Bộ môn Công nghệ phần mềm
Khoa Công nghệ thông tin
Trường Đại học Khoa học Tự nhiên

VCBB© 3.0

PP LT HƯỚNG ĐỐI TƯỢNG

ThS. Đặng Bình Phương
dbphuong@fit.hcmus.edu.vn

PHONG CÁCH LẬP TRÌNH

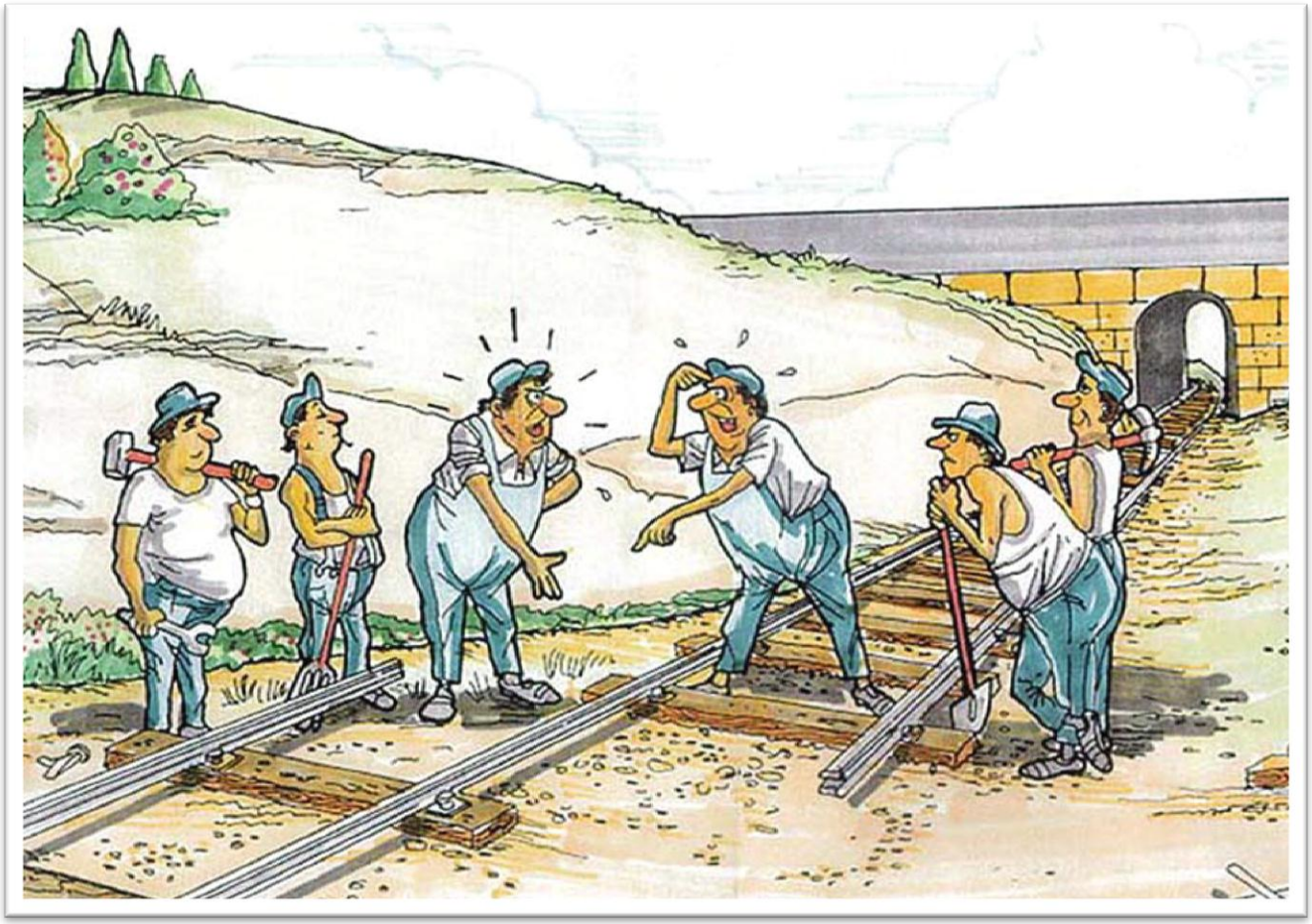
Nội dung

- Quy ước đặt tên (naming convention)
- Quy tắc trình bày tổng thể chương trình
- Quy tắc trình bày dòng lệnh
- Quy tắc liên quan đến hằng số
- Quy tắc liên quan đến kiểu tự định nghĩa
- Quy tắc liên quan đến biến
- Quy tắc liên quan đến hàm
- Quy tắc chú thích chương trình

Sức mạnh của làm việc nhóm

Phim quảng cáo của hãng Coca-Cola

Vì sao phải có chuẩn và quy ước?



