目录

[一、 C++命名规范 7](#_Toc2009379785)

[1. file name 文件命名 7](#_Toc2046789908)

[2. class & struct name 类/结构体命名 8](#_Toc1790193006)

[3. variable & argument name 变量、参数命名 8](#_Toc1159945160)

[4. function name 函数命名 10](#_Toc1366000663)

[5. enumeration name 枚举命名 11](#_Toc1922716500)

[6. macro name 宏定义名 11](#_Toc868358167)

[7. typedef /using 声明类型别名、函数指针等 11](#_Toc996393907)

[8. name abbreviation rules 命名缩写规则 12](#_Toc1879988624)

[二、 头文件 14](#_Toc1253029561)

[1. 使用#include防止重复包含 14](#_Toc1213458613)

[2. 使用前向声明减少文件依赖 14](#_Toc313178967)

[3. 内联函数 16](#_Toc1849077785)

[4. 使用.inl.hpp文件 17](#_Toc1412968944)

[5. 包含文件的次序 18](#_Toc1565094937)

[三、 作用域 20](#_Toc209039510)

[1. 命名空间namespace 20](#_Toc415113076)

[2. 普通函数（关于缺省参数） 22](#_Toc1294629626)

[3. 全局变量 23](#_Toc591307801)

[4. 局部变量 26](#_Toc90128046)

[四、 类 28](#_Toc2036259056)

[1. 结构体和类 28](#_Toc2005612825)

[2. 嵌套类 30](#_Toc290952001)

[3. 默认构造函数 33](#_Toc1638079573)

[4. 使用explicit避免隐式转换 35](#_Toc518265572)

[5. 禁止拷贝和禁止重载“=” 36](#_Toc1674412664)

[6. 防止运算符歧义 39](#_Toc643655838)

[7. 虚函数 41](#_Toc2091743112)

[8. 析构函数总是为虚 42](#_Toc1418113664)

[9. 构造函数和析构函数中不能调用虚函数 44](#_Toc1095893577)

[10. 派生类调用基类函数 47](#_Toc1268013393)

[11. 抽象类 49](#_Toc1280009802)

[12. 类成员 51](#_Toc995199837)

[13. 何时使用多重继承 58](#_Toc910722752)

[14. 优先使用组合代替继承 60](#_Toc292471314)

[15. 继承不要扩到到集合层面 64](#_Toc213716852)

[五、 数据操作 65](#_Toc685955604)

[1. Aggregates类型 65](#_Toc1160829482)

[2. POD类型 67](#_Toc1210110759)

[3. 推荐初始化列表 73](#_Toc418460581)

[4. 避免变量的不确定值 80](#_Toc266375395)

[5. 空指针使用nullptr 82](#_Toc276085724)

[6. 类型转换 84](#_Toc731639548)

[7. const的用法 89](#_Toc2115453181)

[8. 0值比较 94](#_Toc1689054668)

[9. 尽量避免#define 97](#_Toc149250838)

[10. 逗号表达式 101](#_Toc177009043)

[六、 内存管理 103](#_Toc2104167744)

[1. new/delete、new[]/delete[] 103](#_Toc1443880464)

[2. 智能指针 103](#_Toc768316844)

[3. 容器中指针的管理 111](#_Toc46812142)

[4. 对象池 113](#_Toc1332655873)

[5. 内存越界 115](#_Toc626446021)

[6. 使用RAII技术 115](#_Toc337764144)

[七、 面向对象 118](#_Toc823251798)

[1. 调用关系 118](#_Toc1144711594)

[2. 单例模式 & 控制实例的数量 122](#_Toc2012176808)

[3. Single Responsibility Principle单一性原则 126](#_Toc1466907636)

[4. Liskov Substitution Principle替换原则原则 130](#_Toc1088971058)

[5. Dependency Inversion Principle依赖倒置原则 132](#_Toc1282806825)

[6. Open Closed Principle开闭原则 133](#_Toc415317565)

[7. Interface Segregation Principle接口分离 136](#_Toc209500804)

[八、 异常 140](#_Toc415332979)

[1. 只需要传递错误堆栈信息 140](#_Toc1410517402)

[2. 在catch{}中throw异常 142](#_Toc1120223556)

[3. UI层捕获异常并定义Handlers 143](#_Toc707804293)

[4. 中间层/底层抛出异常的同时定义Handler 144](#_Toc1624234255)

[5. 注意事项 146](#_Toc1806179160)

[九、 禁止/慎用事项 148](#_Toc1868633775)

[1. 禁止goto 148](#_Toc686861366)

[2. 慎用友元 148](#_Toc77156093)

[3. 禁止返回局部变量的引用 149](#_Toc2135009171)

[4. 禁止编译警告 150](#_Toc962947091)

[5. 禁止头文件using 151](#_Toc808795642)

[6. 禁止使用magic number 152](#_Toc2102978704)

[7. 禁止将函数的输入参数作为工作变量 152](#_Toc504518111)

[十、 鼓励事项 154](#_Toc958046480)

[1. 鼓励优先使用静态断言调试 154](#_Toc132504099)

[2. 鼓励优先使用标准库 154](#_Toc461202208)

[3. 鼓励优先使用前缀操作符 156](#_Toc254443296)

[4. 鼓励函数内局部变量离第一次使用越近越好 156](#_Toc900820943)

[5. 鼓励使用列表初始化 157](#_Toc508014350)

[6. 鼓励优化循环 158](#_Toc1587099169)

[7. 鼓励优先使用STL循环算法代替手写循环 159](#_Toc1527266965)

[8. 鼓励使用表驱动法 160](#_Toc845778494)

[十一、 代码版面 163](#_Toc262867319)

[1. 注释 163](#_Toc524494911)

[2. 代码格式 170](#_Toc710471655)

[十二、 QML 175](#_Toc1729774956)

[1. QML文件 & Component使用注意事项 175](#_Toc1613465969)

[2. Class、Struct & Enum 178](#_Toc1993278480)

[3. Property属性 178](#_Toc2145092521)

[4. Function函数 184](#_Toc1822966773)

[5. Variable变量 184](#_Toc261127811)

[6. Comment 注释 186](#_Toc1408126276)

[7. 常见QT控件缩写 190](#_Toc795706681)

[8. QtCreator 中翻译注意事项 194](#_Toc968932104)

# C++命名规范

## file name 文件命名

1. 文件名全部小写，不要包含下划线(\_)或短线(-)

举例:

framequeue.hpp, mat.cpp

1. C++头文件的后缀名用“.hpp”

举例:

mat.hpp, matrix.hpp

1. C 语言头文件的后缀名用“.h”

举例:

epnp.h, rho.h, p3p.h

备注:

如下情况皆做为 C 语言头文件对待：

* 包含了C语言标准库的头文件，如include <stddef.h>，include<stdio.h>等。
* 间接包含了C语言头文件的头文件，且该头文件将来可能会与.c文件配对编译纯C程序。比一个头文件include “type.h”,而“type.h”内又有include<stdio.h>等C语言标准库或其它C语言头文件时，并且该头文件需要考虑到不仅匹配.cpp进行混合编程，也要考虑与.c配对纯C编程时，那么该头文件也需要以.h结尾。

1. 如果头文件在公司内部使用只会与.cpp 配对编译不考虑与C配对编译时，那仍建议采用.hpp 后缀即可。
2. 用以实现对应头文件中模板类、模板函数、内联函数的文件名请以.inl.hpp 作为后缀名

举例:

mat.inl.hpp

1. 由于git版本管理及QtCreator对中文的支持不是很好，所有文件路径(包括所有的文件名和目录名)禁止出现**中文，**也不能是‘-’和空格。规范起见，规定只能是字母、数字或者‘\_’的组合，其中‘\_’不能打头。

## class & struct name 类/结构体命名

1. 每个单词得首字母大写，不允许包含下划线

举例:

class UrlTable {…}, struct TableProperty{…}

1. 抽象类需添加大写 I 作为前缀以示区别

举例:

class ICar {…}, class IMotionCard{…}

1. 派生类需将基类名作为后缀

举例:

class EcoSystem : public System{…}

class StepMotor : pulic IMotor{…}

//派生类与基类都是抽象类，无需把抽象基类的 I 放到后缀

class IBigCar : pulibc ICar{…}

特例:

class Circle : public Shape{…}

众所周知，从字面含义即可理解 Circle 是 Shape 的一种细分类型，这种情况下可以不必遵循派生类将基类作为后缀的命名方法。

1. 通常类名是名词属性，但如果只是一组函数接口为主的抽象类，也可以采用形容词属性

举例:

class INonCopyable{…}

class IEnumerable{…}

## variable & argument name 变量、参数命名

1. 函数内局部变量、函数参数采用驼峰命名法则

举例:

height, userName, studentAge, urlText

void setUserName(string name){…}

void setStudentAge(int age){…}

1. 类或结构体内成员变量采用驼峰命名法则，以“m\_”作为前缀

举例:

m\_height, m\_userName, m\_studentAge, m\_urlText

1. 全局变量采用驼峰命名法则，以“g\_”作为前缀

举例:

g\_height, g\_userName, g\_studentAge, g\_urlText

1. const、static 修饰的变量、参数的应用类型多、频次高，因此只需根据实际情况遵守上述 1) & 2)点命名即可，无需额外特殊标示

举例:

【静态类成员变量】

static int m\_sharedCount;

【静态全局变量】

static int count;

Car(const Car& car){…}

void setAge(const int& age){…}

int age( )const{…}

1. 原始指针类型的变量与参数需要使用字母 p 加以标示

举例:

【静态全局变量】static int \*pUserCount;

