `else` - Tách/re-factor code thành các hàm/routine đơn giản hơn

Giảm mức lồng ghép bằng cách kiểm tra lại điều kiện (Retesting part of the condition)

Nếu lồng ghép quá sâu, bạn có thể giảm số cấp bằng cách kiểm tra lại một số điều kiện, ví dụ:

Ví dụ C++ về mã lồng ghép sâu (xấu)

```
if ( inputStatus == InputStatus_Success ) {
    // nhiều mã
    if ( printerRoutine != NULL ) {
        // nhiều mã
        if ( SetupPage() ) {
            // nhiều mã
            if ( AllocMem( &printData ) ) {
                // nhiều mã
            }
        }
    }
}
```

Cải tiến bằng kiểm tra lại (retesting):

```
if ( inputStatus == InputStatus_Success ) {
    // nhiều mã

    if ( printerRoutine != NULL ) {
        // nhiều mã
    }
}
if ( ( inputStatus == InputStatus_Success ) &&
     ( printerRoutine != NULL ) &&
      SetupPage() ) {
    // nhiều mã

    if ( AllocMem( &printData ) ) {
        // nhiều mã
    }
}
```

Cái giá là biểu thức kiểm tra phức tạp hơn, nhưng độ lồng ghép giảm đáng kể.

Giảm lồng ghép bằng khối break (break block)

Một phương pháp khác là dùng khối break để thoát sớm:

```
do { // bắt đầu khối break
    if ( inputStatus != InputStatus_Success ) { break; }
    // nhiều mã
    if ( printerRoutine == NULL ) { break; }
    // nhiều mã
    if ( !SetupPage() ) { break; }
    // nhiều mã
```

```
    if ( !AllocMem( &printData ) ) { break; }
    // nhiều mã
} while (FALSE); // kết thúc khối break
```

Lưu ý: Sử dụng kỹ thuật này chỉ khi cả nhóm phát triển đều quen thuộc, tránh gây khó hiểu.

Chuyển if lồng ghép thành chuỗi if-else

Nhiều cây quyết định if lồng ghép có thể tổ chức lại thành chuỗi if-else:

Ví dụ Java về cây quyết định lồng ghép thừa

```
if ( 10 < quantity ) {
    if ( 100 < quantity ) {
        if ( 1000 < quantity ) {
            discount = 0.10;
        }
        else {
            discount = 0.05;
        }
    }
    else {
        discount = 0.025;
    }
}
else {
    discount = 0.0;
}
```

Các phép so sánh thừa, vì kiểm tra `quantity > 1000` thì đương nhiên lớn hơn các mốc phía trước. (Phần này kết thúc ở đây do bản gốc dừng lại.)

Chú thích: - Dịch các đoạn code được giữ nguyên trong markdown code block. - Các ghi chú, thuật ngữ kỹ thuật (đánh giá short-circuit, macro, block …) đảm bảo nhất quán và làm rõ ngữ nghĩa. - Lỗi nhỏ (ví dụ: dấu phẩy, từ thừa, tách dòng thừa trong bản gốc) đã được chỉnh sửa hợp lý để bản dịch liền mạch, dễ hiểu.

Ví dụ Java về việc chuyển đổi if lồng nhau sang chuỗi if-then-else khi các số "không gọn gàng"

Khi các giá trị kiểm tra không có tính liên tục hoặc gọn gàng, ta có thể viết lại cấu trúc if lồng nhau thành chuỗi các mệnh đề if-then-else như sau:

```
if ( 1000 < quantity ) {
    discount = 0.10;
```

```
}
else if ( ( 100 < quantity ) && ( quantity <= 1000 ) ) {
    discount = 0.05;
}
else if ( ( 10 < quantity ) && ( quantity <= 100 ) ) {
    discount = 0.025;
}
else if ( quantity <= 10 ) {
    discount = 0;
}
```

Sự khác biệt chính giữa đoạn mã này và mã trước đó là các biểu thức trong các mệnh đề else-if không phụ thuộc vào kết quả của các kiểm tra trước đó. Đoạn mã này không cần có các mệnh đề else để hoạt động đúng, và thực tế có thể thực hiện các kiểm tra theo bất kỳ thứ tự nào. Tuy nhiên, sử dụng else giúp tránh việc lặp lại các điều kiện kiểm tra không cần thiết.

Chuyển đổi if lồng nhau sang câu lệnh case

Với một số loại kiểm tra nhất định, đặc biệt là khi xử lý các số nguyên (integer), bạn có thể chuyển từ chuỗi if-then-else sang câu lệnh case (ví dụ: switch-case). Phương pháp này không phải ngôn ngữ nào cũng hỗ trợ, nhưng lại rất hữu ích nếu có.

Ví dụ dưới đây minh họa cách chuyển đổi trong Visual Basic:

```
Select Case quantity
    Case 0 To 10
        discount = 0.0
    Case 11 To 100
        discount = 0.025
    Case 101 To 1000
        discount = 0.05
    Case Else
        discount = 0.10
End Select
```

Ví dụ này **rõ ràng, dễ đọc**, đặc biệt nếu so với các ví dụ về lồng ghép nhiều cấp độ trình bày trước đó.

Phân tách mã lồng nhau thành các routine (thủ tục) riêng

Nếu việc lồng ghép nhiều cấp độ xuất hiện bên trong vòng lặp, có thể cải thiện tính rõ ràng bằng cách đưa phần lõi của vòng lặp vào các routine riêng biệt. Điều này đặc biệt hiệu quả khi nguyên nhân lồng ghép là do cả hai yếu tố: điều

kiện và lặp. Mục đích là giữ lại các nhánh điều kiện (if-then-else) ở mức ngoài cùng để thể hiện nhánh quyết định, đồng thời chuyển phần xử lý cụ thể vào các routine riêng.

Ví dụ C++ về mã lồng nhau cần được tách routine

```
while ( !TransactionsComplete() ) {
    // read transaction record
    transaction = ReadTransaction();
    // process transaction depending on type of transaction
    if ( transactionType == TransactionType_Deposit ) {
        // process a deposit
        if ( transactionAccountType == AccountType_Checking ) {
            if ( transactionAccountSubType == AccountSubType_Business )
                MakeBusinessCheckDep( transactionAccountNum, transactionAmount );
            else if ( transactionAccountSubType == AccountSubType_Personal )
                MakePersonalCheckDep( transactionAccountNum, transactionAmount );
            else if ( transactionAccountSubType == AccountSubType_School )
                MakeSchoolCheckDep( transactionAccountNum, transactionAmount );
        }
        else if ( transactionAccountType == AccountType_Savings )
            MakeSavingsDep( transactionAccountNum, transactionAmount );
        else if ( transactionAccountType == AccountType_DebitCard )
            MakeDebitCardDep( transactionAccountNum, transactionAmount );
        else if ( transactionAccountType == AccountType_MoneyMarket )
            MakeMoneyMarketDep( transactionAccountNum, transactionAmount );
        else if ( transactionAccountType == AccountType_Cd )
            MakeCDDep( transactionAccountNum, transactionAmount );
    }
    else if ( transactionType == TransactionType_Withdrawal ) {
        // process a withdrawal
        if ( transactionAccountType == AccountType_Checking )
            MakeCheckingWithdrawal( transactionAccountNum, transactionAmount );
        else if ( transactionAccountType == AccountType_Savings )
            MakeSavingsWithdrawal( transactionAccountNum, transactionAmount );
        else if ( transactionAccountType == AccountType_DebitCard )
            MakeDebitCardWithdrawal( transactionAccountNum, transactionAmount );
    }
    else if ( transactionType == TransactionType_Transfer ) {
        MakeFundsTransfer(
            transactionSourceAccountType,
            transactionTargetAccountType,
            transactionAccountNum,
            transactionAmount
        );
    }
```

```
    else {
        // process unknown kind of transaction
        LogTransactionError( "Unknown Transaction Type", transaction );
    }
}
```

Mặc dù khá phức tạp, mã trên vẫn chưa phải là tệ nhất. Mã chỉ lồng tối đa bốn cấp, có bình luận, cấu trúc thụt đầu dòng rõ ràng và phân rã hợp lý, đặc biệt với trường hợp TransactionType_Transfer. Tuy vậy, có thể **nâng cao** hơn nữa bằng việc di chuyển các phần kiểm tra điều kiện bên trong thành các routine riêng.

Ví dụ C++ sau khi phân rã mã lồng nhau thành routine

```
while ( !TransactionsComplete() ) {
    // read transaction record
    transaction = ReadTransaction();
    // process transaction depending on type of transaction
    if ( transactionType == TransactionType_Deposit ) {
        ProcessDeposit(
            transactionAccountType,
            transactionAccountSubType,
            transactionAccountNum,
            transactionAmount
        );
    }
    else if ( transactionType == TransactionType_Withdrawal ) {
        ProcessWithdrawal(
            transactionAccountType,
            transactionAccountNum,
            transactionAmount
        );
    }
    else if ( transactionType == TransactionType_Transfer ) {
        MakeFundsTransfer(
            transactionSourceAccountType,
            transactionTargetAccountType,
            transactionAccountNum,
            transactionAmount
        );
    }
    else {
        // process unknown transaction type
        LogTransactionError("Unknown Transaction Type", transaction );
    }
}
```

Các routine mới được lấy trực tiếp từ routine ban đầu. **Lợi ích:**

1. Việc lồng ghép chỉ còn hai mức, cấu trúc đơn giản và dễ hiểu hơn.
2. Vòng lặp while ngắn hơn, có thể đọc, sửa đổi, kiểm tra trên một màn hình mà không cần chia nhỏ.
3. Đưa chức năng vào các routine riêng giúp tận dụng các lợi ích về mô đun hóa.
4. Sau phân rã, rất dễ chuyển đổi đoạn mã thành switch-case để rõ ràng hơn:

Ví dụ C++ sử dụng switch-case sau khi phân rã

```
while ( !TransactionsComplete() ) {
    // read transaction record
    transaction = ReadTransaction();
    // process transaction depending on type of transaction
    switch ( transactionType ) {
        case TransactionType_Deposit:
            ProcessDeposit(
                transactionAccountType,
                transactionAccountSubType,
                transactionAccountNum,
                transactionAmount
            );
            break;
        case TransactionType_Withdrawal:
            ProcessWithdrawal(
                transactionAccountType,
                transactionAccountNum,
                transactionAmount
            );
            break;
        case TransactionType_Transfer:
            MakeFundsTransfer(
                transactionSourceAccountType,
                transactionTargetAccountType,
                transactionAccountNum,
                transactionAmount
            );
            break;
        default:
            // process unknown transaction type
            LogTransactionError("Unknown Transaction Type", transaction );
            break;
    }
}
```

Sử dụng hướng tiếp cận hướng đối tượng

Một cách đơn giản hóa mã kể trên trong môi trường lập trình hướng đối tượng là tạo lớp cơ sở trừu tượng Transaction cùng các lớp con cho các loại giao dịch: Deposit, Withdrawal, Transfer.

Ví dụ C++ sử dụng tính đa hình (polymorphism)

```
TransactionData transactionData;
Transaction *transaction;
while ( !TransactionsComplete() ) {
    // read transaction record
    transactionData = ReadTransaction();
    // create transaction object, depending on type of transaction
    switch ( transactionDataType ) {
        case TransactionType_Deposit:
            transaction = new Deposit( transactionData );
            break;
        case TransactionType_Withdrawal:
            transaction = new Withdrawal( transactionData );
            break;
        case TransactionType_Transfer:
            transaction = new Transfer( transactionData );
            break;
        default:
            // process unknown transaction type
            LogTransactionError("Unknown Transaction Type", transactionData );
            return;
    }
    transaction->Complete();
    delete transaction;
}
```

Trong một hệ thống thực tế quy mô lớn, câu lệnh switch trên thường được thay bằng **factory method** có thể tái sử dụng bất cứ khi nào cần tạo đối tượng kiểu Transaction.

Ví dụ C++ sử dụng đa hình cùng Object Factory

```
TransactionData transactionData;
Transaction *transaction;
while ( !TransactionsComplete() ) {
    // read transaction record and complete transaction
    transactionData = ReadTransaction();
    transaction = TransactionFactory::Create( transactionData );
    transaction->Complete();
    delete transaction;
```

```
}
```

Routine **TransactionFactory::Create()** chỉ đơn giản là chuyển đổi từ switch-case sang phương thức tạo đối tượng:

```
Transaction *TransactionFactory::Create(
    TransactionData transactionData
) {
    // create transaction object, depending on type of transaction
    switch ( transactionDataType ) {
        case TransactionType_Deposit:
            return new Deposit( transactionData );
        case TransactionType_Withdrawal:
            return new Withdrawal( transactionData );
        case TransactionType_Transfer:
            return new Transfer( transactionData );
        default:
            // process unknown transaction type
            LogTransactionError("Unknown Transaction Type", transactionData );
            return NULL;
    }
}
```

Thiết kế lại mã lồng ghép sâu

Một số chuyên gia lập luận rằng việc sử dụng câu lệnh case thường là dấu hiệu của việc thiếu phân rã hợp lý trong lập trình hướng đối tượng và rất ít khi cần thiết (Meyer, 1997). Việc chuyển đổi từ switch-case sang factory kết hợp đa hình là ví dụ điển hình.

Nói chung, **mã phức tạp luôn là dấu hiệu bạn chưa hiểu đủ rõ để đơn giản hóa chương trình của mình**. Lồng ghép quá sâu là cảnh báo cần thiết kế lại hoặc phân rã routine cho hợp lý. Không nhất thiết phải sửa đổi mọi routine lồng ghép sâu, nhưng nếu không thực hiện, cần có lý do chính đáng.

Tóm tắt các kỹ thuật giảm lồng ghép sâu

Sau đây là các kỹ thuật bạn có thể sử dụng để giảm mức độ lồng ghép sâu, kèm theo tham chiếu tới các phần tương ứng trong cuốn sách:

- Kiểm tra lại từng phần của điều kiện (ngay trong mục này)
- Chuyển đổi sang if-then-else (ngay trong mục này)
- Chuyển đổi sang câu lệnh case (ngay trong mục này)
- Phân tách mã lồng ghép sâu thành các routine riêng (ngay trong mục này)
- Sử dụng đối tượng và phân bổ đa hình (ngay trong mục này)

- Viết lại mã để sử dụng biến trạng thái (ở Mục 17.3)
- Sử dụng guard clause (câu điều kiện bảo vệ) để thoát routine, giúp làm rõ luồng xử lý chính (ở Mục 17.1)
- Sử dụng exception (Xử lý ngoại lệ - Mục 8.4)
- Thiết kế lại mã lồng ghép sâu hoàn toàn (ngay trong mục này)

19.5 Nền tảng lập trình: Structured Programming (Lập trình có cấu trúc)

Khái niệm *structured programming* (lập trình có cấu trúc) được khởi xướng trong một bài báo nổi tiếng "Structured Programming" bởi Edsger Dijkstra tại hội nghị NATO năm 1969 (Dijkstra, 1969). Khi khái niệm này phổ biến, thuật ngữ "structured" cũng được áp dụng cho hàng loạt hoạt động phát triển phần mềm như structured analysis (phân tích có cấu trúc), structured design (thiết kế có cấu trúc),... Tuy nhiên, không có điểm chung rõ rệt giữa các phương pháp ngoài việc xuất hiện trong cùng thời kỳ mà từ "structured" đem lại giá trị thời thượng.

Cốt lõi của *structured programming* là ý tưởng đơn giản: một chương trình chỉ nên sử dụng *one-in, one-out control constructs* (cấu trúc điều khiển vào-một, ra-một, còn gọi là single-entry, single-exit control constructs). Nghĩa là, một khối mã chỉ có một điểm bắt đầu và một điểm kết thúc duy nhất, không có điểm vào hoặc ra khác.

Structured programming KHÔNG giống với structured, top-down design (thiết kế từ trên xuống có cấu trúc); nó chỉ áp dụng ở cấp độ mã chi tiết.

Một chương trình được lập trình có cấu trúc sẽ tiến trình một cách có trật tự, kỷ luật, dễ đọc từ trên xuống dưới. Cách tiếp cận ít kỷ luật hơn sẽ khiến mã nguồn khó hiểu và khó bảo trì, từ đó làm giảm chất lượng chương trình.

Các khái niệm trung tâm của *structured programming* vẫn còn hữu ích đến nay, đặc biệt liên quan tới việc sử dụng break, continue, throw, catch, return và các chủ đề liên quan.

Ba thành phần cốt lõi của *structured programming*

Ba cấu trúc điều khiển tạo nên nền tảng của *structured programming* gồm: Sequence (trình tự), Selection (lựa chọn) và Iteration (lặp).

Sequence (Trình tự)

Một sequence là tập hợp các câu lệnh được thực thi theo thứ tự. Thường là các lệnh gán hoặc gọi routine.

Ví dụ Java về mã tuyến tính (sequence):

```
// một chuỗi các lệnh gán
a = "1";
b = "2";
c = "3";

// một chuỗi gọi hàm
System.out.println(a);
System.out.println(b);
System.out.println(c);
```

Selection (Lựa chọn)

Selection là cấu trúc điều khiển cho phép lựa chọn thực thi câu lệnh dựa trên điều kiện. Điển hình là câu lệnh if-then-else. Chỉ một trong hai nhánh sẽ được thực hiện.

Câu lệnh switch-case cũng là ví dụ về selection, như trong C++, Java hoặc select trong Visual Basic.

Ví dụ Java về selection:

```
// lựa chọn với if
if ( totalAmount > 0.0 ) {
    // làm gì đó
}
else {
    // làm gì khác
}

// lựa chọn với switch-case
switch ( commandShortcutLetter ) {
    case 'a':
        PrintAnnualReport();
        break;
    case 'q':
        PrintQuarterlyReport();
        break;
    case 's':
        PrintSummaryReport();
        break;
    default:
        DisplayInternalError("Internal Error 905: Call customer support.");
}
```

Iteration (Lặp)

Iteration là cấu trúc điều khiển cho phép thực hiện lặp lại một nhóm câu lệnh nhiều lần, thường gọi là "vòng lặp". Các loại vòng lặp phổ biến: For-Next trong Visual Basic, while và for trong C++, Java.

Ví dụ Visual Basic về iteration:

```
' vòng lặp For
For index = first To last
    DoSomething(index)
Next

' vòng lặp While
index = first
While (index <= last)
    DoSomething(index)
    index = index + 1
Wend

' vòng lặp Do với điều kiện thoát
index = first
Do
    If (index > last) Then Exit Do
    DoSomething(index)
    index = index + 1
Loop
```

Luận điểm cốt lõi: Bất cứ luồng điều khiển nào cũng có thể được tạo thành từ ba cấu trúc này (sequence, selection, iteration)—(Böhm Jacopini, 1966). Mọi cấu trúc điều khiển khác (break, continue, return, throw-catch, …) nên được cân nhắc kỹ lưỡng khi sử dụng.

19.6 Cấu trúc điều khiển và độ phức tạp

Cấu trúc điều khiển là một trong những yếu tố chính góp phần tạo nên độ phức tạp của chương trình. Việc sử dụng chúng không hợp lý sẽ làm tăng độ phức tạp, ngược lại dùng tốt sẽ giúp mã nguồn đơn giản hơn.

> *Hãy làm mọi việc đơn giản nhất có thể—nhưng không đơn giản hơn.*
> — Albert Einstein

Một tiêu chí để đánh giá "độ phức tạp lập trình" là số lượng đối tượng tinh thần mà bạn phải cùng lúc ghi nhớ để hiểu một chương trình. Việc "giữ bóng tinh thần" này là lý do lập trình đòi hỏi sự tập trung cao, và cũng lý giải vì sao lập trình viên dễ bị ảnh hưởng bởi "các gián đoạn nhanh".

Độ phức tạp của chương trình tỷ lệ thuận với công sức cần thiết để hiểu nó. Tom McCabe đã xuất bản một bài báo có ảnh hưởng lớn, cho rằng độ phức tạp của chương trình được xác định dựa trên luồng điều khiển (control flow) (1976). Một số nghiên cứu khác bổ sung thêm các tiêu chí khác như số lượng biến được sử dụng, nhưng đều đồng thuận rằng control flow là yếu tố then chốt.

Độ quan trọng của độ phức tạp

Các nhà nghiên cứu nhận thức rõ tầm quan trọng của độ phức tạp đã nhiều thập kỷ nay. Edsger Dijkstra từng nhấn mạnh giới hạn nhận thức của con người: *"Lập trình viên giỏi biết rất rõ kích thước hạn chế của bộ não mình, nên phải khiêm tốn khi lập trình"* (Dijkstra, 1972). Điều này không có nghĩa nên tăng khả năng ghi nhớ để xử lý độ phức tạp lớn, mà là nên **giảm độ phức tạp của chương trình càng nhiều càng tốt.**

Một số nghiên cứu thực nghiệm cũng chỉ ra rằng độ phức tạp của control flow tỉ lệ nghịch với độ tin cậy và số lỗi phần mềm. Ví dụ tại Hewlett-Packard, chương trình sử dụng thước đo độ phức tạp của McCabe giúp giảm số lỗi rõ rệt so với các chương trình khác, với tỉ lệ lỗi hậu phát hành thấp hơn nhiều.

Tóm lại, bạn chỉ có thể giảm độ phức tạp của phần mềm bằng hai cách: hoặc rèn luyện khả năng ghi nhớ (giới hạn), hoặc thiết kế để giảm độ phức tạp! Trong thực tế, giải pháp hiệu quả là thiết kế để độ phức tạp trong phạm vi có thể kiểm soát được.

Đo lường độ phức tạp như thế nào?

Bạn có thể cảm nhận trực quan đâu là routine phức tạp hay đơn giản hơn. Các nhà nghiên cứu đã cố gắng chính xác hóa trực giác này thông qua nhiều chỉ số đo lường khác nhau, trong đó nổi bật nhất là chỉ số McCabe's cyclomatic complexity.

Đo độ phức tạp điều khiển dựa trên số điểm quyết định theo Tom McCabe

Tom McCabe đã đề xuất phương pháp đo độ phức tạp bằng cách đếm số **"decision point"** (điểm quyết định) trong một routine (thủ tục/chương trình con). Bảng 19-2 mô tả phương pháp đếm điểm quyết định này.

Bảng 19-2: Kỹ thuật đếm điểm quyết định trong một routine

1. Khởi đầu với số 1 cho đường đi thẳng qua routine.
2. Cộng thêm 1 cho mỗi từ khóa sau, hoặc các từ tương đương: **if**, **while**, **repeat**, **for**, **and**, **or**.
3. Cộng thêm 1 cho mỗi trường hợp (**case**) trong một câu lệnh **case**.

Ví dụ minh họa:

```
if ( ( (status = Success) and done ) or
     ( not done and ( numLines >= maxLines ) ) )
```

Trong đoạn mã này: - Đếm 1 để bắt đầu. - Đếm thêm 1 cho **if**, thành 2. - Đếm thêm 1 cho **and**, thành 3. - Đếm thêm 1 cho **or**, thành 4. - Đếm thêm 1 cho **and** thứ hai, thành 5.

