C++ Coding Standard

Version 2

Tổ Khoa học máy tính

Contents

I. Giới thiệu

Tài liệu này sẽ định nghĩa các quy tắc nhằm nâng cao chất lượng của C++ code dựa trên các chuẩn chung nhất. Các quy tắc sẽ phù hợp nhất với ISO C++ Language sử dụng phương pháp lập trình OOP.

Mục đích của tài liệu này nhằm để giúp cho developer có thể tạo ra các project có khả năng maintain dễ dàng, đồng thời giảm thiểu tối đa các vấn đề do project sử dụng nhiều compiler, được viết bởi nhiều developer với mỗi phong cách lập trình khác nhau.

II. Coding convention

1. Các phong cách đặt tên phổ biến

Kiểu	Mô tả	Ví dụ
Pascal	Viết hoa chữ cái đầu tiên của mỗi từ.	PascalCase
Camel	Viết thường chữ cái đầu của từ đầu tiên. Các từ còn lại viết hoa chữ cái đầu.	camelCase
Uppercase	In hoa toàn bộ tên	UPPERCASE
Hungarian Notation	Gồm phần tiền tố chỉ kiểu dữ liệu hoặc đặc trưng của biến và phần tên với mỗi từ được bắt đầu bằng chữ in hoa.	iTuSo, strName

2. Quy tắc đặt tên class

1. Tên class sử dụng style là Pascal-case (các ký tự đầu tiên của từ sẽ là chữ Hoa).
   Ví dụ: `class ProductManagement;`
2. Tên class thể hiện được ý nghĩa của đối tượng mà nó thể hiện.
   Ví dụ: `class Cusomter`, `class Product`
   Lưu ý: Đối với một số project viết trên nền tảng Windows thì tên class sẽ thêm tiền tố C ở trước. Ví dụ: `class CCusomter`, `class CProduct`
3. Không nên viết tắt tên class (ngoại trừ trường hợp là các từ viết tắt có nghĩa, và cần có tài liệu thống kê các từ viết tắt).

3. Quy tắc về đặt tên biến, tên hàm/method

1. Đặt tên biến sử dụng style là Hungarian Notation (gồm phần tiền tố chỉ kiểu dữ liệu hoặc đặc trưng của biến và phần tên với mỗi từ được bắt đầu bằng chữ in hoa).
   Ví dụ: `string strMacAddress;`
2. Đặt tên hàm/method sử dụng style là Camel-case (từ đầu tiên sẽ viết thường, kể từ từ thứ 2 thì ký tự đầu tiên của từ sẽ viết hoa).
   Ví dụ: `Customer getCustomerInfo();`
3. Đặt tên biến, tên hàm/method thể hiện được ý nghĩa sử dụng.
   Ví dụ: `char szIpAddress[64];`
4. Không nên viết tắt trong tên biến, tên hàm/method (ngoại trừ trường hợp là các từ viết tắt có nghĩa, và cần có tài liệu thống kê các từ viết tắt).
5. Tên hàm/method cần thể hiện được hành động mà hàm/method sẽ thực hiện.
   Ví dụ: `Customer getCustomerInfo();` hoặc `bool parseIpAddress(…);`
6. Đặt tên method mang tính đối tượng hóa
   Ví dụ: Trong `class Customer` thì chỉ cần đặt tên method là `getInfo()` thay vì `getCustomerInfo()`.
7. Tiền tố của tên biến local sẽ đặt theo style Hungarian Notation như sau:
   bool sẽ có tiền tố là b, ví dụ: `bool bResult;`
   int sẽ có tiền tố là i, ví dụ: `int iCount;`
   long sẽ có tiền tố là l, ví dụ: `long lSize;`
   float sẽ có tiền tố là f, ví dụ: `float fRatio;`
   char sẽ có tiền tố là c, ví dụ: `char cKey;`
   đối với chuỗi cơ bản sẽ có tiền tố là sz, ví dụ: `char szIp6Address[64];`
   string sẽ có tiền tố là str, ví dụ: `string strAddress;`
   enum sẽ có tiền tố là e, ví dụ: `State eInstallState;`
   pointer sẽ có tiền tố là p, ví dụ: `char* pBuffer;`
8. Đối với tham số, có thể không cần đặt tiền tố như biến local nêu trên, nhưng cần thống nhất cho toàn project.
9. Đối với biến global cần thêm tiền tố g_ so với quy định biến local nêu trên, ví dụ: `int g_iFlow = 0;`
10. Đối với biến member của class sẽ thêm tiền tố _ so với quy định biến local nêu trên, ví dụ: `int _iFlow = 0;`
    Lưu ý: Đối với một số project viết trên nền tảng Windows thì biến member của class sẽ thêm tiền tố là m_. Ví dụ: `int m_iFlow = 0;`
11. Đối với kiểu dữ liệu pointer thì khai báo theo format như sau:
    `char* pBuffer;` hoặc `int* pNeeded;`