【普通全局变量】extern int \*g\_pUserCount;//需加关键字extern

【类成员变量】int \*m\_pUserCount;

【类静态成员变量】int \*m\_pUserCount;//无需特别标示

【函数参数】void setUserCount( int\* pUserCount)

1. 智能指针会自动释放内存，因此无需特殊标示

举例:

std::shared\_ptr<cv::Mat> m\_backFrame;

std::shared\_ptr<cv::Mat> m\_frontFrame;

1. 变量、参数命名一般应当是名词属性
2. 数组、STL 容器的变量名请根据英语拼写规则在尾部增加“s”或“es”

举例:

int nums[5] = {0，7，2，3，5};

std::vector<string> m\_names;

std::map<string, int> m\_objects;

std::list<Pad> m\_pads; //容器含原始指针的记得含字母p

std::set<Tomato\*> m\_pTomatoes;

std::queue<QPair<Qstring, QString>> pendingObjs

## function name 函数命名

1. 采用驼峰命名法则

举例:

void detectFile( );

void find( );

void findNext( );

1. 对读取函数只需将成员变量名中m\_前缀去除，其余保持一致即可

举例:

私有成员变量 QRect m\_rect;

对应读取函数 QRect rect() const;

1. 对私有成员变量写入函数需将成员变量名中 m\_前缀替换为 set，其余保持一致

举例:

私有成员变量 QRect m\_rect;

对应写入函数 void setRect(const QRect& rect);

1. 函数命名建议采用“动词”或“动词+名词”的方式，其间可以结合形容词与副词

举例:

void move( ); //vi

void moveFast( ); //vi + adv

void runProduction( ); //vt + n

void driveLongJourney( ); //vt + adj + n

## enumeration name 枚举命名

1. 枚举类型名采用首字母大写、枚举值则采取单词全部大写，词与词之间采用“\_”连接

举例:

enum UrlTableError{OK = 0, OUT\_OF\_MEM= 1}；

1. 枚举类型名与枚举值一般都采用名词属性

## macro name 宏定义名

全部大写，单词之间用“\_”连接

举例:

#define PI 3.1415

#define ADD (a)+(b)

#define PI\_ROUNDED 3.0

特例:

涉及系统、跨平台、编程语言等条件编译宏定义时有很多成俗的情况，比如使用双下划线甚至全部小写：

#ifdef \_\_cplusplus

#ifdef \_\_MINGW\_NAME\_AW

## typedef /using 声明类型别名、函数指针等

1. 别名采取首字母大写原则

举例:

【类型别名】

typedef QList<QMimeType>::Iterator Iterator;

using WordCount = QMap<QString, int>

【函数指针】

typedef void(\*StartAddressFunc)(int, int);

using StartAddressFunc = void(\*)(int,int)

备注:函数指针无需特别标示 p 或 Ptr 这类字样

1. 指针类型别名需将 Ptr 作为后缀（包括智能指针）

举例:

typedef Pad\* PadPtr;

using FiducialMarkPtr = FiducialMark\*;

typedef std::shared\_ptr<Shape> ShapePtr;

1. 容器类型别名需后缀容器名称

举例:

typedef QHash<QString, QModelIndex> NameIndexHash;

typedef std::vector<std::string> StringVector;

using ChannelMap = std::map <Name, Channel>;

typedef std::map<int, PeerTracker> PeerTrackerMap;

typedef std::list<int\*> IntPtrList //尖括号有指针需加上 Ptr

using FloatSet = std::set<float>;

1. 申明类型别名与函数指针等通常采用名词属性

## name abbreviation rules 命名缩写规则

1. 只接受常见缩写，禁止初级程序员无法理解的缩写

好的缩写:

|  |  |
| --- | --- |
| √ command -> cmd | √ message -> msg |
| √ parameter -> param | √ argument -> arg |
| √ source -> src | √ destination -> dest |
| √ string -> str | √ image->img |
| √ form -> frm | √ control -> ctrl |
| √ average -> avg | √ password -> pwd |
| √ statistic -> stat | √ user -> usr |
| √ server -> srv | √ temparory -> tmp/temp |
| √ initial -> init | √ library -> lib |
| √ manager -> mgr | √ picture -> pic |
| √ position -> pos | √ pointer -> ptr |
| √ error -> err | √ information -> info |
| √ increment -> inc | √ insert -> ins |
| √ flag -> flg | √ point -> pt |
| √ database -> db | √ function -> func |
| √ return -> rtn |  |

错误缩写:

|  |  |
| --- | --- |
| × shared -> shrd | × shared pointer ->sp |
| ×list -> ls | × unique -> uq |
| ×decrement -> de | × window ->wndw |

1. 静止使用去元音缩写法（只保留辅音）

举例:

|  |  |
| --- | --- |
| ×create ->crt | ×function ->fnctn |

特例:

部分 UI 控件命名，其控件类型缩写作为前缀或后缀时可以破例使用，比如

|  |  |
| --- | --- |
| tbxName | //tbx 代表 textBox |
| lblResult | //lbl 代表 label |
| infoDlg | //Dlg 代表 Dialog |

# 头文件

## 使用#include防止重复包含

为了防止重复包含头文件，规定在声明每个头文件时采用#ifndef宏定义方式。

举例:

【方式一：只被微软支持（**禁止**使用）】

|  |  |
| --- | --- |
| #program once | //只被微软的VC++支持 |
| … | //声明、定义语句 |

【方式二：真正跨平台的方式，C、C++等都支持（**必须**使用）】

|  |  |
| --- | --- |
| //以下两行代码是加载整个头文件的开始 | |
| #ifndef PROJECT\_PATH\_FILE\_H | |
| #define PROJECT\_PATH\_FILE\_H | |
| …… | //声明、定义语句 |
| #endif | //加载整个头文件最后一行 |

## 使用前向声明减少文件依赖

通过使用前向声明，减少文件之间的编译依赖，节省编译时间。

前向声明是一种不完全的类型声明，当需要知道对象的大小和内容时，就不能使用了，所以前向声明的应用场景，仅限如下:

* 前向声明的类的指针/引用作为其它类成员变量。
* 前向声明的类的实例作为其它类的静态成员变量。
* 前向声明的类的实例作为函数声明的参数/返回值。
* 前向声明的类的指针/引用作为函数定义参数/返回值，且在定义中没有用到类的任何成员变量或者成员函数。

举例:

~~#include “student.h”~~ //省略了头文件依赖

class Student; //前向声明，减少编译依赖

class Teacher: Student //× 前向声明不能作为基类

{

public:

...

//>>>---------------------------------------------------------------------

/\*√ 前向声明的类的指针/引用作为函数定义参数/返回值，且在定

义中没有用到类的任何成员变量或者成员函数\*/

Student\* appointMonitor(){...};

Student& appointViceMonitor(){...};

void addStudent(Student\*){...};

void addStudent(Student&){...};

//<<<---------------------------------------------------------------------

//>>>---------------------------------------------------------------------

//√ 前向声明的类的实例作为函数声明的参数/返回值。

void addStudent(Student);

Student getNewStudnet();

//<<<---------------------------------------------------------------------

//>>>---------------------------------------------------------------------

//× 前向声明的类的实例不能作为函数实现的参数/返回值

void removeStudent(Student newStudent)

{

...

//× 不能调用到类的成员变/成员函数

newStudent.register();

...

};

Student getHighestStudent(){...};

//<<<--------------------------------------------------------------------

…

private:

//>>>---------------------------------------------------------------------

//√ 指针/引用可以作为类成员变量

vector<Student\*> m\_pStudents;

Student\* m\_monitor { nullptr };

//√ 静态成员变量可以使用实例

static Student m\_assistant;

//<<<---------------------------------------------------------------------

//× 实例对象不能作为类的成员变量

Student m\_viceMonitor;

...

}

## 内联函数

由于节约了函数调用的开销，使用内联函数在一定程度上能够提高执行效率，但是这以代码膨胀（复制）为代价。如果函数体执行的开销，远远大于函数调用的开销，那么效率的收获会很有限。所以盲目使用内联是非常不明智的做法，在使用内联函数时，应该注意如下几点:

* 代码小于等于5行，没有调用到其它函数(其它函数可能会发生修改，所以最好就是禁止调用到其它函数)；
* 没有递归、复杂的if/switch语句、for/while循环等调用；
* inline关键字必须写在定义而非声明的地方，否则编译时无效；

现在编译器会根据实际情况来决定是否编译为inline，有时候写了inline关键字仍会作为非inline函数处理，而没有inline关键字也有可能作为inline函数来处理，这取决于不同编译器的优化原则。

错误案例:

|  |  |
| --- | --- |
| inline void setIndex(int index)  {  ...  } | 1. setIndex在cpp中 **或** 2. 超过5行 **或** 3. 递归 **或** 4. 复杂if语句 **或** 5. 循环 **或** 6. switch语句 **或** 7. 其他函数 |

正确案例:

class Person

{

public:

int age()const{return m\_age;}

//对于简单的逻辑，如存取函数，直接放在头文件中实现

void setAge(int age){ m\_age = age;}

…

private:

int m\_age;

…

}

只要超过一行的代码(一行代码一般就是存取函数了)放在.inl.hpp中实现，具体做法见下一章节。

注意:

普通的访存函数用内联函数，但是用于C++和Qml进行交互的类，因为和Qml交互就有效率损失，加上在这些访存函数中经常有数据同步，信号发送等行为，所以全部在源文件中实现，不用内联函数。