Như vậy, đoạn mã này chứa tổng cộng **năm điểm quyết định**.

Xử lý kết quả đo độ phức tạp

Sau khi đã đếm số điểm quyết định, bạn có thể sử dụng con số này để phân tích độ phức tạp của routine:

- **0–5**: Routine có thể coi là ổn.
- **6–10**: Nên bắt đầu suy nghĩ về cách đơn giản hóa routine.
- **Trên 10**: Hãy tách một phần routine thành routine khác và gọi từ routine ban đầu.

Việc chuyển một phần của routine sang routine khác không làm giảm tổng độ phức tạp của toàn chương trình; nó chỉ thay đổi vị trí của các điểm quyết định. Tuy nhiên, việc này giúp giảm số lượng điểm quyết định mà bạn phải xử lý cùng lúc trong một routine cụ thể. Bởi mục tiêu quan trọng là giảm số lượng thành phần phải ghi nhớ đồng thời, việc đơn giản hóa từng routine là điều đáng làm.

Giới hạn tối đa 10 điểm quyết định không phải là một quy tắc tuyệt đối. Hãy coi số điểm quyết định như một cờ cảnh báo rằng routine có thể cần thiết kế lại, thay vì là một giới hạn không linh hoạt. Một câu lệnh **case** với nhiều trường hợp có thể dài hơn 10 phần tử và, tùy thuộc vào mục đích của câu lệnh đó, việc chia nhỏ có thể là không hợp lý.

Các loại độ phức tạp khác

> Đọc thêm: Để tham khảo chi tiết về các chỉ số phức tạp, xem **Software Engineering Metrics and Models** (Conte, Dunsmore và Shen, 1986).

Chỉ số phức tạp của McCabe không phải là thước đo duy nhất hợp lý, nhưng nó là chỉ số được nhắc đến nhiều nhất trong tài liệu khoa học máy tính, đặc biệt hữu ích khi bạn xem xét dòng điều khiển. Một số thước đo khác bao gồm: - Lượng dữ liệu được sử dụng. - Số lượng mức lồng nhau trong các cấu trúc điều khiển. - Số dòng mã (**lines of code**). - Số dòng giữa các lần tham chiếu liên tiếp đến biến ("**span**"). - Số dòng một biến được sử dụng ("**live time**"). - Lượng dữ liệu đầu vào/đầu ra (**input/output**).

Một số nhà nghiên cứu đã xây dựng các chỉ số tổng hợp dựa trên việc kết hợp các chỉ số đơn giản này.

Danh sách kiểm tra: Các vấn đề về cấu trúc điều khiển (Control-Structure Issues)

- Các biểu thức có sử dụng **true** và **false** thay vì 1 và 0 không?
- Các giá trị kiểu boolean có được so sánh ngầm với **true** và **false** không?
- Các giá trị số có được so sánh tường minh với giá trị kiểm tra không?
- Các biểu thức đã được đơn giản hóa bằng cách bổ sung biến **boolean** mới và sử dụng hàm boolean hoặc bảng quyết định (**decision table**) chưa?
- Các biểu thức boolean được phát biểu theo hướng khẳng định chưa?
- Các cặp ngoặc nhọn (**brace**) có cân đối không?
- Có sử dụng ngoặc nhọn đầy đủ ở những nơi cần thiết để đảm bảo rõ ràng không?
- Các biểu thức logic đã được đặt dấu ngoặc đầy đủ chưa?
- Các điều kiện kiểm thử đã sắp xếp theo thứ tự trên trục số chưa (**number-line order**)?
- Trong Java, đã sử dụng phong cách **a.equals(b)** thay cho **a == b** khi thích hợp chưa?
- Các câu lệnh rỗng (**null statement**) có rõ ràng không?
- Các cấu trúc lồng nhau đã được đơn giản hóa thông qua kiểm tra lại một phần điều kiện, chuyển đổi sang **if-then-else** hoặc **case statement**, chuyển mã lồng nhau thành routine riêng, chuyển sang thiết kế hướng đối tượng hơn, hoặc cải thiện bằng cách khác chưa?
- Nếu một routine có số điểm quyết định trên 10, có lý do hợp lý nào để không thiết kế lại routine đó không?

Đặt tên cho routine càng dài càng tốt khi cần thiết

Nghiên cứu cho thấy độ dài tối ưu trung bình của một tên biến là từ 9 đến 15 ký tự. Routine (thủ tục/hàm) thường phức tạp hơn biến, và do đó tên gọi phù hợp cho chúng có xu hướng dài hơn. Tuy nhiên, tên routine thường được gắn kèm với tên đối tượng, điều này thực chất đã cung cấp một phần của tên miễn

phí. Nhìn chung, khi tạo tên cho routine, điều quan trọng là làm cho tên trở nên rõ ràng nhất có thể, tức là nên đặt tên dài hoặc ngắn tùy ý miễn sao đảm bảo dễ hiểu.

> **Tham khảo thêm:** Để đặt tên cho function, hãy sử dụng mô tả của giá trị trả về. Một function trả về một giá trị, và function nên được đặt tên theo giá trị mà nó trả về. Ví dụ, `cos()`, `customerId.Next()`, `printer.IsReady()`, và `pen.CurrentColor()` đều là những tên function hợp lý, chỉ rõ ràng giá trị mà function trả về.

Để đặt tên cho procedure (thủ tục), hãy bắt đầu bằng một động từ mạnh, sau đó là đối tượng. Một procedure với functional cohesion (liên kết chức năng) thường thực hiện một thao tác trên một đối tượng nào đó. Tên gọi của procedure nên phản ánh hành động mà thủ tục thực hiện, nghĩa là tên nên có dạng động từ+kết hợp với đối tượng. Ví dụ: `PrintDocument()`, `CalcMonthlyRevenues()`, `CheckOrderInfo()`, và `RepaginateDocument()` đều là những tên procedure tốt.

Trong các ngôn ngữ lập trình hướng đối tượng, bạn không cần phải bao gồm tên đối tượng trong tên procedure vì bản thân đối tượng đã được đề cập trong lệnh gọi. Bạn sẽ gọi các routine như: `document.Print()`, `orderInfo.Check()`, và `monthlyRevenues.Calc()`.

Nếu đặt tên ví dụ như `document.PrintDocument()` thì điều này trở nên thừa và có thể gây sai lệch khi áp dụng cho các lớp dẫn xuất. Ví dụ, nếu `Check` là một lớp dẫn xuất từ `Document`, câu lệnh `check.Print()` sẽ rõ ràng là in một séc, trong khi `check.PrintDocument()` nghe như đang in sổ séc hoặc báo cáo tháng, nhưng lại không rõ là in séc.

> **Tham khảo thêm:** Về các cặp thuật ngữ đối lập thường gặp khi đặt tên biến, xem mục "Common Opposite-pairs in Variable Names" tại Phần 11.1.

Sử dụng các cặp từ đối lập một cách chính xác

Việc áp dụng quy ước đặt tên cho các cặp thuật ngữ đối lập giúp tăng tính nhất quán và nâng cao khả năng đọc mã nguồn. Các cặp đối lập như `first/last` là những cặp phổ biến và dễ hiểu. Tuy nhiên, các cặp đối lập như `FileOpen()` và `_lclose()` lại không đối xứng và dễ gây nhầm lẫn. Dưới đây là một số cặp đối lập thông dụng:

- add/remove
- increment/decrement
- open/close
- begin/end
- insert/delete
- show/hide
- create/destroy
- lock/unlock
- source/target

- first/last
- min/max
- start/stop
- get/put
- next/previous
- up/down
- get/set
- old/new

Thiết lập quy ước đặt tên cho các phép toán phổ biến

Trong một số hệ thống, việc phân biệt các loại phép toán là rất quan trọng. Áp dụng quy ước đặt tên là phương pháp dễ dàng và tin cậy nhất để chỉ ra sự khác biệt này.

Ví dụ, trong một dự án, mỗi đối tượng được gán một định danh duy nhất (unique identifier). Tuy nhiên, nhóm phát triển đã không thống nhất quy ước đặt tên routine để trả về identifier của đối tượng, dẫn đến nhiều tên routine như sau:

```
employee id Get()
dependent GetId()
supervisor()
candidate id()
```

Lớp `Employee` truy cập đối tượng `id`, sau đó gọi routine `Get()`. Lớp `Dependent` cung cấp routine `GetId()`. Lớp `Supervisor` trả về giá trị `id` mặc định. Lớp `Candidate` sử dụng mặc định giá trị trả về của đối tượng `id` và truy xuất `id`. Đến giữa dự án, không ai còn nhớ routine nào dùng cho đối tượng nào, nhưng lúc đó quá nhiều code đã được viết rồi nên không thể sửa đồng bộ được nữa. Hệ quả là mỗi thành viên phải dành không ít thời gian để nhớ cú pháp lấy `id` tương ứng với từng class. Nếu có quy ước đặt tên cho routine truy xuất id ngay từ đầu, sự bất tiện này đã có thể tránh được.

7.4 Routine có thể dài đến mức nào?

Khi đến Mỹ, nhóm Pilgrims tranh luận về độ dài tối đa nên có của một routine. Sau cả chuyến đi dài, họ vẫn chưa đi đến thống nhất, và vì thế vấn đề về độ dài tối đa của routine vẫn còn là chủ đề bàn luận bất tận cho đến ngày nay.

Độ dài tối đa lý tưởng của routine thường được mô tả là vừa một màn hình hoặc một đến hai trang tài liệu chương trình, tức khoảng 50 đến 150 dòng. Ví dụ, IBM từng giới hạn routine ở mức 50 dòng, còn TRW giới hạn ở mức hai trang (McCabe 1976). Trong các chương trình hiện đại, thường có rất nhiều routine ngắn đan xen với một số routine dài. Tuy nhiên, routine dài vẫn chưa hề biến mất. Gần đây, tôi từng làm việc tại hai khách hàng, một nơi các lập

trình viên đang phải vật lộn với một routine khoảng 4.000 dòng, còn nơi kia là routine hơn 12.000 dòng!

Rất nhiều nghiên cứu đã được thực hiện về độ dài routine, một số kết quả có giá trị đối với các chương trình hiện đại, số khác thì không:

- **Nghiên cứu của Basili và Perricone** cho thấy độ lớn của routine tỷ lệ nghịch với số lỗi: khi độ dài routine tăng lên (đến 200 dòng code), số lỗi trên mỗi dòng code lại giảm (Basili và Perricone 1984).
- Một nghiên cứu khác nhận thấy độ lớn routine không liên quan đến số lỗi, mặc dù độ phức tạp về cấu trúc và lượng dữ liệu lại có liên hệ với số lỗi (Shen et al. 1985).
- Nghiên cứu năm 1986 phát hiện rằng routine nhỏ (32 dòng code trở xuống) không có mối tương quan với chi phí phát triển hoặc tỷ lệ lỗi thấp hơn (Card, Church, và Agresti 1986; Card và Glass 1990). Bằng chứng cho thấy routine lớn hơn (65 dòng trở lên) lại có chi phí phát triển trên mỗi dòng code rẻ hơn.
- Nghiên cứu thực nghiệm trên 450 routine nhận thấy các routine nhỏ (dưới 143 câu lệnh nguồn, kể cả chú thích) có số lỗi trên mỗi dòng code cao hơn 23% so với routine lớn, nhưng chi phí sửa lỗi lại thấp hơn 2,4 lần (Selby và Basili 1991).
- Một nghiên cứu khác cho thấy code cần thay đổi ít nhất khi routine có độ dài trung bình từ 100 đến 150 dòng code (Lind và Vairavan 1989).
- Nghiên cứu tại IBM cho thấy các routine dễ mắc lỗi nhất là routine dài hơn 500 dòng. Vượt quá 500 dòng, tỷ lệ lỗi tỷ lệ thuận với độ dài routine (Jones 1986a).

Routine trong chương trình hướng đối tượng nên dài bao nhiêu?

Đa số routine trong các chương trình hướng đối tượng sẽ là accessor routine, vốn rất ngắn. Tuy nhiên, đôi khi các thuật toán phức tạp sẽ dẫn đến routine dài hơn, và trong những trường hợp này, routine nên phát triển tự nhiên lên đến 100–200 dòng (đếm các dòng mã nguồn không bao gồm comment hoặc dòng trống).

Nhiều thập kỷ nghiên cứu cho thấy routine dài như vậy không dễ mắc lỗi hơn routine ngắn. Các yếu tố như cohesion (tính liên kết), độ sâu lồng nhau, số lượng biến, số điểm quyết định, số lượng comment cần thiết để giải thích routine, và các yếu tố liên quan đến độ phức tạp, cần được dùng để xác định độ dài routine, chứ không nên quy định cứng số dòng.

Tuy nhiên, nếu bạn muốn viết routine dài hơn 200 dòng, cần thận trọng. Không có nghiên cứu nào cho thấy việc tăng kích thước routine lớn hơn 200 dòng lại giúp giảm chi phí hay giảm lỗi. Khi vượt ngưỡng 200 dòng, khả năng hiểu code sẽ bị giới hạn.

7.5 Sử dụng tham số (parameter) trong routine như thế nào?

Các interface (giao diện) giữa các routine là một trong những khu vực dễ phát sinh lỗi nhất trong chương trình. Một nghiên cứu của Basili và Perricone (1984) chỉ ra rằng 39% tổng số lỗi là lỗi giao tiếp nội bộ—lỗi trong việc truyền dữ liệu giữa các routine. Sau đây là một số hướng dẫn nhằm giảm thiểu lỗi khi làm việc với interface:

> **Tham khảo thêm:** Về chú thích tham số của routine, xem "Commenting Routines" tại Mục 32.5. Về định dạng tham số, xem Mục 31.7, "Laying Out Routines".

Sắp xếp tham số theo thứ tự input–modify–output

Thay vì sắp xếp tham số một cách ngẫu nhiên hoặc theo thứ tự chữ cái, hãy liệt kê các tham số chỉ-dùng-làm-input trước, tiếp theo là các tham số vừa-input-vừa-output, và cuối cùng là chỉ-output. Quy ước này phản ánh trình tự các thao tác trong routine—nhập liệu, thay đổi, và trả về kết quả. Dưới đây là các ví dụ về danh sách tham số trong ngôn ngữ Ada:

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

Ada sử dụng các từ khóa `in` và `out` để làm rõ vai trò input/output của tham số.

Quy ước này mâu thuẫn với quy ước trong thư viện C, nơi tham số sẽ bị thay đổi thường được đặt lên đầu. Tuy nhiên, theo tôi, thứ tự input–modify–output có ý nghĩa hơn, và nếu bạn kiên định với một quy ước nào đó, điều này vẫn sẽ giúp người đọc hiểu dễ dàng hơn.

Cân nhắc tạo từ khóa in và out cho riêng bạn

Các ngôn ngữ hiện đại khác như C++ không hỗ trợ từ khóa `in` và `out` như Ada, nhưng vẫn có thể sử dụng preprocessor để tạo các từ khóa này cho mục đích chú thích.

```
#define IN
#define OUT

void InvertMatrix(
   IN Matrix originalMatrix,
   OUT Matrix *resultMatrix
);

void ChangeSentenceCase(
   IN StringCase desiredCase,
   IN OUT Sentence *sentenceToEdit
);

void PrintPageNumber(
   IN int pageNumber,
   OUT StatusType &status
);
```

Ở đây, các macro `IN` và `OUT` chỉ mang ý nghĩa tài liệu/chú thích code.

Lưu ý: Bản dịch đã được hiệu chỉnh về ngôn từ và trình bày để tăng tính dễ hiểu, đồng thời sửa một số lỗi định dạng gốc nhằm bảo đảm văn bản nhất quán và mạch lạc.

Chương 7: Các Thủ Tục (Routine) Chất Lượng Cao

Xem xét các rủi ro khi mở rộng cú pháp ngôn ngữ

Trước khi áp dụng kỹ thuật này, bạn cần cân nhắc hai hạn chế quan trọng. Việc tự định nghĩa các từ khóa IN và OUT sẽ mở rộng ngôn ngữ C++ theo cách không quen thuộc với hầu hết những người đọc mã nguồn của bạn. Nếu bạn quyết định mở rộng ngôn ngữ như vậy, hãy đảm bảo thực hiện một cách nhất quán, tốt nhất là trên toàn dự án. Hạn chế thứ hai là các từ khóa IN và OUT này sẽ không được trình biên dịch kiểm soát, nghĩa là về mặt kỹ thuật, bạn có thể đánh dấu một tham số là IN nhưng vẫn sửa đổi giá trị của nó trong thủ tục. Điều này có thể khiến người đọc mã nguồn nghĩ rằng mã là đúng, dù thực tế không phải vậy. Thông thường, việc sử dụng từ khóa `const` của C++ sẽ là phương pháp tốt hơn để xác định tham số chỉ dùng để nhập (input-only).

Đặt thứ tự tham số một cách nhất quán

Nếu nhiều thủ tục sử dụng các tham số tương tự, hãy đặt các tham số đó theo cùng một thứ tự. Thứ tự các tham số trong thủ tục có thể đóng vai trò như một phương tiện ghi nhớ (mnemonic), và sự thiếu nhất quán trong thứ tự này có thể khiến việc ghi nhớ trở nên khó khăn. Ví dụ, trong ngôn ngữ C, thủ tục `fprintf()` giống hệt với `printf()`, chỉ khác là nó thêm đối tượng file vào vị trí tham số đầu tiên. Một thủ tục tương tự, `fputs()`, lại tương tự với `puts()` nhưng thêm file vào vị trí tham số cuối cùng. Sự khác biệt nhỏ nhưng không cần thiết này khiến cho các tham số của các thủ tục này trở nên khó nhớ hơn mức cần thiết.

Ngược lại, thủ tục `strncpy()` trong C nhận các tham số lần lượt là chuỗi đích, chuỗi nguồn và số byte tối đa; thủ tục `memcpy()` cũng nhận các tham số với thứ tự tương tự. Sự nhất quán này giúp việc ghi nhớ tham số trở nên dễ dàng hơn.

Sử dụng hiệu quả tất cả các tham số

Nếu bạn truyền một tham số vào thủ tục, hãy sử dụng nó. Nếu không dùng đến, hãy loại bỏ tham số đó khỏi khai báo thủ tục. Sự tồn tại của các tham số không được sử dụng thường đi đôi với tỷ lệ lỗi tăng cao. Trong một nghiên cứu, 46% các thủ tục không có biến không sử dụng thì cũng hoàn toàn không có lỗi, trong khi chỉ 17-29% các thủ tục có nhiều hơn một biến không tham chiếu là không có lỗi ([Card, Church, và Agresti, 1986]).

> **Dữ liệu thực tế:** Thủ tục có tham số không sử dụng dễ xuất hiện lỗi chương trình hơn.

Quy tắc loại bỏ tham số không sử dụng có một ngoại lệ: Nếu bạn biên dịch điều kiện một phần chương trình, có thể bạn sẽ loại bỏ các phần của thủ tục sử dụng tham số đó. Hãy thận trọng với thực hành này, nhưng nếu bạn chắc chắn nó hoạt động thì vẫn có thể chấp nhận được. Nói chung, nếu bạn có lý do hợp lý để giữ tham số, hãy để nguyên; còn nếu không, hãy nỗ lực dọn sạch mã nguồn.

Đặt biến trạng thái hoặc lỗi ở cuối danh sách tham số

Theo thông lệ, các biến trạng thái (status) hoặc biến báo lỗi nên đứng cuối trong danh sách tham số. Đây là những tham số chỉ dùng để xuất ra kết quả (output-only) và không phải là mục đích chính của thủ tục, vì vậy việc đặt ở cuối là hợp lý.

Không dùng tham số như biến làm việc

Việc sử dụng tham số truyền vào như biến trung gian cho các phép toán là một thực hành nguy hiểm. Hãy sử dụng biến cục bộ thay thế. Ví dụ dưới đây minh họa sai lầm khi sử dụng tham số làm biến trung gian trong Java:

```
int Sample( int inputVal ) {
    inputVal = inputVal * CurrentMultiplier( inputVal );
    inputVal = inputVal + CurrentAdder( inputVal );
    return inputVal;
}
```

> Tại thời điểm trả về, biến `inputVal` không còn chứa giá trị nhập vào ban đầu nữa.

Trong đoạn mã trên, tên biến `inputVal` trở nên gây hiểu nhầm bởi vì sau các thao tác, nó không còn mang giá trị nhập vào gốc nữa mà mang một giá trị đã được tính toán. Nếu sau này bạn cần sử dụng giá trị nhập gốc ở các vị trí khác, rất có thể bạn sẽ giả định nhầm về nội dung của biến này.

Cách giải quyết

Đổi tên biến `inputVal`? Có thể không giải quyết triệt để! Đặt tên là `workingVal` chỉ giải quyết một phần vì nó không nói rõ nguồn gốc bên ngoài của giá trị. Cách đặt tên như `inputValThatBecomesWorkingVal`, `x`, hay `val` cũng không giúp ích nhiều.

Giải pháp tốt hơn là nên tạo biến trung gian làm việc rõ ràng:

```
int Sample( int inputVal ) {
    int workingVal = inputVal;
    workingVal = workingVal * CurrentMultiplier( workingVal );
    workingVal = workingVal + CurrentAdder( workingVal );
    return workingVal;
}
```

> Nếu cần dùng giá trị gốc `inputVal`, vẫn có thể truy cập được bất cứ lúc nào.

Việc thêm biến `workingVal` làm rõ vai trò của `inputVal` và loại bỏ nguy cơ vô tình dùng nhầm giá trị. (*Lưu ý*: Không nên coi đây là lý do để đặt tên biến thực tế là `inputVal` hay `workingVal`. Trong ví dụ chỉ dùng để minh họa vai trò các biến.)

Việc gán giá trị đầu vào cho biến làm việc giúp nhấn mạnh nguồn gốc dữ liệu, đồng thời loại bỏ khả năng sửa đổi một tham số trong danh sách một cách vô ý. Trong C++, bạn còn có thể yêu cầu trình biên dịch kiểm soát việc này bằng cách sử dụng từ khóa `const` cho tham số.

Tài liệu hóa giả định giao diện với tham số

Nếu bạn giả định dữ liệu truyền vào thủ tục có đặc điểm nhất định, hãy ghi chú lại ngay khi bạn xác định giả định đó, cả bên trong thủ tục lẫn tại nơi thủ tục được gọi. Đừng đợi đến khi hoàn thành thủ tục mới quay lại bổ sung chú thích bởi bạn sẽ không nhớ hết giả định. Tốt hơn nữa, hãy sử dụng *assertion* (biểu thức khẳng định) để biến giả định thành mã cho chính xác.

> Tham khảo thêm ở Chương 8 (*Defensive Programming*) và Chương 32 (*Self-Documenting Code*).