Quy ước đặt tên trong lập trình

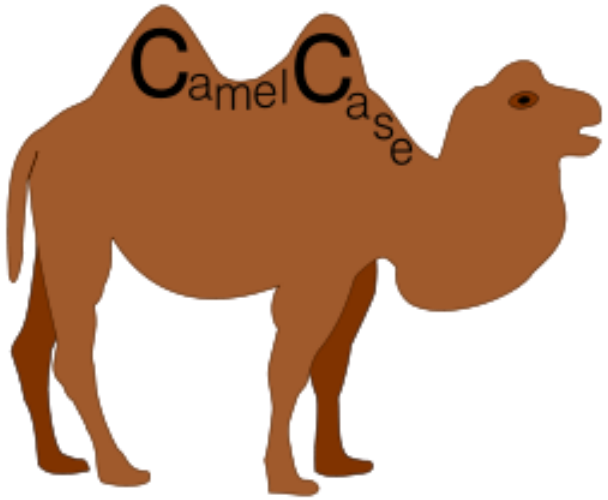
- Ngoài tính đúng đắn, chương trình máy tính cần phải dễ đọc và dễ hiểu.

vs.

- Hai chuẩn đang được sử dụng rộng rãi:
  - Quy tắc đặt tên theo kiểu "lạc đà"
  - Quy tắc đặt tên theo phong cách Hung-ga-ri.

Quy tắc "lạc đà" (Camel Case)

- Khái niệm
  - Cách viết tránh sử dụng các khoảng trắng giữa các từ, được sử dụng trong hầu hết các ngôn ngữ lập trình hiện đại và phổ biến như C, C++, Visual Basic (VB) và JavaScript.



Quy tắc "lạc đà" (Camel Case)

- Một số từ đồng nghĩa với ký hiệu CamelCase
  - BumpyCaps, BumpyCase
  - CamelCaps, CamelHumpedWord
  - CapWordsPython
  - mixedCase trong Python (lowerCamelCase)
  - ClCl (Capital-lower Captital-lower)
  - HumpBackNotation
  - InterCaps, NerdCaps, InternalCapitalization
  - WordsStrungTogether, WordsRunTogether

Quy tắc "lạc đà" (Camel Case)

- Phân loại
  - UpperCamelCase (thường gọi là PascalCase) nếu ký tự đầu tiên của câu được viết hoa.

    Ví dụ:
    TheQuickBrownFoxJumpsOverTheLazyDog

  - lowerCamelCase (thường gọi là camelCase) nếu ký tự đầu tiên của câu được viết thường.

    Ví dụ:
    theQuickBrownFoxJumpsOverTheLazyDog

Ký hiệu Hung-ga-ri

- Khái niệm
  - Là quy ước đặt tên trong lập trình máy tính được phát minh bởi Charles Simonyi.
  - Thường sử dụng trong môi trường lập trình Windows như C, C++ và Visual Basic.
  - Tên của biến cho biết kiểu, ý định sử dụng hoặc thậm chí là tầm vực của biến đó:
    - Tiền tố viết thường chứa thông tin về kiểu của biến.
    - Phần còn lại bắt đầu bằng ký tự hoa cho biết biến chứa thông tin gì.

