C++ Coding Guide

1. Style
    a. Header file
    b. Naming
    c. Block
    d. Indentation
    e. Class
    f. Comment
2. Recommendation

---

1. Style

a, Header file

- Tạo file theo custom boilder plate được cài đặt sẵn vào Qt. Header file tự động gen sẽ có format. Nếu không cài đặt boilder plate thì cop mẫu header này.

```
/**
 *============================================================================
 * Project: %{ProjectName}
 * Module: rand.h
 * Module Short Description:
 * Author: %{AuthorName}
 * Date: 20/05/2020 15:41
 * Organization: Viettel Aerospace Institude - Viettel Group
 * ============================================================================
 */
```

b, Naming

Quy tắc thống nhất cách đặt tên file, class, biến, ... trong nhóm.

- **File name:** Tên file phải viết hoa đầu các chữ cái. **Đặt trùng với tên class.**

```
file_example.h, fileExample.h // Bad
FileExample.h                 // Good
```

- **Class, struct name:** Tên class **phải trùng với tên file**. Viết hoa chữ cái đầu, không chứa _

```
// File name: ClassExample.h
class class_example {} // Bad
class classExample {}  // Bad
class ClassExample {}  // Goood
```

- **Function name:** Bắt đầu bằng động từ và viết thường. Tuôn theo style CamelCase

```
void DoSomething(...)   // Bad
void do_something(...)  // Bad
```

```
void doSomething(...)   // Good
```

- **Variable**
  - **Class property**: Bắt đầu bằng **m_** để thể hiện đó là biến của lớp

```
// Bad
class X {
    private:
        int firstProp;
}

// Good
class X {
    private:
        int m_firstProp;
}
```

  - **constant**: Viết hoa tất cả các chữ cái

```
const pi = 3.14 // Bad
const PI = 3.14 // Good
```

  - **static:** Bắt đầu bằng **s_**

```
static int myStaticVar = "xxx"    // Bad
static int s_myStaticVar = "xxx"  // Good

// In case: combine both class property and static
class X {
    private:
        static X* ms_inst;
}
```

  - **global**: Bắt đầu bằng **g_**
  - **function parameter:** Bắt đầu bằng **t_** để thể hiện đó là params của hàm.

```
// Bad
function myFunc(int param1, std::string _param2, bool Param3_ ){}
// Goood
function myFunc(int t_param1, std::string t_param2, bool t_param3) {}
```

  - **Các biến thường khác**: Bắt đầu bằng chữ cái thường và theo style camelcase.

b, Block (bracket) style

Cặp bracket đặt xuống 1 dòng riêng biệt.

```
// Bad
class XXX {

}

void xyz(...) {

}
```

```
if (...) {

}

// Good
class XXX
{
}

void xyz(...)
{

}

if (...)
{

}
```

c, Indentation

- Độ rộng code tối đa 100. Set up ở Qt creator. Tools → Options → Text Editor → Display
- Code trong các block (bracket) **lùi 1 tab**. (Sử dụng độ rộng tab mặc định của Qt)

```
// Bad
class X
{
public:
....
}

// Goood
class X
{
    public:
        ...
}
```

- Cặp ngoặc () cách các từ khóa if, while, for ... 1 khoảng trắng

```
//Bad
if(...)
for(...)
while(...)

// Good
if (...)
for (...)
while (...)
```

- Các đối số trong hàm cách dấu () 1 khoảng trắng.

```
// Bad
void badFunc(int param1, param2)
{}

// Good
void goodFunc( int param1, int param2 )
{}
```

- Con trỏ và tham chiếu đặt cạnh kiểu dữ liệu

```
// Bad
int * a = nullptr;
int *a = nullptr;
int &a
int & a
// Good
int* a = nullptr;
int& a
```

d, Class

- Thứ tự của 1 class: public → protected → private
- Các biến của lớp phải luôn để private

e, Comment

- Comment hàm

```
/**
 * Get the size (in bytes) of the specified file.
 *
 * @param path the path of the file
 * @return if successful, the size of the file in bytes
 *         otherwise, 0 will be returned.
 *
 * @ingroup filesystem
 */
size_t fileSize( const char* path );
```