Một số giả định giao diện cần tài liệu hóa bao gồm:

- Tham số chỉ dùng để nhập (input-only), tham số bị thay đổi (modified), hoặc tham số chỉ dùng để xuất (output-only).
- Đơn vị đo lường của tham số số học (inch, feet, meter, v.v.).
- Ý nghĩa của mã trạng thái (status codes), giá trị lỗi nếu không dùng kiểu liệt kê (enumerated types).
- Dải giá trị mong đợi.
- Giá trị cụ thể không được phép xuất hiện.

Giới hạn số lượng tham số của thủ tục

Bảy là số lượng "thần kỳ" mà con người có thể hiểu cùng lúc. Nghiên cứu tâm lý cho thấy hầu hết mọi người không thể theo dõi hơn bảy "khối" thông tin cùng lúc ([Miller, 1956]). Do đó, số lượng tham số hợp lý của một thủ tục nên nhỏ hơn hoặc bằng bảy.

Trên thực tế, khả năng giới hạn số lượng tham số còn phụ thuộc vào cách ngôn ngữ xử lý kiểu dữ liệu phức tạp. Nếu sử dụng ngôn ngữ hiện đại hỗ trợ dữ liệu có cấu trúc, bạn có thể truyền một kiểu dữ liệu tổng hợp chứa 13 trường mà vẫn chỉ coi là một "khối" thông tin. Nếu dùng ngôn ngữ nguyên thủy hơn, có thể phải truyền từng trường riêng lẻ.

> Tham khảo thêm về thiết kế giao diện ở mục "Good Abstraction" trong Mục 6.2.

Nếu bạn thấy mình thường xuyên truyền nhiều tham số, nghĩa là coupling (liên kết) giữa các thủ tục quá chặt chẽ. Hãy thiết kế lại để giảm coupling; nếu truyền cùng một dữ liệu cho nhiều thủ tục, hãy nhóm các thủ tục đó vào cùng một lớp và để dữ liệu dùng chung làm dữ liệu lớp (class data).

Xem xét quy ước đặt tên tham số theo vai trò

Nếu bạn cần phân biệt tham số đầu vào, thay đổi hoặc đầu ra, hãy áp dụng quy ước tên, ví dụ: đặt tiền tố *i_*, *m_*, *o_*, hoặc verbose hơn là *Input_*, *Modify_*, *Output_*.

Truyền các biến cần thiết để duy trì tính trừu tượng của giao diện

Có hai quan điểm về cách truyền các thành viên (member) của một object (đối tượng) vào thủ tục. Giả sử bạn có một object cung cấp dữ liệu thông qua 10 hàm truy cập (access routine), và thủ tục được gọi cần dùng ba trường dữ liệu này.

1. **Trường phái thứ nhất**: Chỉ nên truyền ba thành phần cụ thể cần thiết. Lý do là để giữ kết nối (connection) giữa các thủ tục ở mức tối thiểu, giảm coupling và làm cho việc hiểu cũng như tái sử dụng dễ dàng hơn. Họ cho rằng truyền cả object sẽ làm lộ ra toàn bộ 10 hàm truy cập cho thủ tục nhận vào, gây vi phạm nguyên tắc đóng gói (encapsulation).
2. **Trường phái thứ hai**: Nên truyền toàn bộ object để giao diện ổn định hơn, cho phép thủ tục được gọi có quyền truy cập bất kỳ thành viên nào nếu cần về sau mà không phải thay đổi giao diện. Họ lập luận rằng truyền ba thành phần cụ thể mới thực sự vi phạm đóng gói, vì làm lộ ra thành phần cụ thể thủ tục đang sử dụng.

Quan điểm trung dung là: yếu tố quyết định là *tính trừu tượng* mà giao diện thủ tục thể hiện. Nếu ý nghĩa trừu tượng là thủ tục chỉ cần ba thành phần cụ thể (dù tình cờ cả ba cùng nằm trong một object), hãy truyền từng thành phần riêng rẽ. Nếu trừu tượng là luôn có một object nhất định dưới tay (và thủ tục sẽ làm việc với toàn thể object đó), thì truyền cả object để không phá vỡ tính trừu tượng đó.

Các Lưu Ý Trong Thiết Kế Và Sử Dụng Routine

> **Ghi chú:** Một số lỗi đánh máy, xuống dòng bất thường trong bản gốc đã được chỉnh sửa lại cho rõ nghĩa.

Thiết Kế Routine Và Việc Sử Dụng Tham Số

Nói chung, nếu phần code dùng để "thiết lập" trước khi gọi một routine, hoặc "thu dọn" sau khi gọi routine xuất hiện thường xuyên, đó là dấu hiệu cho thấy routine đó không được thiết kế tốt.

Nếu bạn thường xuyên phải thay đổi danh sách tham số (parameter list) của routine, và các tham số đó đều lấy từ cùng một đối tượng, đây là dấu hiệu cho thấy bạn nên truyền cả đối tượng thay vì truyền từng phần tử cụ thể.

Sử Dụng Tham Số Đặt Tên (Named Parameter)

Ở một số ngôn ngữ lập trình, bạn có thể liên kết rõ ràng các tham số hình thức với tham số thực tế, điều này giúp code tự giải thích hơn cũng như tránh lỗi do nhầm lẫn thứ tự tham số.

Ví dụ trong Visual Basic:

```
Private Function Distance3d( _
    ByVal xDistance As Coordinate, _
    ByVal yDistance As Coordinate, _
    ByVal zDistance As Coordinate _
)
End Function
```

```
Private Function Velocity( _
    ByVal latitude As Coordinate, _
    ByVal longitude As Coordinate, _
    ByVal elevation As Coordinate _
)
    Distance = Distance3d(xDistance := latitude, yDistance := longitude, zDistance := elevat
End Function
```

Kỹ thuật này đặc biệt hữu ích khi bạn có danh sách tham số dài với cùng kiểu dữ liệu, điều này dễ dẫn đến nhầm lẫn khi chèn nhầm tham số mà trình biên dịch không phát hiện được. Việc liên kết rõ ràng tham số có thể là dư thừa trong nhiều môi trường, nhưng đối với phần mềm an toàn hay đòi hỏi tính tin cậy cao, sự đảm bảo này rất đáng giá.

Đảm Bảo Tham Số Thực Khớp Với Tham Số Hình Thức

Tham số hình thức (formal parameter), còn gọi là "tham số giả", là các biến được khai báo trong định nghĩa routine. Tham số thực (actual parameter) là biến, hằng số, hoặc biểu thức được sử dụng khi gọi routine.

Một lỗi phổ biến là truyền nhầm kiểu biến vào routine — ví dụ, dùng kiểu integer khi cần floating point. (Vấn đề này chủ yếu xuất hiện ở các ngôn ngữ kiểu yếu như C, đặc biệt khi không sử dụng đầy đủ cảnh báo biên dịch. Ở các ngôn ngữ kiểu mạnh như C++ hay Java, vấn đề này ít gặp hơn.) Đối với tham số chỉ dùng để truyền vào (input only), thông thường trình biên dịch sẽ tự động chuyển đổi kiểu, nhưng nếu là tham số dùng cho cả đầu vào và đầu ra thì lỗi truyền nhầm kiểu có thể gây ra hậu quả nghiêm trọng.

Vì vậy, hãy rèn luyện thói quen kiểm tra kỹ kiểu của các đối số trong danh sách tham số, đồng thời luôn chú ý các cảnh báo của trình biên dịch liên quan đến kiểu tham số không khớp.

7.6 Những Lưu Ý Đặc Biệt Khi Sử Dụng Function

Ngôn ngữ hiện đại như C++, Java, và Visual Basic hỗ trợ cả function và procedure. **Function** là routine trả về giá trị; **procedure** là routine không trả về giá trị. Trong C++, tất cả các routine thường được gọi là "function"; nhưng function có kiểu trả về void về mặt ngữ nghĩa lại là procedure. Việc phân biệt giữa function và procedure không chỉ ở mặt cú pháp mà còn ở mặt ngữ nghĩa, và bạn nên đặt trọng tâm vào ý nghĩa (semantic) khi lựa chọn.

Khi Nào Dùng Function, Khi Nào Dùng Procedure

Theo quan điểm "thuần túy", function chỉ nên trả về một giá trị duy nhất và chỉ nhận tham số đầu vào, đặt tên theo giá trị trả về (như `sin()`, `CustomerID()`,

`ScreenHeight()`). Procedure, ngược lại, có thể nhận/hiệu chỉnh/thay đổi đầu vào, đầu ra, hoặc cả hai.

Một thực tiễn phổ biến là function vừa thực hiện hành động như một procedure, vừa trả về giá trị trạng thái. Ví dụ:

```
if (report.FormatOutput(formattedReport) = Success) then
```

Trong ví dụ này, `report.FormatOutput()` vừa có tham số đầu ra `formattedReport`, vừa trả về kết quả kiểm tra trạng thái. Về nguyên tắc, đây là function nhưng hoạt động giống procedure. Việc sử dụng giá trị trả về để xác định thành công/thất bại là hợp lý nếu bạn giữ sự nhất quán.

Một cách khác là tạo procedure với biến trạng thái như tham số xuất, ví dụ:

```
report.FormatOutput(formattedReport, outputStatus)
if (outputStatus = Success) then
```

Hoặc:

```
outputStatus = report.FormatOutput(formattedReport)
if (outputStatus = Success) then
```

Tóm lại, hãy sử dụng function nếu mục đích chính là trả về giá trị như tên function; còn lại nên dùng procedure.

> **Ý chính:** Nếu routine chủ yếu trả về giá trị được nêu rõ ở tên function, hãy dùng function; nếu không, hãy dùng procedure.

Đặt Giá Trị Trả Về Cho Function

Sử dụng function tiềm ẩn nguy cơ function trả về giá trị sai. Thông thường, điều này xảy ra khi trong function có nhiều đường dẫn (path) thực thi mà một trong số đó không gán giá trị trả về.

- **Kiểm tra mọi đường dẫn trả về:** Khi viết function, hãy hình dung và kiểm thử mọi đường dẫn để đảm bảo luôn gán đủ giá trị trả về.
- **Khởi tạo giá trị mặc định cho biến trả về:** Nên khởi tạo biến trả về ngay từ đầu để tránh trường hợp thiếu gán giá trị.
- **Không trả về reference hoặc pointer tới dữ liệu cục bộ:** Khi routine kết thúc, dữ liệu cục bộ sẽ mất hiệu lực, đồng nghĩa mọi reference hoặc pointer tới chúng đều invalid. Nếu cần trả thông tin về dữ liệu nội bộ, hãy lưu trong thành viên của class (class member), sau đó cung cấp accessor function để truy xuất giá trị đó.

7.7 Macro Routine và Inline Routine

> **Tham khảo thêm:** Ngay cả khi ngôn ngữ lập trình của bạn không hỗ trợ, bạn có thể tự xây dựng các macro routine (routine được tạo bằng preprocessor macro). Tham khảo chi tiết ở mục 30.5 "Building Your Own Programming Tools".

Đặt Dấu Ngoặc Đầy Đủ Cho Macro

Bởi vì macro và đối số của nó sẽ được mở rộng thành code, bạn phải cẩn thận để đảm bảo chúng được mở rộng đúng như mong đợi. Một lỗi phổ biến là viết macro như sau:

```
#define Cube(a) a*a*a
```

Nếu truyền giá trị không nguyên tử cho `a`, phép nhân sẽ không ra kết quả đúng. Ví dụ, `Cube(x+1)` sẽ mở rộng thành `x+1 * x + 1 * x + 1`, không đúng ý định do thứ tự ưu tiên toán tử.

Phiên bản cải tiến như sau (nhưng vẫn chưa hoàn hảo):

```
#define Cube(a) (a)*(a)*(a)
```

Cách tốt nhất là đặt toàn bộ biểu thức trong dấu ngoặc:

```
#define Cube(a) ((a)*(a)*(a))
```

Đặt Dấu Ngoặc Nhọn Cho Macro Nhiều Lệnh

Macro có thể chứa nhiều lệnh, nếu coi như một lệnh duy nhất sẽ gây lỗi, ví dụ:

```
#define LookupEntry(key, index) \
    index = (key - 10) / 5; \
    index = min(index, MAX_INDEX); \
    index = max(index, MIN_INDEX);

for (entryCount = 0; entryCount < numEntries; entryCount++)
    LookupEntry(entryCount, tableIndex[entryCount]);
```

Chỉ có dòng đầu tiên của macro được thực thi trong vòng lặp. Để tránh lỗi, hãy đặt dấu ngoặc nhọn:

```
#define LookupEntry(key, index) { \
    index = (key - 10) / 5; \
    index = min(index, MAX_INDEX); \
    index = max(index, MIN_INDEX); \
}
```

Việc sử dụng macro thay thế cho function call nhìn chung được đánh giá là nguy hiểm, khó hiểu, và là thực hành lập trình không tốt. Chỉ nên dùng kỹ thuật này khi hoàn cảnh thực sự đòi hỏi.

Quy Tắc Đặt Tên Macro

Đối với macro mở rộng thành code (thay thế function), nên đặt tên giống tên function để có thể thay thế lẫn nhau mà không phải sửa đổi code nơi sử dụng. Ở C++, quy ước là đặt tên macro bằng chữ in hoa. Tuy nhiên, nếu macro có thể được thay thế bằng routine, hãy sử dụng quy ước đặt tên cho routine để thuận tiện chuyển đổi về sau.

> **Lưu ý:** Áp dụng các khuyến nghị trên kèm với cân nhắc rủi ro đặc thù của dự án và ngôn ngữ lập trình bạn đang sử dụng.

Hạn chế trong việc sử dụng Macro Routine

Các vấn đề liên quan đến tác dụng phụ

> Cân nhắc đến các vấn đề khác do tác dụng phụ (side effects), đây là một lý do nữa để tránh sử dụng tác dụng phụ.

Những hạn chế khi sử dụng Macro Routine

Các ngôn ngữ hiện đại như C++ cung cấp nhiều lựa chọn thay thế cho việc sử dụng macro, bao gồm:

- `const` để khai báo giá trị hằng số.
- `inline` để định nghĩa hàm sẽ được biên dịch thành mã nội tuyến.
- `template` để định nghĩa các phép toán tiêu chuẩn như `min`, `max`,… theo cách đảm bảo an toàn kiểu dữ liệu (type-safe).
- `enum` để khai báo các kiểu liệt kê (enumerated types).
- `typedef` để định nghĩa thay thế kiểu dữ liệu đơn giản.

Như Bjarne Stroustrup, nhà thiết kế của C++ đã chỉ ra: > “Hầu hết mọi macro đều thể hiện một khiếm khuyết của ngôn ngữ lập trình, của chương trình, hoặc của lập trình viên. Khi bạn sử dụng macro, bạn nên mong đợi chất lượng phục vụ kém hơn từ các công cụ chẳng hạn như debugger (trình gỡ lỗi), cross-reference tools (công cụ tham chiếu chéo), và profiler (công cụ phân tích hiệu năng)” (Stroustrup 1997).

Macro hữu ích trong việc hỗ trợ biên dịch có điều kiện (conditional compilation)—xem thêm ở Mục 8.6, “Debugging Aids”—nhưng các lập trình viên cẩn trọng chỉ sử dụng macro thay cho routine (thủ tục/hàm) như là giải pháp cuối cùng.

Inline Routine

Ngôn ngữ C++ hỗ trợ từ khóa `inline`. Một inline routine cho phép lập trình viên xử lý mã như một routine khi viết mã, tuy nhiên trình biên dịch thường

sẽ chuyển hóa mỗi lần gọi routine thành mã nội tuyến khi biên dịch. Theo lý thuyết, `inline` có thể giúp tạo ra code hiệu quả bằng cách loại bỏ chi phí gọi routine.

Cẩn trọng khi sử dụng inline routine

- Sử dụng inline routine một cách tiết kiệm.
- Inline routine vi phạm nguyên tắc đóng gói (encapsulation), bởi C++ yêu cầu đặt phần mã hiện thực của inline routine vào tập tin header, lộ ra cho mọi lập trình viên sử dụng tập tin đó.
- Inline routine đòi hỏi toàn bộ mã routine được sinh ra mỗi lần routine được gọi, với inline routine có kích cỡ lớn sẽ làm tăng kích thước mã nguồn, kéo theo các vấn đề riêng.

Kết luận về việc tối ưu hiệu năng nhờ inlining giống như mọi kỹ thuật tối ưu khác: cần thực hiện profile và đo lường sự cải thiện. Nếu lợi ích hiệu năng dự kiến không đủ lớn để biện minh cho việc profile kiểm chứng, nó cũng không xứng với việc đánh đổi chất lượng mã.

DANH SÁCH KIỂM TRA: Routine Chất lượng Cao

Các vấn đề tổng thể

- Lý do tạo routine có đủ thỏa đáng?
- Tất cả thành phần của routine nào nên tách thành routine riêng đã được xử lý thích hợp chưa?
- Tên routine có phải dạng động từ-kèm-tân ngữ rõ ràng cho procedure (thủ tục) hoặc mô tả giá trị trả về cho function (hàm) không?
- Tên routine có mô tả đầy đủ những gì routine làm không?
- Đã thống nhất quy ước đặt tên cho các thao tác phổ biến chưa?
- Routine đảm bảo tính cohesion chức năng (functional cohesion) mạnh, chỉ làm một việc tốt nhất chưa?
- Các routine có loose coupling (liên kết lỏng lẻo)—liên kết giữa các routine nhỏ gọn, trực quan, và linh hoạt không?
- Độ dài routine được xác định tự nhiên theo chức năng, không bị gò ép bởi tiêu chuẩn mã hóa cứng nhắc?

Các vấn đề về việc truyền tham số

- Danh sách tham số mang đến abstraction interface (giao diện trừu tượng) nhất quán chưa?
- Tham số có thứ tự hợp lý, tương tự đối với những routine cùng loại?
- Đã ghi chú các giả định về giao diện chưa?
- Routine có tối đa bảy tham số trở xuống không?
- Mỗi tham số đầu vào đều được sử dụng?

- Mỗi tham số đầu ra đều được sử dụng?
- Routine tránh dùng tham số đầu vào làm biến làm việc chưa?
- Nếu là function, hàm có trả về giá trị hợp lệ trong mọi trường hợp không?

Các điểm chính

- **Lý do quan trọng nhất để tạo ra routine là tăng khả năng kiểm soát trí tuệ (intellectual manageability) của chương trình**; có nhiều lý do tốt khác, trong đó tiết kiệm bộ nhớ chỉ là phụ, còn tăng khả năng đọc, tin cậy và khả năng chỉnh sửa là các lý do quan trọng hơn.
- **Đôi khi thao tác đơn giản nhất lại cần tách ra thành routine riêng.**
- **Routine có thể được phân loại theo nhiều mức độ cohesion (độ gắn kết), nhưng hầu hết nên đạt cohesion chức năng (functional cohesion), đó là tốt nhất.**
- **Tên routine phản ánh chất lượng của nó; tên tệ nhưng đúng thì routine có thể thiết kế kém, tên tệ mà sai nghĩa thì nó không cho biết routine làm gì—dù thế nào, tên routine tệ nghĩa là phải sửa mã.**
- **Chỉ nên dùng function khi mục tiêu chính là trả về giá trị cụ thể theo tên hàm.**
- **Lập trình viên cẩn trọng chỉ dùng macro routine khi cần thiết và cuối cùng.**

Chương 8: Lập Trình Phòng Thủ (Defensive Programming)

Nội dung

Chủ đề liên quan

- **Information hiding (che giấu thông tin)**: Xem "Hide Secrets (Information Hiding)" tại Mục 5.3
- **Thiết kế cho sự thay đổi**: "Identify Areas Likely to Change" tại Mục 5.3
- **Kiến trúc phần mềm**: Mục 3.5
- **Thiết kế trong xây dựng**: Chương 5
- **Gỡ lỗi**: Chương 23

Giới thiệu về Lập trình Phòng thủ

Lập trình phòng thủ không có nghĩa là phòng thủ chống lại chỉ trích về mã—kiểu "Tôi làm đúng rồi mà!". Ý tưởng này dựa trên nguyên tắc của defensive driving (lái xe phòng thủ). Trong lái xe phòng thủ, bạn luôn chuẩn bị tinh thần rằng không thể đoán trước người khác sẽ làm gì. Nhờ vậy, nếu người khác làm sai, bạn vẫn hạn chế được rủi ro. Bạn chịu trách nhiệm bảo vệ chính mình, dù lỗi thuộc về người khác.

> **Điểm then chốt:** Trong lập trình phòng thủ, nguyên tắc cốt lõi là nếu routine nhận dữ liệu sai, nó sẽ không gặp sự cố, dù lỗi do routine khác gây ra. Nói rộng hơn, đây là sự thừa nhận rằng chương trình luôn có vấn đề và cần sửa đổi, vì vậy lập trình viên thông minh sẽ mã hóa chương trình tương ứng.

Chương này sẽ hướng dẫn cách bảo vệ bạn trước dữ liệu không hợp lệ, những sự kiện "không bao giờ xảy ra", và lỗi của những lập trình viên khác. Nếu bạn đã có kinh nghiệm, có thể bỏ qua phần tiếp theo về xử lý dữ liệu vào và bắt đầu từ mục 8.2 về assertion.

8.1 Bảo vệ Chương trình khỏi Dữ liệu Vào Không hợp lệ

Khi học ở trường, bạn có thể đã nghe câu "Garbage in, garbage out" (Đầu vào rác, đầu ra rác). Đây cũng chính là dạng caveat emptor (cảnh báo người dùng) trong phát triển phần mềm. Tuy nhiên, đối với phần mềm sản xuất, "garbage in, garbage out" không còn phù hợp. Một chương trình tốt không bao giờ xuất ra dữ liệu rác, bất kể đầu vào nhận được. Một chương trình tốt sẽ dùng các chiến lược như "garbage in, nothing out", "garbage in, error message out", hoặc "no garbage allowed in" (không cho phép đầu vào rác).

> **Điểm then chốt:** Với tiêu chuẩn ngày nay, "garbage in, garbage out" cho thấy một chương trình lỏng lẻo, thiếu an toàn.

Có ba cách tổng quát để xử lý dữ liệu rác:

1. **Kiểm tra giá trị toàn bộ dữ liệu từ nguồn bên ngoài**
   Khi nhận dữ liệu từ file, người dùng, mạng, hoặc nguồn ngoại lai khác, cần kiểm tra đảm bảo dữ liệu nằm trong phạm vi chấp nhận được:

   - Đảm bảo giá trị kiểu số nằm trong phạm vi cho phép.
   - Chuỗi không quá dài.
   - Nếu chuỗi đại diện cho giá trị hạn chế (ví dụ: mã giao dịch tài chính…), phải đảm bảo chuỗi hợp lệ; nếu không, loại bỏ giá trị đó.
   - Nếu làm việc với ứng dụng an toàn, cần đặc biệt đề phòng các kiểu tấn công như:
     - Buffer overflow (tràn bộ đệm)
     - Injected SQL commands (lệnh SQL chèn độc hại)
     - Injected HTML/XML code (mã HTML hoặc XML chèn độc hại)
     - Integer overflow (tràn số nguyên)
     - Dữ liệu truyền tới system calls (lệnh hệ thống)

2. **Kiểm tra giá trị toàn bộ tham số đầu vào của routine** Việc kiểm tra này tương tự kiểm tra dữ liệu từ nguồn ngoài, chỉ khác là dữ liệu đến từ routine khác trong hệ thống chứ không phải giao diện bên ngoài. Mục 8.5, "Barricade Your Program to Contain the Damage Caused by Errors," bàn chi tiết cách xác định những routine nào cần kiểm tra đầu vào.