Ký hiệu Hung-ga-ri

- Phân loại theo mục đích của tiền tố
  - Ký hiệu Hung-ga-ri hướng hệ thống (Systems Hungarian notation) sử dụng tiền tố để thể hiện kiểu dữ liệu thực sự, ví dụ:
    - `lAccountNum`: biến có kiểu là số nguyên dài ("l" – long integer)
    - `arru8NumberList`: biến là một mảng các số nguyên 8 bit không dấu ("arru8" – array of unsigned 8-bit integers)
    - `szName`: biến là một chuỗi có ký tự kết thúc ("sz" – zero-terminated string)

Ký hiệu Hung-ga-ri

- Phân loại theo mục đích của tiền tố
  - Ký hiệu Hung-ga-ri hướng ứng dụng (Apps Hungarian notation) sử dụng tiền tố để thể hiện mục đích sử dụng của biến, ví dụ:
    - `rwPosition`: biến thể hiện một dòng (“`rw`” – row)
    - `usName`: biến thể hiện một chuỗi “không an toàn” (“`us`” – unsafe), nghĩa là nó cần được “xử lý” trước khi sử dụng.
    - `strName`: biến thể hiện một chuỗi chứa một tên (“`str`” – string).

Ký hiệu Hung-ga-ri

- Ví dụ
  - `bBusy`: kiểu luận lý ("`b`" – boolean)
  - `cApples`: số lượng ("`c`" – count)
  - `dwLightYears`: kiểu từ kép ("`dw`" – double word)
  - `fBusy`: cờ trạng thái ("`f`" – flag)
  - `nSize`: kiểu số nguyên và lưu số lượng ("`n`" – count)
  - `iSize`: kiểu số nguyên và lưu chỉ số ("`i`" – index)
  - `fpPrice`: kiểu số chấm động ("`fp`" – floating-point)

Ký hiệu Hung-ga-ri

- Ví dụ
  - `dbPi`: kiểu số thực dài ("`db`" – double)
  - `pFoo`: kiểu con trỏ ("`p`" – pointer)
  - `pfnFunc`: con trỏ hàm ("`pfn`" – pointer to function)
  - `rgStudents`: mảng hoặc một vùng ("`rg`" – range)
  - `szLastName`: chuỗi có ký tự kết thúc ("`sz`" – zero-terminated string)
  - `u32Identifier`: kiểu số nguyên 32-bit không dấu ("`u32`" – unsigned 32-bit integer)
  - `stTime`: cấu trúc thời gian ("`st`" – structure)

Ký hiệu Hung-ga-ri

- Đối với con trỏ và mảng thường có kiểu của phần tử đi kèm:
  - `pszOwner`: con trỏ đến chuỗi ký tự có kết thúc (“`psz`” – pointer to zero-terminated string)
  - `rgfpBalances`: mảng các số chấm động (“`rgfp`” – array/range of floating-point)
  - `aulColors`: mảng các số nguyên dài không dấu (“`aul`” – array of unsigned long)

Ký hiệu Hung-ga-ri

- Thường thấy trong lập trình Windows (trong sách "Programming Windows", sách đầu tiên về lập trình Windows API của Charles Petzold):
  - `wParam`: tham số kiểu word ("`w`" – word-size)
  - `lParam`: tham số kiểu số nguyên dài ("`l`" – long-integer)
  - `hwndFoo`: biến quản lý cửa sổ ("`hwnd`" – handle to a window)
  - `lpszBar`: biến con trỏ số nguyên dài đến một chuỗi có kết thúc ("`lpsz`" – long pointer to a zero-terminated string)

Ký hiệu Hung-ga-ri

- Đôi khi được mở rộng trong C++ để cho biết phạm của biến, phân cách bởi dấu gạch dưới:
  - `g_nWheels`: biến toàn cục ("`g`" – global) và là số nguyên (integer).
  - `_wheels`: biến cục bộ (local) và không xác định kiểu.
  - `s_wheels`: biến tĩnh ("`s`" – static).
  - `m_nWheels`: thành viên ("`m`" – member) của một cấu trúc/lớp và là số nguyên ("`n`" – integer).
  - `m_wheels`: thành viên ("`m`" – member) của một cấu trúc/lớp và không xác định kiểu.