4. Quy tắc đặt tên enum và định nghĩa các giá trị hằng số

1. Tên enum và các giá trị của enum phải thể hiện được ý nghĩa sử dụng, tương tự với tên hằng số.
2. Tên enum sử dụng style là Pascal-case và các giá trị của enum sử dụng là Uppercase (viết hoa toàn bộ đối với các giá trị của enum, các từ phân cách bởi dấu gạch dưới).
   Ví dụ:
```
enum Color
{
  RED   = 1,
  GREEN = 2,
  BLUE  = 3
};
```
3. Tên hằng số sử dụng style là Uppercase (viết hoa toàn bộ đối với tên hằng số, các từ phân cách bởi dấu gạch dưới).
   Ví dụ:
```
const size_t PAGE_SIZE          = 8192;
```

5. Quy tắc khai báo struct

Khai báo một struct sẽ có format như sau:

- Tên của struct sử dụng style là Upper-case.
- Tên các biến thành phần của struct sử dụng style là Pascal-case và không cần đặt tiền tố xác định kiểu dữ liệu.

Ví dụ:
```
typedef struct _STRUCTNAME
{
    int StructMember1;
    int StructMember2;
    std::string StructMember3;
} STRUCTNAME, *PSTRUCTNAME;
```

6. Quy tắc đối với const và reference keyword

1. Đối với biến lưu trữ dữ liệu không thay đổi thì khai báo như sau:
   `const char* szMessage = "This action cannot be undone!";`
2. Đối với method không cho phép thay đổi giá trị các biến member thì khai kháo như sau:
   `Customer getCustomerInfo() const;`
3. Đối với parameter là input parameter (không cho phép thay đổi trong method) thì khai báo như sau: `Customer getInfoById(const int id);`

4. Đối với tham số là kiểu dữ liệu là object, struct, string truyền vào method/hàm thì khai báo như sau:

```
Customer compare(const Customer& other);
Customer compare(const Customer& other) const;
```

7. Thụt đầu hàng và cách khoảng

- Viết cách vào một khoảng tab đối với các lệnh nằm trong khối lệnh { }.
- Viết cách vào một khoảng tab đối với lệnh ngay sau if, else, while, for, foreach.
- Viết cách một khoảng trắng xung quanh các toán tử 2 ngôi và 3 ngôi.
- Viết cách một khoảng trắng sau dấu “,” và “;”

8. Một vài tiêu chí khác về coding style

1. Mỗi câu lệnh riêng rẽ trên một dòng.
   Ví dụ:

```
if(!bVideoInfo)
        return -1;
```

2. Sử dụng cặp dấu { } để đánh dấu một khối mã nguồn. Nên thống nhất một cách sử dụng cặp dấu {} trong suốt project. Trong các lệnh if, for, ... nếu chỉ có một lệnh thì có thể không cần đánh dấu khối mã nguồn.
3. Đối với điều kiện if giá trị Boolean thì không cần có toán tử so sánh. Ví dụ:

```
if(SUCCEEDED(hr)) {…}, if(bProcessing) {…}
```

9. Comment source code

- Sử dụng comment // cho các comment source code.
- Cần comment miêu tả các xử lý phức tạp hoặc dễ gây hiểu lầm
- Cần comment giải thích cho các biến có tên không rõ ràng, khó hiểu.

10. Comment cho method/hàm đối với các dự án tạo mới.

Format comment cho method/hàm sẽ như sau:

```
/*************************************************************
 * @Description Lấy thông tin video
 *
 * @return      Pointer trỏ đến video object
 * @attention   Tham khảo CIP_VideoDecode::VideoInfo
 * @attention   Trả về NULL nếu có lỗi xảy ra
 * @attention   Chẳng hạn trả về lỗi ERROR_INVALID_PARAM
 *************************************************************/
```

11. Các ngôn ngữ khác như SQL hoặc xml thuộc string của code.

Cần viết để dễ hiểu, dễ maintain, và tuân theo standard của ngôn ngữ đó.
Ví dụ về format xml trong chuỗi như sau:

```
const char* szXml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
```

```
"<root>"
    "<device>"
        "<model>%s</model>"
        "<mac>"
            "<string>%s</string>"
            "<value>%s</value>"
        "</mac>"
        "<node_name>"
            "<string>%s</string>"
        "</node_name>"
    "</device>"
"</root>";
```

