**MỘT SỐ KỸ THUẬT – PHONG CÁCH LẬP TRÌNH TỐT**

[1. Cách đặt tên cho biến hàm 2](#_Toc502675932)

[2. Phong cách viết mã nguồn 5](#_Toc502675933)

[3. Tối ưu sự thực thi mã nguồn 7](#_Toc502675934)



**MỞ ĐẦU**

Một chương trình nguồn được xem là tốt không chỉ được đánh giá thông qua thuật giải đúng và cấu trúc dữ liệu thích hợp. Mà còn phụ thuộc vào phong cách và kỹ thuật mã hoá (coding) của người viết chương trình.

Nếu một người lập trình viết một chương trình tuy thực hiện đúng yêu cầu đặt ra nhưng mã nguồn quá lộn xộn và phong cách lập trình cẩu thả, thì mã nguồn này sẽ gây khó khăn cho chính người lập trình!

Đôi khi người mới lập trình không quan tâm đến vấn đề này do ban đầu chỉ làm việc với chương trình nhỏ. Tuy nhiên, vấn đề phát sinh khi họ phải làm việc với dự án lớn và chương trình lúc này không còn đơn giản vài chục dòng lệnh nữa. Nếu không rèn luyện một phong cách và trang bị một số kỹ thuật lập trình tốt thì người lập trình đối mặt với nhiều khó khăn…

Trong chương đầu tiên xin giới thiệu một số kỹ thuật và phong cách lập trình cơ bản, ít nhiều giúp cho người học viết chương trình được tốt hơn.

1. Cách đặt tên cho biến hàm

Thông thường tùy theo ngôn ngữ và môi trường lập trình, người viết chương trình thường chọn cho mình một phong cách nhất quán trong việc đặt tên các định danh. Một số quy tắc cần quan tâm khi đặt tên như sau:

1. Tên của định danh phải thể hiện được ý nghĩa: thông thường các biến nguyên như i, j, k dùng làm biến lặp; x, y dùng làm biến lưu tọa độ…Còn những biến lưu trữ dữ liệu khác thì nên đặt gợi nhớ: biến đếm số lần dùng “*count*” hay So\_Luong, biến lưu trọng lượng “*weight*”, chiều cao “*height*”…Nếu đặt quá ngắn gọn như c cho biến đếm, hay w cho khối lượng thì sau này khi nhìn vào chương trình sẽ rất khó hiểu!
2. Tên phải xác định được kiểu dữ liệu lưu trữ: phong cách lập trình tốt là khi người đọc nhìn vào một biến nào đó thì xác định ngay được kiểu dữ liệu mà biến đó lưu trữ. Giả sử có biến đếm số lần thì ta có thể đặt *iCount*, trong đó *i* là kiểu của dữ liệu, *strContent* là kiểu chuỗi…Có nhiều cú pháp quy ước đặt tên biến, người lập trình có thể chọn cho mình một quy ước thích hợp. Có thể tham khảo một số quy ước trong phần 3 bên dưới.

1. Theo một quy ước cụ thể:
   1. Cú pháp Hungary: hình thức chung của cú pháp này là thêm tiền tố chứa kiểu dữ liệu vào tên biến. Bảng 1.1 bên dưới là một số tiền tố quy ước được nhiều lập trình viên sử dụng. Các công ty phần mềm thường có các quy ước về cách đặt tên biến cho đội ngũ lập trình viên. Tuy nhiên đa số các quy ước này đều dựa trên cú pháp Hungary.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tiền tố** | | **Kiểu dữ liệu** | **Minh họa** |
|  | |  |  |
| **b** | | boolean | bool bStop |
|  | |  |  |
| **c** | | char | char cLetterGenre |
|  | |  |  |
| **str/s** | | C++ string | string strFirstName |
|  | |  |  |
| **si** | | short integer | short siTables |
|  | |  |  |
| **i/n** | | integer | int iCars |
|  | |  | int nCars |
|  | |  |  |
| **li** | | long integer | long liStars |
|  | |  |  |
| **f** | | floating point | float fPercent |
|  | |  |  |
| **d** | | Double precision floating point | double dMiles |
|  | |  |  |
| **ld** | | long double precision floating | long double ldPI |
|  | | point |  |
|  | |  |  |
| **sz** | | Null terminated string | char szName[NAME\_LEN] |
|  | |  |  |
| **if** | | Input file stream | ifstream ifNameFile |
|  | |  |  |
| **is** | | Input stream | istream isInput |
|  | |  |  |
| **of** | | Output file stream | ofstream ofNameFile |
|  | |  |  |
| **os** | | Output stream | ostream osOut |
|  | |  |  |
| **S** | | Struct | struct sPoint {…} |
|  | |  |  |
| **C** | | Class | class CStudent {…} |
|  | |  |  |
| **w** | | Word | word wChar |
|  | |  |  |
|  | |  |  |
|  | | **u** | Unsigned.. |  |  |
|  | |  |  |  |  |
|  | | **m\_** | biến thành viên của hàm | class CStudent |  |
|  | |  |  | { |  |
|  | |  |  | private: |  |
|  | |  |  | string m\_strName; |  |
|  | |  |  | } |  |
|  | |  |  |  |  |
|  | | **g\_** | biến toàn cục | string g\_strBuff |  |
|  | |  |  |  |  |
|  | | **lp** | long pointer | LPCTSTR lpszClassName |  |
|  | |  |  |  |  |
|  | | **h** | handle trong windows | HINSTANCE hInstance |  |
|  | |  |  |  |  |