Đánh giá ký hiệu Hung-ga-ri

- Ưu điểm
  - Kiểu biến có thể thấy được từ tên biến đó.
  - Nhiều biến các nhau với cùng ngữ nghĩa có thể được sử dụng trong cùng một khối mã nguồn: `iWidth`, `fWidth`, `dWidth`
  - Các tên biến được thống nhất hơn.
  - Có thể phát hiện dễ dàng việc ép kiểu không phù hợp hoặc các biểu thức sử dụng kiểu không tương thích khi đọc mã nguồn.
  - Tránh sử dụng nhầm thao tác: `heightWindow = window.getWidth()`

Đánh giá ký hiệu Hung-ga-ri

- Khuyết điểm
  - Việc kiểm tra bằng mắt là thừa khi việc kiểm tra kiểu được thực hiện tự động bởi trình biên dịch.
  - Một số IDE hiện đại tự động đánh dấu những chỗ sử dụng kiểu không tương thích.
  - Khi kiểu của biến thay đổi, tên của biến đó cũng phải thay đổi nếu không sẽ không thống nhất.
  - Trong đa số trường hợp, việc biết ý nghĩa sử dụng của một biến bao hàm việc biết kiểu của nó. Vì vậy, nếu không biết biến dùng để làm gì thì việc biết kiểu của nó cũng vô ích!

Một số quan điểm đáng chú ý

Linus Torvalds (*cha đẻ của hệ điều hành mã nguồn mở Linux*) không ủng hộ ký hiệu Hung-ga-ri hướng hệ thống.

"Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged—the compiler knows the types anyway and can check those, and it only confuses the programmer"

Một số quan điểm đáng chú ý

Steve McConnell (*tác giả của nhiều sách CNPM nổi tiếng, được tạp chí Software Development xem là một trong ba người có ảnh hưởng nhất trong công nghiệp phần mềm cùng với Bill Gates và Linus Torvalds*) ủng hộ ký hiệu Hung-ga-ri.

"Although the Hungarian naming convention is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value. ... Standardized prefixes allow you to check types accurately when you're using abstract data types that your compiler can't necessarily check"

Một số quan điểm đáng chú ý

Bjarne Stroustrup (*nhà khoa học người Đan Mạch phát triển NNLT C++*) không ủng hộ ký hiệu Hung-ga-ri hướng hệ thống cho NNLT C++.

"No I don't recommend 'Hungarian'. I regard 'Hungarian' (embedding an abbreviated version of a type in a variable name) a technique that can be useful in untyped languages, but is completely unsuitable for a language that supports generic programming and object-oriented programming—both of which emphasize selection of operations based on the type an arguments (known to the language or to the run-time support). In this case, 'building the type of an object into names' simply complicates and minimizes abstraction"

Một số quan điểm đáng chú ý

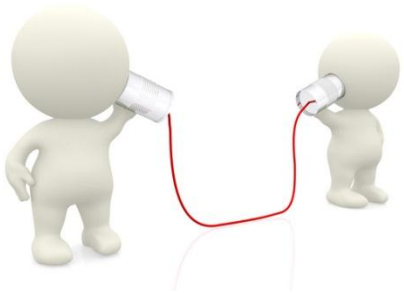
Joel Spolsky (*kỹ sư phần mềm và tác giả của blog về phát triển phần mềm đặc biệt về phần mềm Windows*) ủng hộ ký hiệu Hung-ga-ri hướng ứng dụng.

"If you read Simonyi's paper closely, what he was getting at was the same kind of naming convention as I used in my example above where we decided that us meant "unsafe string" and s meant "safe string." They're both of type string. The compiler won't help you if you assign one to the other and Intellisense won't tell you bupkis. But they are semantically different; they need to be interpreted differently and treated differently and some kind of conversion function will need to be called if you assign one to the other or you will have a runtime bug. If you're lucky. (...) There's still a tremendous amount of value to Apps Hungarian, in that it increases collocation in code, which makes the code easier to read, write, debug, and maintain, and, most importantly, it makes wrong code look wrong".

Quy tắc 1

Chương trình nên được tách thành nhiều đơn thể (mô-đun), mỗi đơn thể thực hiện một công việc và càng độc lập với nhau càng tốt. Điều này sẽ giúp chương trình dễ bảo dưỡng hơn và khi đọc chương trình, ta không phải đọc nhiều, nhớ nhiều các đoạn lệnh nằm rải rác để hiểu được điều gì đang được thực hiện.

Quy tắc 2



Nên sử dụng các tham số khi truyền thông tin cho các chương trình con. Tránh sử dụng các biến toàn cục để truyền thông tin giữa các chương trình con vì như vậy sẽ làm mất tính độc lập giữa các chương trình con và rất khó khăn khi kiểm soát giá trị của chúng khi chương trình thi hành.