## 使用.inl.hpp文件

类模板、函数模板及内联函数不能在.cpp中实现。类内实现会破坏声明的简洁性。类外实现又会让.hpp文件变得冗长。比较好的做法是将它们单独封装在一个**.inl.hpp**文件中。

举例:

//文件cuda.**inl.hpp**(from OpenCV)

inline GpuMat& GpuMat::operator =(const GpuMat& m)

{

if (this != &m)

{

GpuMat temp(m);

swap(temp);

}

return \*this;

}

注意:

* inl.hpp会在对应的头文件尾部include(包含在头文件中)，因此本质上是头文件的一部分，因此同样需要防止重复包含，请记得采用跨平台宏定义方式来确保这一点，具体做法请参见:

[二/1使用＃include防止重复包含](#_使用#include防止重复包含)。

* 对于简单的逻辑(如存取函数)，直接放在hpp中实现，具体做法请参见：

[一/3. variable & argument name 变量、参数命名](#_variable & argument name 变量、参数命名)。

* 如果需要包含inl.hpp文件, 请在头文件的最后include。

目前的Qt Creator版本中（Qt Creator 4.5.0）对于inl文件支持并不是很好，不能够在inl文件中进行单步调试(F11)，所以内联函数还是建议在**头文件中直接定义**。同时为了最大程度保持简洁性，除了内联函数，其它函数全部在**类外实现**。

## 包含文件的次序

良好的文件包含次序风格可增强可读性、避免隐藏依赖，规范的文件包含次序如下:对应该源文件声明的头文件、C++库、C库、其他库的.h、项目内的.h。

对应该源文件声明的头文件、标准库、第三方库、本项目其它头文件等不同的类型层次的包含文件应该空一行来区别。

相同目录下的头文件按字母进行排序，但注意较老的代码可以不符合这条规则，主要为了后续的调式改正方便。

举例:

imageTool.cpp包含的头文件顺序如下:

|  |  |
| --- | --- |
| #include “imageTool.hpp” | //对应该cpp的头文件 |
|  | |
| //注意，上面需要另起一行 | |
| #include <stdio> | // C库 |
| #include <iostream> | // C++库 |
|  | |
| //注意，上面需要另起一行 |  |
| #include <boost/any.hpp> | //第三方通用库 |
| #include <cvMat> | //第三方专业库 |
|  | |
| //注意，上面需要另起一行 | |
| //相同目录下按照字母顺序进行排列 | |
| #include “**f**ileTool.hpp” | //项目内头文件 |
| #include “**i**oTool.hpp” | //项目内头文件 |
| #include “**o**therTool.hpp” | //项目内头文件 |

# **作用域**

## 命名空间namespace

namespace可以很好的解决命名污染问题，同时也为逻辑分层提供了更多的选择性，使用namespace必须遵守如下规范:

1. 对于非UI项目至少有一个主namespace

项目最上层有一个主namespace，这主要是为了该项目被调用时调用方很清晰的知道调用关系，同时还可以避免项目间的命名冲突。如OpenCV中的CV，Boost中的boost。

特例:

最上层的UI(App)项目可以不需要拥有主namespace，因为其本身在绝大多数情况下是不可能再被其它项目调用了。

非UI项目不用namespace的，需要和主管申请获得批准方可执行。确定不需要namespace的项目，为了避免命名污染，所有类型都加一个约定的前缀(如QT都是以Q开头)。

1. 职责清晰，功能不重叠

尽量做到职责清晰，namespace之间不产生功能重叠，同一层级的namespace不产生耦合。

举例:

|  |  |
| --- | --- |
| namespace | Description |
| cv | 核心模块，所有视觉模块都封装在这里 |
| cvflann | OpenCV中调用到的一个第三方库（最近邻算法库） |
| cvv | 用来配合调试OpenCV |
| matlab | 提供一些工具，能够让算法生成为matlab算法 |
| NcvCTprep | 和CUDA加速相关 |
| Tinyxml2 | 操作xml相关 |

如上表，OpenCV在最上层的namespace设计其实就一个cv核心模块，其他namespace都是围绕cv服务的，几大namespace在功能上没有重叠交叉。

1. 嵌套namespace

namespace可以进行嵌套，不管是C#、Java还是C++，这里强调下C#和C++就namespace的一些区别：

* C#的namespace是默认和目录进行关联，并且会默认按照目录的结构建立命名上的关系。如在ImageProcess/ SearchOutLine目录下创建CircleSearcher 类，编译器会按照当前目录结构自动为该类加上namespace。

举例:

namespace ImageProcess.SearchOutLine

{

class CircleSearcher {}

}

* 对于C++的namespace和目录不会任何关系，也不会自动创建，全部需要靠自己手动编写。同样的使用上面的例子，namespace就要手动打上。

举例:

namespace ImageProcess

{

namespace SearchOutLine

{

class CircleSearcher {}

...

}// End of namespace SearchOutLine

}// End of namespace ImageProcess

1. 不具名namespace

如果仅仅把一个全局函数或者全局变量的作用域限制在单个cpp中，使用不具名namespace。使用不具名namespace具有如下优点:

* 支持namespace嵌套，可以对函数进行逻辑归类。
* 属于内链接，避免不同文件间的命名冲突。
* 提高效率，不具名命名空间中的变量和函数相当于内联。

举例：

namespace

{

void checkCompression (Compression compression)

{

…

}

} // End of namespace

## 普通函数（关于缺省参数）

C++在声明函数原型时，可为一个或多个参数指定缺省参数值，以后调用此函数，若省略其中某一参数，C++自动地以缺省值作为相应参数的值。

1. 只能在头文件中声明时设置默认参数

举例:

void setNumber( int num = 1); //默认参数num=1

1. 缺省参数都必须是从右到左连续定义，中间不能跳开

错误举例:

//× 编译错误，height被跳开

void setStudentInfo ( int id, **int age = 7**, int height)

{

…

}

正确举例:

//√ 正确写法，缺省参数从右到左连续定义

void setStudentInfo (int id, int height, **int age = 7**) {…}

1. 缺省参数在定义中一定要特别注释，使用/\*\*/注释

举例:

void Student::setNumber( int num /\*= 1\*/);

任何项目第一版不准使用缺省函数，明确的输入参数有助于避免程序员因为粗心漏写的参数而引发的隐性bug。

错误举例:

//Student.hpp中setStudentInfo函数age缺省默认值为1

void setStudentInfo(int id,int age /\*= 1\*/);

//程序员忘记了age的输入参数

Student lili;

lili.setStudentInfo(2);

当程序员调用setStudentInfo时，忘记了age的输入，此时是能够编译通过并正常运行的，因为编译器是无法判断调用方是忘记了age的输入还是本来就想要使用缺省的默认值，这无疑增加了程序员犯错的概率。

正确的做法是避免缺省参数，明确每一个参数的输入。

正确举例:

void setStudentInfo(int id,int age);

//程序员必须输入所有参数才能保证编译通过，降低了出错率

Student lili;

lili.setStudentInfo(2,20);

在修改升级时为了向下兼容推荐使用缺省参数函数，这样可以保证在不需要修改原有调用代码的基础上进行扩展。

举例:

//代码修改升级，增加了weight 和height 的参数输入

void setStudentInfo(int id,int age,double weight =

120,double height = 170);

//原来的调用代码不需要修改，lili默认weight为120,height为170

Student lili;

lili.setStudentInfo(2,20);

//如果需要另外设置weight，height的地方，只需要手动输入

Student wang;

wang.setStudentInfo(2,20,130,180);

## 全局变量

C++的全局变量有两种，分别用static和extern修饰。

1. static全局变量

对于只希望单个源文件内起作用的全局变量用static。static全局变量声明在头文件中，源文件中进行定义并初始化。因为static全局变量每次进入源文件都会进行初始化，所以static全局变量只会在源文件范围内起作用，不同源文件内的static全局变量既不会命名冲突，也不会相互影响。

举例:

//Image.hpp，头文件中声明了静态全局变量

static string ImgsFolderPath;

...

//Fov.cpp，在源文件中初始化并使用

#include “Image.hpp”

Fov::export()

{

ImgsFolderPath= "../Fov/Image";

...

}

//Job.cpp，在源文件中初始化并使用

#include “Image.hpp”

Job::export()

{

ImgsFolderPath= "../Job/Image";

...

}

注意:

对于静态全局变量static，可以使用不具名的namespace代替，不具名namespace和static全局变量同样属于内链接，能够将变量的作用域限制在源文件范围内，同时不具名namespace具有更好的封装、更好支持泛型(在namespace中可以声明类/结构体作为模板参数类型)，职责更加单一(static可以修饰函数，局部变量)。

1. extern 全局变量

extern告诉编译器存在着这样一个变量或者一个函数，如果在当前编译语句中没有找到变量或函数的定义，说明会在其它文件中定义。

extern全局变量声明在头文件中，源文件中进行定义并初始化。和static全局变量不同，extern只会进行一次初始化，所以extern的作用域跨文件，而static的作用域只是限定在当前的源文件中。

举例:

//image.hpp，头文件中声明了全局变量

extern int g\_imgId;

//Fov.cpp，在源文件中初始化并使用

#include “Fov.hpp”

Fov::export()

{

g\_imgId = 1;

...

}

//Job.cpp，在源文件中初始化并使用

#include “Fov.hpp”

Job::export()

{

cout<<g\_imgId<<endl; //输出为1

/\*g\_imgId为2，所有引用Image.hpp的源文件

中g\_imgId全部为2\*/

g\_imgId = 2;

...

