C++ Coding Standards

郑华

2017 年 2 月 8 日

目录

第一章　代码管理

1.1　Compile Cleanly at High warning levels

1.2　*Use a version Control system (Git)

Summary　:

The palest of ink is better than the best memory.

Use a version cotrol system.

Never keep files checked out for long periods.

Check in freequently after your updated unit tests pass.

Ensure that checked-in code does not break the build.

1.3　Invest in code reviews

Summary　:

Re-view code:More eyes will help make more quality. Show your code,and read others'.You will all learn and benefit.

第二章　设计风格

2.1 Give one entity on cohesive responsibility

Summary ：单一职责原则

Focus on one thing at a time:Prefer to give each entity(variable, class, function, namespace, module, library) one well-defined responsibility.

As an entity grows,its scope of responsiblity naturally increases, but its responsibility should bot diverge.

2.2 Correctness, simplicity, And clarity come first

Summary ：

Keep it Simple Software:

Correct is better than fast.

Simple is better than Complex.

Clear is better than cute.

Safe is better than insecure.

2.3 Know when and how to code for scalability

Summary ：

Be aware of explosice datea growth:Without optimizing prematurely, keep an eye on asymptotic complexity.

Algorithms that work on user data should take a predictabl and preferably no worse thanlinear,time with the amount of data processed.

When optimization is provably necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.

2.4 Don't optimize prematurely

Summary ：

Spur not a willing horse:Premature optimization is as addictive as it is unproductive.

This first rule of optimization is:Don't do it.

The second rule of optimization is :Don't do it yet.

Measure Twice,Optimize Once.

2.5 Don't pessimize prematurely

Summary :

Easy on yourself,easy on the code: All other things being equal,notably code complexity and readablility,certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives.

It is avoiding gratuitous pessimization.

2.6 Minimize global and shared data

Summary :

Sharing causes contention: Avoid shared data,especially global data.

Shared data increase coupling(耦合),which reduces maintainablility and often performance.

2.7 Hide information

Summary :

Don't tell:Don't expose internal information from an entity that provides an abstraction.

2.8 *Know when and how to code for concurrency(并发)

Summary :

Thread safely:If your application uses multiple threads or processes,konw how to minimize sharing objects where possible and share the right ones safely.

2.9 Ensure resources are owned by objects

Summary :

Don't saw by hand when you have power tools:C++'s "Resouce Acquisition Is Initialization(RAII)" idiom is the power tool for correct resource handing.

RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms.

When allocating a raw resource,Immediately pass it to an owning object.

Never allocate more than one resource in a single statement.

Use explicit(显式) RAII and smart pointers

第三章　编程方式

3.1 Prefer compile- and link-time errors to run-time errors

Summary :

Don't put off until run time what you can do at build time:Prefer to write code that uses the compiler to check for invariants during compilation,instead of checking them at run time.

Run-time checks are control- and data-dependent,which means you will seldom konw whether they are exhausive.

In contrast,Compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.

3.2 Use const proactively

Summary :

const is your friend: Immutable values are easier to understand,track,and reason about,so prefer constants over variables whereever it is sensible and make const your default choice when you define a value:It's safe,it's checked at compile time,and it's intergrated with c++'s type system.

Don't cast away const except to call a const-incorrect function.

3.3 Avoid macros(宏命令)

Summary :避免使用宏

Macros are the bluntest instrument of C and C++'s abstraction facilities, ravenous wolves in function's clothing , hard to tame, marching to their own beat all over your scopes. Avoid them.

Macros remain the only solution for a few important tasks, such as **#include** guards,**#ifdef** and **#if defined** for conditional compilation,and implementing assert.

3.4 Avoid Magic numbers

Summary :

Programming isn't magic,so don't incant it:Avoid spelling literal constants like **43** or **3.1415** in Code.

They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication.

Use symbolic names and expressions instead,such as **width*aspectRatio**.

3.5 Declare variables as locally as possible

Summary :

Avoid scope bloat, as withrequirements so too with variables: Variables introduce state, and ou should have to deal with as little state as possible, with lifetimes as short as possible.

3.6 Always initialize variables

Summary :

Start with a clean slate：Uninitialized variables are a common source of bugs in C and C++ programs.

Avoid such bugs by being disciplined about **cleaning memory before you use it**;

Initilize variables upon definition.

3.7 Avoid long functions,Avoid deep nesting

Summary :

Short is better than long, Flat is better than deep: Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility, and both are usually solved by better refactoring.

3.8 Avoid initialization dependencies across compilation units

Summary :

keep (initialization) order:Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined.

Doing otherwise causes headaches ranging from mysterious crashes when you make small changes in your projects to severe non-portablility even to new releases of the same compiler.

3.9 Minimize definitional dependencies,Avoid cyclic dependencies(循环依赖)

Summary :

Don't be over-dependent:Don't #include a definition when a forward declaration will do.

Don't be co-dependent:Cyclic dependencies occur when two modules depend directly or indirectly on one another.

A module is a cohesive unit of release;

Moudules that are interdependent are not really individual modules, but superglued together into what's really a larger module, a larger unit of release.