Ví dụ về format SQL trong chuỗi như sau:

```
const char* szSql = "SELECT "
                    "Id, "
                    "FullName, "
                    "Address, "
                    "PhoneNumber, "
                    "Profile "
                "FROM Customer "
                "WHERE Id > 500 ";
```

12. Về format, bố trí thứ tự trong source code.

1. Biến dùng ở đâu sẽ khai báo ở đó, không khai báo tất cả ở trên đầu hàm như trong ANSI C. *(Từ khi C++ ra đời thì việc này coi như là một cải tiến nhằm block hóa các đoạn code trong chương trình C++)*.
2. Thứ tự biến, thứ tự các hàm nên bố trí sao nhìn cho đẹp mắt sẽ làm người review happy hơn.
3. Thứ tự các method/hàm trong file *.h và file *.cpp cần thống nhất (trong file *.h nằm ở vị trí nào thì file *.cpp cũng nằm ở vị trí đó)

13. Dùng switch đối với các kiểu dữ liệu enumerable

```
//Không nên viết code như sau:
 if(state == State::PAUSED)
 {
     //TODO: Do something
 }
 if(state == State::WORKING)
 {
     //TODO: Do something
 }
 if(state == State::STOP)
 {
```

```
        //TODO: Do something
    }

    //Nên viết code như sau:
    switch (state)
    {
        case State::PAUSED:
            break;
        case State::WORKING:
            break;
        case State::STOP:
            break;
        default:
            break;
    }
```

14. Thêm TODO Comment vào các vị trí chưa hoàn thành.

Các vị trí sẽ TODO comment gồm có:

1. Các phần source code cần implement nhưng chưa implement
2. Các phần source code cần phải xem xét lại về việc implement
3. Các method cần phải implement nhưng chưa implement
4. Các method mà không implement (empty method)

Format của TODO comment như sau:

```
//TODO: Not Implemented

//TODO: Need to review and check again

//TODO: Empty Method
```

15. Tối giản các câu lệnh if

Việc tối giản câu lệnh if sẽ giúp chính bạn maintain tốt hơn.

```
//Không nên viết source code phức tạp như sau
 if (img.Width > width && img.Height > height)
 {
     fProportion = proWidth > proHeight ? proWidth : proHeight;
 }
 else if (img.Width > width && img.Height < height)
 {
     fProportion = proWidth;
 }
 else if (img.Width < width && img.Height > height)
 {
     fProportion = proHeight;
 }
 else if (img.Width <= width && img.Height <= height)
```

```
{
    fProportion = 1;
}

//Nên viết source code đơn giản như sau:
fProportion = 1;
if (img.Width > width || img.Height > height)
{
    fProportion = proWidth > proHeight ? proWidth : proHeight;
}
```

16. Loại bỏ các câu lệnh if không cần thiết

```
//Không nên viết như sau
 if(g_bProcessed)
 {
     bProcessing = FALSE;
 }
 else
 {
     bProcessing = TRUE;
 }

 //Nên viết đơn giản như sau
 bProcessing = !g_bProcessed;
```

17. Có new thì phải có delete, có malloc thì phải có free

Thông thường thì các developer vẫn hay mắc phải sự cố này (đặc biệt là các developer quen viết code C# hoặc Java). Đây là nguyên tắc nhằm ngăn chặn memory, tuy nhiên đối với một chương trình lớn thì điều này vẫn chưa đủ mà cần thêm một nguyên tắc nữa "Object/function quản lý cấp phát thì Object/function đó phải đi dọn dẹp".

- Nếu new/malloc ở contructor thì sẽ delete/free ở destructor, nếu viết hàm để new/malloc thì cần có hàm để delete/free.
- Không nên new/malloc ở một module và delete/free ở một module khác.
- Trong trường hợp bắt buộc phải delete/free không đúng theo nguyên tắc trên thì phải có comment miêu tả để người review nắm được.

18. Có open thì phải có close

Nguyên tắc này không chỉ trong C/C++ mà tất cả các ngôn ngữ đối với việc quản lý resource (file, disk, …). Cũng tương tự như nguyên tắc về cấp phát bộ nhớ, thì đối với một chương trình lớn cần thêm nguyên tắc "Object/function quản lý đối tượng cần open thì Object/function đó phải close".