Quy tắc 3

Cách trình bày chương trình càng nhất quán sẽ càng dễ đọc và dễ hiểu. Từ đó, ta sẽ mất ít thời gian để nghĩ về cách viết chương trình và như vậy sẽ có nhiều thời gian hơn để nghĩ về các vấn đề cần giải quyết.

Quy tắc 4

Chương trình nên giữ được tính đơn giản và rõ ràng trong hầu hết các tình huống. Việc sử dụng các mẹo lập trình chỉ thể hiện sự khéo léo của lập trình viên và làm tăng hiệu quả chương trình lên một chút, trong khi điều đó sẽ đánh mất đi tính đơn giản và rõ ràng của chương trình.

Quy tắc 4

Ví dụ

```
1   a[i++] = 1;
2   a[i] = 1;
3   i++;
4
5   if (doSomething() == false) …
6   bool bResult = doSomething();
7   if (bResult == false) …
8
9   *x += (*xp = (2*k < (n - m) ? c[k + 1] : d[k--]));
10  if (2 * k < n - m)
11      *xp = c[k + 1];
12  else
13      *xp = d[k--];
14  *x = *x + *xp;
```

Quy tắc 5

Chương trình nên thực hiện như một dòng chảy từ trên xuống dưới

- Các chỉ thị `#include`, `#define` trên cùng.
- Sau đó đến khai báo các biến toàn cục, `class` hay `struct`.
- Không nên có những thay đổi bất chợt do sử dụng `goto` hay `continue`.

Quy tắc 6

Mỗi câu lệnh nên được đặt riêng trên một dòng để chương trình dễ đọc và dễ quan sát
cách thực hiện trong quá trình tìm lỗi (debug).

```
1   x=a+b-c*d;for(int i=0; i<n; i++)      // Khó đọc!
2
3   x = a + b - c * d;                    // Viết thành 2 dòng
4   for (int i = 0; i < n; i++)           // dễ đọc hơn
5
6   // Thực hiện bao nhiêu lần khi debug?
7   while (i < 100) i++;
8
9   // Phải theo dõi i để biết if có thực hiện không!
10  if (i < 100) i++;
```




Quy tắc 7

Câu lệnh phải thể hiện đúng cấu trúc chương trình bằng cách sử dụng hợp lý thụt đầu dòng (tab)

```
1  if (nCount == 0) printf("No data!\n");
2
3  if (nCount == 0)
4  printf("No data!\n");
5
6  if (nCount == 0)
7      printf("No data!\n");
```



Quy tắc 8

Các dấu { } bao các khối lệnh phải được canh thẳng hàng theo một trong hai cách sau:

```
1  // Cách 1                    // Cách 2
2  if (…)                      if (…) {
3  {                               …
4      …                       }
5  }                           else {
6  else                        …
7  {                           }
8      …
9  }
```

Nên viết cặp dấu { } trước rồi viết các lệnh vào giữa để tránh thiếu các dấu ngoặc này.

Quy tắc 9

Các câu lệnh nằm giữa cặp dấu { } được viết thụt vào một khoảng tab (các lệnh ngang cấp thì phải thụt vào như nhau).

```
1  if (a == 1)
2  {
3      printf(“Mot\n”);
4      if (a == 2)
5          printf(“Hai\n”);
6      else
7          printf(“Khac Mot va Hai!\n”);
8  }
```

Quy tắc 10

Các toán tử và toán hạng trong một biểu thức nên được tách rời nhau bởi 1 khoảng trắng nhằm làm biểu thức dễ đọc hơn (trừ các toán tử ++, – –, –)

```
1  a=b*c;        // Khó đọc!
2  a = b * c;    // Dễ đọc hơn!
3
4  a = b ++;     // Không nên!
5  a = b++;      // Nên!
6
7  a = - b;      // Không nên!
8  a = -b;       // Nên!
```

Quy tắc 11

Có khoảng trắng ngăn cách giữa dấu phẩy hay chấm phẩy với các tham số.

```
1  // Khoảng trắng sau dấu , trong khai báo hàm
2  void hoanVi(int a,int b);
3  void hoanVi(int a, int b);
4
5  // Khoảng trắng sau dấu , trong lời gọi hàm
6  hoanVi(x,y);
7  hoanVi(x, y);
8
9  // Khoảng trắng sau dấu ; trong vòng lặp for
10 for (int i = 1;i < n;i++) …
11 for (int i = 1; i < n; i++) …
```