3. **Quyết định cách xử lý đầu vào không hợp lệ** Sau khi phát hiện tham số không hợp lệ, bạn sẽ xử lý ra sao? Tùy trường hợp có thể lựa chọn một trong nhiều biện pháp, các phương pháp này được trình bày chi tiết tại Mục 8.3, "Error-Handling Techniques."

Lập trình phòng thủ là phương án hỗ trợ thêm cho các kỹ thuật nâng cao chất lượng được đề cập trong sách. Hình thức tốt nhất của coding phòng thủ là **không làm phát sinh lỗi ngay từ đầu**: thiết kế lặp, viết giả mã trước khi viết mã, viết test case trước khi viết code, kiểm tra thiết kế ở mức thấp… đều là những hoạt động giúp phòng tránh lỗi; chúng cần được ưu tiên cao hơn so với lập trình phòng thủ. Tuy vậy, bạn hoàn toàn có thể phối hợp các kỹ thuật này với phòng thủ để đạt kết quả tốt nhất.

Như Hình 8-1 minh họa, việc bảo vệ mình trước các vấn đề tưởng chừng nhỏ nhặt có thể tạo ra khác biệt lớn hơn bạn nghĩ. Phần còn lại của chương sẽ trình bày cụ thể các tùy chọn kiểm tra dữ liệu từ nguồn ngoài, kiểm tra tham số đầu vào và xử lý dữ liệu không hợp lệ.

Hình 8-1: Một phần của cầu nổi Interstate-90 tại Seattle đã bị chìm trong bão do các bể nổi không được che chắn, dẫn đến việc nước tràn vào và khiến cầu chìm. Trong quá trình xây dựng, việc bảo vệ mình trước những chi tiết nhỏ lại là điều quan trọng hơn bạn nghĩ.

Khi một assertion (khẳng định) là đúng

Khi một assertion là đúng, điều này có nghĩa mọi thành phần của hệ thống đang hoạt động như mong đợi. Khi assertion sai, điều đó cho thấy đã phát hiện một lỗi bất ngờ trong mã. Ví dụ, nếu hệ thống giả định rằng tệp thông tin khách hàng sẽ không bao giờ có nhiều hơn 50.000 bản ghi, chương trình có thể chứa một assertion kiểm tra rằng số lượng bản ghi phải nhỏ hơn hoặc bằng 50.000. Khi số bản ghi nhỏ hơn hoặc bằng 50.000, assertion sẽ không đưa ra cảnh báo. Tuy nhiên, nếu gặp nhiều hơn 50.000 bản ghi, assertion sẽ thông báo mạnh mẽ rằng có lỗi trong chương trình.

Assertions đặc biệt hữu ích trong các chương trình lớn, phức tạp và các chương trình yêu cầu độ tin cậy cao. Chúng giúp lập trình viên nhanh chóng phát hiện các giả định không nhất quán giữa các interface (giao diện), những lỗi phát sinh khi mã bị thay đổi, v.v.

Thông thường, một assertion nhận hai đối số: một biểu thức boolean (đúng/sai) mô tả giả định cần phải đúng, và một thông báo sẽ được hiển thị nếu giả định đó không đúng. Ví dụ về assertion trong Java, khi giả sử biến `denominator` (mẫu số) phải khác không:

```
assert denominator != 0 : "denominator is unexpectedly equal to 0 ";
```

Assertion trên xác nhận rằng `denominator` không bằng 0. Đối số thứ nhất `denominator != 0` là biểu thức boolean. Đối số thứ hai là thông báo sẽ in ra nếu assertion sai, tức là nếu điều kiện kiểm tra không đúng.

> **Lưu ý quan trọng:** Hãy sử dụng assertion để ghi nhận các giả định trong mã và phát hiện các tình huống bất ngờ.

Các ví dụ về giả định có thể kiểm tra bằng assertion

- Giá trị của một input parameter (tham số đầu vào) nằm trong phạm vi dự kiến (hoặc của output parameter).
- Một file hoặc stream mở (hoặc đóng) khi một routine (thủ tục) bắt đầu (hoặc kết thúc) thực thi.
- File hoặc stream ở đầu (hoặc cuối) khi routine bắt đầu (hoặc kết thúc).
- File hoặc stream mở theo chế độ chỉ đọc, chỉ ghi, hoặc vừa đọc vừa ghi.
- Giá trị của một biến chỉ đọc không bị thay đổi trong routine.
- Một pointer (con trỏ) khác null.
- Mảng hoặc container truyền vào một routine có thể chứa tối thiểu X phần tử dữ liệu.
- Bảng đã được khởi tạo với các giá trị thực tế.
- Container rỗng (hoặc đầy) khi routine bắt đầu (hoặc kết thúc).
- Kết quả từ một routine phức tạp, tối ưu hóa mạnh khớp với kết quả từ một routine chậm hơn nhưng được viết rõ ràng.

Đây chỉ là các ví dụ cơ bản; các routine của bạn sẽ còn có nhiều giả định cụ thể khác có thể ghi nhận bằng assertion.

Thời điểm sử dụng assertion

Thông thường, bạn không muốn người dùng nhìn thấy các thông báo assertion trong mã vận hành thực tế (production code); assertion chủ yếu phục vụ trong quá trình phát triển và bảo trì phần mềm. Các assertion thường được biên dịch vào mã khi phát triển và loại bỏ khỏi mã khi đóng gói cho sản xuất, nhằm tránh ảnh hưởng đến hiệu năng hệ thống.

Xây dựng cơ chế assertion riêng

Nhiều ngôn ngữ lập trình như C++, Java, Microsoft Visual Basic đều hỗ trợ assertion. Nếu ngôn ngữ bạn sử dụng không hỗ trợ trực tiếp, việc xây dựng routine assertion là khá đơn giản. Ví dụ, macro `assert` chuẩn trong C++ không hỗ trợ thông báo chi tiết. Sau đây là phiên bản mở rộng của macro ASSERT bằng C++:

```
#define ASSERT( condition, message ) { \
    if ( !(condition) ) { \
        LogError( "Assertion failed: ", \
        #condition, message ); \
        exit( EXIT_FAILURE ); \
    } \
}
```

Hướng dẫn sử dụng assertion

1. Phân biệt giữa assertion và xử lý lỗi

- **Sử dụng mã xử lý lỗi** cho các tình huống bạn dự đoán sẽ xảy ra.
- **Sử dụng assertion** cho các tình huống không bao giờ được xảy ra.

Mã xử lý lỗi kiểm tra các trường hợp dữ liệu không hợp lệ mà lập trình viên đã lường trước và cần được xử lý trong sản phẩm. Assertion thường dùng để kiểm tra các lỗi logic trong mã.

Nếu sử dụng mã xử lý lỗi cho điều kiện bất thường, chương trình có thể phục hồi gracefully (mượt mà). Nếu assertion được kích hoạt, hành động khắc phục không chỉ là xử lý lỗi mà là sửa mã nguồn, biên dịch lại và phát hành phần mềm mới.

Assertion là một dạng tài liệu khả thực (executable documentation) giúp ghi nhận các giả định, chủ động hơn so với lời chú thích (comment) trong code.

2. Không đặt mã khả thực vào assertion

Nếu bạn đặt lệnh thực thi vào assertion, khi loại bỏ assertion lúc biên dịch cho môi trường sản xuất, bạn cũng loại bỏ cả lệnh thực thi đó. Thay vào đó, hãy viết mã khả thực trên dòng riêng, gán kết quả cho biến trạng thái và kiểm tra biến đó bằng assertion, ví dụ:

```
actionPerformed = PerformAction()
Debug Assert( actionPerformed )  ' Không thể thực hiện hành động
```

3. Sử dụng assertion để ghi nhận và kiểm tra precondition và postcondition

Precondition (tiền điều kiện) và postcondition (hậu điều kiện) là thành phần quan trọng trong thiết kế phần mềm, còn gọi là "design by contract" (thiết kế dựa trên hợp đồng).

- **Precondition** là các điều kiện mà phía client (mã gọi procedure hoặc class) cam kết đúng trước khi gọi.
- **Postcondition** là các điều kiện mà routine hoặc class đảm bảo sẽ đúng khi kết thúc thực thi.

Assertion là công cụ hữu ích để kiểm tra động các precondition và postcondition, thay vì chỉ ghi chú bằng comment.

Ví dụ sử dụng assertion để kiểm tra precondition và postcondition trong Visual Basic:

```
Private Function Velocity ( _
    ByVal latitude As Single, _
    ByVal longitude As Single, _
    ByVal elevation As Single _
) As Single
    ' Preconditions
    Debug Assert ( -90 <= latitude And latitude <= 90 )
    Debug Assert ( 0 <= longitude And longitude < 360 )
    Debug Assert ( -500 <= elevation And elevation <= 75000 )

    ' Postcondition
    Debug Assert ( 0 <= returnVelocity And returnVelocity <= 600 )
    ' return value
    Velocity = returnVelocity
End Function
```

Nếu các biến `latitude`, `longitude`, `elevation` đến từ nguồn bên ngoài, hãy kiểm tra giá trị bất hợp lệ bằng mã xử lý lỗi (error-handling code) thay vì assertion. Nếu các giá trị này đến từ nguồn tin cậy nội bộ và routine giả định chúng luôn hợp lệ, assertion là phù hợp.

4. Kết hợp cả assertion và mã xử lý lỗi khi cần thiết

Trong các hệ thống lớn, phức tạp do nhiều nhóm thiết kế, phát triển trong thời gian dài và môi trường khác nhau, đôi khi cần sử dụng cả assertion lẫn xử lý lỗi cho cùng một loại lỗi để đảm bảo độ chắc chắn và an toàn cho phần mềm.

Tự kiểm tra:
Bản dịch đảm bảo giữ nguyên các đoạn code, sử dụng đúng các thuật ngữ chuyên ngành, phân đoạn rõ ràng và ngắn gọn, logic liền mạch với văn phong học thuật, trang trọng. Các chú thích bổ sung đã được lồng ghép tại vị trí thích hợp để làm rõ nội dung gốc.

Ứng dụng lớn, phức tạp và lâu đời: Vai trò của assertion (khẳng định)

Đối với các ứng dụng cực kỳ lớn, phức tạp và có tuổi thọ kéo dài như Word, assertion (khẳng định) rất có giá trị vì chúng giúp loại bỏ càng nhiều lỗi xuất hiện trong quá trình phát triển càng tốt. Tuy nhiên, với độ phức tạp lớn (hàng triệu dòng mã lệnh) và đã qua nhiều thế hệ sửa đổi, sẽ không thực tế nếu giả định rằng mọi lỗi có thể đều được phát hiện và khắc phục trước khi phần mềm được phát hành. Vì thế, các lỗi phát sinh cũng cần được xử lý trong phiên bản sản xuất (production) của hệ thống.

Ví dụ sử dụng assertion trong Velocity

Dưới đây là ví dụ minh họa cho cách sử dụng assertion và xử lý lỗi đầu vào trong ngôn ngữ Visual Basic, áp dụng cho hàm Velocity:

```
' Visual Basic Example of Using Assertions to Document Preconditions and Postconditions
Private Function Velocity ( _
    ByRef latitude As Single, _
    ByRef longitude As Single, _
    ByRef elevation As Single _
) As Single
    ' Preconditions
    Debug Assert ( -90 <= latitude And latitude <= 90 )
    Debug Assert ( 0 <= longitude And longitude < 360 )
    Debug Assert ( -500 <= elevation And elevation <= 75000 )

    ' Sanitize input data. Values should be within the ranges asserted above,
    ' but if a value is not within its valid range, it will be changed to the
    ' closest legal value.
    If ( latitude < -90 ) Then
        latitude = -90
    ElseIf ( latitude > 90 ) Then
        latitude = 90
    End If
    If ( longitude < 0 ) Then
        longitude = 0
    ElseIf ( longitude > 360 ) Then
```

```
    ' ...
End Function
```

Trong ví dụ này, assertion được sử dụng để ghi chú các điều kiện tiên quyết (precondition) và hậu điều kiện (postcondition). Ngoài ra, mã nguồn cũng có đoạn xử lý "sanitize input data" nhằm hiệu chỉnh giá trị đầu vào về giá trị hợp lệ gần nhất nếu phát hiện dữ liệu sai phạm.

8.3 Các kỹ thuật xử lý lỗi

Assertion được dùng để xử lý những lỗi lẽ ra không nên xảy ra trong mã nguồn. Vậy với các lỗi dự kiến có thể xảy ra, chúng ta nên xử lý thế nào? Tùy vào từng tình huống, bạn có thể áp dụng các biện pháp sau:

- Trả về giá trị trung tính (neutral value)
- Thay thế bằng phần dữ liệu hợp lệ tiếp theo
- Trả về kết quả giống như lần trước
- Thay thế bằng giá trị hợp lệ gần nhất
- Ghi lại thông báo cảnh báo vào file log
- Trả về mã lỗi (error code)
- Gọi một routine/object xử lý lỗi
- Hiển thị thông báo lỗi cho người dùng
- Tắt chương trình
- Hoặc kết hợp các biện pháp trên

Các phương án xử lý lỗi chi tiết

1. Trả về giá trị trung tính (Return a neutral value) Đôi khi cách ứng phó tốt nhất với dữ liệu sai là tiếp tục hoạt động và trả về giá trị được biết là vô hại. Ví dụ: - Tính toán số học trả về 0 - Hàm thao tác chuỗi trả về chuỗi rỗng - Hàm thao tác con trỏ trả về con trỏ rỗng

Một routine vẽ trong trò chơi video nhận màu sai có thể sử dụng màu nền hoặc màu mặc định. Tuy nhiên, đối với routine hiển thị dữ liệu chụp X-quang cho bệnh nhân ung thư, việc trả về giá trị "trung tính" là không phù hợp; trường hợp này nên kết thúc chương trình thay vì hiển thị dữ liệu bệnh nhân sai.

---

2. Thay thế bằng phần dữ liệu hợp lệ tiếp theo (Substitute the next piece of valid data) Khi xử lý luồng dữ liệu, đôi khi chỉ cần trả về phần dữ liệu hợp lệ tiếp theo. Ví dụ: - Khi đọc bản ghi từ cơ sở dữ liệu và gặp bản ghi bị lỗi, chỉ cần tiếp tục cho đến khi gặp bản ghi hợp lệ. - Lấy dữ liệu từ nhiệt kế 100 lần mỗi giây, nếu có lần không lấy được số đo hợp lệ thì chỉ cần chờ 1/100 giây và lấy số liệu tiếp theo.

---

3. Trả về kết quả giống lần trước (Return the same answer as the previous time) Nếu phần mềm đo nhiệt độ không lấy được số liệu, có thể trả về giá trị lần đo trước, vì xác suất nhiệt độ thay đổi nhiều trong 1/100 giây là thấp. Trong trò chơi, khi nhận yêu cầu hiển thị màu không hợp lệ, cũng có thể dùng lại màu lần trước. Tuy nhiên, với các ứng dụng như máy rút tiền (ATM), việc dùng lại kết quả trước đó (chẳng hạn trả về số tài khoản của khách lần trước) là không thể chấp nhận.

---

4. Thay thế bằng giá trị hợp lệ gần nhất (Substitute the closest legal value) Một phương án hợp lý là trả về giá trị hợp lệ gần nhất, như ví dụ về Velocity ở trên. Khi cảm ứng nhiệt độ chỉ đo được từ 0 đến 100 độ C, nếu số đo dưới 0 sẽ trả về 0; nếu lớn hơn 100 sẽ trả về 100. Đối với thao tác chuỗi, nếu độ dài chuỗi báo là nhỏ hơn 0, có thể dùng 0 thay thế. Một số thiết bị (ví dụ đồng hồ tốc độ xe hơi) cũng áp dụng cách này, ví dụ khi xe lùi, đồng hồ tốc độ hiển thị 0 thay vì giá trị âm.

---

5. Ghi lại thông báo cảnh báo vào file log (Log a warning message to a file) Khi phát hiện dữ liệu sai, có thể ghi lại một thông báo cảnh báo vào file log và tiếp tục chạy chương trình. Biện pháp này có thể kết hợp với kỹ thuật thay thế giá trị hợp lệ gần nhất hoặc thay thế dữ liệu tiếp theo. Khi sử dụng log, cần cân nhắc tính bảo mật, xác định xem log có thể công khai hay phải mã hóa, bảo vệ.

---

6. Trả về mã lỗi (Return an error code) Bạn có thể thiết kế sao cho chỉ một số bộ phận hệ thống chịu trách nhiệm xử lý lỗi. Phần còn lại chỉ phát hiện lỗi và báo cáo lên các routine cấp cao hơn trong call hierarchy để xử lý. Cơ chế báo lỗi có thể là: - Thiết lập giá trị biến trạng thái (status variable) - Trả mã trạng thái như giá trị trả về của hàm - Ném ra exception (ngoại lệ) nhờ cơ chế exception tích hợp của ngôn ngữ lập trình

Điều quan trọng là xác định rõ những phần nào trực tiếp xử lý lỗi, phần nào chỉ báo lỗi. Nếu liên quan đến bảo mật, các routine gọi luôn phải kiểm tra mã trả về.

---

7. Gọi routine/object xử lý lỗi (Call an error-processing routine/object) Một cách khác là tập trung xử lý lỗi vào một routine hoặc object xử lý lỗi toàn cục (global error-handling routine/object). Ưu điểm là giúp tập trung trách nhiệm, dễ gỡ lỗi. Tuy nhiên, toàn bộ chương trình sẽ phải biết đến routine này và bị ràng buộc với nó. Nếu muốn tái sử dụng một số đoạn mã ở hệ thống khác, sẽ phải mang theo module xử lý lỗi này.

Lưu ý quan trọng về bảo mật: Nếu chương trình bị tràn bộ đệm (buffer overrun), kẻ tấn công có thể kiểm soát địa chỉ routine/object xử lý lỗi. Do đó, sau khi xảy ra buffer overrun, không còn an toàn khi tiếp tục sử dụng cơ chế này.

---

8. Hiển thị thông báo lỗi tại vị trí gặp lỗi (Display an error message wherever the error is encountered) Biện pháp này giúp giảm chi phí xử lý lỗi. Tuy nhiên, nhược điểm là có thể khiến các thông báo liên quan đến giao diện người dùng (UI - User Interface) rải rác khắp ứng dụng. Điều này gây khó khăn trong việc tạo ra UI nhất quán, tách biệt rõ ràng UI với hệ thống xử lý nghiệp vụ, hoặc khi muốn bản địa hóa phần mềm sang ngôn ngữ khác. Ngoài ra, cần tránh tiết lộ quá nhiều thông tin cho kẻ tấn công qua thông báo lỗi, vì chúng có thể tận dụng để tìm lỗ hổng.

---

9. Xử lý lỗi phù hợp tại từng vị trí (Handle the error in whatever way works best locally) Một số thiết kế cho phép từng đoạn mã tự quyết định cách xử lý từng lỗi cụ thể. Phương pháp này cung cấp sự linh hoạt cho từng lập trình viên, nhưng cũng tiềm ẩn rủi ro lớn là hiệu năng toàn hệ thống khó đáp ứng các tiêu chí về tính đúng đắn (correctness) hoặc độ bền vững (robustness). Các vấn đề liên quan đến UI cũng có thể phát sinh nếu hiện diện khắp hệ thống.

---

10. Tắt chương trình (Shut down) Một số hệ thống sẽ dừng hoạt động ngay khi phát hiện lỗi, đặc biệt trong các ứng dụng an toàn, trọng yếu (safety-critical applications). Ví dụ, nếu phần mềm điều khiển thiết bị xạ trị phát hiện dữ liệu liều xạ sai, hành động đúng là phải dừng hoạt động. Dùng lại giá trị cũ, giá trị gần đúng, hay giá trị trung tính đều không phù hợp, vì rủi ro ảnh hưởng nghiêm trọng. Tốt nhất nên khởi động lại thiết bị.

Tương tự, Windows có thể được cấu hình để dừng máy chủ nếu security log (nhật ký bảo mật) bị đầy, điều này phù hợp với môi trường yêu cầu bảo mật cao.

Độ bền vững (Robustness) so với Tính đúng đắn (Correctness)

Như hai ví dụ về trò chơi và máy x-quang đã chỉ ra, phong cách xử lý lỗi phù hợp phụ thuộc vào loại phần mềm. Xử lý lỗi thường thiên về ưu tiên tính đúng đắn hoặc độ bền vững:

- **Tính đúng đắn (correctness):** Không bao giờ trả về kết quả sai. Không trả về gì còn tốt hơn trả về kết quả không chính xác.

- **Độ bền vững (robustness):** Luôn cố gắng duy trì hoạt động của phần mềm, ngay cả khi có thể dẫn đến kết quả đôi lúc không chính xác.

Ứng dụng an toàn, trọng yếu (safety-critical) thường ưu tiên tính đúng đắn. Ngược lại, các phần mềm tiêu dùng (consumer applications) lại ưu tiên độ bền vững — bất kỳ kết quả nào cũng tốt hơn là phần mềm bị dừng đột ngột. Ví dụ, trình xử lý văn bản đôi khi hiển thị một phần dòng văn bản ở cuối màn hình, nhưng vẫn tiếp tục hoạt động, đảm bảo người dùng không bị gián đoạn.

Tác động của Xử lý Lỗi tới Thiết kế ở Cấp Độ Cao

Tôi biết rằng lần tới khi tôi nhấn Page Up hoặc Page Down, màn hình sẽ làm mới và hiển thị sẽ trở lại trạng thái bình thường.

Tác động của Thiết kế Cấp Độ Cao đối với Xử lý Lỗi

Với rất nhiều lựa chọn về cách xử lý, bạn cần thận trọng trong việc xử lý các tham số không hợp lệ một cách nhất quán trong toàn bộ chương trình. Phương thức xử lý lỗi sẽ ảnh hưởng đến khả năng phần mềm đáp ứng các yêu cầu liên quan đến tính đúng đắn, độ vững chắc (robustness) và các thuộc tính phi chức năng (nonfunctional attributes) khác.

> **Lưu ý quan trọng:** Việc quyết định phương pháp tổng thể để xử lý tham số xấu là một quyết định kiến trúc hoặc thiết kế cấp cao, cần được giải quyết ở các cấp độ đó.