}

在上面的举例中，Job和Fov的源文件中都会存在共享的g\_imgId变量，改变了其中一个变量的值影响的另一个源文件。

1. static/extern 只能修饰内置类型

无论是类外的全局变量(static/extern)，还是类内的静态成员变量，建议只使用语言自带的基本类型，无论是用户自定义类型，还是类似STL标准库中的类等除了在特殊情况下都不要使用。

举例:

|  |  |
| --- | --- |
| static Fov fov; | 【全局变量(自定义类型)】 //× |
| extern string flag; | 【全局变量(标准库)】  //× |
| static vector<int> m\_ids; | 【成员变量(标准库)】 //× |
| static vector<int> IDs; | 【全局变量(标准库)】 //× |

## 局部变量

在函数中定义的变量称为局部变量，作用域是定义该变量的“{}”内，使用局部变量需要注意以下事项:

1. 虽然局部变量和全局变量、成员变量可以同名，但公司禁止这么做，因为容易产生混淆。

举例:

#include <iostream>

int number { 9 };

void main( )

{

int number { 6 };

//输出6，这里代表的是局部变量的number

std::cout<<number<<std::endl; //输出6

//输出9，特别注意“::”，这里代表的是全局变量的number

std::cout<<**::number**<<std::endl;

}

注意

1. 如果发生同名的情况，全局变量和成员变量必须通过::来确定，这很容易发生混淆。
2. 如果按照第一章的命名规范来写，根本就不会出现局部变量和全局变量重名的情况，所以要求变量严格按照规定命名。
3. 尽量靠近使用的地方声明并初始化局部变量。

错误举例:

//× 局部变量离实际使用的地方太远

int number {6}；

//省略号代表此处包含了大量代码

…

std::cout<<number<<std::endl;

正确举例:

…

int number {6}; //√ 在最靠近使用的地方声明并初始化

std::cout<<number<<std::endl;

1. 静态局部变量只进行一次初始化，每次执行函数时保持上一次执行时的值,它和全局变量的区别在于全局变量全局范围内可见，而静态局部变量只有定义自己的函数体可见。

举例:

int addValue( int value)

{

//不管调用几次函数，静态局部变量sum只初始化一次

static int sum = 0;

sum = sum + value;

}

# 类

## 结构体和类

在C++中，struct与class主要差别有以下几点：

1. 默认访问权限

struct的默认访问权限为public，class的默认访问权限为private。

注意:

虽然class在不写访问权限修饰符的情况默认为private，编译并不会出错，但是公司还是要求显式的指定数据成员的访问权限修饰符，除非没有该访问权限的成员变量，比如在没有继承关系的类中，就可以省略protected。

1. 列表的初始化

如果自定义了构造函数，无论是类还是结构体，列表初始化调用合适的构造函数。

举例:

【自定义了构造函数的struct】

**struct** Student

{

int m\_age{0};

int m\_height{0};

//定义了构造函数的Struct

**Student**( int age, int height);

...

}；

//√ 调用构造函数

Student student { 20, 180 };

//× 没有找到合适的构造函数

Student student { };

【自定义了构造函数的class】

**class** Student

{

public:

//定义了构造函数的Class

**Student** ( int age, int height);

...

private:

int m\_age{0};

int m\_height{0};

};

//√ 调用构造函数

Student student { 20, 180 };

//× 没有找到合适的构造函数

Student student { };

如果没有定义构造函数，因为struct成员默认的访问权限是public，class的访问权限是private，所以struct默认可以用大括号初始化，除非手动将成员变量设置为private/protected；而class默认不能使用列表初始化，只有在所有成员变量全是public的情况下才可以使用列表初始化。

举例:

【struct可以列表初始化public的成员变量】

struct Student

{

int m\_age{0};

int m\_height{0};

...

}；

//√ struct默认可以进行列表初始化

Student student { 20, 180 };

【class不可以列表初始化private的成员变量】

class Student

{

public:

...

private:

int m\_age{0};

int m\_height{0};

};

//× class默认不能进行列表初始化

Student student { 20, 180 };

【class可以列表初始化public的成员变量】

class Student

{

**public**:

int m\_age{0};

int m\_height{0};

...

};

//√ class所有成员变量public的情况下可以列表初始化

Student student { 20, 180 };

## 嵌套类

在一个类中定义另一个类，嵌套类也称成员类，当嵌套（成员）类只在被嵌套类中使用时很有用，可以有效的进行逻辑封装和避免命名污染。

1. 何时使用嵌套类

被嵌套类在逻辑上和嵌套类有明显的**主从**关系和**独占**关系，即被嵌套类无法脱离嵌套类被调用。

举例:

/\*以List为例，List属于一个双向链表，包含了前向节点和

后向节点，然后节点本身这种数据结构就是为了配合List

使用，具有高度的关联，所以适合使用嵌套类\*/

class List

{

public:

List();

/\*这里并不希望外部知道Node的细节，所以直接设置

为private\*/

**private**:

class Node

{

public:

int m\_data { 0 };

Node\* m\_pNext{ nullptr };

Node\* m\_pPrev { nullptr };

};

Node\* m\_pHead;

Node\* m\_pTail;

};

1. 嵌套类类型的访问权限

鉴于嵌套类和被嵌套类独占的从属关系，**原则上嵌套类定义为private**，但是还是有一些特例情况：

1. 基类，嵌套类需要被继承到派生类。

举例:

//public/protected保证派生类中能够正常访问和调用Buzzer

class SignalTower

{

**public**:

class Buzzer{…};

…

};

class SPISignalTower: public SignalTower

{

...

}

class AOISignalTower: public SignalTower

{

...

}

1. 参数，嵌套类被作为实参类型/返回类型被传递。

举例:

class BufferedIO

{

**public**:

//输入型号

class **InputSignal**

{

...

};

//输出型号

class **OutputSignal**

{

...

};

void setInputSignals(list<**InputSignal**> inputs)

{

...

}

std::list<**OutputSignal**> outputSignals()const

{

...

}

...

};

//如何调用

BufferedIO bufferedIO{};

std::list<BufferedIO::InputSignal> tempInputs;

...

bufferedIO.setInputSignals(tempInputs);

...

auto outputs = bufferedIO.outputSignals();

被嵌套类不管何种访问权限，必须要保证被嵌套类的**成员变量**只会出现在**被嵌套类**中。

举例:

/\*InputSignal 和 OutputSignal 为BufferedIO 的嵌入类，

同时也是其嵌入成员对象的类型，这种方式可以被接受\*/

class BufferedIO

{

public:

//输入信号

class InputSignal

{

...

};

//输出信号

class OutputSignal

{

...

};

...

private:

/\*√ 作为BufferedIO的嵌入类，InputSignal和

OutputSignal类型对象出现在BufferedIO是对的。\*/

std::list<InputSignal> m\_inputSignals;

std::list<OutputSignal> m\_outputSignals;

};

/\*Motor类型中包含InputSignal 和 OutputSignal

类型的成员变量，这种方式不能被\*/

class Motor

{

public:

...

private:

/\*× InputSignal和OutputSignal类型对象不建议现

BufferedIO以外的其它类\*/

InputSignal m\_inputSignals;

OutputSignal m\_outputSignals;

};

## 默认构造函数

默认构造函数由编译器提供，它可以是没有任何形参的构造函数，也可以是实参全部有默认值的构造函数。

举例:

//虽然没有显式的写构造函数，但是编译器会默认实现

class Box

{

public:

double m\_length{ 1.0 };

double m\_width{ 1.0 };

double m\_height{ 1.0 };

};

//调用编译器自动生成的默认构造函数

Box box{};

虽然编译器提供的默认构造函数有时可以提供方便，但是毕竟有潜在的隐患：

1. 初始化问题

默认构造函数**不会初始化成员变量**。如果没有指定构造函数，容易造成成员变量值不确定的风险。具体见:[五/4 避免变量的不确定值](#_避免变量的不确定值)。

举例:

/\*对象创建后m\_age的值是不确定的\*/

class Person

{

public:

...

private:

int m\_age;

...

};

//调用编译器自动生成的默认构造函数

Person person { };

1. 效率问题

如果成员变量没有初始化的情况下，成员变量会进入**默认的构造函数**，然后再进入**指定的构造函数**。为了减少这一次调用提高执行效率，最好自己实现使用初始化列表的方式实现默认构造函数。

举例:

class Cube

{

...

private:

/\*如果有一个成员没有进行初始化，同时没有显式定义默认 的构造函数的话，那么会调用,编译器默认生成的默认构造 函数，会多一次调用\*/

int m\_side/\*{ 1 }\*/;

double m\_area{0.1\*2/3};

float m\_ratio{0.01};

}

如何调用:

auto start = chrono::steady\_clock::now();

int cnt = 50000000;

for (int i = 0; i < cnt; ++i)

{

Cube cube1;

}

auto end = chrono::steady\_clock::now();

cout << chrono::duration <double, milli> (end-start).count() <<endl;

分别在注释m\_side和不注释m\_side的情况下测试，时间分别为155ms和124ms，原因就是前者多了一次调用默认构造函数。

1. 访问权限问题

自动实现的构造函数总是public，不适用于单例模式或者禁止拷贝的场合,具体见：

[四/5 禁止拷贝和禁止重载 “=”；](#_禁止拷贝和禁止重载\“=\”)

[七/2 单例模式 & 控制实例的数量。](#_单例模式 & 控制实例的数量)

## 使用explicit避免隐式转换

当类存在单一参数的构造函数时，编译器在类的构造时会帮我们进行隐式转换，虽然这一定程度上简化了编码，但是也会带来隐患。

举例:

//函数单一参数构造函数的Student类

class Student

{

public:

Student( int age );

...

private:

int m\_age{0};