Quy tắc 12

- Nên có khoảng trắng giữa từ khóa và dấu '(', nhưng không nên có khoảng trắng giữa tên hàm và dấu '('

```
1  // Không nên có khoảng trắng giữa hoanVi và (
2  void hoanVi (int a, int b);
3  void hoanVi(int a, int b);
4
5  // Nên có khoảng trắng giữ if và (
6  // Không nên có khoảng trắng giữa strcmp và (
7  if(strcmp (strInput, "info") == 0)
8      …
9  if (strcmp(strInput, "info") == 0)
10     …
```




Quy tắc 13

Nên sử dụng các dấu ( ) khi muốn tránh các lỗi về độ ưu tiên toán tử

```
1  int i = a >= b && c < d && e <= g + h;
2  int i = (a >= b) && (c < d) && (e <= (g + h));
```

Quy tắc 14

Nên dùng các dòng trắng để phân chia các đoạn lệnh trong một hàm như: đoạn nhập/xuất dữ liệu, đoạn tương ứng với các bước xử lý khác nhau.

```
1   void main()
2   {
3       // (Các) Đoạn lệnh nhập ở đây
4       …
5
6       // (Các) Đoạn lệnh xử lý ở đây
7       …
8
9       // (Các) Đoạn lệnh xuất Xuất
10      …
11  }
```

Quy tắc 15

Mỗi dòng lệnh không nên dài quá 80 ký tự, điều này giúp việc đọc chương trình dễ dàng hơn khi không phải thực hiện các thao tác cuộn ngang màn hình. Trong trường hợp dòng lệnh quá dài thì nên ngắt thành nhiều dòng.

```
1  int myComplexFuncuntion(unsigned int uiValue, int iValue, …
2
3  int myComplexFuncuntion(unsigned int uiValue,
4                          int iValue,
5                          char cValue,
6                          int *piValue);
```

Quy tắc 16

Tên hằng không theo quy tắc PascalCase, camelCase hay ký hiệu Hung-ga-ri mà được viết in toàn bộ và giữa các từ cách nhau bằng dấu _

```
1  const int NumberOfElements 100;
2
3  const int NUMBEROFELEMENTS 100;
4
5  const int NUMBER_OF_ELEMENTS 100;
```



Quy tắc 17

Các hằng số không nên viết trực tiếp vào chương trình mà nên sử dụng `#define` hay `const` để định nghĩa. Điều này sẽ giúp lập trình viên dễ kiểm soát những chương trình lớn vì giá trị của hằng số khi cần thay đổi thì chỉ phải thay đổi một lần duy nhất ở giá trị định nghĩa (ở `#define` hay `const`).

Tuy vậy, không nên dùng `#define` thường xuyên để định nghĩa các hằng số, bởi vì trong quá trình debug, ta sẽ không thể xem được giá trị của một hằng số định nghĩa bằng `#define`.

Quy tắc 18

Tên kiểu tự định nghĩa là danh từ được đặt theo PascalCase và bắt đầu bằng một ký tự đại diện:

- `TMyTypeName`: Kiểu định nghĩa bằng typedef
- `SMyStructName`: Kiểu cấu trúc (structure)
- `UMyUnionName`: Kiểu hợp nhất (union)
- `EMyEnumName`: Kiểu tập hợp (enumeration)
- `CMyClassName`: Lớp đối tượng (class)

Quy tắc 19

Tên biến được đặt theo ký hiệu Hung-ga-ri sao cho đủ nghĩa, có thể là là các từ hoàn chỉnh hoặc viết tắt nhưng phải dễ đọc (dễ phát âm)

```
1  int ntuso, nmauso;
2
3  int nTuso, nMauso;
4
5  int nTuSo, nMauSo;
```



Quy tắc 20

Tên biến kiểu dữ liệu tự định nghĩa (typedef, struct, union, class, …) được viết theo camelCase và không có tiền tố.

```
1  CPhanSo psPhanSo;
2
3  CPhanSo PhanSo;
4
5  CPhanSo phanSo;
```



Quy tắc 21

Biến nên được khai báo ở gần vị trí mà nó bắt đầu được sử dụng nhằm tránh việc khai báo một loạt các biến dư thừa ở đầu hàm hay chương trình.