- Comment dòng: Sử dụng //

```
// fileSize
size_t Utils::File::fileSize( const char* path )
{
    // Check path existence
    ...

    // Get file information

    return;
}
```

2, Recommendation

- **Sắp xếp thứ tự các điều kiện trong câu lệnh *'if'***

Nếu xác xuất nhận giá trị true/false của các điều kiện con bên trong là tương đương nhau thì nên đặt các điều kiện đơn giản, có thời gian xử lý nhanh lên trước, đặt các điều kiện phức tạp, có thời gian xử lý lâu hơn ở phía sau
Nếu xác suất nhận giá trị true/false của các điều kiện con bên trong là chênh lệch nhau thì

  - Với phép "AND", ví dụ (A && B): Nên đặt điều kiện có xác suất nhận giá trị false nhiều hơn lên trước.
  - Với phép "OR", ví dụ (A || B): Nên đặt điều kiện có xác suất nhận giá trị true nhiều hơn lên trước

- **Sử dụng lookup table thay cho câu lệnh switch**

```c
int n = …;

switch (n) {
case 0:
    printf("Alpha");
    break;
case 1:
    printf("Beta");
    break;
case 2:
    printf("Gamma");
    break;
case 3:
    printf("Delta");
    break;
default:
    break;
}
```

```c
static char const * const
Greek[4] = {
    "Alpha",
    "Beta",
    "Gamma",
    "Delta"
};

int n = …;

if (n >= 0 && n < 4) {
    printf(Greek[n]);
}
```

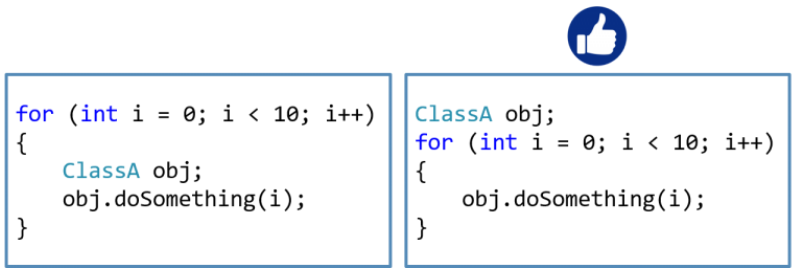
- **Sử dụng 'switch' thay vì một loạt các lệnh 'if'**

```c
if (a == 1) {
    f();
}
else if (a == 2) {
    g();
}
else if (a == 5) {
    h();
}
else {
    k();
}
```

```c
switch (a)
{
case 1:
    f();
    break;
case 2:
    g();
    break;
case 5:
    h();
    break;
default:
    k();
    break;
}
```

- **Tối ưu phạm vi của biến**

Nếu biến là một đối tượng của một class và được sử dụng bên trong vòng lặp nhưng không bị thay đổi trong vòng lặp thì nên khai báo biến ngay trước vòng lặp. Điều này là để tránh việc hàm khởi tạo và hàm hủy của đối tượng được gọi liên tục một cách không cần thiết trong mỗi lần lặp.

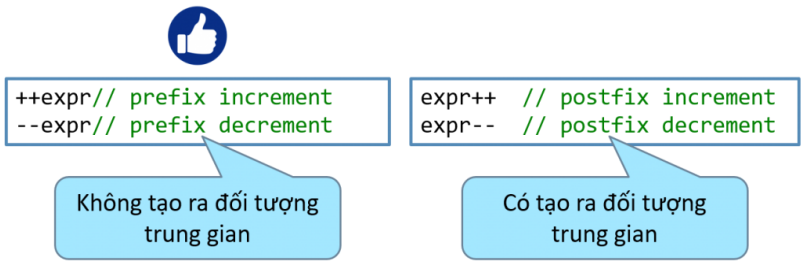


```
for (int i = 0; i < 10; i++)
{
    ClassA obj;
    obj.doSomething(i);
}
```