- Nếu open ở contructor thì sẽ close ở destructor, nếu viết hàm để open thì cần có hàm để close.
- Không nên open ở một module và close ở một module khác.

19. Không nên hard-code các magic number

Ví dụ như sau:

```
   //Không nên viết như sau, vì sau này maintain có thể không biết
3.14 là số gì
   fArea = fRadius * 3.14;

   //Nên viết như sau
   const float M_PI = 3.14;
   fArea = fRadius * M_PI;
```

20. Không lạm dụng biến global

Thường thì các developer mới hay lạm dụng biến global, nhưng việc này sẽ làm cho hệ thống trở nên khó bảo trì và khó extend nên cần phải hạn chế sử dụng tới mức thấp nhất.

21. Sử dụng biến local nếu có thể

Rất nhiều trường hợp các developer lạm dụng biến member hoặc global mặc dù biến đó chỉ sử dụng trong 1 method hoặc một hàm dẫn đến khó khăn cho việc maintain và dễ làm người khác hiểu nhầm ý nghĩa của biến.

22. Cần khởi tạo biến đã khai báo.

C/C++ không quy định việc biến khai báo sẽ khởi tạo giá trị nào dẫn đến việc có trường hợp Debug thì chạy, Release không chạy (hoặc ngược lại) dẫn đến mất thời gian fix bug. Ví dụ: cần phải khởi tạo giá trị null cho biến con trỏ.

23. Hạn chế viết function quá dài và các điều kiện lồng nhau nhiều.

Đối với một function quá dài và các điều kiện lồng nhau quá nhiều sẽ là một thử thách lớn đối với việc maintain. Vì thế cần chia nhỏ function và không sử dụng các điều kiện lồng nhau trong khi viết code.

24. Header file (*.h)

- Header file nên định nghĩa quy tắc để đảm bảo không bị include nhiều lần.

Phổ thông là quy tắc sau:

```
#ifndef MYHEADER_H_INCLUDED_
#define MYHEADER_H_INCLUDED_
//contents of the header file...
#endif
```

- Đối với Microsoft C++ (Visual C++) thì có thể dùng quy tắc sau ở đầu header file.

```
#pragma once
```

- Không sử dụng #include trong file *.h nếu không thật sự cần thiết.
- Không sử dụng using namespace trong file *.h nếu không thật sự cần thiết.

25. Không gộp nhiều class trong 1 file

- Việc gộp nhiều class trong 1 file cũng sẽ gây ra sự khó khăn đối với quá trình maintain. Ngay cả developer tạo ra nó một thời gian cũng không biết nó ở đâu.
- Hãy đặt tên file cùng tên với tên class.
- Trong C++ một class nên có file header (*.h) và file implementation (*.cpp).

26. Không sử dụng goto label

Việc sử dụng goto label sẽ làm cho code trở nên rất phức tạp để hiểu vì thế không được sử dụng trong bất kỳ hoàn cảnh nào.

27. Sử dụng this-> và classname::

Để dễ phân biệt với các hàm API, các macro khi sử dụng biến member hoặc gọi một method của chính nó, hãy sử dụng `this->` hoặc `ClassName::`
Ví dụ: `this->Start();this->_memberVar;` hoặc `Enginer::staticMethod();`

III. Các lỗi thường gặp

1. Thừa/thiếu điều kiện

Việc thừa hoặc thiếu điều kiện, không check null sẽ ảnh hưởng rất nhiều đến chất lượng chương trình. Ví dụ điển hình mà nhiều developer hay gặp phải như sau:

```
//Check thiếu điều kiện null
 Enginer* eng = this->_factory.GetEngier();
 if(eng != 0 && state == State::PAUSED)
 {
     eng->Resume();
 }
 else if(state == State::WORKING)
 {
     eng->Pause();
 }
 else
 {
     eng->Stop();
 }
```

Để không xảy ra hiện tượng check thừa thiếu, thì cần validate data trước khi sử dụng.

