PHONG CÁCH LẬP TRÌNH C++

Bạn đang học lập trình, vậy bạn nghĩ về một chương trình máy tính như thế nào?

- *Một chương trình máy tính có thể xem như một tác phẩm, bởi vì nó được đọc bởi bạn (có thể bây giờ, mà cũng có thể là 10 năm sau!), và bởi những lập trình viên khác sau bạn (để phát triển, sửa chửa, cập nhật ...). Chính vì lẽ đó, một chương trình máy tính nên đáp ứng cả 3 yêu cầu sau: đúng, dễ đọc và dễ hiểu.*
- *Mục đích của style là làm cho chương trình trở nên dễ đọc đối với người viết và những người khác, một style tốt là một phần thiết yếu của việc lập trình tốt. Viết một chương trình chạy đúng là chưa đủ bởi chương trình không chỉ để cho máy tính đọc mà còn để các lập trình viên khác đọc. Hơn nữa, một chương trình có style tốt luôn có nhiều khả năng chạy đúng hơn một chương trình có style tồi.*

Tài liệu này cung cấp cho bạn:

1. Một tập hợp các chuẩn trình bày chương trình thông dụng.

2. Một thói quen để từ đó bạn có một phong cách lập trình tương đối chuyên nghiệp.

Tuy nhiên, tài liệu này không có tham vọng đề cập đến toàn bộ các khía cạnh của một phong cách lập trình. Nó chỉ nói đến những gì cần thiết để cho bạn, một sinh viên, khi chưa tìm được một phong cách phù hợp tạo được những thói quen tốt ngay từ đầu.

Tổ chức chương trình

1. Môđun hóa chương trình của bạn

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

Khi muốn chuyển thông tin cho các chương trình con, bạn nên sử dụng các tham số. Tránh sử dụng các biến toàn cục để chuyển thông tin giữa các chương trình con. làm như vậy bạn sẽ triệt tiêu tính độc lập giữa các chương trình con và rất khó khăn khi kiểm soát giá trị của chúng khi chương trình thi hành. (Chú ý, bạn nên phân biệt giữa biến toàn cục và hằng số toàn cục)

2. Định nghĩa và cài đặt của các lớp phải được chia thành nhiều file để ta có thể dễ dàng tái sử dụng. **Định nghĩa các lớp được lưu trong các file header với mở rộng *.h. Cài đặt của các thành viên của lớp lưu trong file nguồn với mở rộng *.cpp.** Thông thường mỗi lớp có một cặp file *.H và *.CPP, nhưng có thể gộp các lớp có liên quan vào một cặp file.Cuối mỗi file *.H là chỉ thị #include đến file *.CPP.Cuối mỗi file *.CPP là các "chương trình chính" dùng để test file CPP đó kèm theo các #define thích hợp cho việc test. Chương trình chính được lưu trong một

file nguồn riêng và include các file header của các lớp được dùng đến.

3. Mỗi file header của lớp nên sử dụng các định hướng #ifndef, #define, và #endif để đảm bảo mỗi file header chỉ được include 1 lần. Ký hiệu được #define nên là tên

của file header viết toàn bằng chữ hoa với một dấu gạch dưới (_) thay cho dấu chấm.

Ví dụ:

```
//counter.h
#ifndef COUNTER_H
#define COUNTER_H
   class Counter
   {


      //...
   };  // end Counter

   #include "counter.cpp"
#endif  // COUNTER_H
```

Chuẩn tài liệu

1. Sử dụng // cho các chú thích. Chỉ dùng /* */ để tạm thời vô hiệu hóa các đoạn chương trình để test và debug.

2. Mỗi file nguồn, cả .CPP và .H, đều phải bắt đầu bằng một khối chú thích đủ để người đọc có thể kết nối các file nếu chúng bị tách ra. Mẫu như sau:

```
//---------------------------------------------------------------------
// Name:    Họ tên
// Class:   Lớp
// Project: mô tả/tên dự án (một dòng, giống nhau tại mọi file)
// Purpose: Mục đích sử dụng của mã chương trình hoặc các khai báo trong file này
//---------------------------------------------------------------------
```

Mỗi lớp, hàm, phương thức phải có một khối chú thích mô tả ngắn gọn lớp, hàm, phương thức đó làm gì; đối với hàm/phương thức: liệt kê tất cả các tham số, nêu rõ ý nghĩa của tham số; và mô tả điều kiện trước và sau của hàm/phương thức đó.Chọn các tên có nghĩa sẽ đơn giản hóa các chú thích.