Khi đã quyết định phương pháp xử lý, bạn phải đảm bảo thực hiện nhất quán. Nếu bạn quyết định để mã ở cấp cao hơn xử lý lỗi, và mã ở cấp thấp hơn chỉ báo cáo lỗi, hãy chắc chắn rằng code cấp cao thực sự có xử lý các lỗi đó! Một số ngôn ngữ cho phép bạn bỏ qua việc một hàm trả về mã lỗi—trong C++ chẳng hạn, bạn không bắt buộc phải xử lý giá trị trả về của một hàm—tuy nhiên, đừng bỏ qua thông tin lỗi! Hãy kiểm tra giá trị trả về của hàm. Dù bạn không mong đợi hàm đó bao giờ trả về lỗi, hãy kiểm tra nó. Toàn bộ mục tiêu của lập trình phòng thủ (defensive programming) là bảo vệ khỏi những lỗi bạn không lường trước.

Hướng dẫn này áp dụng cả với hàm hệ thống cũng như hàm bạn tự viết. Trừ khi bạn đã đặt ra định hướng kiến trúc không kiểm tra lỗi từ lệnh gọi hệ thống, hãy kiểm tra mã lỗi sau mỗi lần gọi. Nếu phát hiện lỗi, hãy đưa vào số hiệu lỗi và mô tả lỗi tương ứng.

8.4 Ngoại lệ (Exceptions)

Ngoại lệ (exception) là một phương tiện cụ thể để mã có thể truyền đạt lỗi hoặc sự kiện ngoại lệ đến mã gọi nó. Nếu mã trong một routine gặp một điều kiện bất ngờ mà không biết cách xử lý, nó sẽ ném (throw) một ngoại lệ, về bản chất là "giơ tay đầu hàng" và nói: "Tôi không biết xử lý thế nào với điều này—hy vọng ai đó ở tầng trên sẽ biết cách!" Khi mã không thể nhận diện ngữ cảnh lỗi, nó có thể trả quyền kiểm soát về các phần khác của hệ thống nơi có khả năng xử lý hoặc phản ứng phù hợp hơn.

Ngoại lệ cũng có thể dùng để làm mạch lạc các logic rối rắm bên trong một đoạn mã đơn lẻ, như ví dụ "Viết lại với try-finally" ở Mục 17.3. Cấu trúc cơ bản của ngoại lệ là một routine sử dụng `throw` để ném một đối tượng ngoại lệ, và một routine khác trong chuỗi gọi sẽ bắt (catch) nó trong khối try-catch.

Các ngôn ngữ phổ biến khác nhau về mặt triển khai ngoại lệ. Bảng sau tóm tắt các khác biệt chủ yếu giữa ba ngôn ngữ:

Thuộc tính ngoại lệ (Exception Attribute)	C++	Java	Visual Basic
Hỗ trợ try-catch	Có	Có	Có
Hỗ trợ try-catch-finally	Không	Có	Có
Có thể ném gì (what can be thrown)	Đối tượng ngoại lệ (exception object) hoặc đối tượng dẫn xuất từ lớp Exception; con trỏ đối tượng; tham chiếu đối tượng; kiểu dữ liệu như string hoặc int	Đối tượng ngoại lệ hoặc đối tượng dẫn xuất từ lớp Exception	Đối tượng ngoại lệ hoặc đối tượng dẫn xuất từ lớp Exception
Tác động khi ngoại lệ không được bắt	Gọi `std::unexpected()`, mặc định gọi `std::terminate()`, tiếp tục gọi `abort()`	Dừng luồng thực thi nếu ngoại lệ là "checked exception"; không ảnh hưởng nếu là "runtime exception"	Dừng chương trình

Thuộc tính ngoại lệ (Exception Attribute)	C++	Java	Visual Basic
Ngoại lệ ném ra phải được định nghĩa trong interface	Không	Có	Không
Ngoại lệ bắt phải được định nghĩa trong interface	Không	Có	Không

> Các chương trình sử dụng ngoại lệ như một phần của xử lý bình thường sẽ gặp phải tất cả các vấn đề về khả năng đọc và bảo trì tương tự như code spaghetti truyền thống.
>
> — *Andy Hunt and Dave Thomas*

Những lưu ý khi sử dụng ngoại lệ

- **Dùng ngoại lệ để thông báo cho phần khác của chương trình về lỗi không nên bị bỏ qua:** Lợi ích lớn nhất của ngoại lệ là khả năng thông báo điều kiện lỗi mà không thể bị làm ngơ (Meyers 1996). Các phương pháp xử lý lỗi khác tạo ra khả năng một lỗi bị truyền đi mà không ai phát hiện. Ngoại lệ loại trừ nguy cơ này.
- **Chỉ ném ngoại lệ cho các điều kiện thực sự ngoại lệ:** Ngoại lệ nên dùng cho các tình huống thực sự hiếm hoi—những điều kiện không thể xử lý bằng các phương pháp lập trình thường thấy khác. Mục đích của ngoại lệ tương tự như assertion (khẳng định) — cho những sự kiện không chỉ là hy hữu mà còn "không bao giờ nên xảy ra".
- **Ngoại lệ là sự đánh đổi giữa khả năng xử lý điều kiện bất ngờ và sự phức tạp tăng lên:** Khi dùng ngoại lệ, tính đóng gói (encapsulation) bị suy yếu vì mã gọi phải biết có thể có ngoại lệ nào được ném ra bên trong. Điều này tăng độ phức tạp của mã, mâu thuẫn với Mệnh lệnh Kỹ thuật Chính của Phần mềm: Quản lý Độ phức tạp.

Đừng dùng ngoại lệ chỉ để "đẩy trách nhiệm"

Nếu một điều kiện lỗi có thể được xử lý tại chỗ, hãy xử lý nó ở đó. Đừng ném một ngoại lệ chưa được xử lý lên trên nếu bạn có thể tự xử lý tại chỗ.

Tránh ném ngoại lệ trong constructor và destructor trừ khi có bắt tại chỗ

Quy tắc xử lý ngoại lệ trở nên phức tạp rất nhanh nếu ném ngoại lệ trong constructor hoặc destructor. Ví dụ, trong C++, destructor sẽ không được gọi nếu một đối tượng chưa được xây dựng đầy đủ, nghĩa là nếu code trong constructor ném ngoại lệ, destructor sẽ không được gọi, dễ dẫn đến rò rỉ tài nguyên. Quy tắc tương tự áp dụng cho exceptions trong destructor.

Các chuyên gia ngôn ngữ có thể nói nhớ các quy tắc này là việc "tầm thường", nhưng lập trình viên thường sẽ khó nhớ hết. Hãy tránh viết dạng code này để

giảm bớt sự phức tạp không cần thiết.

> **Tham khảo chéo:** Để biết thêm về duy trì tính nhất quán của tầng trừu tượng giao diện, xem "Good Abstraction" ở mục 6.2.

Ném ngoại lệ ở mức trừu tượng phù hợp

Hàm nên thể hiện một tầng trừu tượng đồng nhất trong giao diện, và lớp cũng vậy. Các ngoại lệ ném ra là một phần của giao diện routine, cũng như các kiểu dữ liệu cụ thể.

Khi chọn truyền một ngoại lệ cho caller, cần đảm bảo mức trừu tượng của ngoại lệ phù hợp với giao diện routine. Dưới đây là ví dụ về việc làm sai:

Ví dụ Java xấu: Ném ngoại lệ ở mức trừu tượng không nhất quán

```
class Employee {
    public TaxId GetTaxId() throws EOFException {
    }
}
```

Ở đây, `GetTaxId()` truyền một ngoại lệ cấp thấp `EOFException` lên caller. Nó không tự chịu trách nhiệm với exception này mà còn phơi bày chi tiết cách hiện thực bên trong Employee, khiến code phía client phải phụ thuộc cả vào code lớp dưới của Employee (nơi phát sinh EOFException). Điều này phá vỡ đóng gói và làm quản lý trở nên khó khăn.

Thay vào đó, nên ném ngoại lệ có cùng tầng trừu tượng với giao diện lớp:

Ví dụ Java đúng: Ném ngoại lệ ở mức trừu tượng phù hợp

```
class Employee {
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {
    }
}
```

Code xử lý ngoại lệ trong `GetTaxId()` có thể ánh xạ lỗi cấp thấp (ví dụ: `io_disk_not_ready`) thành ngoại lệ `EmployeeDataNotAvailable`, đảm bảo duy trì tầng trừu tượng của interface.

Bao gồm tất cả thông tin liên quan trong thông điệp ngoại lệ

Mỗi ngoại lệ xảy ra trong hoàn cảnh cụ thể, và thông tin này rất quan trọng cho người đọc thông điệp ngoại lệ. Hãy đảm bảo thông điệp có chứa các thông tin để hiểu rõ lý do ngoại lệ phát sinh. Nếu ngoại lệ do lỗi chỉ số mảng, thông điệp phải ghi rõ giới hạn trên, dưới và giá trị chỉ số không hợp lệ.

Tránh khối catch rỗng (empty catch blocks)

Đôi khi bạn có thể cảm thấy nên "lơ đi" một ngoại lệ như sau:

Ví dụ Java xấu: Bỏ qua ngoại lệ

```
try {
    // lots of code
} catch (AnException exception) {
}
```

Cách này chứng tỏ hoặc code trong try là sai (vì sinh ra exception mà không có lý do chính đáng), hoặc code trong catch là sai (vì không xử lý ngoại lệ hợp lệ). Xác định nguyên nhân thực sự, rồi sửa code ở khối try hoặc catch cho hợp lý!

Nếu thực sự có những trường hợp hiếm mà ngoại lệ ở tầng dưới không còn là ngoại lệ ở tầng trừu tượng caller, ít nhất nên ghi chú rõ lý do khối catch rỗng. Có thể dùng chú thích hoặc ghi log vào file, ví dụ:

Ví dụ Java đúng: Ghi nhận ngoại lệ được bỏ qua

```
try {
    // lots of code
} catch (AnException exception) {
    LogError("Unexpected exception");
}
```

Nắm rõ các ngoại lệ mà thư viện trả về

Nếu bạn làm việc với một ngôn ngữ không yêu cầu routine hoặc lớp phải định nghĩa rõ ngoại lệ nó có thể ném, hãy chắc chắn rằng bạn biết các ngoại lệ mà mọi mã thư viện bạn dùng có thể sinh ra.

Nếu mã thư viện không ghi chú các ngoại lệ mà nó phát sinh

Nếu mã thư viện không ghi chú rõ các ngoại lệ (exception) mà nó phát sinh, hãy xây dựng một đoạn mã thử nghiệm (prototyping code) để kiểm tra các thư viện này và phát hiện các ngoại lệ tiềm ẩn.

Xây dựng bộ báo cáo ngoại lệ tập trung (Centralized Exception Reporter)

Một cách tiếp cận nhằm đảm bảo tính nhất quán trong xử lý ngoại lệ là sử dụng bộ báo cáo ngoại lệ tập trung (centralized exception reporter). Bộ báo cáo này cung cấp một kho lưu trữ tập trung về các loại ngoại lệ, cách thức xử lý từng ngoại lệ, định dạng thông báo ngoại lệ và các thông tin liên quan.

Dưới đây là ví dụ về một **trình xử lý ngoại lệ đơn giản** chỉ in ra thông báo chẩn đoán:

Ví dụ Visual Basic về bộ báo cáo ngoại lệ tập trung - Phần 1

```
Sub ReportException( _
   ByVal className, _
   ByVal thisException As Exception _
)
   Dim message As String
   Dim caption As String
   message = "Exception: " & thisException.Message & " " & ControlChars.CrLf & _
             "Class: " & className & ControlChars.CrLf & _
             "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf
   caption = "Exception"
   MessageBox.Show(message, caption, MessageBoxButtons.OK, _
                   MessageBoxIcon.Exclamation)
End Sub
```

Bạn sẽ sử dụng **trình xử lý ngoại lệ tổng quát này** như sau:

Ví dụ Visual Basic về bộ báo cáo ngoại lệ tập trung - Phần 2

```
Try

Catch exceptionObject As Exception
   ReportException(CLASS_NAME, exceptionObject)
End Try
```

Đoạn mã trong phiên bản này của `ReportException()` rất đơn giản. Trong một ứng dụng thực tế, bạn có thể làm đoạn mã này đơn giản hoặc phức tạp tùy thuộc nhu cầu xử lý ngoại lệ của mình.

Khi quyết định xây dựng một bộ báo cáo ngoại lệ tập trung, cần cân nhắc các vấn đề tổng quan liên quan đến xử lý lỗi tập trung, như đã được đề cập trong mục **"Gọi một routine/đối tượng xử lý lỗi" ở Mục 8.3**.

Chuẩn hóa việc sử dụng ngoại lệ trong dự án

Để việc xử lý ngoại lệ trở nên dễ quản lý hơn về mặt tư duy, bạn có thể chuẩn hóa quy ước sử dụng ngoại lệ theo các cách sau:

- Nếu bạn đang làm việc với ngôn ngữ như C++ cho phép phát sinh nhiều loại đối tượng, dữ liệu và con trỏ khác nhau, hãy chuẩn hóa loại đối tượng cụ thể mà bạn sẽ phát sinh (throw). Để đảm bảo khả năng tương thích với các ngôn ngữ khác, nên cân nhắc chỉ phát sinh các đối tượng kế thừa từ lớp cơ sở `Exception`.
- Xem xét tạo một lớp ngoại lệ riêng cho từng dự án, lớp này có thể làm lớp cơ sở cho tất cả các ngoại lệ phát sinh trong dự án. Việc này giúp tập trung và chuẩn hóa việc ghi nhật ký, báo cáo lỗi, v.v.
- Xác định rõ các trường hợp mã nguồn được phép sử dụng cú pháp throw-catch để xử lý lỗi cục bộ.

- Xác định rõ các trường hợp mã nguồn được phép phát sinh ngoại lệ mà không xử lý cục bộ.
- Quyết định xem có sử dụng bộ báo cáo ngoại lệ tập trung hay không.
- Định nghĩa rõ việc ngoại lệ có được phép phát sinh trong constructor và destructor hay không.

Tham khảo thêm

> Với nhiều phương pháp xử lý lỗi thay thế, xem mục "Error-Handling Techniques" (Kỹ thuật xử lý lỗi) trước đó trong chương này.

Cân nhắc các phương án thay thế ngoại lệ

Nhiều ngôn ngữ lập trình đã hỗ trợ ngoại lệ từ 5-10 năm hoặc hơn, tuy nhiên vẫn chưa hình thành nhiều quan điểm chung về cách sử dụng chúng một cách an toàn.

Một số lập trình viên sử dụng ngoại lệ chỉ vì ngôn ngữ họ dùng hỗ trợ cơ chế xử lý lỗi này. Tuy nhiên, bạn nên cân nhắc đầy đủ các phương án xử lý lỗi, bao gồm:

- Xử lý lỗi ngay tại nơi phát sinh (local handling)
- Truyền mã lỗi (error code)
- Ghi nhật ký thông tin gỡ lỗi vào tập tin
- Tắt hệ thống
- Hoặc sử dụng các cách tiếp cận khác

Việc sử dụng ngoại lệ chỉ vì ngôn ngữ hỗ trợ cơ chế này là một ví dụ điển hình của **lập trình theo ngôn ngữ (programming in a language)** thay vì **lập trình khai thác ngôn ngữ (programming into a language)** (xem thêm chi tiết ở mục "Your Location on the Technology Wave" và mục "Program into Your Language, Not in It").

Cuối cùng, cần cân nhắc liệu chương trình của bạn **thực sự cần xử lý ngoại lệ hay không**. Như Bjarne Stroustrup đã chỉ ra, đôi khi phản ứng tối ưu với một lỗi thời gian chạy nghiêm trọng là giải phóng toàn bộ tài nguyên đã cấp phát rồi kết thúc chương trình. Hãy để người dùng chạy lại chương trình với đầu vào hợp lệ (Stroustrup 1997).

8.5 Dựng rào chắn trong chương trình để hạn chế thiệt hại do lỗi gây ra

Barricade là chiến lược ngăn chặn thiệt hại (damage-containment strategy). Điều này tương tự lý do các khoang tàu thường được cách biệt; nếu xảy ra sự cố thủng vỏ tàu, khoang bị ảnh hưởng sẽ được cách ly để phần còn lại không bị tác động. Tương tự, tường lửa trong tòa nhà được sử dụng để ngăn chặn cháy lan.

(Trước đây, "firewall" được dùng đồng nghĩa với "barricade" nhưng hiện tại chủ yếu chỉ dùng để chỉ các biện pháp chặn lưu lượng mạng độc hại.)

Một cách để tạo **rào chắn** trong lập trình phòng thủ là xác định một số giao diện (interface) như ranh giới giữa "vùng an toàn". Hãy kiểm tra tính hợp lệ của dữ liệu vượt qua ranh giới này và xử lý hợp lý nếu dữ liệu không hợp lệ.

Hình minh họa bên dưới cho thấy:

> Một số phần của phần mềm sẽ làm việc với dữ liệu "bẩn" (dirty data) và không tin cậy, chúng chịu trách nhiệm làm sạch dữ liệu và tạo nên rào chắn. Các phần bên trong rào chắn có thể giả định dữ liệu đã sạch (clean) và đáng tin cậy (trusted).

Phương pháp này có thể áp dụng ở cấp lớp (class level): các phương thức public của lớp sẽ kiểm tra và làm sạch dữ liệu đầu vào, còn các phương thức private có thể giả định dữ liệu đã hợp lệ.

Một cách ví von khác là coi đây như kỹ thuật phòng phẫu thuật: Dữ liệu phải được "tiệt trùng" trước khi đưa vào phòng mổ. Quyết định thiết kế then chốt là xác định đối tượng nào có thể "vào phòng mổ", cái nào thì không, và đặt các "cửa" (routine) ở đâu — routine nào nằm trong vùng an toàn, routine nào ngoài, routine nào chịu trách nhiệm làm sạch dữ liệu.

Thông thường, thuận tiện nhất là làm sạch dữ liệu từ bên ngoài ngay khi nhận được. Tuy nhiên, đôi khi cần làm sạch ở nhiều cấp độ khác nhau.

Chuyển đổi dữ liệu đầu vào về đúng kiểu ngay tại thời điểm nhập liệu

Thông thường, dữ liệu đầu vào dưới dạng chuỗi (string) hoặc số (number), đôi khi ánh xạ sang kiểu boolean như "yes" hoặc "no", hoặc sang dạng liệt kê (enumerated type) như `Color_Red`, `Color_Green`, `Color_Blue`. Nếu duy trì dữ liệu chưa xác định kiểu trong thời gian dài sẽ làm tăng độ phức tạp và rủi ro (ví dụ nhập giá trị "Yes" làm cho chương trình lỗi). Vì vậy, hãy chuyển đổi dữ liệu về đúng kiểu càng sớm càng tốt.

8.6 Công cụ hỗ trợ gỡ lỗi (Debugging Aids)

Một khía cạnh then chốt khác của lập trình phòng thủ là sử dụng các công cụ hỗ trợ gỡ lỗi (debugging aids), giúp phát hiện lỗi nhanh chóng.

Không áp dụng mặc định các ràng buộc của phiên bản sản xuất vào phiên bản phát triển

Nhiều lập trình viên thường bị mắc kẹt trong quan niệm rằng các giới hạn của phần mềm sản xuất (production software) cũng áp dụng cho phiên bản phát triển. Thật ra, phiên bản sản xuất cần chạy nhanh và tiết kiệm tài nguyên,

trong khi phiên bản phát triển có thể hi sinh tốc độ, tiêu thụ nhiều tài nguyên hơn để phục vụ cho các công cụ hỗ trợ phát triển.

Ví dụ: Trong một dự án, chúng tôi sử dụng danh sách liên kết bốn chiều (quadruply linked list) dễ xảy ra lỗi. Tôi đã bổ sung chức năng kiểm tra tính toàn vẹn của danh sách vào menu.

Trong chế độ gỡ lỗi (debug mode), Microsoft Word có đoạn mã kiểm tra trạng thái của đối tượng Document liên tục vài giây một lần, giúp phát hiện sớm lỗi trên dữ liệu.

> Hãy sẵn sàng đánh đổi tốc độ và tài nguyên trong giai đoạn phát triển để đổi lấy các công cụ hỗ trợ giúp quá trình phát triển dễ dàng và trơn tru hơn.

Đưa vào các công cụ hỗ trợ gỡ lỗi từ sớm

Càng sớm bổ sung các công cụ hỗ trợ gỡ lỗi, bạn càng hưởng lợi nhiều hơn. Thông thường, bạn sẽ không viết các công cụ hỗ trợ này cho đến khi gặp phải một vấn đề nào đó nhiều lần.

Ghi chú dịch thuật: - Thuật ngữ như: exception, class, routine, assertion, debug mode, v.v. được giữ nguyên và giải thích lần đầu tiên. - Một số lỗi đánh máy/gõ thừa của bản gốc đã được sửa hoặc bỏ qua để tăng sự mạch lạc (như các dấu câu thiếu, cách dòng lặp lại). - Bài dịch được trình bày ngắn gọn, đúng chuẩn truyền đạt học thuật, nhất quán về thuật ngữ và cấu trúc.

Sử Dụng Offensive Programming (Lập trình Tấn công)

> **Tham khảo chéo:** Để biết thêm chi tiết về xử lý các trường hợp không lường trước, xem mục “Tips for Using case Statements” trong Phần 15.2.

Những tình huống ngoại lệ (exceptional cases) cần được xử lý sao cho chúng trở nên rõ ràng trong quá trình phát triển và có thể khôi phục được khi code vận hành thực tế. Michael Howard và David LeBlanc gọi phương pháp này là “offensive programming” (lập trình tấn công) (Howard và LeBlanc, 2003).

Giả sử bạn có một câu lệnh `case` dự kiến sẽ xử lý chỉ năm loại sự kiện. Trong quá trình phát triển, nhánh `default` nên được sử dụng để tạo ra cảnh báo, ví dụ: “Có một trường hợp mới! Hãy sửa chương trình!”. Tuy nhiên, trong môi trường vận hành (production), nhánh `default` nên thực hiện một hành động nhẹ nhàng hơn, như ghi thông báo vào file nhật ký lỗi (error-log file).

> Một chương trình bị dừng hoàn toàn (dead program) thường gây ra ít thiệt hại hơn một chương trình hoạt động chập chờn (crippled

program).