- Trong C chuẩn, tất các biến bắt buộc phải khai báo ở đầu khối trước khi sử dụng.
- Trong C++ biến có thể khai báo ở bất kỳ nơi đâu trước khi sử dụng.

Quy tắc 22

Mỗi biến nên khai báo trên một dòng nhằm dễ chú thích về ý nghĩa của mỗi biến.

```
1  float fHeSoA, fHeSoB, fNghiem;   // …!
2
3  float fHeSoA, fHeSoB;    // Hệ số a, b của pt bậc 1
4  float fNghiem;           // Nghiệm của pt bậc 1
```

Quy tắc 23

Các biến không nên được sử dụng lại với nhiều nghĩa khác nhau trong cùng một hàm.

```
1  for (i = 0; i < n; i++)      // n là số lần lặp
2      …
3
4  for (n = 0; n < 10; n++)     // n là biến điều khiển
5      …
```

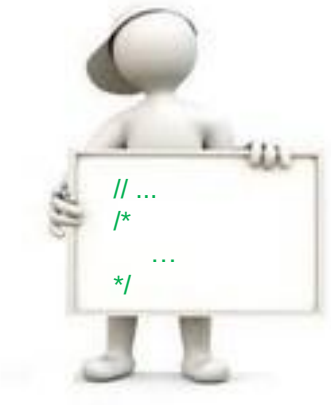
Quy tắc 24

Tên hàm được viết theo camelCase và phải là động từ phản ánh công việc hàm sẽ thực hiện hoặc giá trị trả về của nó.

```
1  int SoLonNhat(int a[], int n);        // Không rõ hành động
2
3  int TimSoLonNhat(int a[], int n);     // Sai quy tắc đặt tên
4
5  int timSoLonNhat(int a[], int n);     // OK
```



Quy tắc 25



Nên sử dụng dấu // vì dấu này không có ảnh hưởng khi sử dụng cặp ký hiệu /* */ để vô hiệu hóa một đoạn lệnh trong quá trình sửa lỗi chương trình. Nên nhớ rằng trong C/C++ không cho phép các cặp dấu /* */ lồng nhau.

Quy tắc 26

Nên viết chú thích ngắn gọn nhưng đầy đủ, dễ hiểu qua việc trình bày chương trình rõ ràng, đơn giản và cách đặt tên hợp lý. Với các chú thích ngắn, nên đặt nó trên cùng dòng lệnh cần chú thích. Với các chú thích dài hơn, hoặc chú thích cho cả một đoạn lệnh hãy đặt câu chú thích trên một dòng riêng ngay phía trên câu lệnh cần chú thích.

Quy tắc 26

Ví dụ

```
1   if (a == 2)
2       return true;                    // Trường hợp đặc biệt
3   else
4       return laSoNguyenTo(a));        // Trường hợp khác
5
6   if (argc > 1)
7   {
8       // Mở và xử lý tập tin…
9       FILE *fp = fopen(argv[1], “rb”);
10      if (fp != NULL)
11          …
12      fclose(fp);
13  }
```



Quy tắc 27

Không nên lạm dụng chú thích, cố gắng giữ cho chương trình của bạn dễ đọc, dễ hiểu.

```
1   i++;     // Câu lệnh này nhằm tăng i thêm 1 đơn vị!!!
```

Quy tắc 28

Viết chú thích cho từng hàm và cho từng tập tin mã nguồn (.h và .cpp).

```
1  // Mô tả          : Chương trình xử lý mảng một chiều
2  // Tác giả        : Đặng Bình Phương
3  // Email          : dbphuong@fit.hcmus.edu.vn
4  // Ngày cập nhật : 15/06/2011
5
6  // Tên hàm        : timSoLonNhat
7  // Mô tả          : Tìm số lớn nhất trong mảng cho trước
8  // Kiểu tra về    : int
9  // Tham số: int a[] – Mảng một chiều các phần tử kiểu int
10 // Tham số: int n   – Số lượng phần tử
11 int timSoLonNhat(int a[], int n);
12 …
```