C++ Code Style

Style

سازگاری مهمترین جنبه style است. دومین جنبه مهم، پیروی از سبکی است که برنامه نویس C++ معمولی به خواندن آن عادت دارد.

C++ اجازه می دهد تا نام های identifier با طول دلخواه را انتخاب کنید، بنابراین دلیلی برای کوتاه کردن نام آن ها وجود ندارد. از نام های توصیفی استفاده کنید و در style ثابت باشید.

- CamelCase
- snake_case

این دو استایل نمونه های رایج هستند. snake_case این مزیت را دارد که در صورت تمایل می تواند با spell checkers نیز کار کند.

ایجاد یک دستورالعمل Style

هر دستورالعمل Style که ایجاد می کنید، مطمئن شوید که یک فایل با فرمت clang. که سبک مورد انتظار شما را مشخص میکند، پیاده سازی کنید. در حالی که این نمی تواند به نامگذاری کمک کند، به ویژه برای یک پروژه منبع باز حفظ یک Style ثابت مهم است.

هر IDE و بسیاری از ویرایشگرها از فرمت clang پشتیبانی می کنند یا به راحتی با یک افزونه قابل نصب هستند.

- VSCode: Microsoft C/C++ extension for VS Code
  CLion: https://www.jetbrains.com/help/clion/clangformat-as-alternative-formatter.html
  VisualStudio https://marketplace.visualstudio.com/items?itemName=LLVMExtensions.ClangFormat#review-details
- C++: Resharper https://www.jetbrains.com/help/resharper/2017.2/Using_Clang_Format.html
- Vim
  - https://github.com/rhysd/vim-clang-format
  - https://github.com/chiel92/vim-autoformat
- XCode: https://github.com/travisjeffery/ClangFormat-Xcode

قراردادهای رایج نامگذاری در C ++

- تایپ با حروف بزرگ شروع می شوند: MyClass
- توابع و متغیرها با حروف کوچک شروع می شوند: myMethod
- ثابت ها همه حروف بزرگ هستند: const double PI= 3.14159265358979323

کتابخانه استاندارد C++ (و دیگر کتابخانه‌های معروف C++ مانند Boost) از این دستورالعمل‌ها استفاده می‌کنند:

- نام‌های ماکرو از حروف بزرگ با underscores استفاده می‌کنند: INT_MAX
- نام پارامترهای الگو از camel case استفاده می کند: InputIterator
- همه نام‌های دیگر از snake case استفاده می‌کنند: unordered_map

متمایز کردن پارامترهای تابع

مهمترین چیز ثبات در codebase شما است. این یک امکان برای کمک به ثبات است.

پارامترهای تابع را با پیشوند t_ نامگذاری کنید. t_ را می توان به عنوان "the" در نظر گرفت، اما معنایش قراردادی است. نکته این است که پارامترهای تابع را از سایر متغیرها در scope متمایز کنیم و در عین حال یک استراتژی نامگذاری ثابت به ما ارائه می دهد.

```
struct Size
{
  int width;
  int height;

  Size(int t_width, int t_height) : width(t_width), height(t_height) {}
};

// This version might make sense for thread safety or something,
// but more to the point, sometimes we need to hide data, sometimes we don't.
class PrivateSize
{
  public:
    int width() const { return m_width; }
    int height() const { return m_height; }
```

```
    PrivateSize(int t_width, int t_height) : m_width(t_width), m_height(t_height) {}

  private:
    int m_width;
    int m_height;
};
```

نام چیزی را با _ شروع نکنید

اگر این کار را انجام دهید، در معرض ریسک برخورد با نام های رزرو شده برای استفاده از کامپایلر و اجرای کتابخانه استاندارد هستید:
http://stackoverflow.com/questions/228783/what-are-the-rules-about-using-an-underscore-in-a-c-identifier

Well-Formed Example

```
class MyClass
{
public:
  MyClass(int t_data)
    : m_data(t_data)
  {
  }

  int getData() const
  {
    return m_data;
  }

private:
  int m_data;
};
```

build های خارج از دایرکتوری source را فعال کنید
مطمئن شوید که فایل های تولید شده به پوشه خروجی جدا از پوشه source می روند.

از nullptr استفاده کنید
Nullptr C++11 را یک مقدار ویژه معرفی می کند که نشانگر null را نشان می دهد. این باید به جای 0 یا NULL برای نشان دادن یک اشاره گر تهی استفاده شود.

Comments

بلوک های Comment باید از // استفاده کنند نه /* */. استفاده از // باعث می‌شود در هنگام debugging، کامنت کردن یک بلوک کد بسیار آسان‌تر شود.

```
// this function does something
int myFunc()
{
}
```

برای کامنت کردن این بلوک تابع در حین debugging، ممکن است این کار را انجام دهیم:

```
/*
// this function does something
int myFunc()
{
}
*/
```

غیر ممکن خواهد بود اگر کامنت بالای تابع از /* */ استفاده شود.

هرگز از namespace در Header File استفاده نکنید

این باعث می شود namespace که استفاده می کنید به namespace همه فایل هایی که شامل فایل هدر هستند کشیده شود. فضای نام را آلوده می کند و ممکن است در آینده منجر به تداخل نام شود. هر چند نوشتن با استفاده از namespace در فایل پیاده سازی خوب است.