— Andy Hunt và Dave Thomas

Dưới đây là một số cách bạn có thể lập trình theo hướng tấn công (offensive):

- **Đảm bảo các lệnh xác nhận (assert) sẽ dừng chương trình.** Đừng để lập trình viên chỉ việc nhấn phím Enter để bỏ qua một vấn đề đã biết. Hãy làm cho vấn đề này đủ khó chịu để bắt buộc phải sửa.
- **Ghi đầy bộ nhớ đã được cấp phát** để có thể phát hiện lỗi cấp phát bộ nhớ (memory allocation errors).
- **Ghi đầy các file hoặc luồng dữ liệu được cấp phát** để phát hiện lỗi định dạng file (file-format errors).
- **Đảm bảo mã lệnh ở nhánh `default` hoặc `else` trong mỗi câu lệnh `case` phải "fail hard" (dừng chương trình), hoặc gây chú ý lớn, không thể bị bỏ sót.**
- **Làm đầy đối tượng với dữ liệu "rác" trước khi xóa** để dễ phát hiện sử dụng nhầm dữ liệu đã xóa.
- **Cấu hình chương trình gửi file log lỗi đến email của bạn**, nếu phù hợp với loại phần mềm bạn phát triển, để bạn biết các loại lỗi đang diễn ra khi phần mềm đã phát hành.

Đôi khi, phòng ngự tốt nhất là tấn công mạnh mẽ. Hãy làm chương trình "fail hard" (lỗi lớn) trong giai đoạn phát triển, nhằm giúp chương trình "fail soft" (lỗi nhẹ nhàng) khi đã đưa vào sản xuất.

Lập Kế Hoạch Loại Bỏ Công Cụ Gỡ Lỗi (Debugging Aids)

Nếu bạn viết code cho mục đích cá nhân, việc giữ lại toàn bộ code gỡ lỗi trong chương trình có thể không sao. Tuy nhiên, khi phát triển phần mềm thương mại, cái giá phải trả về dung lượng và tốc độ sẽ rất lớn. Vì thế, hãy lên kế hoạch tránh việc liên tục thêm/bớt code gỡ lỗi khỏi chương trình. Sau đây là vài cách thực hiện:

> **Tham khảo chéo:** Để biết chi tiết về quản lý phiên bản, xem Phần 28.2 "Configuration Management".

- **Sử dụng công cụ quản lý phiên bản (version-control tools)** và công cụ build như `ant` hoặc `make`. Các công cụ quản lý phiên bản cho phép xây dựng nhiều phiên bản chương trình từ cùng một bộ mã nguồn gốc. Ở chế độ phát triển, bạn có thể cấu hình để build bao gồm toàn bộ code gỡ lỗi. Ở chế độ vận hành, code debug sẽ bị loại bỏ khỏi phiên bản thương mại.
- **Sử dụng preprocessor (tiền xử lý) có sẵn.** Nếu môi trường lập trình có hỗ trợ preprocessor, như C++, bạn có thể bao gồm hoặc loại trừ code

gỡ lỗi chỉ bằng một chuyển đổi trong quá trình biên dịch. Bạn có thể dùng trực tiếp preprocessor hoặc định nghĩa macro tương ứng. Ví dụ:

```
#define DEBUG

#if defined( DEBUG )
// debugging code
#endif
```

Với chủ đề này, có thể có nhiều biến thể khác nhau. Thay vì chỉ định nghĩa `DEBUG`, bạn có thể gán cho nó một giá trị và kiểm tra giá trị đó để phân biệt các mức độ debug khác nhau. Ví dụ, một số đoạn code gỡ lỗi cần tồn tại mọi lúc thì có thể đặt điều kiện như `#if DEBUG > 0`, hoặc một số đoạn code chỉ cho mục đích nhất định thì dùng như `#if DEBUG == POINTER_ERROR`.

Nếu bạn không thích việc lặp đi lặp lại nhiều dòng `#if defined()` trong code, hãy tạo macro tiền xử lý như sau:

```
#define DEBUG
#if defined( DEBUG )
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif

DebugCode(
    statement 1;
    statement 2;
    statement n;
);
```

Kỹ thuật này cũng có thể được mở rộng để tinh chỉnh hơn, không chỉ hoàn toàn loại bỏ hoặc giữ lại toàn bộ code debug.

> **Tham khảo chéo:** Xem thêm về tiền xử lý (preprocessor) tại "Macro Preprocessors" trong Phần 30.3.

- **Viết tiền xử lý riêng** nếu ngôn ngữ không hỗ trợ có sẵn. Bạn có thể đặt quy ước đánh dấu các đoạn code debug (ví dụ, trong Java dùng `//#BEGIN DEBUG` và `//#END DEBUG`) và viết script xử lý trước khi biên dịch. Như vậy, bạn sẽ tiết kiệm thời gian về lâu dài và tránh việc vô tình biên dịch code chưa được xử lý qua preprocessor.
- **Sử dụng debugging stubs (chương trình con giả để kiểm tra).** Trong nhiều trường hợp, bạn có thể gọi một routine để kiểm tra khi debug. Trong bản phát triển, routine này sẽ thực hiện nhiều phép kiểm tra, còn trong bản sản xuất, bạn chỉ cần routine trả về ngay lập tức hoặc thực hiện tối thiểu thao tác, giúp giảm thiểu tác động đến hiệu năng. Bạn nên giữ cả hai phiên bản phát triển và sản xuất để tiện hoán đổi khi cần.

Ví dụ:

```
void DoSomething(
    SOME_TYPE *pointer;
) {
    CheckPointer( pointer );
}
```

Trong phát triển, routine `CheckPointer()` sẽ kiểm tra kỹ đầu vào:

```
void CheckPointer( void *pointer ) {
    // kiểm tra pointer không phải NULL
    // kiểm tra dogtag hợp lệ
    // kiểm tra vùng trỏ tới không bị hỏng
    // ...
}
```

Ở môi trường sản xuất, bạn có thể dùng routine rỗng:

```
void CheckPointer( void *pointer ) {
    // Không làm gì, trả về ngay
}
```

Đây không phải là danh sách đầy đủ các cách loại bỏ công cụ gỡ lỗi, nhưng đủ để bạn lựa chọn giải pháp phù hợp cho môi trường phát triển của mình.

8.7 Xác Định Mức Độ Defensive Programming Cần Giữ Lại Trong Code Sản Xuất

Một nghịch lý trong defensive programming (lập trình phòng ngừa) là trong quá trình phát triển, bạn muốn lỗi xuất hiện thật rõ rệt, thậm chí "đáng ghét" để đảm bảo không bỏ sót. Nhưng khi tiến hành vận hành, bạn lại mong lỗi càng nhẹ nhàng, không ảnh hưởng lớn, chương trình có khả năng phục hồi hoặc dừng một cách "êm ái".

Sau đây là một số hướng dẫn về việc quyết định giữ/loại bỏ defensive programming trong phiên bản sản xuất:

- **Giữ lại code kiểm tra các lỗi quan trọng.** Xác định khu vực nào của chương trình cho phép bỏ sót lỗi (undetected errors), khu vực nào không. Ví dụ, trong chương trình bảng tính, bạn có thể bỏ qua lỗi không phát hiện trong phần cập nhật màn hình vì hậu quả chỉ là giao diện bị rối. Tuy nhiên, không thể bỏ qua lỗi trong engine tính toán, vì có thể dẫn đến kết quả sai ngầm, gây ra hậu quả nghiêm trọng như khai thuế bị kiểm toán.
- **Loại bỏ code kiểm tra các lỗi không quan trọng.** Nếu một lỗi có hậu quả thật sự tầm thường, hãy loại bỏ code kiểm tra lỗi đó — nghĩa là sử dụng công cụ quản lý phiên bản, preprocessor, hoặc kỹ thuật khác để không biên dịch đoạn code này vào chương trình thương mại. Nếu dung

lượng không phải vấn đề, bạn có thể giữ code kiểm tra này nhưng để nó chỉ ghi lỗi một cách âm thầm vào file nhật ký.

- **Loại bỏ code gây crash mạnh (hard crashes).** Trong giai đoạn phát triển, khi chương trình phát hiện lỗi, bạn muốn lỗi xuất hiện thật rõ ràng, kể cả với các lỗi nhỏ, thường là bằng cách in ra thông báo debug rồi crash chương trình. Tuy nhiên, ở bản vận hành, người dùng cần có thời gian lưu dữ liệu trước khi chương trình gặp sự cố, và sẵn sàng chấp nhận một số sai sót nhỏ để tiếp tục sử dụng tiếp. Người dùng không thích mất dữ liệu, bất kể điều đó giúp cho việc debug về sau. Vì vậy, nếu chương trình chứa đoạn mã debug có thể gây mất dữ liệu, hãy loại bỏ chúng ở bản sản xuất.

Ví dụ từ Mars Pathfinder về việc sử dụng mã gỡ lỗi trong sản phẩm

Trong dự án Mars Pathfinder, các kỹ sư đã chủ động giữ lại một số đoạn mã gỡ lỗi (debug code) trong phần mềm. Một lỗi đã phát sinh sau khi Pathfinder hạ cánh xuống sao Hỏa. Nhờ vào các công cụ hỗ trợ gỡ lỗi còn được giữ lại, các kỹ sư tại JPL đã có thể chẩn đoán vấn đề, sau đó tải lên phiên bản mã đã chỉnh sửa cho Pathfinder, giúp thiết bị hoàn thành nhiệm vụ một cách hoàn hảo (tháng 3 năm 1999).

Ghi nhận lỗi cho đội ngũ hỗ trợ kỹ thuật

- Hãy ghi lại (log) các lỗi cho đội ngũ hỗ trợ kỹ thuật. Xem xét việc giữ lại các công cụ hỗ trợ gỡ lỗi trong mã sản phẩm, nhưng điều chỉnh hành vi của chúng sao cho phù hợp với phiên bản chính thức.
- Nếu bạn đã thêm nhiều biểu thức xác nhận (assertion) khiến chương trình dừng lại trong quá trình phát triển, bạn nên cân nhắc thay đổi routine assertion sang ghi lại thông báo vào tệp (log) khi chạy ở môi trường sản xuất, thay vì loại bỏ hoàn toàn chúng.

Đảm bảo thông báo lỗi thân thiện

- Đảm bảo rằng các thông báo lỗi để lại trong chương trình là thân thiện với người dùng. Nếu giữ lại các thông báo lỗi nội bộ, hãy chắc chắn chúng sử dụng ngôn ngữ phù hợp với người dùng cuối.
- Trong một chương trình trước đây của tôi, một người dùng đã gọi và báo rằng cô ấy nhận được thông báo: “You’ve got a bad pointer allocation, Dog Breath!” (Bạn có một phân bổ con trỏ lỗi, Hơi Thở Chó!). May mắn là cô ấy có khiếu hài hước.
- Một cách tiếp cận phổ biến và hiệu quả là thông báo với người dùng về một “lỗi nội bộ” và cung cấp địa chỉ email hoặc số điện thoại để người

dùng báo cáo lỗi đó.

8.8 Cảnh giác với lập trình phòng thủ quá mức

> "Quá nhiều bất cứ điều gì đều không tốt, nhưng quá nhiều whisky thì vừa đủ."
> — Mark Twain

Lập trình phòng thủ (defensive programming) thái quá cũng gây ra những vấn đề riêng. Nếu bạn kiểm tra dữ liệu được truyền dưới mọi hình thức, tại mọi nơi có thể, chương trình của bạn sẽ trở nên "phình to" và chậm chạp. Nghiêm trọng hơn, mã nguồn bổ sung cho mục đích phòng thủ còn làm tăng độ phức tạp phần mềm; bản thân mã lập trình phòng thủ cũng không miễn nhiễm với lỗi, thậm chí khả năng có lỗi còn cao hơn nếu bạn viết chúng một cách sơ sài. Do đó, hãy cân nhắc vị trí cần có sự phòng thủ và ưu tiên những nơi thực sự cần thiết.

Danh sách kiểm tra: Lập trình phòng thủ (Defensive Programming)

Tổng quan

- Routine có tự bảo vệ khỏi dữ liệu đầu vào xấu không?
- Bạn đã sử dụng assertion để ghi rõ các giả định, gồm tiền điều kiện (precondition) và hậu điều kiện (postcondition) chưa?
- Assertion chỉ được dùng để ghi nhận các trường hợp không bao giờ được phép xảy ra?
- Kiến trúc hoặc thiết kế cấp cao liệu có xác định tập hợp các kỹ thuật xử lý lỗi cụ thể chưa?
- Kiến trúc hoặc thiết kế cấp cao có quy định việc nên ưu tiên tính bền vững (robustness) hay tính chính xác (correctness) khi xử lý lỗi?
- Có tạo các hàng rào (barricades) để giới hạn phạm vi ảnh hưởng của lỗi và giảm số lượng mã cần quan tâm đến xử lý lỗi hay không?
- Công cụ hỗ trợ gỡ lỗi đã được sử dụng trong mã nguồn chưa?
- Công cụ hỗ trợ gỡ lỗi đã được thiết lập để có thể bật/tắt một cách linh hoạt chưa?
- Lượng mã để lập trình phòng thủ đã hợp lý — không quá nhiều cũng không quá ít?
- Bạn đã sử dụng các kỹ thuật lập trình chủ động (offensive-programming) để khiến lỗi khó bị bỏ qua khi phát triển chưa?

Xử lý ngoại lệ (Exception)

- Dự án đã định nghĩa phương pháp chuẩn cho việc xử lý ngoại lệ chưa?
- Đã xem xét các giải pháp thay thế việc sử dụng exception chưa?
- Lỗi được xử lý cục bộ ở mức routine thay vì ném (throw) exception ra ngoài không cần thiết?
- Mã tránh ném ngoại lệ trong constructor và destructor chưa?
- Mọi exception đều ở đúng mức trừu tượng đối với routine ném chúng?
- Mỗi exception đều kèm theo đầy đủ thông tin nền liên quan?
- Đã loại bỏ các khối bắt rỗng (empty catch block)? Nếu bắt rỗng là cần thiết, đã được ghi chú rõ ràng?

Vấn đề bảo mật

- Mã kiểm tra dữ liệu đầu vào có kiểm soát tràn bộ đệm (buffer overflow), SQL injection, HTML injection, tràn số nguyên (integer overflow) và các dạng đầu vào độc hại khác không?
- Mọi mã trả về báo lỗi (error-return code) đều được kiểm tra?
- Mọi exception đều được bắt (caught)?
- Thông báo lỗi không tiết lộ thông tin có thể giúp kẻ tấn công xâm nhập hệ thống?

Tài liệu tham khảo về Lập trình phòng thủ (Defensive Programming)

Bảo mật

- Howard, Michael và David LeBlanc. *Writing Secure Code*, lần thứ 2. Microsoft Press, 2003. Tác giả phân tích các hệ lụy bảo mật của việc tin tưởng đầu vào, minh họa nhiều cách một chương trình có thể bị tấn công. Nội dung bao gồm yêu cầu về bảo mật, thiết kế, lập trình và kiểm thử.

Assertion

- Maguire, Steve. *Writing Solid Code*. Microsoft Press, 1993. Chương 2 bàn rất sâu về cách dùng assertion, cùng nhiều ví dụ thực tế từ sản phẩm của Microsoft.
- Stroustrup, Bjarne. *The C++ Programming Language*, lần thứ 3. Addison-Wesley, 1997. Mục 24.3.7.2 trình bày một số biến thể của triển khai assertion trong C++ cũng như mối quan hệ giữa assertion với precondition và postcondition.
- Meyer, Bertrand. *Object-Oriented Software Construction*, lần thứ 2. Prentice Hall PTR, 1997. Sách này đề cập sâu sắc đến tiền/hậu điều kiện.

Ngoại lệ (Exception)

- Meyer, Bertrand. *Object-Oriented Software Construction*, lần thứ 2. Prentice Hall PTR, 1997. Chương 12 trình bày chi tiết về xử lý ngoại lệ.
- Stroustrup, Bjarne. *The C++ Programming Language*, lần thứ 3. Addison-Wesley, 1997. Chương 14 thảo luận sâu về xử lý ngoại lệ trong C++. Mục 14.11 tóm lược 21 lưu ý khi xử lý ngoại lệ trong C++.
- Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1996. Các mục 9–15 đề cập nhiều khía cạnh tinh tế của xử lý ngoại lệ trong C++.
- Arnold, Ken, James Gosling, và David Holmes. *The Java Programming Language*, lần thứ 3. Addison-Wesley, 2000. Chương 8 bàn về xử lý ngoại lệ trong Java.
- Bloch, Joshua. *Effective Java Programming Language Guide*. Addison-Wesley, 2001. Các mục 39–47 nói về các khía cạnh tinh tế của xử lý ngoại lệ trong Java.
- Foxall, James. *Practical Standards for Microsoft Visual Basic.NET*. Microsoft Press, 2003. Chương 10 nói về xử lý ngoại lệ trong Visual Basic.

Những điểm mấu chốt

- **Mã sản xuất phải xử lý lỗi tinh vi hơn nhiều so với kiểu "rác vào, rác ra" (garbage in, garbage out).**
- **Kỹ thuật lập trình phòng thủ giúp phát hiện, sửa lỗi dễ dàng và giảm hậu quả lỗi với hệ thống khi vận hành thực tế.**
- **Assertion giúp phát hiện lỗi sớm, đặc biệt hiệu quả trong hệ thống lớn, đòi hỏi độ tin cậy cao hoặc có mã thường xuyên thay đổi.**
- **Việc quyết định xử lý dữ liệu đầu vào xấu như thế nào là hai quyết định then chốt — cả trong xử lý lỗi lẫn thiết kế cấp cao.**
- **Ngoại lệ (exception) cung cấp một phương thức xử lý lỗi khác với luồng điều khiển thông thường, là công cụ hữu ích nếu sử dụng đúng mức và nên được cân nhắc so với các kỹ thuật khác.**
- **Những ràng buộc áp dụng với hệ thống sản xuất không nhất thiết phải dùng cho phiên bản phát triển. Bạn nên tận dụng để thêm mã hỗ trợ nhận diện lỗi cho phiên bản phát triển.**

Chương 9: Quá trình Lập trình Pseudocode (Pseudocode Programming Process)

Nội dung

Chủ đề liên quan

- Tạo lớp chất lượng cao: Chương 6
- Đặc điểm routine chất lượng cao: Chương 7
- Thiết kế trong quá trình kiến tạo: Chương 5
- Phong cách chú thích code: Chương 32

Mặc dù toàn bộ cuốn sách này có thể được xem là mô tả mở rộng về quy trình lập trình cho việc xây dựng lớp và routine, chương này đặt các bước đó vào một bối cảnh cụ thể. Trọng tâm của chương là việc lập trình "ở tầm nhỏ" – các bước cụ thể để xây dựng một lớp riêng lẻ cùng các routine, các bước này đều có vai trò quan trọng ở mọi quy mô dự án. Chương còn trình bày về Quá trình Lập trình Pseudocode (Pseudocode Programming Process - PPP), giúp giảm khối lượng công việc khi thiết kế và tài liệu hóa, đồng thời nâng cao chất lượng cho cả hai.

Nếu bạn là lập trình viên chuyên nghiệp, bạn có thể chỉ cần đọc lướt chương này, nhưng hãy xem tóm tắt các bước và các lời khuyên xây routine với PPP ở Mục 9.3. Ít lập trình viên khai thác triệt để sức mạnh của PPP, dù nó mang lại rất nhiều lợi ích.

PPP không phải là quy trình duy nhất để xây dựng lớp và routine. Mục 9.4 ở cuối chương sẽ giới thiệu thêm về các lựa chọn phổ biến khác, như phát triển theo hướng kiểm thử trước (test-first development) hoặc thiết kế theo hợp đồng (design by contract).

9.1 Tóm tắt các bước xây dựng lớp và routine

Việc xây dựng lớp có thể được tiếp cận bằng nhiều cách, nhưng thông thường, đó là một quy trình lặp lại giữa việc tạo một thiết kế tổng quát cho lớp, liệt kê các routine cụ thể bên trong lớp, xây dựng các routine chi tiết, kiểm tra và đánh giá tổng thể lớp. Như Hình 9-1 minh họa, quá trình xây dựng lớp có thể

khá "lộn xộn", do bản chất "lộn xộn" sẵn có của quy trình thiết kế (như được trình bày ở Mục 5.1, "Các thách thức thiết kế").

```
Begin
Tạo thiết kế tổng quát
cho lớp
Xem xét và Kiểm tra
toàn bộ lớp trong khi
xây dựng routine chi tiết
Kết thúc
```

Hình 9-1: Các chi tiết trong xây dựng lớp có thể khác nhau, nhưng các hoạt động thường diễn ra theo trình tự như trên.

Các bước then chốt khi xây dựng lớp

1. **Tạo thiết kế tổng quát cho lớp:** Thiết kế lớp bao gồm nhiều yếu tố cụ thể.
2. **Xác định trách nhiệm cụ thể của lớp, "bí mật" mà lớp che giấu cùng các trừu tượng giao diện lớp đại diện.**
3. **Xác định xem lớp có kế thừa từ lớp khác hay không, và các lớp khác có thể kế thừa lại lớp này không.**

Lưu ý: Một số lỗi đánh máy, dấu câu thiếu hoặc sai ở bản gốc đã được chỉnh sửa lại trong bản dịch nhằm đảm bảo tính mạch lạc và dễ hiểu.

Lặp lại các chủ đề này nhiều lần nếu cần để xây dựng một thiết kế rõ ràng cho routine

Những cân nhắc này, cùng nhiều vấn đề khác, sẽ được thảo luận chi tiết hơn trong **Chương 6, "Làm việc với Classes"**.

9.1 Tóm tắt Các Bước Xây Dựng Classes và Routines

Xây dựng từng routine trong class
Sau khi đã xác định các routine chính của class ở bước đầu tiên, bạn cần triển khai từng routine cụ thể. Quá trình xây dựng mỗi routine thường sẽ làm phát sinh nhu cầu thiết kế thêm các routine bổ sung (bao gồm cả routine nhỏ và lớn), và các vấn đề liên quan đến việc tạo ra các routine này thường sẽ tác động ngược trở lại thiết kế tổng thể của class.

Rà soát và kiểm thử toàn bộ class
Thông thường, mỗi routine sẽ được kiểm thử ngay khi được tạo ra. Sau khi toàn bộ class đã vận hành được, việc rà soát và kiểm thử tổng thể cho class là cần thiết nhằm phát hiện các vấn đề không thể kiểm tra ở cấp độ routine riêng lẻ.

Các Bước Xây Dựng một Routine

Nhiều routine trong một class sẽ rất đơn giản và dễ thực hiện—chẳng hạn như **accessor routines** (routine truy xuất), hoặc các routine chuyển tiếp tới routine của đối tượng khác. Tuy nhiên, cũng có những routine yêu cầu triển khai phức tạp hơn; việc xây dựng chúng càng được lợi nhiều từ một quy trình hệ thống.

Các hoạt động chính trong quá trình tạo routine—**thiết kế routine, kiểm tra thiết kế, viết code cho routine, kiểm tra code**—thường được thực hiện theo trình tự minh họa ở Hình 9-2.

```
Bắt đầu
Thiết kế routine
Kiểm tra thiết kế
Lặp lại nếu cần thiết
Viết và kiểm thử code của routine
Hoàn thành
```

Hình 9-2: Các hoạt động chính để xây dựng một routine thường được thực hiện theo thứ tự trên.