Bảng 1.1: Minh họa tiền tố của cú pháp Hungary.

Đối với những hằng thì tất cả các ký tự đều viết hoa

#define MAXSIZE 100

const int MAXLENGTH 200

Cách đặt tên cho hàm: hàm bắt đầu với ký tự đầu tiên là ký tự hoa và không có tiền tố. Tuy nhiên, điều này cũng không bắt buộc tuỳ theo ngôn ngữ lập trình. Nói chung là hàm có chức năng thực hiện một chức năng nào đó, cho nên chúng thường bắt đầu bằng động từ: get, set, do…

Cstring GetName(); // Microsoft VC++ standard

String setName(); // Sun Java standard

2. Phong cách viết mã nguồn

Sử dụng tab để canh lề chương trình: khi soạn thảo mã nguồn nên dùng tab với kích thước là 4 hay 8 để canh lề. Thói quen này giúp cho chương trình được rõ ràng và dễ quản lý.



|  |  |  |  |
| --- | --- | --- | --- |
|  | for (i = 0;i < N; i++) | for (i = 0; i < N; i++) |  |
|  | { | { |  |
|  | if (Check(i)) | if (Check(i)) |  |
|  | { | { |  |
|  | Action1(); | Action1(); |  |
|  | Action2(); | Action2(); |  |
|  | } | } |  |
|  | else | else |  |
|  | Action3(); | Action3(); |  |
|  | ActionMain(); | ActionMain(); |  |
|  | } | } |  |
|  |  |  |  |

Sử dụng khoảng trắng: chương trình sẽ dễ nhìn hơn.



|  |  |
| --- | --- |
| int count; | int count; |
| for(count=0;count<10;count++) | for (count = 0; count < 10; count++) |
| { | { |
| printf(“%d”,count\*count+count); | printf(“%d”, count \* count + count); |
| } | } |
|  |  |

Tránh viết nhiều lệnh trên cùng một dòng:



|  |  |
| --- | --- |
| if(a>5){b=a;a++;} | if (a > 5) |
|  | { |
|  | b=a; |
|  | a++; |
|  | } |

Định nghĩa các hằng số: một thói quen là người lập trình không định nghĩa những hằng số thường xuyên sử dụng. Dẫn đến những con số khó hiểu xuất hiện trong chương trình, một số tài liệu lập trình gọi những con số này là “*magic mumber*”.



|  |  |  |
| --- | --- | --- |
| … | #define MAX\_LEN | 100 |
| for(int i=0; i < 100; i ++) | #define MAX\_NUM | 100 |
| A[i] = Rand(100); | … |  |
| … | for(int i=0; i < MAX\_LEN; i++) | |
| k = InputNum(); | A[i] = Rand(MAX\_NUM); | |
| j=0; | … |  |
| while (A[j] != k && j < 100) | k = InputNum(); |  |
| j++; | j=0; |  |
| … | while (A[j] != k && j < MAX\_LEN) | |
|  | j++; |  |
|  | … |  |
|  |  |  |

Trong đoạn chương trình bên trái rất khó phân biệt giá trị 100 ở ba vị trí có mối quan hệ gì với nhau. Tuy nhiên, trong đoạn bên phải ta dễ dàng thấy được ý nghĩa của từng giá trị khi thay bằng định danh. Ngoài ra khi cần thay đổi giá trị của MAX\_LEN, MAX\_NUM thì chỉ cần thay một lần trong phần định nghĩa. Do đó đoạn chương trình B dễ nhìn hơn và dễ thay đổi hơn!