Thus,cyclic dependencies work against modularity and are a bane of large projects. Avoid them

3.10 Make header files self-sufficient

Summary :

Behave responsibly: Ensure that each header you write is compilable standalone(每个头文件都能够独自的进行编译), by having it include any headers its contents depend upon.

3.11 Always write internal #include guards. Never wirte External #include guards.

Summary :

Weaer head(er) protection: Prevent unintended multiple inclusions by using #include guards with unique names for all of your header files.

第四章　函数与操作符

4.1 Take parameters appropriately by value, (smart) pointer, or reference

Summary ：

Parameterize well: **Distinguish** among input,output,and input/output parameters, and **between** value and reference parameters.

Take them appropriately.

4.2 Preserve natural semantics for overloaded operators

Summary ：

programmers hate surprises: Overload operators only for good reasons, and preserve natural semantics; if that's difficult,you might be misusing operator overloading.

4.3 Prefer the canonical forms of arithmatic and assignment operators

Summary ：

if you provide A+B, then also A+=B

4.4 Prefer the canonical form of ++ and –,Prefer calling the prefix forms

Summary ：

if you provide ++c; then also provide c++;

4.5 Consider overloading to avoid implicit type type conversions

Summary :

4.6 Avoid overloading &&, ||,or,(comma)

Summary :

4.7 Don’t write code that depends on the order of evaluation of function arguments

Summary :

Keep evaluation order: The order in which arguments of a function are evaluated is unspecified, so don’t rely on a specific ordering.

如：Func(++count, ++count); 这两个执行的顺序是不一定的.

第五章　类的设计与继承

5.1 Be clear what kind of class you're writing

Summary :

5.2 Prefer minimal classes to monolithic classes

Summary :

5.3 Prefer composition to inheritance

Summary :

5.4 Avoid inheriting from classes that were not designed to be base classes

Summary :

5.5 Prefer providing abstract interfaces

Summary :

5.6 Public inheritance is substitutablity.Inherit,not to reuse,but to be reused.

Summary :

5.7 Practice safe overriding

Summary :

5.8 Consider making virtual functions nonpublic,and public functions nonvirtual

Summary :

5.9 Avoid providing implicit conversions

Summary :

5.10 Make data members private,except in behaviorless aggregates(C-style structs)

Summary :

5.11 Don't give away your internals

Summary :

5.12 Pimpl judiciously

Summary :

5.13 Prefer writing nonmember nonfriend functions

Summary :

5.14 Always provide new and delete together

Summary :

第六章　构造函数、析构函数与拷贝构造

6.1 Define and initialize member variables in the same order

Summary :

6.2 Prefer initialization to assignment in constructors

Summary :

6.3 Avoid calling virtual functions in consturctors and destructors

Summary :

6.4 Make base class desturctors public and virtual,or protected and nonvirtual

Summary :

6.5 Destructors,deallocation,and swap never fail

Summary :

6.6 Copy and destroy consistently

Summary :

6.7 Explicitly enable or disable copying

Summary :

6.8 Avoid slicing.Consider Clone instead of copying in base classes.

Summary :

6.9 Prefer the cannoical form of assignment

Summary :

6.10 Whenever it makes sense,provide a no-fail swap(and provide it correctly)

Summary :

第七章　命名空间与模块

7.1 Keep a type and its nonmember function interface in the same namespace

Summary :

7.2 Keep types and functions in separate namespaces unless they're specifically intended to work together

Summary :

7.3 Don't write namespace usings in a header file or before an #include

Summary :

7.4 Avoid allocationg and deallocating memory in different modules

Summary :

7.5 Don't define entities with linkage in a header file

Summary :

7.6 Don't allow exceptions to propagate across module boundaries

Summary :

7.7 Use sufficiently portable types in a module's interface

Summary :

第八章　错误处理与异常

8.1 Assert liberally to ducument internal assumptions and invariants

Summary ：

8.2 Establish a rational error handling policy,and follow it strictly

Summary ：

8.3 Distinguish between errors and ono-errors

Summary ：

8.4 Design and write error-safe code

Summary ：

8.5 Prefer to use exceptions to report errors

Summary ：

8.6 Throw by value,catch by reference

Summary ：

8.7 Report,handle,and translate errors appropriately

Summary ：

8.8 Avoid exception specifications

Summary :

第九章　类型安全

9.1　Avoid tupe switching; prefer polymorphism(多态)

Summary ：

9.2　Rely on types,not on representations

Summary ：

9.3　Avoid using reinterpret_cast

Summary ：

9.4　Avoid using static_cast on pointer

Summary ：

9.5　Avoid casting away const

Summary ：

9.6　Don't use C-style casts

Summary ：

9.7　Don't memcpy or memcmp ono-PODs

Summary ：

9.8　Don't use unions to reinterpret representation

Summary ：

9.9 Don't use varargs(可变长参数)

Summary :

9.10 Don't use invalid objects

Summary :

9.11 Don't use unsafe functions

Summary :

9.12 Don't treat arrays polymorphically(多态的)

Summary :