Lưu ý, tài liệu về phương thức đặt tại định nghĩa lớp (*.H) ta có thể sao chép tài liệu đó vào file *.CPP nhưng không bắt buộc.

3. Có thể chú thích các đoạn code bên trong hàm, tuy nhiên chỉ nên chú thích đủ hiểu. Quá nhiều chú thích và chú thích thừa làm code trông rối. Tất cả các chú thích phải được lùi đầu dòng cùng đoạn code quanh nó.

Tên:

• Sử dụng các tên có nghĩa.**Tên giàu tính mô tả cho các biến toàn cục và tên ngắn gọn cho các biến cục bộ.**Tên có nghĩa sẽ giúp chương trình dễ viết và dễ debug hơn. Nếu bạn phải dùng tên không có nghĩa cho một cái gì đó thì có thể bạn chưa hoàn toàn hiểu bài toán mình đang giải. Hãy cố hiểu rõ trước khi tiếp tục lập trình.

Theo thônglệ, các tên i và j được dành cho các chỉ số, p và q dành cho các con trỏ, s và t dành cho các xâu.Người ta dùng các tên bắt đầu hoặc kết thúc bởi chữ "p" cho các biến con trỏ (chẳng hạn nodep, intp, intpp, doublep), các tên bắt đầu bằng chữ hoa cho biến toàn cục (chẳng hạn Globals) và tất cả chữ cái hoa cho các hằng số (chẳng hạn CONSTANTS).

Khuyến cáo sử dụng tên tiếng Anh kiểu camel (xem bên dưới)

Các namespace trong C++ góp phần làm rõ nghĩa của các tên mà không cần sử dụng các tên dài.

• Đặt tên một cách nhất quán

Các biến có liên quan phải được đặt các tên có liên quan, đồng thời phải làm nổi bật được sự khác nhau của chúng. Các tên trong lớp sau đây vừa quá dài vừa không hề nhất quán:

```
class
  UserQueue
{
  int noOfItemsInQ, frontOfTheQueue, queueCapacity;
  public int noOfUsersInQueue() {...}
};
```

Thứ nhất, cùng một nội dung là queue nhưng được biểu diễn bởi ba dấu hiệu: Q, Queue và queue. Thứ hai, các biến và các hàm thành phần của lớp UserQueue chỉ có thể được sử dụng bởi các đối tượng của lớp này, do vậy viết

```
queue.queueCapacity
```

hay

```
queue.noOfUsersInQueue()
```

rõ ràng là thừa. Chúng ta có thể viết lại lớp này với các tên mới như sau:

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

Không chỉ bản thân đoạn mã định nghĩa lớp đó dễ hiểu hơn, mà những đoạn mã sử dụng lớp UserQueue cũng dễ hiểu hơn:

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

Lớp UserQueue vẫn có thể cải tiến thêm, bởi nitems và nusers thực chất là cùng biểu diễn một khái niệm và do đó chỉ cần sử dụng một trong hai tên đó mà thôi.

• Tên của các project, form, và component sinh bởi môi trường lập trình: các project và form phải có tên hợp lý, không để nguyên là Form1. Các component phải được đặt tên có nghĩa, ngoại trừ các component như Label, Group Box, etc., nếu chúng không có mặt trong code. Các component nên được đặt hậu tố là kiểu đối tượng:

Ex: widthScale, nameText, leftScrollbar, mainForm, myLabel, printerDialog ...

• Tên biến và tên hàm:

o Thường phải là các từ hoặc cụm từ. Ngoại lệ duy nhất: con đếm vòng for() đôi khi có thể chỉ cần dùng tên là 1 chữ cái chẳng hạn i
o không viết tắt trừ các từ viết tắt thông dụng chẳng hạn HTML, khi đó coi từ viết tắt như từ thông thường (tên sẽ có dạng convertToHtml thay vì convertToHTML)
o Đặt tên cho các namespace nên bằng chữ in thường toàn bộ:
Ex: mynamespace, com.company.application.ui
o Tên biến là một danh từ bắt đầu bằng một ký tự in thường, các từ tiếp theo được bắt đầu bằng một ký tự in hoa:
Ex: line, audioSystem…
o Đặt các tên "động" cho hàm:
Tên hàm nên là một động từ theo sau bởi một danh từ. Ví dụ:

```
now = date.getTime();
```