Viết chú thích cho chương trình: biến, hàm khi định nghĩa nên viết chú thích ý nghĩa và chức năng rõ ràng. Đôi khi các lệnh thực thi cũng cần có giải thích nếu chúng quá phức tạp.



int CheckFactor(int n)

{

/\*

* *nghĩa: kiểm tra xem 1 số có phải là nguyên tố hay không Tham số vào: n số cần kiểm tra*

*Tham số ra: giá trị trả về*

* 1. *không phải số nguyên tố*
  2. *là số nguyên tố*

\*/

….// phần thực hiện của hàm

}

Ví dụ chú thích cho biến

|  |  |  |
| --- | --- | --- |
| byte | Image; | // buffer ảnh |
| int | Rows, Cols; | // số dòng, số cột |
| int | r, c; | // dòng cột hiện hành |
| int | PixelCount; | // tổng số pixel |

Tuy nhiên không phải bất cứ lệnh nào cũng chú thích, việc chú thích tràn lan ngay cả với câu lệnh đơn giản cũng không có ý nghĩa gì. Đôi khi còn làm cho chương trình khó nhìn hơn!

Nên viết biểu thức điều kiện mang tính tự nhiên: biểu thức nên viết dưới dạng khẳng định, việc viết biểu thức dạng phủ định sẽ làm khó hiểu!



if ( !(iBlock < Block1 ) || !(iBlock >= Block2))

…

Mỗi biểu thức trong điều kiện được viết dưới dạng phủ định, ta nên viết lại dưới dạng khẳng định cho dễ hiểu hơn:

if ( (iBlock >= Block1 ) || (iBlock < Block2))

…

Dùng chính ngôn ngữ đó để tính kích thước của đối tượng: không nên dùng giá trị tường minh cho kích thước của dữ liệu. Khi cần lấy kích thước của biến int, ta có thể dùng **sizeof(int)** thay cho các giá trị 2 hay 4. Tương tự như vậy khi lấy kích thước của phần tử trong một mảng int ta dùng sizeof(array[0]) thay cho sizeof(int). Sau này khi mảng array có thay đổi kiểu dữ liệu thì cách viết sizeof(array[0]) cũng không ảnh hưởng.



3. Tối ưu sự thực thi mã nguồn

Mã nguồn nếu được viết tốt sẽ làm cho tốc độ chương trình cải thiện đáng kể. Có thể ngày nay năng lực xử lý của máy tính khá mạnh, do đó người lập trình không quan tâm đến việc tối ưu mã nguồn. Nhưng cũng không vì thế mà bỏ qua kỹ thuật này. Vậy thế nào là tối ưu mã nguồn? ở đây không đề cập đến giải thuật, vì chắc chắn giải thuật tốt thì sẽ cho chương trình tối ưu. Tuy nhiên, việc cài đặt cũng cần phải có kỹ thuật, nếu không thì chính khả năng cài đặt của lập trình viên làm hạn chế sự thực thi của thuật giải hay chương trình.

Mục đích của việc tối ưu mã nguồn là nâng cao tốc độ xử lý và hạn chế không gian bộ nhớ mà chương trình chiếm dụng. Thông thường có thể mâu thuẫn giữa tốc độ và không gian lưu trữ, do đó tuỳ theo điều kiện cụ thể mà người lập trình có thể lựa chọn thích hợp.

Trong phần dưới xin trình bày một số thủ thuật chọn lọc có thể giúp ích để hình thành nên phong cách lập trình tốt cho người đọc.

Thu gọn những biểu thức dùng nhiều lần: nếu một biểu thức tính toán được dùng nhiều lần thì chúng ta nên tính kết quả một lần rồi lưu vào một biến và dùng lại.



Ví dụ:

F = sqrt(dx\*dx+dy\*dy) + (sqrt(dx\*dx + dy\*dy)\*sqrt(dx\*dx)-sqrt(dy\*dy))…

Trong dãy biểu thức trên có sqrt(dx\*dx+dy\*dy), dx\*dx, dy\*dy được dùng nhiều chỗ, ta có thể tính trước bên ngoài và lưu vào biến tạm để dùng lại sau này. Hạn chế việc tính toán với cùng một biểu thức nhiều lần!

Đưa những biểu thức không phụ thuộc vòng lặp ra ngoài: trong một số vòng lặp ta có sử dụng biểu thức tính toán nhưng giá trị của biểu thức không phụ thuộc vào sự thay đổi của vòng lặp thì có thể đưa biểu thức này ra ngoài.



Ví dụ:

for(i =0; i < strlen(str); i++)

….

chuyển thành:

int n = strlen(str)

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

….