}

//以下几种构造方式可以编译通过

|  |  |
| --- | --- |
| Student wangfei = 1 ; | /\*通过，但是很难理解数字“1”的具体含义\*/ |
| Student student = 's' ; | /\*通过，但是s字符从ASCII隐式转换到int，不仅存在效率问题，同时m\_age也会被错误赋值\*/ |

C++提供了禁止这种隐式转换的方式，使用explicit关键字, 通常建议大家使用explict来避免隐性bug出现。

举例:

class Student

{

public:

explicit Student( int age ) ;

private:

int m\_age { 0 };

}

//以下几种构造方式无法编译通过

|  |
| --- |
| Student wangfei = 1 ; |
| Student student = 's' ; |

## 禁止拷贝和禁止重载“=”

在某些情况下，需要禁止对象复制操作。具体分两种情形:

1. 禁止赋值

使用编译器自动实现的赋值重载有很多隐患，如类成员变量的赋值问题，派生类的赋值问题，类后期的扩展修改等。正确的做法是对于需要赋值重载应当是根据实际**手动实现**，而**其余大部分类需要禁止“=”赋值**操作，具体的做法就是**继承接口INonAssignable**。

举例:

//接口INonAssignable

class INonAssignable

{

**protected**:

INonAssignable () {}

~ INonAssignable () {}

**private**:

const INonAssignable & operator = (const INonAssignable &) ;

};

//禁止Pad的“=”重载，只需要继承INonAssignable

class Pad: private **INonAssignable**

{

…

}

int main()

{

…

Pad pad1;

…

Pad pad2 ;

//×编译不过

pad2 **=** pad1;

…

}

如果需要进行重载赋值操作时，记得一定要进行自赋值检查。

举例:

class Student

{

public:

…

Student& operator = ( const Student& student )

{

//检查自赋值

if( **this != &student** )

{

…

}

return \*this;

}

…

};

1. 禁止复制

对于某些场合，如检测对象Target，是不可能存在完全一样的Target的，为了安全起见就可以禁止对象的复制，禁止复制要比禁止“=”重载严格一些，除了禁止对象的“=”复制操作以外还需要禁止对象的拷贝行为，具体的做法就是**继承接口INonCopyable**。

举例:

//接口INonCopyable

class INonCopyable ：private **INonAssignable**

{

**protected**:

INonCopyable( ) { }

~INonCopyable( ) { }

**private**:

INonCopyable( const INonCopyable& ) ;

};

//禁止Target的复制，只需要继承INonCopyable

class Target : private INonCopyable

{

…

}

int main()

{

…

Target target1;

…

//× 编译不过

Target target2 = target1;

//× 编译不过

Target target3 = Target(target1);

//× 编译不过

Target target3(target1);

…

}

1. 鼓励使用clone函数实现复制

复制操作还可以使用实例方法**clone**()实现，通过返回指针的方式复制，这样做优点是可以完全实现摆脱对赋值“=”重载和复制构造的依赖，不需要进行自赋值检查，同时更加适用于具有继承关系的基类和派生类中，基类中的声明clone的虚方法，派生类中根据具体情况进行重写，更加安全。

举例:

//基类IBird，声明clone的虚方法

class IBird

{

public:

…

**virtual** const IBird\* clone( ) const = 0;

…

};

//派生类sparrow复制只需要重写clone即可

class sparrow : public IBird

{

public:

**virtual** const IBird\* clone()**override** const

{

//具体的复制逻辑代码

...

}

};

## 防止运算符歧义

运算符重载时，坚持运算符的通用含义，比如“+”是加，“-”是减，“\*”是乘，“/”是除，同时兼顾具体的应用场景，否则很容易引发理解上的混乱。

错误举例:

class Pad

{

public:

…

/\*× 该函数把Pad相加理解为2个Pad的x坐标、y坐标、

长和宽分别相加，这明显和实际的应用不符合\*/

const Pad operator + (const Pad & pad) const

{

return Pad( m\_x + pad.m\_x,

m\_y + pad.m\_y,

m\_w + pad.m\_w,

m\_h + pad.m\_h);

}

private:

double m\_x{0};

double m\_y{0};

double m\_w{0};

double m\_h{0};

…

}

根据实际的应用，Pad相加简单地把几何参数相加是没有意义的，而应该是两个Pad相加得到一个新的Pad，这个新的Pad是形状为2个Pad的外接矩形。

正确举例:

class Pad

{

public:

…

/\*√ Pad相加理解为求取一个新的Pad，该Pad为两个

Pad的外接矩形\*/

const Pad operator + (const Pad & pad) const

{

/\*以下代码为求两个Pad外接矩形的代码，实际应

用比这里复杂，仅仅作为参考\*/

return Pad( (m\_x + pad.m\_x) /2,

(m\_y + pad.m\_y)/2,

min(m\_x - m\_w/2,pad.m\_x - pad.m\_w/2),

max(m\_x + m\_w/2,pad.m\_x + pad.m\_w/2));

}

private:

double m\_x{0};

double m\_y{0};

double m\_w{0};

double m\_h{0};

…

}

注意:

* 但凡涉及到运算符重载的，不管简单的通用含义还是具体特殊的应用，必须要100%写详细的注释，**详细交代改运算符的使用目的，以免调用者理解不正确。**
* 运算符重载一定是**成双成对**的出现的，比如重载了“+”，一定要重载“-”，对于确实由于某种原因还没有实现的函数请写好“**//ToDo”**。
* 为了和STL进行对应，如果是类似容器仅仅只是简单的添加、移除操作，建议还是使用add和remove等函数方法的方式实现。

## 虚函数

虚函数是以virtual关键字声明的基类函数。基类中的函数指定为virtual，派生类对该函数进行重写，实现多态。关于虚函数需要注意几点。

1. virtual关键字

在派生类中重写的虚函数virtual关键字不写也能够编译通过和正常运行，但是为了定位及理解代码方便，使阅读者一眼就能够明白该函数是继承重写而来，所以必须要加上**virtual关键字**。

1. override关键字

在派生类中重写的虚函数后面不加override也能够编译通过和正常运行，但是加上override的话：

* 可以方便阅读者知道该方法重写了基类。
* 编译器会对函数的参数及返回值和基类函数进行验证，提高编译时便发现问题的可能性。

举例:

class Box

{

public:

…

//基类Box中volume函数设置为虚

**virtual** double volume( ) const

{

return 5\*m\_length;

}

…

private:

double m\_length;

};

class GlassBox : public Box

{

public:

…

//派生类GlassBox中重写volume

**virtual** double volume( ) const **override**

{

return 10\*m\_length;

}

};

//如何调用

**Box** myBox{ 2.0 };

**GlassBox** myGlassBox{ 2.0 };

//调用基类的showVolume方法

myBox.**volume**( );

//调用派生类的showVolume方法

myGlassBox.**volume**( );

运行结果：

10 20

## 析构函数总是为虚

基类的析构函数应该总是为virtual的，因为通过虚函数机制可以保证派生类能够彻底的被释放。

错误案例:

//Box为基类，其析构函数不为虚

class Box

{

public:

Box ( ){...}

**~Box**()

{

cout<<"Box is destructed"<<endl;

}

};

//CandyBox为派生类，继承了Box

class CandyBox : public Box

{

public:

CandyBox( );

**~CandyBox** ()

{

cout<<"CandyBox is destructed"<<endl;

}

};

int main()

{

…

Box\* pBox = new CandyBox();

delete pBox;

…

}

//× Box被销毁，CandyBox没有被销毁。

结果：

Box be destructed

当Box的析构函数不为虚时，当销毁pBox时，只是销毁的对象的基类部分，CandyBox部分的数据并没有被销毁，这就造成了内存泄露。解决的办法也很简单，只要把Box的析构函数设置为virtual

正确案例:

//设置Box的析构函数为virtual

class Box

{

public:

Box ( ){...}

**virtual** **~Box**()

{

cout<<"Box is destructed"<<endl;

}

};

//CandyBox为派生类，继承了Box

class CandyBox : public Box

{

public:

CandyBox( );

/\*以防CandyBox 再被继承，其析构函数也设置为虚\*/

**virtual** **~CandyBox** ()

{

cout<<"CandyBox is destructed"<<endl;

}

};

//CandyBox得到彻底释放

int main()

{

…

Box\* pBox = new CandyBox;

delete pBox;

…

}

//√ CandyBox的派生类部分和基类部分全部销毁

结果：

CandyBox is destructed

Box is destructed

注意：

对于新人来说：

* 应该每写一个类就把析构函数写成虚函数。
* 当需要写一个继承类，发现基类中析构函数不为虚时，要请示主管是否有蹊跷以便确认该类是否有特殊应用的原因，避免用错。

## 构造函数和析构函数中不能调用虚函数

在构造函数里调用虚函数，编译器是可以通过的，但是没有意义的。因为派生类的正确构造必须以基类的正确构造为前提，此时调用虚函数，会发生调用基类本身的虚函数，并不是想要的结果。

错误案例:

class IAlg

{

public:

//基类Alg中的init为虚函数

IAlg( )

{

**init**( );

}

}

virtual void **init**( )

{

cout<< " IAlg:: init\n";

};

class BWAlg: public IAlg

{

public:

BWAlg( ) : IAlg()

{

...