2. **Khai báo biến nhưng không sử dụng**

```
void MyFunction()
{
    std::string strComponents;
}
```

3. **Thiếu xử lý return, default trong switch case**

```
Color color = this->GetCurrentColor();
switch(color)
{
    case Red:
        return 2;
    case Green:
        return 0;
    case Blue:
    case Black:
        return 1;
}
```

4. **Không quan tâm đến warning của compiler**

- Không hiểu ý nghĩa của các warning
- Chương trình vẫn chạy được không quan tâm

Việc không quan tâm đến các warning của compiler sẽ có khả năng tạo ra bug "phong thủy" sau này. Ví dụ như warning biến không được khởi tạo, hay warning đối với function memcpy

5. **Trùng tên biến trong nested for**

Ví dụ như sau:

```
for (int i = 0; i < max ; ++i)
{
    for (i = 0; i < max ; ++i)
    {
        // Do something here
    }
}
```

6. **Viết code dài dòng, không cần thiết**

Ví dụ như sau:

```
//Không nên viết như sau:
int iPosition = 0;
```

```
iPosition = iLeft;
iPosition += iWidth;

//Nên viết như sau:
int iPosition = iLeft + iWidth;
```

Ví dụ 2:

```
//Không nên viết như sau:
std::string strFilePath = strHomePath;
strFilePath += PLUGIN_IO_FILE_PATH;

//Nên viết như sau:
std::string strFilePath = strHomePath + PLUGIN_IO_FILE_PATH;
```

7. **Copy lại code hoặc comment code nhưng quên không edit**

Ví dụ :

```
// Sai:
string strStudentName; // Name of Student
string strStudentAddress ; // Name of Student


// Đúng:
string strStudentName; // Name of Student
string strStudentAddress ; // Address of Student
```

8. **2 hàm làm chức năng tương tự nhưng format không thống nhất:**

Ví dụ:

```
// Không nên viết như sau
JobIDs getRecoverableRootJobIDs();
void getPausedRootJobIDs(JobIDs jobIDs);

// Nên viết như sau:
JobIDs getRecoverableRootJobIDs();
JobIDs getPausedRootJobIDs();
```

9. **Xử lý không tối ưu:**

Ví dụ:

```
for (std::map<long, long>::iterator i = jobMap.begin(); i != jobMap.end(); ++i) {
    jobMap.erase(i);
}

//Chỉ cần gọi lệnh sau thay cho đoạn code trên
jobMap.clear();
```

Ví dụ 2: Khi 1 hàm dài khoảng 100 line (lớn hơn 1 page màn hình) xử lý chung cho cả 2 trường hợp success và error thì nên tách thành 2 hàm riêng biệt: 1 hàm xử lý trường hợp success và 1 hàm xử lý trường hợp error. Ví dụ:

```
void writeFile();
// Tách thành 2 hàm:
void writeFileSuccess();
void writeFileError();
```

10.Không define const mà hardcode.

Ví dụ:

```
size_t newPageSize = 8192;

// đúng phải như sau:
const size_t DEFAULT_PAGE_SIZE = 8192;
size_t newPageSize = DEFAULT_PAGE_SIZE;
```

11. Include và using namespace nhưng không sử dụng

Ví dụ:

```
#include <stdlib.h>
using namespace clib;
```

Không sử dụng đến include và namespace này.

12. Đã dùng "using namespace std;", nhưng vẫn gọi "std::"

Ví dụ:

```
using namespace std;
std::string strStudentName;

// đúng thì chỉ cần như sau
string strStudentName;
```

13. Sử dụng == true là không cần thiết:

Ví dụ:

```
if (bFlag == true) {
      ...
}

// nên sử dụng như sau:
if (bFlag) {
      ...
}
```

14. Dấu {} không thống nhất

Ví dụ, như sau là không thống nhất :

```
if (bFlag) {
      ...
}

if (iNum > 1)
{
      ...
}
```

15. Không nên chia ra giữa việc khai báo biến, khởi tạo và nhập giá trị

Ví dụ:

```
CTestClass *testClass = 0;
testClass = getInstanceTestClass();

// gộp xử lý lại như sau
CTestClass *testClass = getInstanceTestClass();
```

16. Chia cho giá trị 0

Ví dụ:

```
//Có thể bị lỗi divide by zero trong trường hợp iRecordPerPage là 0
int iPage = iTotalRecords / iRecordPerPage;
```

17. Sử dụng đường dẫn tuyệt đối trong code

Ví dụ:

```
//Viết code như sau là không nên
this->saveToTempFolder("C:\\Windows\\Tmp");
```

18. Sử dụng pointer không đúng

Ví dụ, gán pointer sang vùng nhớ khác và không kiểm soát vùng nhớ cấp phát ban đầu:

```
CTestClass *testClass = new CTestClass();
//Do other job
testClass = manager->getInstanceTestClass();
//Do something
delete testClass;
```

THAM KHẢO

1. Fsoft Coding Standard (2015)
2. http://geosoft.no/development/cppstyle.html
3. https://gist.github.com/lefticus/10191322
4. http://www.possibility.com/Cpp/CppCodingStandard.html
5. https://www.mantidproject.org/C++_Coding_Standards