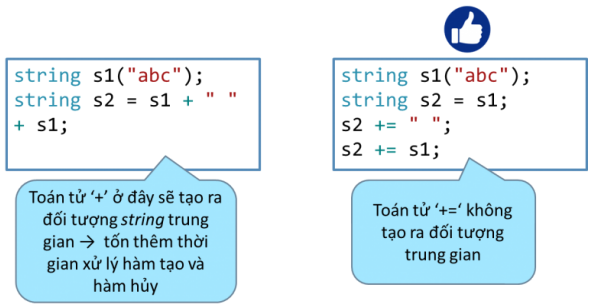
```
ClassA obj;
for (int i = 0; i < 10; i++)
{
    obj.doSomething(i);
}
```

- **Sử dụng toán tử tăng/giảm**

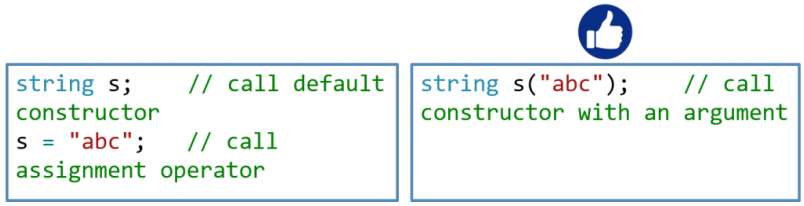
Khi sử dụng toán tử tăng/giảm ở dạng hậu tố thì khi chạy chương trình sẽ tạo ra đối tượng trung gian, do đó làm tăng thời gian xử lý (do phải gọi hàm copy constructor và hàm hủy), thời gian xử lý đó sẽ khá đáng kể với những biến là đối tượng của class. Vì vậy nên sử dụng toán tử tăng/giảm ở dạng tiền tố thay vì hậu tố trong trường hợp giá trị của biểu thức không được sử dụng.



Nên sử dụng toán tử gán kết hợp toán tử số học thay vì sử dụng toán tử toán học và toán tử gán riêng biệt



Khi khởi tạo đối tượng, nên sử dụng hàm khởi tạo thay vì toán tử gán



```
string s;    // call default
constructor
s = "abc";   // call
assignment operator
```

```
string s("abc");    // call
constructor with an argument
```

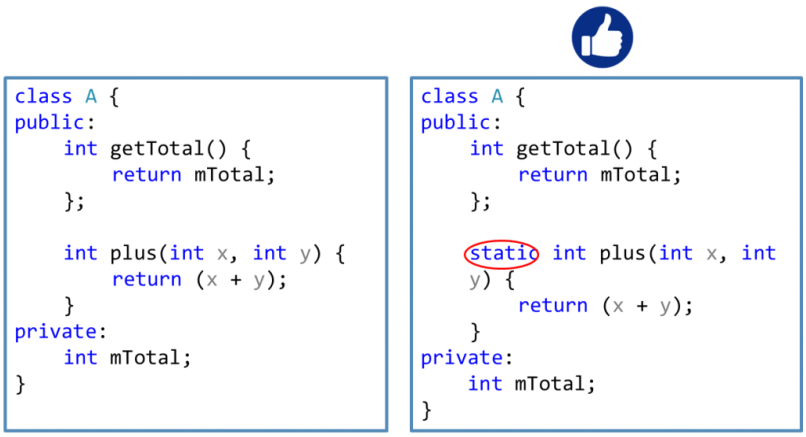
- **Nguyên tắc truyền tham số cho hàm**

Khi truyền một biến *x* có kiểu dữ liệu là *T* vào một hàm như là tham số *func* thì nên tuân theo nguyên tắc sau:

  - Nếu *x* là tham số chỉ dùng để đọc (input-only):
    - Nếu *x* có kích thước lớn và có thể nhận giá trị *NULL* → Truyền bằng con trỏ trỏ tới hằng số: *func(const T* x);*
    - Nếu *x* có kích thước lớn nhưng không thể nhận giá trị *NULL* → Truyền bằng tham chiếu hằng số: *func(const T& x);*
  - Nếu *x* là tham số dùng để output hoặc cả input và output
    - Nếu *x* có thể *NULL* → Truyền bằng con trỏ: *func(T* x);*
    - Nếu *x* không thể nhận giá trị *NULL* → Truyền bằng tham chiếu: *func(T& x)*

- **Sử dụng hàm thành viên static**

Trong các class, với các hàm thành viên không truy cập vào biến non-static thì nên implement các hàm đó dưới dạng hàm *static*. Bởi vì khi call hàm *static* thì không cần truyền tham số ngầm định *this* vào cho hàm, điều đó giúp tiết kiệm memory và giảm thời gian xử lý

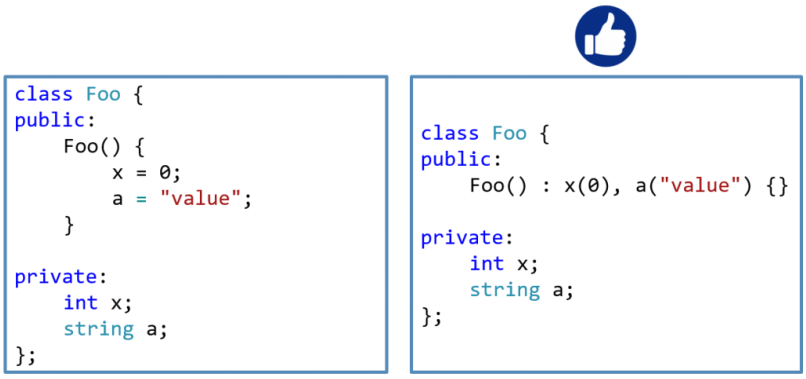


```
class A {
public:
    int getTotal() {
        return mTotal;
    };

    int plus(int x, int y) {
        return (x + y);
    }
private:
    int mTotal;
}
```

```
class A {
public:
    int getTotal() {
        return mTotal;
    };

    static int plus(int x, int
    y) {
        return (x + y);
    }
private:
    int mTotal;
}
```

- **Sử dụng Constructor Initialization List**

Trong hàm khởi tạo của class, đối với các biến thành viên có kiểu dữ liệu là class thì tốt nhất là nên khởi tạo chúng trong cái gọi là *ConstructorInitialization List* thay vì gán giá trị cho biến đó bên trong hàm khởi tạo.

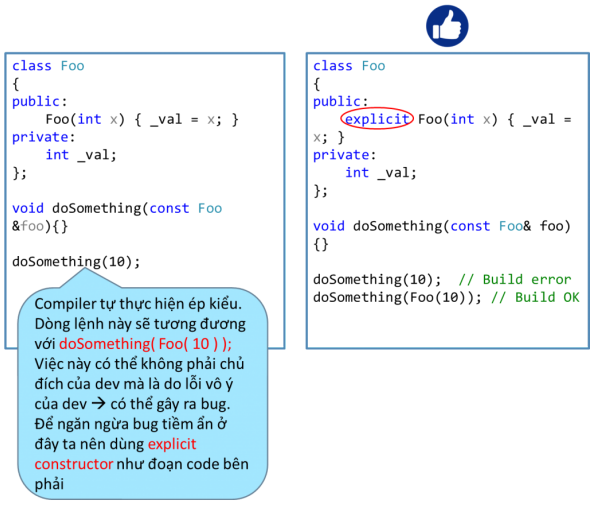


```
class Foo {
public:
    Foo() {
        x = 0;
        a = "value";
    }

private:
    int x;
    string a;
};
```

```
class Foo {
public:
    Foo() : x(0), a("value") {}

private:
    int x;
    string a;
};
```

Việc khởi tạo cho biến *a* như hình đoạn code bên trái sẽ tốn 2 công đoạn, đầu tiên là khởi tạo *a* với hàm khởi tạo mặc định của nó, sau đó gán cho nó giá trị là *"value"*. Tuy nhiên nếu code theo như đoạn code bên phải thì biến *a* sẽ được khởi tạo luôn chỉ với một lần call hàm khởi tạo 1 tham số, nhờ đó rút ngắn được thời gian xử lý.