}

//派生类BWAlg重写了init

**virtual** void **init**( )**override**

{

cout<<" BWAlg:: init\n";

}

};

//如何调用

int main()

{

…

BWAlg bwAlg{};

…

}

结果：

**IAlg :: init** //× 调用了基类init，并不是想要的

析构函数和构造函数一样，同样不能调用虚函数，虚函数机制已经不起作用。

错误案例:

class IAlg

{

public:

virtual ~IAlg( )

{

**release**( );

}

//基类Alg中的release为虚函数

virtual void **release**( )

{

cout<< " IAlg:: release\n";

}

};

class BWAlg: public IAlg

{

public:

virtual ~BWAlg( )

{

...

}

//派生类BWAlg重写了release

**virtual** void **release**( )**override**

{

cout<<" BWAlg:: release\n";

}

};

//如何调用

int main()

{

…

BWAlg bwAlg{};

…

}

//× 调用了基类release，并不是想要的

结果：

IAlg::release

## 派生类调用基类函数

在派生类中经常涉及到需要调用基类中函数，这时候需要注意:

1. 调用基类的构造函数

在派生类构造函数后面加上“**:**”及“**基类的构造函数**”。

举例:

Class Box{…};

Class CandyBox : public Box

{

public:

CandyBox ( ) : **Box ( )**

{

…

}

…

};

如果基类中构造函数比较多的情况下，派生类要实现全部对应实现比较麻烦，可以在派生类中通过**继承构造函数**实现。

举例:

class Person

{

public：

Person( int age, int height );

Person( int age, int height, int weight );

…;

protected:

int m\_age{0};

int m\_height{0};

int m\_weight{0};

};

//通过继承构造函数实现基类构造函数的直接调用

class Student : public Person

{

public:

//关于基类各构造函数的继承一句话搞定

using Person::Person;

void printID( ) const

{

std::cout << m\_id << std::endl;

}

}

…;

private:

int m\_id{0};

};

//√ 调用基类的构造函数

Student zhangSan(1,100);

//√ 调用基类的构造函数

Student xiaoLi(2, 100, 50 );

注意:

虽然继承构造函数在一定程度上可以带来编程的便捷，但是也有如下陷阱，如:

* 对于在派生类中声明的成员变量没有进行初始化，其值可能不确定(除非在声明的时候就初始化默认值)。虽然可以在类内进行赋初值，但是该新的成员变量很难通过构造函数进行灵活赋值了。含有默认参数的构造函数不会被继承，参数的默认值会导致基类产生多个构造函数版本。
* 当派生类继承多个类时，可能导致派生类中的继承构造函数冲突。
* 如果基类中的构造函数为私有，那么就不能从派生类继承。
* 一旦使用继承构造函数，编译器不会为派生类生成默认构造函数。

综上，继承构造函数不推荐使用，如果觉得真的有必要使用，先请示主管获得批准后再用。

1. 调用基类的成员函数
2. 如果不是重名函数直接调用即可
3. 如果重名，使用这样的形式“类型名：函数名”

举例:

class Printer

{

public:

void **print**( );

void **reset**();

…

};

class StringPrinter : public Printer

{

public:

void print()

{

**reset();** //不重名，直接调用即可

**Printer::print();** //基类类型加重名函数

…

}

…

};

## 抽象类

包含纯虚函数的类称为抽象类，具有如下特点：

1. 抽象类不能实例化，但是可以定义抽象类的指针和引用。
2. 如果抽象类的派生类没有实现其所有派生类的虚函数，则该派生类还是抽象类。
3. 只要有一个virtual函数= 0时，该类就是抽象类。
4. 为区别于实体基类，所有的抽象类建议都大写I字母。详见 [一/2.class & struct 类/结构体命名](#_class & struct name 类/结构体命名)。

判断一个类是否为抽象类，有如下几种情况：

举例:

//只要有一个virtual函数=0，该类就是抽象类

class Box

{

public:

**virtual** int calVolume( ) **= 0**;

**virtual** int calArea( ) = 0;

...

protected:

double m\_w{1};

double m\_h{1};

double m\_l{1};

};

//有未实现的纯虚函数，该类依旧为抽象类

class CandyBox : public IBox

{

public:

virtual int calVolume( ) override

{

return m\_h \* m\_w \* m\_l;

}

...

};

//× 抽象类不能实例化

CandyBox candyBox{};

//实现了全部virtual函数后，该类就是实体类

class CakeBox : public Box

{

public:

virtual int calVolume( ) override

{

return m\_h \* m\_w \* m\_l;

}

virtual int calArea() override

{

return m\_w \* m\_l;

}

...

};

//√ 实现了所有的虚函数，变成实体类，可以实例化

CakeBox cakeBox{};

## 类成员

1. 访问权限

类的成员变量必须优先声明为private或protected，通过存取函数去获得和设置它的值，可以实现禁止访问、只读访问和读写访问。

举例:

class Student

{

public:

int age( ) const;

void setAge( int value );

...

private:

int m\_age{ 0 };

...

}

1. 嵌入对象类型

设计类的数据成员时，有以下三种选择：

1. 嵌入**实例**:

使用实例，通过类的构造函数创建对象，当对象类型的声明与定义发生变化时，被嵌入的类会重新编译。

嵌入实例的优点是不用关心对象的生命周期，从而不必担心对象的销毁。缺点是对象占用的内存大小会影响到被嵌入的类对象。

比较轻量(如几何中的点、线、向量)的对象不需要考虑类型占用的内存大小，推荐使用嵌入实例。

举例:

class Point{…};

class Rectangle

{

…

private:

//嵌入实例

**Point m\_centerPoint;**

int m\_width;

int m\_height;

…

};

1. 嵌入引用:

使用引用，构造时就必须将一个已经存在的实例传入并与嵌入引用对象绑定，否则会导致编译不通过。

嵌入引用的优点是不会因嵌入类的修改导致被嵌入类占用的内存发生变化；当嵌入类的声明与定义发生变化，被嵌入类不需要在为此而重新编译;嵌入引用支持多态，同时和嵌入实例一样，对象自动销毁，使得类不用关心对象的释放问题，具有良好的可读性和安全性。

当不希望使用嵌入实例，同时仅仅是用到了嵌入对象的一些成员变量和成员函数，不涉及到对象的创建和销毁，不涉及一些特殊的设计模式时，可以使用嵌入引用。

举例:

class Book{…};

class Teacher

{

public:

//嵌入引用必须在构造函数的初始化列表中初始化

Teacher(**const Book& professionalBook**):

m\_professionalBook(**professionalBook**)

{

...

}

…

private:

//嵌入引用

**Book& m\_professionalBook;**

…

};

注意:

关于嵌入引用有以下陷阱,如果使用要特别注意:

* 特别注意嵌入引用对象的生命周期，必须保证在被嵌入类的生命周期内引用对象不被释放。
* 嵌入引用只能在构造函数的初始化列表中进行初始化，无法像指针一样实现延迟初始化。
* 使用嵌入引用对象的类默认是不支持赋值拷贝的，如果要支持，需要手动实现这个函数，也就是说，使用了嵌入引用以后，被嵌入的对象隐式的变为Nonasignnable。
* 嵌入引用一旦绑定之后就不能为nullptr或者再绑定其它对象。

1. 嵌入指针:

使用指针作为嵌入对象。

指针的优点不需要依赖嵌入类类型，支持多态。可以按需实现延迟初始化，无须在构造时就绑定实例，缺点是需要花费额外的精力维护指针(创建和销毁以及nullptr的判断等)。

当嵌入对象需要按需创建，绑定不同的对象，如柴郡猫、特定模式设计等推荐使用嵌入指针。

举例:

class Bluetooth{…};

class Camera{...};

class Phone

{

public:

...

private:

**Bluetooth\* m\_pBluetooth {nullptr};**

**Camera\* m\_pCamera {nullptr};**

};

注意:

使用嵌入指针时，关于指针的删除要加倍小心。

* 使用时要记得判断指针是否为**nullptr**；
* 如果是由该类的构造函数或成员函数使用new创建了该对象，那么应该记得要在析构函数中delete指针。如果是外部传入指针时裸指针，通常不应该由该类负责释放，而应该由外部new它的地方来负责相对应的delete。
* 大部分情况下需要手动实现或者禁止赋值拷贝函数和复制构造函数，如果不手动实现，类的拷贝就是浅拷贝，这回造成一个类对象的销毁会影响到复制的另一个类对象。

举例:

class Fov: private **INonCopyable**

{

public:

/\*1.m\_pads都是由外部传入，Fov不负责pads中指针元素

的释放2.m\_pImg内部创建，Fov需要负责m\_pImg的释放\*/

Fov(std::vector<Pad\*> pPads):

m\_pPads(pPads),

m\_pImg(**new Image()**)

{

}

Fov()

{

m\_pImg = **new Image();**

}

~Fov()

{

if(nullptr!=m\_pImg)

delete m\_pImg;

}

private:

std::vector<Pad\*> m\_pPads;

Image\* m\_pImg;

};

指针不是数量很多时完全可以通过智能指针。这样会更安全。如果数量很大时，使用裸指针，具体何时如何选择可以参见

[六/3. 容器中指针的管理。](#_容器中指针的管理)

1. 声明次序

成员的声明次序需要注意以下几点：

1. 如果用到了类的前向声明，那么放在类外。
2. 访问权限顺序: public、protected、private，中间空行隔开。
3. 同一权限内的成员的声明顺序如下,如果没有则省略。

* 嵌入类的声明;
* #define/typedef/using;
* enums;
* 常量;
* 成员变量，含静态成员变量(静态在前);
* 构造函数(默认构造函数、参数构造函数、复制构造函数);
* 析构函数;
* 运算符重载函数;
* 存取函数;
* 其它成员函数，含静态成员函数;

1. 相关功能的成员放在一起，不相关功能的成员之间用空行隔开。

举例:

//在Fov中使用到Target的前向声明

class Target;

class Fov:INonCopyable

{

public:

/\*如果需要用到嵌入类，把嵌入类的声明放在成员的第一\*/

class Image{...};

/\*不同形式的类型声明之间加空行分隔\*/

using Targets = std::vector<Target>;

using RawImage= std::unique\_ptr<Image\*>;

enum FovType

{

Fiducial,

Normal

};

/\*

1.多个构造函数全部紧挨放在一起

2.默认构造函数总是在构造函数中放第一个

3.带参构造函数放在默认构造函数后面

4.如果需要支持复制，需要实现复制构造函数

5.析构函数紧挨着构造函数后面

\*/

Fov ();

Fov ( FovType type, int id, int width, int height);

~virtual Fov( );

/\*

1. 运算符重载函数总是放在普通成员函数前面
2. 成双成对的运算符重载函数总是紧挨着放

\*/

Fov & operator+=(Target& pad);

Fov & operator-=(Target& pad);

/\*

1. 存取函数总是放在普通函数前面

2. 对于自定义类型的读取函数，总是需要加上const修饰

3. 不同对象的存取函数用空行隔开

4. 存取函数直接在头文件中实现即可

\*/

int id() const { return m\_id; };

int order() const{return m\_order;}

int setOrder(int order){ m\_order = order;}

const RawImage image const{ return m\_img;}

protected:

/\*

1. 相关功能的成员放在一起

2. 不相关功能的成员以空行隔开

\*/

const int m\_id { -1 };//Fov的ID,构造函数初始化

int m\_order { -1 };//Fov的Order

FovType m\_fovType{ Noraml }

private:

Targets m\_targets;//待检查的Target

Targets m\_ngTargets;//ng的Target

RawImage m\_img;//图像对象

};

1. 成员函数尽量短小简单

成员函数尽量短小、凝练。如果函数超过**40**行(这个行数保证了正常分辨率和字体大小下一屏代码能够显示的行数)，就应该考虑在不影响程序结构的情况下将其分割一下。函数越短小、简单，就越便于他人阅读和修改代码。

错误案例:

/\*计算体积

函数中包括了计算面积、高和体积三部分。这个函数是个长函数，包括了计算面积和计算高两部分，总共代码超过了40行，显示器屏幕已不能完整显示完全代码\*/

void calcPadVolume ()

{

//1.计算面积

……

//2.计算高

……

//3.计算体积

……

}

考虑到计算体积主体在于计算面积和高，可以将这两部分分别封装成函数，通过调用函数来避免代码过长，同时也更容易理解。

正确案例:

/\*把原先1个函数的代码拆解成了3个函数，然后这里只有3行调用了，这样看起来更简洁，而且拆解的函数之间还可以灵活组合以供其它地方调用，降低了耦合度\*/

double calcPadVolume ()

{

//1.计算面积

double area=**calcPadArea**();

//2.计算高

double height = **calcPadHeight**();

//3.计算体积

return area\*height;

}

## 何时使用多重继承

C++支持多重继承，但多重继承会带来理解上的混乱，破坏单一性。

错误案例:

//声明一个Sofa的基类，可以坐，可以清洁

class Sofa

{

public:

void sit();

void clean();

…

}

//一个Bed的基类，可以睡觉，可以清洁

class Bed

{

public:

void sleep ();

void clean();

…

}

//× 声明一个SofaBed派生类，需要同时具备sleep和sit的功能

class SofaBed:public Sofa,public Bed ×

{…}

例子中同时继承了Sofa和Bed，这不仅仅是不好理解的问题，当调用到SofaBed的clean方法会造成不明确的调用。除非显式指定类型调用，这又牺牲了多态性。

另外一种由多重继承产生的典型问题是“死亡之钻(DOD)”。

错误案例:

//一个基类Animal

class Animal

{

public:

const int weight() const{return weight;};

private：

int m\_weight;

…

};

//Tiger和Lion都继承于Animal

class Tiger: public Animal { ... }

class Lion : public Animal { ... }

//Liger继承于Tiger和Lion

Class Liger:public Tiger,public Lion{ ... };

//Liger继承于Tiger和Lion

int main( )

{

Liger liger;

//× 编译错误，下面的代码不会被任何C++编译器通过

int weight = lg. weight ();

}

Liger多重继承了Tiger和Lion类，因此Liger类会有两份Animal类的成员（数据和方法），Liger对象" liger "会包含Animal两个子对象。编译器并不知道是调用Tiger类的weight ()还是调用Lion类的weight ()。所以，调用weight方法是不明确的，因此不能通过编译。虽然该问题可以用virtual继承的方式解决，但是公司不推荐使用这种方式，所以请务必牢记通常情况下禁止使用多重继承。

当然如果只是作为继承接口抽象类的多重继承，那么允许继承一个实体类，其余都必须继承的是接口抽象类，此作为可以多重继承的特例。

特例:

//声明一个ISitable、ISleepable和ICleanable的接口抽象类

class ISitable

{

public:

virtual void sit() = 0;

}

class ISleepable

{

public:

virtual void sleep() = 0;

}

class ICleanable

{

public:

virtual void clean() = 0;

}

//声明Sofa和Bed的实体基类

class Sofa: ISitable

class Bed: ISleepable

//√ 下面三种继承方式都可以

class SofaBed: public ISitable, public ISleepable, public ICleanable

class SofaBed: public Sofa, public ISleepable, public ICleanable

class SofaBed: public Bed, public ISitable, public ICleanable

## 优先使用组合代替继承

在类的关系中，继承属于强关系，耦合度最高，在设计类的时候，应该先考虑组合，其次才是继承。

1. 继承

继承在逻辑表示Is-A的关系。关系上有明显的属于关系，一旦基类发生变化，派生类也发生变化，派生类无法脱离基类存在。

举例:

/\*Human有年龄、身高、体重等，Teacher和Student也具有，

两者都属于Human，适合使用继承\*/

class IHuman

{

public:

…

virtual int age( ) const = 0;

virtual int height( ) const = 0;

virtual int weight( ) const = 0;

…

private:

int m\_age{0};

int m\_height{0};

int m\_weight{0};

…

};

class Student : public IHuman {…};

class Teacher : public IHuman {…};

1. 组合

组合，在逻辑上表示的是Has-A的关系,表示一种包含归属的关系。

例如眼(Eye)、鼻(Nose)、口(Mouse)、耳(Ear)是头(Head)的一部分，所以类Head应该由类Eye、Nose、Mouse、Ear组合而成，被包含类的生命周期受包含类的控制，被包含类会随着包含类的创建而创建，随包含类的消亡而消亡。采用继承的方式，Head可以实现看、闻、吃、听的功能，但是这样要求Head继承多个接口。

不好示例:

//设计ISeeable和ISmellable接口

class ISeeable

{

public:

virtual void look( ) = 0;

};

class ISmellable

{

public:

virtual void smell( ) = 0;

};

//Head通过继承来实现ISeeable和ISmellable的功能

class Head : public ISeeable, public ISmellable

{

virtual void look() override;

virtual void smell() override;

...

};

在上述场合使用继承会造成如下问题：

* Head和ISeeable、ISmellable属于强耦合关系，一旦接口抽象类发生修改，Head也需要进行修改：如接口抽象类函数改名，Head也需要改名，否则会编译不通过；如果接口抽象类增加了虚函数，因为Head忘了重写增加的函数，也会变成了抽象类，容易造成隐患。
* Head于Eye，Nose有重复代码，违背了单一性原则。同时如果Head需要增加更多的功能，如听的功能，Head需要再继承IHearable的接口，同时实现其虚函数。随着需求的增加，Head会变得越来越臃肿，如下面的例子：

不好示例:

//原来的ISeeable和ISmellable的接口

class ISeeable

{

public:

virtual void look( ) = 0;

};

class ISmellable

{

public:

virtual void smell( ) = 0;

};

//Head增加了hear的需求，设计新的接口

class IHearable

{

public:

virtual void hear() = 0;

}

class Head : public ISeeable, public ISmellable, public IHearable

{

virtual void look() override;

virtual void smell() override;

//如果忘记了重写hear，Head会变成抽象类

virtual void hear() override;

...

};

使用组合，被包含对象的内部细节对外是不可见的，所以它的封装性相对较好。在上面的例子中，使用组合可以实现Head对具体子模块的解耦。

较好示例:

//声明Eye和Nose的实体类

class Eye

{

public:

void look( );

};

class Nose

{

public:

void smell( );

};

//Eye和Nose作为嵌入对象放在Header中

class Head

{

private:

Eye\* m\_pEye { nullptr };

Nose\* m\_pNose { nullptr };

…

};

## 继承不要扩到集合层面

不要将对象的继承关系扩到至集合层面。比如说车是一种交通工具，所以车这个类型可以继承交通工具这个类型，由此，将停车场定为一种交通工具的停泊场，在逻辑上是可以的使用继承的。

举例:

class Vehicle { … } ;

class Car : public Vehicle { … } ;

class VehicleParkingLot

{

public:

//添加交通工具

void addNewVehicle( const Vehicle& vehicle );

}；

//汽车停车场继承交通工具停车场

class CarParkingLot : public VehicleParkingLot { … };

基类VehicleParkingLot可以是飞机、汽车、轮船等的停车场。当CarParkingLot继承了这一能力，那汽车停车场(CarParkingLot)也将具有停飞机、轮船的能力，这严重违背的单一性原则，所以继承关系不能扩展至对应的集合类型中。

# 数据操作

在正式讨论数据操作的规范之前，先介绍下2个比较重要、并在后续章节会经常提到的概念，Aggregate 聚合类型和POD类型。

## Aggregates类型

以下是C++对于聚合体的定义:

1. 类型是一个普通数组(如int[10]、char[]、long[2][3])。
2. 类型是一个类(class、struct、union),且

* 无用户自定义的构造函数。
* 无私有(private)或保护(protected)的非静态成员变量。
* 无基类。
* 无虚函数。
* 不能有{}和=直接初始化(brace-or-equal-initializer)的非静态数据成员。

举例:

//× 非聚合类型，包含虚函数

class NotAggregate1

{

...

virtual void func(){}

}

//× 非聚合类型，包含private的非静态成员变量

class NotAggregate2

{

...

private:

int m\_id;

}

//× 非聚合类型，包含用户自定义构造函数

class NotAggregate3

{

public:

NotAggregate3( int ){}

}

//× 非聚合类型，成员变量声明完就进行列表初始化

class NotAggregate4

{

public:

vector<int> m\_ids { 1, 2, 3 }

}

//√ 聚合类型

class Aggregate1

{

public:

//√ 拷贝赋值函数

Aggregate1& operator = (Aggregate1 const & aggregate);

//√ 公开成员变量

NotAggregate1 m\_notAggregate { nullptr };

...

private：

void func(){}//√ private函数

...