Thay thế một biểu thức bằng một biểu thức tương đương nhưng lợi về thực thi: một số chương trình xử lý ảnh đòi hỏi tốc độ cao, thì người lập trình có thể thay thế các phép nhân chia bằng phép dịch chuyển bit. Thay thế sử dụng chỉ mục trong mảng C/C++ bằng con trỏ…



Ví dụ: khi so sánh khoảng cách của hai điểm ta thường làm như sau

if (sqrt(dx1\*dx1+dy1\*dy1) < sqrt(dx2\*dx2+dy2\*dy2))

…

Thay bằng

if ((dx1\*dx1+dy1\*dy1) < (dx2\*dx2+dy2\*dy2))

...

Dùng số nguyên thay cho số thực: do việc xử lý số thực chậm hơn xử lý số nguyên nên ta có thể dùng số nguyên thay cho số thực có phần lẻ nhỏ.



Ví dụ: điểm trung bình của sinh viên là số thực ta có thể thay bằng số nguyên: DTB là 8.72 thì lưu theo số nguyên 872, khi xuất ra thì chia cho 100.

Loại bỏ vòng lặp: nếu thân vòng lặp đơn giản và số lần lặp cũng không nhiều, ta có thể làm cho đoạn chương trình hiệu quả hơn bằng cách bỏ vòng lặp.



Ví dụ:

for(i =0; i < 3; i++)

A[i] = B[i] + C[i];

Thay bằng

A[1] = B[1] + C[1];

A[2] = B[2] + C[2];

A[3] = B[3] + C[3];

Đoạn chương trình thay thế loại bỏ vòng lặp, tức là lệnh rẽ nhánh, lệnh rẽ nhánh làm chậm chương trình do ngắt luồng thực thi.

Nếu vòng lặp dài và cùng dạng biểu thức ta có thể cải tiến như ví dụ sau

for(i=0; i < 3\*n; i++)

A[i] = B[i] + C[i];

Thay bằng

for(i=0; i < 3\*n; i+=3)

{

A[i] = B[i] + C[i];

A[i+1] = B[i+1] + C[i+1];

A[i+2] = B[i+2] + C[i+2];

}

Ví dụ trên chỉ áp dụng khi chiều dài vòng lặp là bội số của bước nhảy!

Loại bỏ câu lệnh rẽ nhánh trong vòng lặp: xem ví dụ sau



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  |  | **Chương trình A** | |  | **Chương trình B** | |  |
|  |  | for i to 1000 do |  |  | if (w) then |  |  |
|  |  | { |  |  | for i to 1000 do |  |  |
|  |  | x[i] = x[i] + y[i]; |  |  | { |  |  |
|  |  | if (w) then y[i] = 0; |  |  | x[i] = x[i] + y[i]; |  |  |
|  |  | } |  |  | y[i] = 0; |  |  |
|  |  |  |  |  | } |  |  |
|  |  |  |  |  | else |  |  |
|  |  |  |  |  | for i to 1000 do |  |  |
|  |  |  |  |  | x[i] = x[i] + y[i]; | |  |
|  |  |  |  |  |  |  |  |

Trong chương trình A, mỗi lần lặp thì phải kiểm tra thêm điều kiện của w. Trong khi chương trình B thì ta kiểm tra giá trị của w trước khi vào vòng lặp. Do đó B có hai vòng lặp nhưng chỉ thực hiện một trong hai và chỉ kiểm tra giá trị w duy nhất 1 lần!

Thoát khỏi vòng lặp sớm nhất: một số trường hợp không cần phải lặp hết toàn bộ vòng lặp mà đã đạt được mục đích thì có thể thoát ra khỏi vòng lặp.



Ví dụ: chỉ cần xác định giá trị -99 có xuất hiện trong danh sách hay không ta có hai chương trình A và B minh họa như sau:

|  |  |
| --- | --- |
| **Chương trình A** | **Chương trình B** |
| found = FALSE; | found = FALSE; |
| for(i=0;i<10000;i++) | for(i=0; i<10000; i++) |
| { | { |
| if( list[i] == -99 ) | if( list[i] == -99 ) |
| { | { |
| found = TRUE; | found = TRUE; |
| } | **break**; |
| } | } |
| if( found ) printf("Yes, there is a -99."); | } |
|  | if( found ) printf("Yes, there is a -99."); |
|  |  |

Chương trình A khi tìm thấy thì vẫn cứ lặp cho đến hết, trong khi B thì sẽ thoát ngay. Rõ ràng khi đã tìm thấy thì không cần phải lặp tiếp, khi đó B sẽ tối ưu hơn!