Các hàm trả về giá trị boolean nên được đặt tên thể hiện giá trị mà nó trả về. Ví dụ:

```
isOctal( c )
```

thì tốt hơn là:

```
checkOctal( c );
```

vì cách đặt tên thứ nhất cho biết ngay rằng hàm trả về giá trị true nếu c là một số octal và trả về false trong trường hợp ngược lại.
o Tên hàm thể hiện chức năng
Các tiền tố thường được sử dụng: get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume ...
Ex:
+"set/get" được đặt trong các phương thức truy cập trực tiếp đến thuộc tính:
getName(), setSalary(int) …
+"find" có thể được sử dụng trong các phương thức tìm kiếm:

```
vertex.findNearestVertex();   matrix.findSmallestElement();   node.findShortestPath(Node destinationNode);
```

+Tập hợp nhiều đối tượng nên được đặt tên được đặt tên ở số nhiều:

```
vector<Point> points; int[] values;
```

+Những biến chỉ số lượng đối tượng nên có tiền tố "n":
nPoints, nLines…

• Tên class (và struct):

Dùng chữ hoa tất cả các chữ cái đầu mỗi từ, còn lại là các chữ cái thường.

Ví dụ: GameBoard, Game.

Định dạng

• Lùi đầu dòng các đoạn code, mỗi mức dùng 3 hoặc 4 ký tự, tốt nhất là dùng tab.

- Phải thống nhất, luôn dùng 3 hoặc luôn dùng 4 ký tự
- Chú ý không được dùng lẫn lộn giữa ký tự tab và space để lùi đầu dòng,

(các môi trường soạn thảo có thể quy ước khác nhau về độ dài của tab).

• Mỗi dòng chỉ chứa nhiều nhất 1 lệnh và không dài quá 79 ký tự. Một lệnh có thể

được chia thành nhiều dòng, khi đó các phần sau phải được lùi đầu dòng hợp lý.

Ví dụ :

```
cout << "The cost for 1 loaf of bread" << endl
     << "is $1.89" << endl;
```

• Có thể căn thẳng hàng để nâng cao hightlight.
VD:

```
int songuyen    = 100;

double sothuc   = 3.14 ;
char kytu       = 'a' ;
char ten[]      = { "tam", "lan", "hiep", "bao", "yen", "tuan", "hoa" };
bool gt[]       = {  0    , 0    , 0      , 1     , 0     , 1      , 0      };
```

• Các khối với cặp ngoặc {} phải được trình bày 1 trong 2 cách sau:

```
if (!done)
{
        doSomething();
        moreToDo();
}
else
{
        //...
}

if (!done)  {
        doSomething();
        moreToDo();
}  else {
        //...
}
```

Nếu trong khối chỉ có 1 lệnh thì có thể bỏ ngoặc (tốt hơn là không nên) nhưng vẫn lùi đầu dòng:

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

        field[ n ] = 0;

*i = 0;
return 'n';
```

• Nên có khoảng trắng giữa từ khóa và dấu '(', nhưng không nên có khoảng trắng giữa tên hàm và dấu '('.
Ví dụ:

```
// no space between 'strcmp' and '(',
// but space between 'if' and '('
if ( strcmp( input_value, "done" ) == 0 )
  return 0;
```

Ngoài ra nên có khoảng giữa dấu ngoặc của hàm và đối số như trên.
• Nên có 1 space trước và sau mỗi toán tử đôi số học hoặc lôgic, chẳng hạn +, <<, và ||. Nên dùng 1 space sau dấu phẩy, nhưng trước dấu phẩy hoặc chấm phẩy không nên có dấu cách.Ngoại lệ: không chèn khoảng trắng vào giữa toán hạng và toán tử ++ và –-

• Chèn dòng trắng giữa các đoạn khác nhau trong chương trình.

Thiết kế

Một số vấn đề thường gặp mà sinh viên cần chú ý:

• Không để dữ liệu của lớp dạng public.

• Hạn chế tối đa việc dùng biến toàn cục.

• Nguyên tắc quyền ưu tiên tối thiểu: chỉ cho hàm đủ quyền truy nhập để thực hiện nhiệm vụ của mình, không cho nhiều quyền hơn.

- const được sử dụng cho một biến khi hàm không cần thay đổi biến được tham chiếu đến đó.
- Nếu hàm không được sửa giá trị của một tham số, chỉ truyền tham số vào là giá trị đối với các kiểu đơn giản, mảng phải được truyền dưới dạng "const []", và các kiểu struct/class nên truyền dạng "const &" hoặc "const *".

• Không để dữ liệu trong lớp mà nó không thực sự thuộc về lớp đó. Chẳng hạn, nếu một hàm cần một biến để lưu một kết quả tạm thời, không khai báo một biến thuộc lớp mà hãy khai báo một biến địa phương của hàm đó.

• Các hàm hoặc phương thức của lớp không nên tạo output trừ khi đó là nhiệm vụ của phương thức đó. Ví dụ, một phương thức print có thể sẽ ghi thông tin ra cout, nhưng một thao tác để thêm hoặc bớt cái gì đó từ một danh sách thì không.

• Mỗi phương thức/hàm (kể cả hàm main()) chỉ chứa tối đa 30 dòng kể cả tính từ ngoặc mở hàm "{" tới ngoặc kết thúc hàm "}".

CODE

1.Viết code theo chuẩn ISO dù compiler có bắt buộc hay không.

• Hạn chế #include ngoài chuẩn, VD: conio.h

• Nên để int main()và return 0;thay vì void main()

• Khai báo using std:: ngay cả khi IDE không bắt buộc. (dùng using namespace std; không tốt lắm)

2. Các hằng số không nên viết trực tiếp vào chương trình.

Thay vì thế, người ta thường sử dụng lệnh #define hay const để đặt cho những hằng số này những tên có ý 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).
Ví dụ:

```
popChange = (0.1758 - 0.1257) * population;
```

nên được viết là:

```
const double BIRTH_RATE = 0.1758, DEATH_RATE = 0.1257;
```

hoặc:

```
#define BIRTH_RATE 0.1758

#define DEATH_RATE 0.1257


//...
popChange = (BIRTHRATE - DEATH_RATE) * population;
```

Ghi chú: bạn không nên dùng #define thường xuyên để định nghĩa các hằng số, bởi vì trong quá trình debug, bạn sẽ không thể xem được giá trị của một hằng số định nghĩa bằng #define.

Khi làm việc với các kí tự, hãy sử dụng các hằng kí tự thay vì các số nguyên. Ví dụ để kiểm tra xem c có phải một chữ cái hoa hay không, có thể dùng đoạn mã sau:

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

Nhưng đoạn mã này hoàn toàn phụ thuộc vào bộ mã biểu diễn kí tự đang được sử dụng. Cách tốt hơn là viết như sau:

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

3.Luôn viết new và delete thành từng cặp.

4. Khi khai báo con trỏ, dấu con trỏ nên được đặt liền với tên, nhằm tránh trường hợp sau:

```
char* p, q, r; // q, r không là con trỏ
```

Trong trường hợp này nên viết là:

```
char *p, *q, *r;
```

(luật này cũng được dùng khi khai báo tham chiếu với dấu &)

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

Ví dụ:

```
// No!
int i = a >= b && c < d && e <= g + h;
// Better
int j = (a >= b) && (c < d) && (e <= (g + h));
```

Bảng sau trong sách **C Programming Language** chỉ ra thứ tự ưu tiên các toán tử trong C. Hàng trên cùng có mức ưu tiên cao nhất.

Toán tử	Dịch
([- .	Từ trái qua phải
! -- ++ { * & (type-cast) sizeof + - (1 ngôi)	Từ phải qua trái
* / %	Từ trái qua phải
+ -	Từ trái qua phải
<< >>	Từ trái qua phải
< <= > >=	Từ trái qua phải
== !=	Từ trái qua phải
&	Từ trái qua phải
^	Từ trái qua phải
|	Từ trái qua phải
&&	Từ trái qua phải
||	Từ trái qua phải
?:	Từ trái qua phải
= += -= *= /= %= &= ^= |= <<= >>=	Từ phải qua trái
,	Từ trái qua phải

Dùng bảng này, có thể thấy rằng **char *a[10];** là một mảng 10 con trỏ kí tự. Bạn cũng thấy rằng tại sao lại cần dấu ngoặc khi dùng **(*p).i**. Sau khi thực hành, bạn sẽ nhớ bảng này.

6.Tách các biểu thức phức tạp thành các biểu thức đơn giản hơn

Biểu thức sau đây rất ngắn gọn nhưng lại chứa quá nhiều phép toán:

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

Chúng ta nên viết lại như sau:

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

7.Viết các lệnh dễ hiểu, không viết các lệnh "khôn ngoan"

Các lập trình viên thường thích viết các lệnh càng ngắn gọn càng tốt. Tuy nhiên điều này thường gây phiền toái cho người khác.

Hãy xem biểu thức sau đây làm gì:

```
subkey = subkey >> ( bitoff – ( ( bitoff >> 3 ) << 3 ) );
```

Biểu thức trong cùng (bitoff >> 3) dịch phải bitoff 3 bit. Kết quả thu được lại được dịch trái 3 bit. Bởi vậy 3 bit cuối cùng của bitoff được thay thế bởi các số 0. Kết quả này lại được trừ đi bởi giá trị ban đầu của bitoff, kết quả của phép trừ chính là 3 bit cuối cùng trong giá trị ban đầu của bitoff. Ba bit này được dùng để dịch subkey sang phải.

Bởi vậy, biểu thức nói trên tương đương với biểu thức sau đây:

```
subkeu = subkey >> ( bitoff & 0×7 );
```

Rõ ràng cách viết thứ hai dễ hiểu hơn nhiều. Một ví dụ khác về cách viết biểu thức ngắn gọn nhưng làm phức tạp hóa vấn đề:

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

Cách viết dưới đây dài hơn, nhưng dễ hiểu hơn nhiều:

```
if( LC == 0 && RC == 0 )
```

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

Toán tử ? : chỉ thích hợp cho những biểu thức ngắn kiểu như sau đây:

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

hoặc:

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

Hãy nhớ rằng mục tiêu của chúng ta là viết những đoạn mã dễ hiểu, chứ không phải các đoạn mã ngắn gọn.

8.Cẩn thận với dấu =

= và == là 2 toán tử gây nhầm lẫn nhất trên C, nhưng bạn có thể tránh gặp nó bằng thói quen viết r-value (biểu thức bên phải phép gán) sang bên trái phép so sánh:

```
if ( a == 42 ) { ... }// Cách viết thông thường.
if ( 42 == a ) { ... }// Nên viết thế này.
```

Và đây là sự khác biệt, khi bạn nhầm ...

```
if ( a = 42 ) { ... }// Chạy bình thường, khó tìm ra lỗi
if ( 42 = a ) { ... }// Báo lỗi ngay chỗ này
```

9. Các idiom

Cũng giống như ngôn ngữ tự nhiên, ngôn ngữ lập trình cũng có các idiom (thành ngữ !?), là các cách viết code chính tắc cho các trường hợp thông dụng, tạm hiểu idiom là các chuẩn không bắt buộc nhưng được đa số người dùng tuân theo. Sử dụng các idiom giúp giảm bớt khả năng mắc lỗi đồng thời làm chương trình dễ đọc hơn và nhất là có vẻ “chuyên nghiệp” hơn Sau đây là một số idiom phổ biến:

1.Các idiom cho mảng

Để duyệt qua n phần tử của một mảng và khởi tạo chúng, có các cách viết sau đây:

```
i = 0;
while ( i <= n – 1 )
   array[ i++ ] = 1.0;
hoặc
for( i = 0; i < n; )
   array[ i++ ] =  1.0;
hoặc
```

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

Tất cả những cách viết trên đều đúng, tuy nhiên idiom cho trường hợp này là:

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

Một lưu ý nhỏ là sự khác biệt giữa i++ và ++i:

- i++ lấy giá trị của i trước rồi tăng nó lên.
- ++i tăng giá trị của i rồi lấy giá trị mới.

Do đó đối với các con đếm vòng lặp (for(),while()) nên dùng ++i để tăng tốc độ.

Idiom của vòng lặp duyệt qua các phần tử của một danh sách (list) là

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

Đối với container:

```
vector<string>::iterator it;
for(it = v.begin(); it != v.end(); ++it)
    std::cout << *it;
```

Đối với các vòng lặp vô hạn, idiom là: `for ( ; ; )` hoặc `while( 1 )`

Khởi tạo danh sách:

```
struct info
{
    char *name;
    char *job;
    char *address;
};

info *array[] = {
          { "name1", "job1", "add1" },
          { "name1", "job1", "add1" },
          { "name1", "job1", "add1" },
          //...
};
```

Hàm tìm kiếm tuyến tính:

```
template <class T>
int find (T obj,T* array,int size,int from = 0)
{
        for(int i = from; i<size; ++i)
                if(array[i] == T) return i;
        return size;
}
```

Sao chép mảng:

Giả sử 2 mảng double *a,*b; thay vì:

```
for(int i=0; i<n; ++i)
    b[i]=a[i];
```

ta có thể dùng:

```
//#include<string.h>
memcpy(b,a,n*sizeof(double));
```

Cấp phát động cho mảng 2 chiều:

```
int **pp = new type*[n];
int *p = new type[n*m];
          for (int i = 0; i < n; ++i)
                    pp[i] = p + i * m;
          //...
          //use array here
delete[] p;
delete[] pp;
```

2.Idiom cho lệnh if

Tiếp theo là một idiom dành cho câu lệnh if. Hãy xem đoạn mã loằng ngoằng sau đây làm gì

```
if ( argc==3 )
    if ( ( fin = fopen(argv[l] , “r” ) ) != NULL )
        if ( ( fout = fopen( argv[2], “w” ) ) != NULL ) {

            while ( ( c = getc( fin ) ) != EOF )
                putc( c, fout );
            fclose( fin );
            fclose( fout );

        } else
            printf ( “Can’t open output file %s\n”, argv[2] ) ;
    else
        printf( “Can’t open input file %s\n”, argv[l] ) ;
else
    printf ( “Usage: cp input file outputfile\n” ) ;
```

Viết lại đoạn mã này theo đúng idiom như sau:

```
if ( argc != 3 )
    printf ( “Usage: cp input file outputfile\n” ) ;
else if ( ( fin = fopen( argv[l] , “r” ) ) == NULL )
    printf( “Can’t open input file %s\n”, argv[l] );
else if ( ( fout = fopen( argv[2], “w” ) ) == NULL )
```

```
    {

        printf ( “Can’t open output file %s\n”, argv[2] ) ;
        fclose( fin ) ;
    } else
    {

        while ( ( c = getc( fin ) ) != EOF)
            putc( c, fout );
        fclose( fin ) ;
        fclose( fout ) ;
    }
```

Nguyên tắc khi viết các lệnh if() là đặt các phép toán kiểm tra điều kiện càng gần các hành động tương ứng càng tốt.

3.Idiom cho switch() case:
Xét ví dụ:

```
switch (c)
{
  case '-': sign = -1;
  case '+': c = getchar();
  case '.': break;
  case '0': case 'o': default:  if (!isdigit(c)) return 0;
}
```

cách viết sau tuy dài nhưng dễ đọc hơn:

```
switch (c)
{
   case '-':
      sign = -1;
   case '+':
      c = getchar();
      break;
   case '.':
      break;
   default: case '0': case 'o':
      if (!isdigit(c))
         return 0;
      break;
}
```

4.Số 0 trong chương trình

Số 0 thường xuyên xuất hiện trong các chương trình với nhiều ý nghĩa khác nhau. Trình dịch sẽ tự động chuyển số 0

thành kiểu thích hợp. Tuy nhiên nên viết ra một cách tường minh bản chất của số 0 mà chúng ta đang nói đến. Cụ thể, hãy sử dụng (void*)0 hoặc NULL để biểu diễn con trỏ null trong C, sử dụng '\0' cho kí tự null ở cuối mỗi xâu và sử dụng 0.0 cho các số float hoặc double có giá trị không. Đừng viết đoạn mã như sau

```
p = 0;
name[ i ] = 0;
x = 0;
```

Hãy viết:

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

Số 0 nên để dành cho các số nguyên có giá trị bằng không. Tuy nhiên trong C++, 0 (thay vì NULL) lại được sử dụng rộng rãi cho các con trỏ null, điều này không được khuyến khích.

Mọi sự vi phạm đều được cho phép nếu nó giúp cho tối ưu đoạn mã của bạn.

Mục đích chính của các quy tắc này là làm cho mã nguồn dễ đọc hiểu hơn, dễ dàng sửa lỗi và bảo trì, nâng chất lượng chung của mã nguồn. Tuy nhiên, nó sẽ không thể áp dụng đúng với mọi trường hợp cụ thể, và các lập trình viên phải sử dụng mềm dẻo các quy ước này.