Các chuyên gia đã phát triển nhiều cách tiếp cận khác nhau để xây dựng routines, và phương pháp yêu thích của tôi là **Pseudocode Programming Process (PPP)**, sẽ được trình bày ở phần tiếp theo.

9.2 Pseudocode dành cho Chuyên gia

Pseudocode đề cập đến một ký hiệu xác định thuật toán, routine, class hoặc chương trình dưới dạng phi chính thức, gần với tiếng Anh tự nhiên. **Pseudocode Programming Process** (PPP) xác định một phương pháp sử dụng pseudocode để tối ưu hóa quá trình tạo code cho các routine.

Vì pseudocode giống với tiếng Anh, bạn có thể nghĩ rằng bất kỳ mô tả nào mang tính chất tiếng Anh đều mang lại hiệu quả tương tự. Tuy nhiên, thực tiễn chứng minh một số phong cách pseudocode hiệu quả hơn những phong cách khác. Để sử dụng pseudocode hiệu quả, hãy tuân theo các hướng dẫn sau:

- **Sử dụng câu tiếng Anh mô tả chính xác các thao tác cụ thể.**
- **Tránh các thành phần cú pháp của ngôn ngữ lập trình đích.** Pseudocode cho phép bạn thiết kế ở cấp độ trừu tượng cao hơn so với code thực tế. Nếu bạn sử dụng các cấu trúc của ngôn ngữ lập trình, bạn sẽ giảm cấp độ thiết kế, đánh mất lợi ích của việc thiết kế ở cấp cao, đồng thời tự đặt giới hạn bởi các quy tắc cú pháp không cần thiết.
- **Viết pseudocode ở mức độ mong muốn (intent).** Miêu tả ý nghĩa của phương pháp thay vì trình bày quá trình sẽ được hiện thực hóa trong

ngôn ngữ đích.

- **Viết pseudocode chi tiết ở mức mà quá trình chuyển hóa thành code gần như tự động.** Nếu pseudocode quá trừu tượng, có thể bỏ sót các chi tiết quan trọng trong code. Tiếp tục bổ sung chi tiết cho đến khi pseudocode trở nên dễ dàng chuyển đổi thành code.

Sau khi pseudocode được viết xong, bạn phát triển code dựa trên nó và pseudocode này sẽ trở thành phần chú thích trong code. Nhờ đó, công việc chú thích được tối thiểu hóa—nếu pseudocode tuân thủ đúng hướng dẫn, các chú thích sẽ đầy đủ và có ý nghĩa.

Ví dụ: Pseudocode Kém

Dưới đây là một pseudocode vi phạm hầu hết các nguyên tắc trên:

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
return 0
```

Vấn đề:
Khối pseudocode này khó hiểu do sử dụng các chi tiết đặc thù của ngôn ngữ lập trình như *hRsrcPtr (cú pháp trỏ của C) và malloc() (hàm cụ thể của C). Pseudocode này tập trung vào cách viết code hơn là ý định thiết kế.

Ví dụ: Pseudocode Tốt

Thiết kế cho cùng nhiệm vụ ở mức pseudocode tốt hơn:

```
Theo dõi số lượng tài nguyên đang được sử dụng
Nếu còn tài nguyên khả dụng
    Cấp phát một cấu trúc hộp thoại
    Nếu cấu trúc hộp thoại được cấp phát thành công
        Ghi nhận thêm một tài nguyên được sử dụng
        Khởi tạo tài nguyên
        Lưu số hiệu tài nguyên vào vị trí do người gọi cung cấp
    Kết thúc nếu
Kết thúc nếu
Trả về true nếu đã tạo được tài nguyên mới; ngược lại trả về false
```

Pseudocode này tốt hơn ví dụ trước vì chỉ dùng tiếng Anh thông thường, không sử dụng bất kỳ thành phần nào của ngôn ngữ lập trình đích. Nó cũng được trình bày ở cấp độ ý định (intent). Dễ theo dõi hơn, và đủ chi tiết để dùng làm nền tảng chuyển sang code.

Lợi ích của Pseudocode

- **Pseudocode giúp quá trình rà soát (review) dễ dàng hơn:** Có thể xem xét thiết kế chi tiết mà không cần nhìn vào mã nguồn. Các cuộc kiểm tra thiết kế ở cấp thấp trở nên nhẹ nhàng, giảm nhu cầu kiểm tra từng dòng code.
- **Pseudocode hỗ trợ quy trình lặp tinh chỉnh (iterative refinement):** Bắt đầu bằng thiết kế tổng quát, dần chi tiết hóa thành pseudocode, rồi tiếp tục đến mã nguồn. Qua từng bước, bạn sẽ phát hiện lỗi ở đúng cấp độ và sửa chữa kịp thời trước khi lan sang các phần khác.
- **Pseudocode giúp việc thay đổi dễ dàng hơn:** Sửa đổi một vài dòng pseudocode đơn giản hơn nhiều so với việc chỉnh sửa cả trang mã đã viết. Nguyên lý sửa đổi sản phẩm ở giai đoạn "giá trị thấp nhất" (ít chi phí nhất) luôn đúng trong phát triển phần mềm.
- **Pseudocode giảm công sức chú thích:** Thay vì vừa viết code vừa bổ sung chú thích, các phát biểu pseudocode trở thành chú thích, dễ duy trì hơn và tiết kiệm công sức.
- **Pseudocode dễ duy trì hơn so với các dạng tài liệu thiết kế khác:** Với các phương pháp khác, thiết kế thường bị tách rời khỏi code; một khi một trong hai thay đổi, sự đồng bộ sẽ bị phá vỡ. Trong PPP, pseudocode được giữ trong code dưới dạng chú thích, luôn đồng nhất với code nếu được cập nhật.

  Lưu ý quan trọng: Theo khảo sát (Ramsey, Atwood và Van Doren, 1983), lập trình viên ưa thích pseudocode vì nó giúp triển khai code dễ dàng, dễ phát hiện các thiết kế thiếu chi tiết, và thuận tiện cho việc tài liệu hóa, bảo trì.

Pseudocode không phải là công cụ duy nhất, nhưng nó và PPP là những lựa chọn hiệu quả nên có trong bộ công cụ của lập trình viên.

9.3 Xây Dựng Routine Bằng PPP

Phần này mô tả các hoạt động khi xây dựng một routine, bao gồm:

- Thiết kế routine
- Viết code cho routine
- Kiểm tra code
- Hoàn thiện các vấn đề còn tồn đọng
- Lặp lại quy trình nếu cần

Thiết kế routine

Sau khi đã xác định các routine cho class, bước đầu tiên khi xây dựng bất kỳ routine phức tạp nào là thiết kế routine đó. Giả sử bạn cần viết một routine để xuất thông điệp lỗi dựa trên mã lỗi, với tên gọi `ReportErrorMessage()`. Sau đây là đặc tả không chính thức cho `ReportErrorMessage()`:

`ReportErrorMessage()` nhận một mã lỗi dưới dạng đối số đầu vào và xuất ra thông điệp lỗi tương ứng với mã đó. Routine này chịu trách nhiệm xử lý các mã lỗi không hợp lệ. Nếu chương trình đang hoạt động ở chế độ tương tác, `ReportErrorMessage()` hiển thị thông báo tới người dùng; nếu ở chế độ dòng lệnh, routine sẽ ghi thông báo vào file thông điệp. Sau khi xuất thông báo, `ReportErrorMessage()` trả về giá trị trạng thái, chỉ ra thành công hay thất bại.

Phần còn lại của chương sẽ sử dụng routine này làm ví dụ minh họa và mô tả cách thiết kế routine.

Xác định vấn đề và đo lường yêu cầu

Xem Chương 3, "Đo hai lần, cắt một lần: Các điều kiện tiên quyết ở mức thượng nguồn (Upstream Prerequisites)," và Chương 4, "Các quyết định then chốt về xây dựng (Key Construction Decisions)" để đảm bảo rằng routine thực sự được gọi theo yêu cầu dự án, ít nhất là gián tiếp.

Xác định vấn đề mà routine sẽ giải quyết

Hãy trình bày vấn đề mà routine sẽ giải quyết với đủ chi tiết để có thể xây dựng routine đó. Nếu thiết kế cấp cao (high-level design) đã đủ chi tiết thì bước này có thể đã được thực hiện. Thiết kế cấp cao nên thể hiện tối thiểu các nội dung sau:

- **Thông tin mà routine sẽ che giấu**
- **Dữ liệu đầu vào (inputs) cho routine**
- **Dữ liệu đầu ra (outputs) từ routine**
- **Điều kiện tiên quyết (preconditions) đảm bảo đúng trước khi gọi routine**
  (ví dụ: giá trị đầu vào trong phạm vi xác định, stream đã được khởi tạo, file đã mở hoặc đóng, buffer đã đầy hoặc đã làm sạch, v.v.)
- **Điều kiện hậu mãn (postconditions) mà routine đảm bảo đúng trước khi trả quyền điều khiển về cho routine gọi nó**
  (ví dụ: giá trị đầu ra trong phạm vi xác định, stream đã được khởi tạo, file đã mở hoặc đóng, buffer đã đầy hoặc đã làm sạch, v.v.)

  Tham khảo chéo: Để biết chi tiết về điều kiện tiên quyết (preconditions) và điều kiện hậu mãn (postconditions), xem mục "Sử dụng assert để mô tả và kiểm tra các điều kiện trước và sau" (Use assertions to document and verify preconditions and postconditions) tại Mục 8.2.

Ví dụ về xử lý các yếu tố trên trong ReportErrorMessage():

- Routine này che giấu hai thông tin: văn bản thông báo lỗi (error message text) và phương thức xử lý hiện tại (interactive hay command line).

- Không có điều kiện tiên quyết đảm bảo cho routine.
- Đầu vào của routine là mã lỗi (error code).
- Hai loại đầu ra được yêu cầu: thứ nhất là thông báo lỗi, thứ hai là trạng thái (status) mà `ReportErrorMessage()` trả về cho routine gọi nó.
- Routine đảm bảo giá trị trạng thái (status value) sẽ là Success hoặc Failure.

Đặt tên cho routine

> *Tham khảo chéo*: Để biết chi tiết về cách đặt tên routine, xem Mục 7.3, "Tên routine tốt" (Good Routine Names).

Việc đặt tên routine có thể tưởng như đơn giản, nhưng một cái tên rõ ràng, không mơ hồ là dấu hiệu của một chương trình xuất sắc và đây là điều không dễ thực hiện. Nếu gặp khó khăn khi đặt tên, đó thường là do mục đích của routine chưa rõ ràng. Một tên gọi mơ hồ giống như một chính trị gia trên đường vận động tranh cử – nghe có vẻ như đang nói điều gì đó, nhưng khi kiểm tra kỹ, chẳng rõ ràng ý nghĩa. Nếu có thể làm tên gọi trở nên rõ ràng hơn, hãy thực hiện điều đó. Nếu tên gọi mơ hồ xuất phát từ thiết kế chung chung, bạn nên xem đó là cảnh báo và cải thiện lại thiết kế.

Trong ví dụ này, `ReportErrorMessage()` là tên không gây nhầm lẫn và là một tên tốt.

Quyết định cách kiểm thử routine

> *Đọc thêm*: Đối với một phương pháp xây dựng chú trọng viết case kiểm thử trước, xem "Test-Driven Development: By Example" (Beck 2003).

Khi viết routine, hãy cân nhắc cách kiểm thử routine đó. Điều này hữu ích cho cả lập trình viên khi kiểm thử đơn vị (unit testing) và người kiểm thử độc lập. Trong ví dụ này, do dữ liệu đầu vào đơn giản, bạn có thể kiểm thử `ReportErrorMessage()` với tất cả mã lỗi hợp lệ và nhiều mã lỗi không hợp lệ.

Tìm hiểu các chức năng sẵn có trong thư viện chuẩn

Phương pháp quan trọng nhất để nâng cao cả chất lượng code lẫn năng suất là tái sử dụng code tốt có sẵn. Nếu nhận thấy routine bạn định thiết kế quá phức tạp, hãy tự hỏi liệu một phần hoặc toàn bộ chức năng bạn cần đã có sẵn trong thư viện của ngôn ngữ, nền tảng hoặc các công cụ mà bạn đang sử dụng hay chưa. Hãy kiểm tra cả thư viện code nội bộ của công ty. Nhiều thuật toán đã được phát minh, kiểm thử, thảo luận, đánh giá và cải thiện từ lâu. Thay vì tự

phát minh lại bánh xe, hãy dành chút thời gian rà soát các thư viện sẵn có để đảm bảo bạn không làm việc dư thừa.

Suy nghĩ về xử lý lỗi

Hãy cân nhắc kỹ mọi vấn đề có thể xảy ra trong routine: giá trị đầu vào xấu, giá trị không hợp lệ được trả về từ routine khác, v.v. Routine có thể xử lý lỗi theo nhiều cách khác nhau, bạn cần lựa chọn phương pháp phù hợp. Nếu kiến trúc hệ thống đã quy định chiến lược xử lý lỗi, bạn chỉ cần tuân theo chiến lược đó. Ngược lại, hãy chọn phương pháp phù hợp với routine cụ thể đó.

Suy nghĩ về hiệu suất (efficiency)

Tùy vào từng trường hợp, có hai hướng để giải quyết vấn đề hiệu suất:

1. **Trong đa số hệ thống**: hiệu suất không phải là yếu tố quan trọng nhất. Hãy đảm bảo giao diện routine trừu tượng tốt, code dễ đọc, để khi cần tối ưu sau này có thể thực hiện dễ dàng nhờ tính đóng gói (encapsulation). Bạn có thể thay thế một routine cũ thực hiện chậm hoặc tiêu tốn tài nguyên bằng một thuật toán tốt hơn hoặc bằng cài đặt ở ngôn ngữ cấp thấp, và không ảnh hưởng đến routine khác.
2. **Trong một số ít hệ thống**: hiệu suất là yếu tố quan trọng sống còn (ví dụ: kết nối cơ sở dữ liệu hạn chế, bộ nhớ hạn chế, số lượng handle hạn chế, thời gian đáp ứng ngắn, hoặc tài nguyên khan hiếm khác). Kiến trúc nên chỉ rõ số lượng tài nguyên mỗi routine (hoặc class) được phép sử dụng và tốc độ thực hiện mong muốn.

Hãy thiết kế routine sao cho đáp ứng yêu cầu về tài nguyên và tốc độ. Nếu một trong hai yếu tố này quan trọng hơn, hãy cân nhắc đánh đổi tài nguyên lấy tốc độ hoặc ngược lại. Trong giai đoạn xây dựng ban đầu, chỉ cần tối ưu vừa đủ sao cho routine đạt được mục tiêu về tài nguyên và tốc độ.

> Ngoài các phương án kể trên, thông thường không nên quá tập trung tối ưu hoá ở cấp độ routine riêng lẻ. Những cải thiện lớn thường đến từ tối ưu thiết kế cấp cao, chứ không phải routine riêng lẻ. Thường chỉ thực hiện tối ưu nhỏ lẻ (micro-optimization) khi thiết kế cấp cao không đáp ứng được yêu cầu hiệu năng và chỉ khi hệ thống đã hoàn thành mới xác định điều này. Đừng tốn thời gian cho các cải thiện vụn vặt khi chưa cần thiết.

Tìm hiểu thuật toán (algorithm) và kiểu dữ liệu (data type)

Nếu chức năng của routine chưa có trong thư viện sẵn có, hãy kiểm tra trong các sách về thuật toán. Trước khi bắt đầu viết code phức tạp từ đầu, hãy xem thử có thuật toán nền tảng nào có thể áp dụng không. Nếu dùng thuật toán có sẵn, đảm bảo bạn điều chỉnh nó đúng cho ngôn ngữ lập trình đang sử dụng.

Viết mã giả (pseudocode)

Quá trình chuẩn bị ở các bước trên chủ yếu giúp bạn có nhận thức tốt về routine trước khi bắt tay vào viết.

> *Tham khảo chéo*: Giả định rằng bạn đang sử dụng phương pháp thiết kế tốt để tạo ra phiên bản mã giả (pseudocode) của routine. Để biết thêm về thiết kế, xem Chương 5, "Thiết kế trong quá trình xây dựng" (Design in Construction).

Sau các bước chuẩn bị, bạn có thể bắt đầu viết routine bằng mã giả ở mức cao. Hãy sử dụng trình biên tập lập trình hoặc môi trường tích hợp (IDE) để viết mã giả, vì mã giả sẽ sớm trở thành nền tảng cho code thực sự.

Bắt đầu từ tổng quát rồi chuyển dần tới chi tiết. Phần tổng quát nhất là phần chú thích đầu (header comment) nêu rõ mục đích của routine. Trước tiên, hãy viết mô tả ngắn gọn về routine – điều này cũng giúp bạn làm rõ ý tưởng và mục đích của routine. Nếu gặp khó khăn trong việc viết chú thích tổng quát, bạn cần làm rõ hơn vai trò của routine trong chương trình.

> Nói chung, nếu khó tóm tắt vai trò của routine, bạn nên giả định rằng có điều gì đó chưa đúng trong thiết kế.

Ví dụ chú thích đầu cho routine

> Routine này xuất ra thông điệp lỗi dựa trên mã lỗi (error code) do routine gọi nó truyền vào. Cách xuất thông điệp tùy thuộc vào trạng thái xử lý hiện tại, và routine này tự xác định trạng thái đó. Routine trả về giá trị biểu thị thành công hoặc thất bại.

Sau khi viết chú thích tổng quát, hãy bổ sung mã giả ở mức cao cho routine.

Ví dụ mã giả cho routine

```
Routine này xuất ra thông điệp lỗi dựa trên mã lỗi do routine gọi truyền vào.
Cách xuất thông điệp phụ thuộc vào trạng thái xử lý hiện tại, routine sẽ tự xác định trạng t
Routine trả về giá trị thể hiện thành công hoặc thất bại.

Đặt giá trị trạng thái (status) mặc định là "fail"
Tra cứu thông điệp dựa trên mã lỗi
Nếu mã lỗi hợp lệ
```

```
    Nếu đang xử lý tương tác (interactive), hiển thị thông điệp lỗi trên giao diện và xác nh
    Nếu đang xử lý dòng lệnh (command line), ghi lại thông điệp lỗi lên dòng lệnh và xác nhậ
Nếu mã lỗi không hợp lệ, thông báo cho người dùng biết có lỗi nội bộ
Trả về thông tin trạng thái
```

Một lần nữa, lưu ý rằng mã giả được viết ở mức khá cao, không phải là ngôn ngữ lập trình cụ thể mà dùng tiếng Anh chính xác để biểu đạt yêu cầu của routine.

Suy nghĩ về dữ liệu

Bạn có thể thiết kế dữ liệu cho routine ở nhiều điểm khác nhau trong quá trình trên. Trong ví dụ này, dữ liệu đơn giản và xử lý dữ liệu không phải là phần trọng tâm. Tuy nhiên, nếu xử lý dữ liệu là chủ đạo, bạn nên xác định các dữ liệu quan trọng trước khi thiết kế logic cho routine. Định nghĩa rõ các kiểu dữ liệu then chốt sẽ rất hữu ích khi bạn xây dựng logic của routine.

Tham khảo: Để sử dụng biến hiệu quả, xem các Chương 10 đến 13.

Lùi lại và suy ngẫm

Chương 21: "Cộng tác Xây dựng" (Collaborative Construction)

Hãy suy nghĩ về cách bạn sẽ giải thích quá trình này cho người khác. Hãy nhờ ai đó xem xét hoặc lắng nghe bạn trình bày. Có thể bạn sẽ nghĩ rằng việc nhờ người khác xem xét 11 dòng mã giả (pseudocode) là điều ngớ ngẩn, nhưng bạn sẽ ngạc nhiên vì mã giả giúp làm rõ các giả định và sai sót ở cấp độ cao dễ dàng hơn so với mã lập trình thực tế. Bên cạnh đó, mọi người thường sẵn lòng xem xét vài dòng mã giả hơn là 35 dòng mã C++ hoặc Java.

9.3 Xây dựng các routine bằng PPP (Pseudocode Programming Process)

Hãy đảm bảo bạn hiểu rõ một cách dễ dàng và thoải mái về mục đích và cách thực hiện của routine. Nếu bạn chưa hiểu rõ về routine ở cấp độ mã giả, thì xác suất bạn hiểu tại cấp độ mã lập trình sẽ càng thấp. Và nếu bạn không hiểu, thì ai sẽ hiểu?

Tham khảo chéo

> Để biết thêm về lặp lại, hãy xem Mục 34.8, "Lặp đi lặp lại, nhiều lần liên tiếp" (Iterate, Repeatedly, Again and Again).

Hãy thử nghiệm ý tưởng trên mã giả, và giữ lại giải pháp tốt nhất (lặp lại). Hãy thử càng nhiều ý tưởng càng tốt trên mã giả trước khi bắt đầu lập trình thực

tế. Một khi bạn đã viết mã, bạn sẽ có xu hướng gắn bó về mặt cảm xúc với mã của mình, khiến việc loại bỏ một thiết kế kém hiệu quả và bắt đầu lại trở nên khó khăn hơn.

Ý tưởng chung là hãy lặp lại routine trên mã giả cho đến khi các câu lệnh mã giả đơn giản đủ để bạn có thể triển khai mã thực phía dưới mỗi câu lệnh đó, đồng thời giữ nguyên mã giả để làm tài liệu. Một phần mã giả ở lần thử đầu tiên của bạn có thể vẫn còn ở mức độ quá cao, yêu cầu bạn phải phân rã thêm. Hãy chắc chắn thực hiện việc đó. Nếu bạn chưa chắc chắn cách viết mã cho một phần nào đó, hãy tiếp tục làm việc với mã giả cho đến khi bạn thật sự chắc chắn. Tiếp tục tinh chỉnh và phân rã mã giả cho đến khi việc viết mã giả thay vì mã thực trở nên không còn cần thiết nữa.

Bắt đầu xây dựng routine

Khi bạn đã thiết kế routine, hãy tiến hành xây dựng nó. Bạn có thể thực hiện các bước xây dựng theo một trình tự gần như tiêu chuẩn, nhưng cũng có thể điều chỉnh linh hoạt khi cần thiết. Hình 9-3 trình bày các bước xây dựng một routine.

- **Bắt đầu với mã giả**
- **Viết khai báo routine**
- **Viết câu lệnh đầu và cuối, chuyển mã giả thành chú thích cấp cao**
- **Lặp lại nếu cần thiết**
- **Hoàn thiện mã phía dưới mỗi chú thích**
- **Kiểm tra mã**
- **Dọn dẹp các phần còn sót lại**
- **Hoàn thành**

Hình 9-3: Bạn sẽ thực hiện tất cả các bước này khi thiết kế routine, nhưng không nhất thiết phải theo trình tự cố định.

Ghi chú về khai báo routine

Hãy viết câu lệnh khai báo interface của routine—tuyên bố hàm trong C++, khai báo method trong Java, hoặc khai báo function/sub procedure trong Microsoft Visual Basic, hoặc theo các yêu cầu của ngôn ngữ sử dụng. Chuyển phần chú thích đầu (header comment) gốc thành chú thích trong ngôn ngữ lập trình, đặt chúng phía trên mã giả đã viết. Dưới đây là ví dụ về khai báo interface và phần chú thích đầu routine trong C++:

Ví dụ C++ về interface của routine và phần đầu chú thích

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
```

```
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
)
set the default status to "fail"
look up the message based on the error code
if the error code is valid
    if doing interactive processing, display the error message
        interactively and declare success
    if doing command line processing, log the error message to the
        command line and declare success
if the error code isn't valid, notify the user that an
    internal error has been detected
return status information
```

Ở giai đoạn này, đây là thời điểm thích hợp để ghi chú về các giả định interface. Trong trường hợp này, biến interface `error` khá rõ ràng và đã được gán kiểu dữ liệu cụ thể, nên không cần tài liệu bổ sung.