Gom các vòng lặp: các vòng lặp cùng số lần lặp thì nên gom lại



Ví dụ:

for( int i=0; i<n; i++)

a[i]= 0;

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

b[i]= 0;

Viết lại:

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

a[i]= b[i]= 0;

Sử dụng phép shift thay cho nhân chia:



1. Shift trái 1 bit: nhân 2
2. Shift phải 1 bit: chia 2

|  |  |  |
| --- | --- | --- |
| Ví dụ: |  |  |
| a \*= 4 | ⇒ | a<<2 |
| b /=8 | ⇒ | b>>3 |
| a = 2\*(b+c) | ⇒ | a = (b+c)<<1 |

Sử dụng phép “&”: thay cho phép chia dư n, với n là 2i {2, 4, 8, 16, 32…}



Ví dụ:

m = n % 2

m = n % 8

m = n % 16

Lấy byte thấp:

* m = n & 1
* m = n & 7
* m = n & 15
* m = n & 0x1
* m = n & 0x7
* m = n & 0xF

m = n % 256 ⇒ m = n & 0xFF

Cải tiến tính toán cho biến cờ:



if (x >y)

flag =1;

else

flag =0;

Cải tiến thành:

flag = x>y;

Lưu tạm giá trị thường sử dụng: trong chương trình đôi khi một giá trị được tính toán một lần nhưng lại thường được sử dụng mà ít có thay đổi giá trị. Khi đó ta có thể dùng biến lưu trữ giá trị của biểu thức này, khi nào cần thì có thể sử dụng biến đó thay vì phải tính toán lại.



Ví dụ: đoạn chương trình giải phương trình bậc hai.

…

if ((b\*b)-4\*a\*c < 0)

printf(“Phuong trinh vo nghiem!”);

else if ((b\*b)-4\*a\*c == 0)

printf(“Phuong trinh co nghiem kep”);

…

else

{

x1= (-b + sqrt((b\*b)-4\*a\*c))/(2\*a);

x2= (-b - sqrt((b\*b)-4\*a\*c))/(2\*a);

…

}

Trong đoạn chương trình trên delta được tính lại 4 lần, ta có thể cải tiến chỉ tính duy nhất một lần!

delta = (b\*b)-4\*a\*c;

if ( delta < 0)

printf(“Phuong trinh vo nghiem!”);

else if (delta == 0)

printf(“Phuong trinh co nghiem kep”);

…

else

{

x1= (-b + sqrt(delta))/(2\*a);

x2= (-b - sqrt(delta))/(2\*a);

…

}

Tránh lãng phí bộ nhớ: bằng cách sử dụng kiểu dữ liệu nhỏ nhất có thể được để lưu trữ: không gian bộ nhớ hiện tại có thể không còn eo hẹp như trước, nhưng không vì thế mà người lập trình có thể tự do phung phí cấp cho chương trình. Việc sử dụng quá nhiều tài nguyên hơn mức đòi hỏi của chương trình là thói quen xấu mà người lập trình hay mắc phải. Hơn nữa tốc độ chương trình sẽ nhanh hơn khi sử dụng kiểu dữ liệu nhỏ hơn.



Khai báo biến cục bộ trong phạm vi gần nhất: đúng như tên gọi là biến cục bộ do đó khi sử dụng nên khai báo gần với điểm sử dụng nhất. Việc khai báo ở phạm vị rộng hơn chỉ làm lãng phí và khó kiểm soát.



Sử dụng macro: một số hàm đơn giản và thường sử dụng có thể chuyển thành macro để tăng tốc độ thực thi của chương trình. Do mỗi lần gọi hàm sẽ tốn chi phí cho việc gọi và trả về từ hàm.



Ví dụ:

int max(int a, int b)

{

return a>b? a: b;

}

Chuyển thành macro:

#define max(a, b) ((a)>(b)) ? (a) : (b)

Hàm hoán chuyển giá trị 2 số nguyên

void swap(int &a, int &b)

{

int t;

t = a;

a = b;

b = t;

}

Chuyển thành macro swap

#define swap(a, b) {int t = a; a = b; b = t;}

Giảm số lượng tham số truyền vào hàm: việc sử dụng hàm có quá nhiều tham số được truyền vào có thể làm ảnh hưởng đến ngăn xếp dành cho việc gọi hàm. Nhất là trường hợp tham số là kiểu dữ liệu cấu trúc. Sử dụng con trỏ hay tham chiếu trong trường hợp này để đơn giản hoá.



Ví dụ :

void Print(struct Student s)

{

printf(“%d”, s.StudentCode);

…

}

Thay bằng:

void Print(const struct Student \*s)

{

printf(“%d”, s->StudentCode);

…

}