- **Nên sử dụng *explicit constructor* đối với tất cả hàm khởi tạo một tham số, ngoại trừ copy constructor.**



```
class Foo
{
public:
    Foo(int x) { _val = x; }
private:
    int _val;
};

void doSomething(const Foo
&foo){}

doSomething(10);
```

Compiler tự thực hiện ép kiểu. Dòng lệnh này sẽ tương đương với doSomething(Foo(10)); Việc này có thể không phải chủ đích của dev mà là do lỗi vô ý của dev → có thể gây ra bug. Để ngăn ngừa bug tiềm ẩn ở đây ta nên dùng explicit constructor như đoạn code bên phải

```
class Foo
{
public:
    explicit Foo(int x) { _val =
x; }
private:
    int _val;
};

void doSomething(const Foo& foo)
{}

doSomething(10);  // Build error
doSomething(Foo(10)); // Build OK
```

- **Nguyên tắc khi sử dụng "smart pointers"**

- Smart pointer là 1 class wrap lại raw pointer của C++. Mục đích chính của việc sử dụng smart pointer là để đảm bảo đối tượng được xóa và memory được giải phóng khi đối tượng không còn được sử dụng nữa. → Tóm lại là ngăn ngừa leak memory.
- Có 3 loại smart pointers trên C++11:
    unique_ptr: Không làm giảm performance khi sử dụng
    shared_ptr: Làm giảm performance khi sử dụng
    weak_ptr: Làm giảm performance khi sử dụng

Chính vì sử dụng smart pointer có thể làm giảm performance nên cần phải cẩn thận khi sử dụng, không sử dụng bừa bãi. Vậy khi nào nên/không nên sử dụng smart pointer ? Hãy xem

> Nếu việc cấp phát và giải phóng đối tượng / memory là do cùng 1 hàm hoặc một class chịu trách nhiệm thì không cần thiết phải sử dụng smart pointer.

Chỉ nên sử dụng smart pointer trong trường hợp một đối tượng / memory được cấp phát bởi một hàm và sau đó bị xóa bởi một hàm khác và hai hàm này không liên quan đến nhau (không phải là hàm thành viên của cùng một class)

- **Truy cập bộ nhớ theo chiều thứ tự tăng dần của địa chỉ**

Thực tế cho thấy bộ nhớ cache của CPU hoạt động hiệu quả nhất khi dữ liệu được truy cập tuần tự theo chiều thứ tự tăng dần của địa chỉ. Nó hoạt động kém hiệu quả khi dữ liệu được truy cập ngược và ít hiệu quả hơn nữa khi dữ liệu được truy cập một cách ngẫu nhiên. Điều này áp dụng cho việc đọc cũng như ghi dữ liệu. Chính vì vậy nên truy cập bộ nhớ theo chiều thứ tự tăng dần của địa chỉ.

- **Sử dụng union để tiết kiệm bộ nhớ**

Nếu 2 hoặc nhiều biến thành viên của structure không bao giờ được sử dụng cùng một thời điểm thì nên sử dụng *union* thay vì *struct* để chia sẻ vùng nhớ giữa các biến này → tiết kiệm bộ nhớ.

- **Phương pháp sắp xếp các trường dữ liệu của structure để tiết kiệm bộ nhớ**

Nên sắp xếp các biến thành viên của class/structure từ trên xuống dưới theo chiều giảm dần của kích thước, điều đó sẽ giúp làm giảm kích thước của class/structure