Include Guards

فایل‌های header باید حاوی یک guard با نام مشخص باشند تا از مشکلاتی در چندین بار قرار دادن هدر یکسان و جلوگیری از تداخل با هدرهای پروژه‌های دیگر جلوگیری شود.

```
#ifndef MYPROJECT_MYCLASS_HPP
#define MYPROJECT_MYCLASS_HPP

namespace MyProject {
  class MyClass {
  };
}

#endif
```

همچنین ممکن است به جای آن از دستور #pragma Once استفاده کنید که در بسیاری از کامپایلرها شبه استاندارد است. کوتاه است و هدف را روشن می کند.

{} برای بلوک ها مورد نیاز است

کنار گذاشتن آنها می تواند منجر به خطاهای معنایی در کد شود.

```
// Bad Idea
// This compiles and does what you want, but can lead to confusing
// errors if modification are made in the future and close attention
// is not paid.
for (int i = 0; i < 15; ++i)
  std::cout << i << std::endl;

// Bad Idea
// The cout is not part of the loop in this case even though it appears to be.
int sum = 0;
for (int i = 0; i < 15; ++i)
  ++sum;
  std::cout << i << std::endl;


// Good Idea
// It's clear which statements are part of the loop (or if block, or whatever).
int sum = 0;
for (int i = 0; i < 15; ++i) {
  ++sum;
  std::cout << i << std::endl;
}
```

طول خط هایتان را معقول نگه دارید

```
// Bad Idea
// hard to follow
if (x && y && myFunctionThatReturnsBool() && caseNumber3 && (15 > 12 || 2 < 3)) {
}

// Good Idea
// Logical grouping, easier to read
if (x && y && myFunctionThatReturnsBool()
    && caseNumber3
    && (15 > 12 || 2 < 3)) {}
```

بسیاری از پروژه ها و استانداردهای کدنویسی دارای یک دستورالعمل ساده هستند که باید کمتر از 80 یا 100 کاراکتر در هر خط استفاده شود. خواندن چنین کدی به طور کلی آسان تر است. همچنین این امکان را فراهم می کند که دو فایل مجزا در کنار هم در یک صفحه بدون داشتن فونت ریز وجود داشته باشد.

از "" برای Include کردن فایل های local استفاده کنید
<> برای include فایل های سیستمی میباشد

```
// Bad Idea. Requires extra -I directives to the compiler
// and goes against standards.
#include <string>
#include <includes/MyHeader.hpp>

// Worse Idea
// Requires potentially even more specific -I directives and
// makes code more difficult to package and distribute.
#include <string>
#include <MyHeader.hpp>


// Good Idea
// Requires no extra params and notifies the user that the file
// is a local file.
#include <string>
#include "MyHeader.hpp"
```

Initialize Member Variables

```
// Good Idea
// There is no performance gain here but the code is cleaner.
class MyClass
{
public:
  MyClass(int t_value)
    : m_value(t_value)
  {
  }

private:
  int m_value;
};

// Good Idea
// The default constructor for m_myOtherClass is never called here, so
```

```
// there is a performance gain if MyOtherClass is not is_trivially_default_constructible.
class MyClass
{
public:
  MyClass(MyOtherClass t_myOtherClass)
    : m_myOtherClass(t_myOtherClass)
  {
  }

private:
  MyOtherClass m_myOtherClass;
};
```

در C++11 می توانید مقادیر پیش فرض را به هر عضو اختصاص دهید (با استفاده از = یا با استفاده از {}).

تخصیص مقادیر پیش فرض با کروشه

```
// Best Idea

// ... //
private:
  int m_value{ 0 }; // allowed
   unsigned m_value_2 { -1 }; // narrowing from signed to unsigned not allowed, leads to a compile time error
// ... //
```

مقدار اولیه {} را بر = ترجیح دهید مگر اینکه دلیل محکمی برای انجام ندادنش داشته باشین.

فراموش کردن مقداردهی اولیه یک عضو منبع bug های رفتاری تعریف نشده است که پیدا کردن آنها اغلب بسیار سخت است.

همیشه از namespace استفاده کنید

تقریباً هرگز دلیلی برای declare an identifier در global namespace وجود ندارد. در عوض، توابع و کلاس‌ها باید در یک namespace مناسب یا در کلاسی در داخل namespace وجود داشته باشند. شناسه‌هایی که در global namespace قرار می‌گیرند ریسک مغایرت با شناسه‌های کتابخانه‌های دیگر را دارند (بیشتر C که فضای نامی ندارد).

هرگز Tabs و فاصله ها را با هم ترکیب نکنید

برخی از ویرایشگرها به طور پیش‌فرض دوست دارند با ترکیبی از tabs و فاصله‌ها indent ایجاد کنند. این باعث می‌شود کد برای کسی که دقیقاً از همان تنظیمات indent تب استفاده نمی‌کند، غیر قابل خواندن باشد. ویرایشگر خود را پیکربندی کنید تا این اتفاق نیفتد.