Chuyển mã giả thành chú thích cấp cao

Tiếp tục viết câu lệnh đầu và cuối: `{` và `}` trong C++. Sau đó, chuyển mã giả thành chú thích. Ví dụ minh họa như sau:

Ví dụ C++: Viết câu lệnh đầu và cuối quanh mã giả

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    // look up the message based on the error code
    // if the error code is valid
        // if doing interactive processing, display the error message
            // interactively and declare success
        // if doing command line processing, log the error message to the
            // command line and declare success
    // if the error code isn't valid, notify the user that an
        // internal error has been detected
    // return status information
}
```

Ở thời điểm này, bạn đã có thể cảm nhận được cách hoạt động của routine,

ngay cả khi chưa có dòng mã thực nào. Khi chuyển đổi mã giả sang mã lập trình nên là một quá trình tự nhiên, cơ học và dễ dàng. Nếu chưa đạt được cảm giác này, hãy tiếp tục thiết kế trên mã giả cho tới khi bạn cảm thấy bản thiết kế đã vững chắc.

Hoàn thiện mã phía dưới mỗi chú thích

> Đây là trường hợp phù hợp với phép ẩn dụ viết văn—trong phạm vi nhỏ. Bạn viết đề cương, sau đó phát triển từng đoạn chi tiết dựa trên các ý chính; tương tự, mỗi chú thích từ mã giả mô tả một khối hoặc đoạn mã. Độ dài của từng khối mã cũng tương tự như độ dài các đoạn văn, phụ thuộc vào nội dung và mức độ tập trung của ý tưởng.

Trong ví dụ sau, hai chú thích mã giả đầu tiên được hiện thực hóa thành hai dòng mã:

Ví dụ C++: Chuyển các chú thích mã giả thành code

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up the message based on the error code
    Message errorMessage = LookupErrorMessage(errorToReport);

    // if the error code is valid
    // if doing interactive processing, display the error message
        // interactively and declare success
    // if doing command line processing, log the error message to the
        // command line and declare success
    // if the error code isn't valid, notify the user that an
        // internal error has been detected
    // return status information
}
```

Ở bước này, biến `errorMessage` được sử dụng nên bạn cần khai báo nó. Nếu bạn chỉ viết chú thích sau khi mã đã hoàn thành thì hai dòng chú thích cho hai dòng mã có thể là dư thừa. Tuy nhiên, trong phương pháp này, điều quan trọng là nội dung ý nghĩa của từng chú thích, không phải số lượng dòng. Các

chú thích giúp giải thích rõ ý định của code, vì vậy hãy giữ chúng lại.

Bạn cần hoàn thiện mã dưới mỗi chú thích còn lại.

Ví dụ C++: Một routine hoàn chỉnh theo PPP

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine. The way it outputs the message
   depends on the current processing state, which it retrieves
   on its own. It returns a value indicating success or failure
*/
Status ReportErrorMessage(
    ErrorCode errorToReport
) {
    // set the default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up the message based on the error code
    Message errorMessage = LookupErrorMessage(errorToReport);

    // if the error code is valid
    if (errorMessage.ValidCode()) {
        // determine the processing method
        ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

        // if doing interactive processing, display the error message
        // interactively and declare success
        if (errorProcessingMethod == ProcessingMethod_Interactive) {
            DisplayInteractiveMessage(errorMessage.Text());
            errorMessageStatus = Status_Success;
        }
        // if doing command line processing, log the error message to the
        // command line and declare success
        else if (errorProcessingMethod == ProcessingMethod_CommandLine) {
            CommandLine messageLog;
            if (messageLog.Status() == CommandLineStatus_Ok) {
                messageLog.AddToMessageQueue(errorMessage.Text());
                messageLog.FlushMessageQueue();
                errorMessageStatus = Status_Success;
            }
        }
        else {
            // can't do anything because the routine is already error processing
        }
    }
    else {
```

```
        // can't do anything because the routine is already error processing
    }
    // if the error code isn't valid, notify the user that an
    // internal error has been detected
    else {
        DisplayInteractiveMessage(
            "Internal Error: Invalid error code in ReportErrorMessage()"
        );
    }
    // return status information
    return errorMessageStatus;
}
```

Mỗi chú thích dẫn tới một hoặc nhiều dòng mã, và mỗi khối mã là một ý tưởng hoàn chỉnh dựa trên chú thích đó. Việc giữ lại các chú thích giúp giải thích rõ ràng ở mức độ tổng quát hơn cho code bên dưới.

Mỗi chú thích nên mở rộng thành khoảng 2 đến 10 dòng mã

Thông thường, mỗi chú thích nên mở rộng thành khoảng từ 2 đến 10 dòng code. (*Vì ví dụ này chỉ mang tính minh họa, việc mở rộng mã code ở đây thấp hơn so với thực tế mà bạn sẽ thường gặp.*)

Hãy xem lại phần đặc tả (specification) ở trang 000 và mã giả (pseudocode) ban đầu ở trang 000. Đặc tả ban đầu gồm năm câu đã được mở rộng thành 15 dòng mã giả (phụ thuộc vào cách bạn đếm dòng), và sau đó lại được phát triển thành một thủ tục có độ dài gần một trang. Mặc dù đặc tả khá chi tiết, việc tạo ra thuật toán đòi hỏi phải thiết kế đáng kể trong cả mã giả lẫn code thực tế. Thiết kế ở mức thấp như vậy là một trong những lý do khiến "việc lập trình" không phải là một nhiệm vụ tầm thường, và cũng vì vậy mà chủ đề của cuốn sách này trở nên quan trọng.

Kiểm tra xem mã có cần tiếp tục phân tách hay không

Trong một số trường hợp, bạn sẽ thấy lượng code phát sinh dưới một trong các dòng mã giả ban đầu là khá lớn. Trong trường hợp như vậy, bạn nên cân nhắc một trong hai hướng xử lý sau:

- **Tách mã dưới chú thích thành một routine (thủ tục) mới**
  Nếu bạn thấy một dòng mã giả mở rộng thành nhiều code hơn mong đợi, hãy tách đoạn code đó thành một routine riêng. Viết phần code gọi đến routine vừa tách, bao gồm tên routine. Nếu bạn đã áp dụng tốt Quy trình lập trình bằng mã giả (PPP - Pseudocode Programming Process), tên routine mới thường sẽ phát sinh tự nhiên từ mã giả. Khi hoàn thiện routine ban đầu, bạn có thể tiếp tục áp dụng PPP cho routine mới này.

- **Áp dụng PPP một cách đệ quy**
  Thay vì viết vài chục dòng code phía dưới một dòng mã giả, hãy dành thời gian phân rã dòng mã giả đó thành nhiều dòng mã giả nhỏ hơn. Sau đó, tiếp tục viết code cho từng dòng mã giả mới này.

  Tham khảo:
  Để tìm hiểu kỹ hơn về tái cấu trúc (refactoring), xem Chương 24, "Refactoring".

Kiểm tra mã

Sau khi thiết kế và triển khai routine, bước lớn thứ ba trong quá trình xây dựng là kiểm tra xem những gì bạn tạo ra có chính xác không. Bất kỳ lỗi nào bị bỏ sót ở giai đoạn này đều sẽ chỉ được phát hiện khi kiểm thử sau này, và chi phí phát hiện cùng sửa chữa sẽ tăng cao hơn nhiều. Vì thế, bạn nên cố gắng tìm ra mọi lỗi ngay ở giai đoạn này.

> **Tham khảo:**
> Để biết chi tiết về kiểm tra lỗi trong kiến trúc và yêu cầu, xem Chương 3, "Measure Twice, Cut Once: Upstream Prerequisites."

Một số lỗi chỉ xuất hiện khi routine đã được lập trình đầy đủ vì nhiều lý do: lỗi ở mã giả có thể trở nên dễ thấy hơn khi triển khai chi tiết; một thiết kế trông có vẻ tinh tế trên mã giả lại có thể vụng về khi hiện thực hóa; làm việc với hiện thực chi tiết có thể bộc lộ các lỗi từ kiến trúc, thiết kế cấp cao, hoặc yêu cầu; cuối cùng là các lỗi coding thông thường—không ai hoàn hảo cả! Vì vậy, hãy xem lại mã (code) trước khi chuyển sang giai đoạn tiếp theo.

Kiểm tra tinh thần đối với routine

Bước kiểm tra hình thức đầu tiên đối với routine là kiểm tra tinh thần (mental checking). Các bước dọn dẹp và kiểm tra không chính thức đã nhắc đến trước đó đều là các loại kiểm tra tinh thần. Một loại khác là tự thực thi từng đường đi (path) trong đầu. Việc kiểm tra này khá khó, đó cũng là lý do nên giữ routine nhỏ gọn. Hãy đảm bảo bạn kiểm tra cả các đường đi chính (nominal path), các điểm kết thúc, và mọi tình huống ngoại lệ. Thực hiện việc này cả một mình ("desk checking") và cùng đồng nghiệp ("peer review", "walk-through", hoặc "inspection" tùy cách thức tiến hành).

> Một trong những khác biệt lớn nhất giữa những người tự học lập trình (hobbyists) và lập trình viên chuyên nghiệp là việc chuyển từ niềm tin cảm tính (superstition) sang sự hiểu biết thực sự. Trong ngữ cảnh này, "superstition" (niềm tin cảm tính) không ám chỉ những chương trình kỳ dị, mà là việc thay thế sự hiểu biết bằng cảm xúc đối với code. Nếu bạn thường xuyên nghi ngờ compiler (bộ biên dịch) hoặc phần cứng gây ra lỗi, bạn vẫn còn ở mức độ "superstition". Một nghiên cứu nhiều năm trước cho thấy chỉ khoảng 5% lỗi đến từ phần

> cứng, compiler hoặc hệ điều hành (Ostrand và Weyuker, 1984); ngày nay, con số này còn thấp hơn nữa. Lập trình viên chuyên nghiệp luôn nghi ngờ chính công việc của họ đầu tiên, vì họ biết 95% lỗi đến từ chính lập trình viên. Hãy hiểu vai trò của từng dòng code và lý do tồn tại của nó. Không có gì "đúng" chỉ vì nó đang chạy. Nếu bạn không biết tại sao nó chạy được, có lẽ thực ra nó chưa chạy đúng—chỉ là bạn chưa nhận ra mà thôi.

Tóm lại: Một routine chạy được là chưa đủ. Nếu bạn chưa hiểu rõ vì sao nó hoạt động tốt, hãy nghiên cứu, thảo luận và thử nghiệm các thiết kế thay thế cho đến khi bạn nắm chắc nguyên tắc vận hành.

Biên dịch routine

Sau khi đã review routine, hãy biên dịch (compile) nó. Có thể bạn sẽ cho là chờ tới lúc này mới biên dịch thì hơi không hiệu quả, bởi code có thể đã hoàn thiện cách đây vài trang. Thật ra, bạn có thể tiết kiệm được một chút thời gian nếu biên dịch sớm để máy kiểm tra các biến chưa khai báo, xung đột tên, v.v.

Bạn sẽ thu được nhiều lợi ích nếu trì hoãn biên dịch cho đến các bước sau cùng. Nguyên nhân chủ yếu là sau lần biên dịch đầu tiên, một chiếc "đồng hồ bấm giờ" bên trong bạn sẽ bắt đầu đếm ngược. Khi đó, bạn sẽ có xu hướng "Chỉ biên dịch thêm lần nữa là xong", dẫn đến sự hấp tấp, thay đổi vội vàng và phát sinh thêm nhiều lỗi cũng như tốn nhiều thời gian hơn về lâu về dài. Hãy tránh vội vàng kết thúc bằng cách chỉ biên dịch khi bạn đã thật sự tin tưởng routine đã đúng.

Điểm mấu chốt của cuốn sách này là giúp bạn vượt ra khỏi vòng luẩn quẩn "nối ghép vội vàng rồi chạy thử xem có được không". Việc biên dịch quá sớm khi bạn chưa dám chắc đầu ra hợp lý là dấu hiệu của tư duy hacker chưa chuyên nghiệp. Nếu bạn đã thoát khỏi vòng lặp "hacking, compiling, and fixing", hãy biên dịch tại thời điểm phù hợp, nhưng hãy luôn ý thức sức ép tự nhiên hướng tới việc "hacking, compiling and fixing" một cách thiếu bài bản.

Một số hướng dẫn để tận dụng tối đa quá trình biên dịch routine:

- Đặt mức cảnh báo (warning level) của compiler lên mức nghiêm ngặt nhất. Bạn có thể phát hiện được rất nhiều lỗi tinh vi chỉ bằng cách để compiler tìm chúng.
- Sử dụng các công cụ xác thực (validator). Ở các ngôn ngữ như C, việc kiểm tra của compiler có thể được bổ trợ bởi các công cụ như lint. Ngay cả với các loại code không biên dịch như HTML và JavaScript, cũng nên sử dụng công cụ xác thực (validator).
- Loại bỏ hết các lỗi và cảnh báo. Hãy chú ý đến thông điệp cảnh báo từ compiler. Nếu có nhiều cảnh báo, đó thường là dấu hiệu chất lượng code thấp; bạn nên cố gắng hiểu lý do của từng cảnh báo. Thực tế, những cảnh báo hay gặp sẽ dẫn tới hai hệ quả: bạn bỏ qua chúng và vô tình bỏ sót

các cảnh báo nguy hiểm hơn, hoặc chúng trở nên phiền phức. Giải pháp an toàn và hiệu quả là viết lại mã để khắc phục tận gốc, loại bỏ hoàn toàn các cảnh báo.

Thực thi từng bước với trình gỡ lỗi (debugger)

Khi routine đã biên dịch thành công, hãy đưa nó vào debugger và thực thi từng bước một. Đảm bảo rằng từng dòng code chạy đúng như mong đợi. Chỉ với thao tác đơn giản này, bạn có thể phát hiện ra rất nhiều lỗi.

> **Tham khảo:**
> Để biết chi tiết, xem Chương 22, "Developer Testing". Xem thêm mục "Building Scaffolding to Test Individual Classes" tại Mục 22.5.

Kiểm thử routine

Kiểm thử routine bằng các bộ test case (trường hợp kiểm thử) đã được lên kế hoạch hoặc xây dựng trong quá trình phát triển routine. Bạn có thể cần phát triển các đoạn mã bổ trợ (scaffolding) để hỗ trợ quá trình kiểm thử—đây là những đoạn mã dùng trong quá trình kiểm thử routine nhưng không được đưa vào sản phẩm cuối cùng. Scaffolding có thể là các routine kiểm thử gọi routine của bạn với dữ liệu kiểm thử (test harness) hoặc các stub được gọi bởi routine của bạn.

> **Tham khảo:**
> Để biết chi tiết, xem Chương 23, "Debugging".

Loại bỏ lỗi khỏi routine

Sau khi phát hiện ra lỗi, bạn cần loại bỏ chúng. Nếu routine đang phát triển xuất hiện nhiều lỗi ở giai đoạn này, khả năng cao là nó sẽ tiếp tục mắc lỗi trong tương lai. Nếu routine quá nhiều lỗi, hãy bắt đầu lại từ đầu; đừng "vá víu" tạm bợ. Việc "hack" chỉ phản ánh sự hiểu biết chưa đầy đủ và sẽ đảm bảo tiếp diễn lỗi trong cả hiện tại lẫn tương lai. Thiết kế lại hoàn toàn một routine nhiều lỗi thực sự đáng giá. Ít điều gì thỏa mãn hơn việc viết lại một routine từng gây rắc rối, và sau đó không còn thấy lỗi nào nữa.

Dọn Dẹp Sau Cùng (Clean Up Leftovers)

Khi bạn đã kiểm tra code để phát hiện vấn đề, hãy kiểm tra nó thêm một lần nữa theo các tiêu chí chất lượng tổng quát đã trình bày trong toàn cuốn sách.

Bạn có thể thực hiện một số bước dọn dẹp để đảm bảo routine đạt đến những tiêu chuẩn mong muốn:

- **Kiểm tra interface của routine:** Đảm bảo tất cả dữ liệu vào/ra đều đã được xét đến và tất cả các tham số (parameter) đều được sử dụng. Để biết thêm chi tiết, xem Mục 7.5, "How to Use Routine Parameters".
- **Kiểm tra chất lượng thiết kế tổng thể:** Đảm bảo routine chỉ thực hiện một việc và thực hiện tốt việc đó, liên kết lỏng lẻo (loose coupling) với các routine khác, và có thiết kế phòng thủ (defensive design). Tham khảo Chương 7, "High-Quality Routines".
- **Kiểm tra biến của routine:** Xem lại các tên biến không chính xác, đối tượng không dùng đến, biến chưa khai báo, đối tượng khởi tạo không đúng, v.v. Xem chi tiết trong các chương 10 đến 13, về sử dụng biến.
- **Kiểm tra các lệnh (statement) và logic:** Tìm lỗi "lệch một đơn vị" (off-by-one), vòng lặp vô hạn, lồng ghép không hợp lý, rò rỉ tài nguyên, v.v. Tham khảo các chương từ 14 đến 19.
- **Kiểm tra bố cục (layout) routine:** Đảm bảo bạn sử dụng khoảng trắng để làm rõ cấu trúc logic của routine, biểu thức, và danh sách tham số. Xem thêm Chương 31, "Layout and Style".
- **Kiểm tra tài liệu chú thích:** Đảm bảo phần mã giả đã chuyển thành chú thích trong code vẫn còn chính xác; kiểm tra mô tả thuật toán, tài liệu về giả định giao diện (interface assumption) và các phụ thuộc không rõ ràng, lý do cho các thói quen lập trình không rõ ràng, v.v. Tham khảo Chương 32, "Self-Documenting Code".
- **Loại bỏ các chú thích thừa.**

Lặp lại các bước khi cần thiết

Nếu chất lượng của routine (thủ tục) kém, hãy quay lại bước viết pseudocode (mã giả). **Lập trình chất lượng cao là một quá trình lặp**, do đó, đừng ngần ngại thực hiện lại các hoạt động xây dựng.

9.4 Các phương pháp thay thế PPP

Theo quan điểm cá nhân, phương pháp PPP (Pseudocode Programming Process – Quy trình Lập trình bằng Mã giả) là phương pháp tốt nhất để tạo ra các class (lớp) và routine (thủ tục/chức năng). Tuy nhiên, dưới đây là một số phương pháp khác được các chuyên gia khuyến nghị. Bạn có thể sử dụng các phương pháp này như những lựa chọn thay thế, hoặc bổ sung cho PPP.

Test-first development (Phát triển hướng kiểm thử trước)

Test-first development là một phong cách phát triển phổ biến, trong đó các test case (trường hợp kiểm thử) được viết **trước** khi viết bất kỳ đoạn code nào. Cách tiếp cận này được mô tả chi tiết hơn trong mục "Test First or Test Last?"

tại Phần 22.2. Một cuốn sách hay về *test-first programming* là **Test-Driven Development: By Example** của Kent Beck (Beck 2003).

Refactoring (Tái cấu trúc mã nguồn)

Refactoring là một phương pháp phát triển, trong đó bạn cải tiến code thông qua loạt các biến đổi giữ nguyên ý nghĩa (semantic preserving transformations). Lập trình viên sử dụng các mẫu nhận diện code kém hay còn gọi là "smells" (mùi mã nguồn yếu kém), để xác định các phần code cần cải thiện. Chương 24, "Refactoring", mô tả chi tiết phương pháp này. Một cuốn sách tốt về chủ đề này là **Refactoring: Improving the Design of Existing Code** của Martin Fowler (Fowler 1999).

Design by contract (Thiết kế theo hợp đồng)

Design by contract là một phương pháp phát triển, trong đó mỗi routine được xem xét theo các điều kiện tiên quyết (preconditions) và điều kiện hậu kiểm (postconditions). Cách tiếp cận này được trình bày trong mục "Use assertions to document and verify preconditions and postconditions" tại Phần 8.2. Nguồn thông tin tốt nhất về *design by contract* là cuốn **Object-Oriented Software Construction** của Bertrand Meyer (Meyer 1997).

Hacking? ("Code chay" không có hệ thống)

Một số lập trình viên cố gắng "hack" để có code hoạt động thay vì sử dụng các phương pháp hệ thống như PPP. Nếu bạn từng rơi vào tình huống phải viết lại một routine do lâm vào ngõ cụt, đó chính là dấu hiệu cho thấy PPP có thể hữu ích hơn. Nếu bạn thường xuyên bị mất luồng suy nghĩ khi đang code một routine, đó cũng là một dấu hiệu cho thấy PPP sẽ mang lại lợi ích. Bạn đã bao giờ "quên" viết một phần của class hoặc routine chưa? Nếu sử dụng PPP, điều này hiếm khi xảy ra. Nếu bạn cảm thấy lúng túng không biết bắt đầu từ đâu trước màn hình máy tính, đó là dấu hiệu rõ ràng PPP sẽ giúp công việc lập trình của bạn dễ dàng hơn.

---

cc2e.com/0943 CHECKLIST: The Pseudocode Programming Process

> Mục đích của danh sách này là kiểm tra xem bạn có tuân thủ đúng các bước cần thiết để xây dựng một routine hay không. Đối với một checklist tập trung vào chất lượng của routine, tham khảo thêm mục "High-Quality Routines" ở Chương 7, trang 185.

Checklist:

- Bạn đã kiểm tra liệu các điều kiện tiên quyết đã được đáp ứng chưa?

- Bạn đã xác định vấn đề mà class sẽ giải quyết chưa?
- Thiết kế cấp cao có đủ rõ ràng để đặt tên hợp lý cho class và từng routine chưa?
- Bạn đã suy nghĩ về cách kiểm thử class và từng routine chưa?
- Bạn đã cân nhắc hiệu năng chủ yếu dưới góc độ giao diện ổn định và triển khai dễ đọc, hay chủ yếu dựa theo tiêu chí tài nguyên và tốc độ?
- Bạn đã kiểm tra các thư viện chuẩn và thư viện mã nguồn khác để tìm các routine hoặc component (thành phần) phù hợp chưa?
- Bạn đã tra cứu các sách tham khảo để tìm các thuật toán hữu ích chưa?