}

//√ 聚合类型,

class Aggregate2

{

public:

Aggregate2() = default;//要求编译器自动生成默认构造函数

...

}

聚合类型支持使用列表“{}”的方式来初始化：

举例:

struct Point

{

int m\_x;

int m\_y;

}

struct Rectangle

{

string m\_title;

Point m\_pt;

int m\_size[2];

}

/\*

m\_title为“rect”;

m\_pt的m\_x和m\_y都为0;

m\_size[0]为2,m\_size[1]为4

\*/

Rectangle rect = {“rect”,{ 0, 0 }, { 2, 4 }};

## POD类型

POD(Plain Old Data，POD)类型，简单来说，是C++为兼容C的内存布局而设计的，主要用于用户自定义类型。POD类型说明该数据是普通的，不会有虚函数，虚继承，内嵌很复杂的数据类型等情况，和C语言中的struct一样，POD类型可以用最老的memcpy()函数进行复制，使用memset()来进行初始化等。

在C++98/03中，POD类型指:

* 没有非静态类型的non-POD类型。
* 没有引用类型。
* 没有用户自定义的赋值操作符和析构函数的Aggregate类型的类。

在C++11中，POD类型指:

* 既为trivial类型又为standard-layout类型。
* 没有非静态的non-POD类型。

下面分开，详细说明trivial类型和standard layout。

1. Trivial类型

* 拥有平凡的默认构造函数(trivial constructor)和析构函数(trivial destructor)。

平凡的默认构造函数就是说构造函数“什么都不干”。通常情况下，不定义类的构造函数，编译器就会为我们生成一个平凡的默认构造函数，对于析构函数也是如此。注意哦，是通常情况下，如果该类含有**虚函数**或者成员变量没有默认构造函数，编译器是不会为我们生成平凡的默认构造函数。

一旦定义了构造函数，即使构造函数不包含参数，函数体没有任何代码，那么该构造函数也不再是“平凡”的。仔细想一想，既然POD类型与Ｃ语言的结构体等价，C语言哪来的用户定义构造函数呢。当然可以使用default关键字来强制使编译器提供默认的平凡构造函数。

简单概括总结上面的条件：即没有用户自定义的构造函数、类内部没有虚函数，同时所有成员类型及其基类都满足这两条规则。

* 拥有平凡的拷贝构造函数(trivial copy constructor)和移动构造函数(trivial move constructor)。

如果用户自定义的拷贝构造函数/移动构造函数，那么该类型就是不平凡的。

* 拥有平凡的拷贝赋值运算符(trivial assignment operator)和移动赋值运算符(trivial move operator)。

基本上与平凡的拷贝构造函数和平凡的移动构造函数类似，只不过把“拷贝构造函数和移动构造函数”换成“拷贝赋值函数和移动赋值函数”理解就可以了。

举例:

【非trivial类型的一些情形】

class People

{

public:

People();//× 声明了用户自定义的构造函数不是Trivial类型

virtual ~People();//× 含有虚析勾函数不是Trivial类型

//× 含有虚函数不是Trivial类型

virtual void sleep();

...

}

//× 基类不是Trivial，那么派生类也必定不是Trivial类型

class Student: public People

{

...

private:

//× 含有非Trivial类型的非静态成员变量

People m\_teacher { nullptr };

}

【trivial类型的一些情形】

class Point

{

public:

//√ 平凡的默认构造函数

Point() = default;

Point(int x,int y):

m\_x(x),

m\_y(y){};

/\*√ trivial类型对访问控制修饰符没有要求，可以存在不同权限的成员变量\*/

protected:

int m\_x { 0 };

int m\_y { 0 };

private:

int m\_id { -1 } ;

};

//√ trivial类型的基类必须也是trivial类型

class Point3D: public Point

{

...

};

class Rectangle

{

public:

/\*√ trivial类型中除了不能使用虚函数以外对函数没有其它限制要求\*/

void draw();

private:

/\*√ trivial类型中的非静态成员变量也必须是trivial类型,包括集合对象\*/

Point m\_centerPoint { nullptr };

Point m\_vertexes[4] {};

int m\_sizeX { 0 };

int m\_sizeY { 0 };

//√ trivial类型中可以包含非trivial类型的静态成员变量

static People m\_drawer { nullptr };

}

1. Standard Layout 标准布局

POD包含的另外一个概念就是标准布局, 定义如下:

* 没有虚函数和虚基类。
* 非静态成员变量的访问控制必须相同。
* 所有的非静态成员变量必须是Standard Layout类型。
* 所有基类必须是Standard Layout类型。
* 派生类的第一个非静态成员变量不能为基类。
* 满足以下条件之一：
  + 在继承体系中，最底层也就是继承最多基类的派生类中没有非静态成员变量，而且在最多一个基类中拥有非静态数据成员。
  + 没有基类含有非静态成员变量。

举例:

【非standard-layout类型】

class People

{

public:

...

//× 含有虚析勾函数不是standard-layout类型

virtual ~People();

//× 含有虚函数不是standard-layout类型

virtual void sleep();

...

private:

int m\_age { 1 };

}

/\*× 基类不是standard-layout类型，那么派生类也必定不是standard-layout类型\*/

class Student: public People

{

...

/\*× 所有成员变量的访问权限必须一致，否则就是非standard-layout类型\*/

protected:

//× 含有非standard-layout类型的非静态成员变量

People m\_teacher { nullptr };

...

private:

int m\_id { -1 };

}

//× 基类不能作为第一个成员变量

class Monitor: public People

{

public:

People m\_teacher { nullptr };

}

//× 在继承体系中，不能超过一个类含有非静态的成员变量

class President: public People

{

public:

People m\_teacher { nullptr };

}

【standard-layout类型】

class Point

{

/\*√ 所有的成员变量的访问权限全部相同(private)\*/

private:

/\*√ 和trivial不同的是，用户自定义的默认构造函数对standard-layout没有影响\*/

Point() {};

Point(int x,int y):

m\_x(x),

m\_y(y){};

};

//√ 能够包含基类对象作为成员变量，但是前提不能排第一个

class Point3D: public Point

{

...

private：

Point m\_2dPoint { nullptr };

double m\_z { 0 };

};

//√ 可以包含静态的非standard-layout类型的成员变量

class Color

{

public:

static People m\_owner { nullptr };

}

//√ 可以多重继承，但是只能有一个类含有非静态成员变量

class RenderPoint: public Point3D, public Color

{

}

1. 使用<type\_traits>中方法进行POD类型的判断

在新的标准下，放宽了旧标准对POD的限制，导致很多新类型成为POD类型，可以使用标准库模板(STL)在头文件<type\_traits>中的相关函数对这些类型的检测:

举例:

template <typename T>

struct std::**is\_pod**;

template <typename T>

struct std::**is\_trivial**;

template <typename T>

struct std::**is\_standard\_layout;**

1. POD类型的好处

POD类型可以带来的好处如下:

* 字节赋值(bytewise copy)，可以使用memset和memcpy对POD类型进行初始化。
* 提供对C内存的兼容。C++程序可以与C函数进行交互操作，POD类型保证这种在C与C++之间的操作总是安全的。

## 3.推荐初始化列表

在声明变量时，需要给它赋予初值。将初始值赋给变量的声明称为初始化。初始化有三种形式：

* 直接括号赋值：（expression list）
* 等号拷贝： = expression
* 初始化列表： {initializer-list}

举例:

|  |
| --- |
| //直接括号赋值 |
| int value(0);  int count(10);  int number(5); |
|  |
| //等号赋值 |
| int value = 0;  int count = 10;  int number = 5; |
|  |
| //初始化列表 |
| int value{};  int count{10};  int number{5}; |

在C++98/03中，就允许使用“{}”对聚合([五/1. Aggregates](#_Aggregates类型))类型进行统一的初始值设定，但是对于一些自定义类型(如vector这样的容器)却无法使用，这对于泛型编程无疑是非常不利的。在C++11中，使用“{}”初始化方式统一叫做“初始化列表”，该方式支持更多的类型，具有一些新的特性。相比直接括号赋值和等号拷贝的方式，具有以下优势:

1. 通用性好

在C++11标准中, 初始化列表的适用性大大增加，可以用于任何类型对象的初始化，包括简单的结构体/聚合类型，数组，标准库容器，类等。

举例:

struct Student

{

//√ 非静态成员变量初始化

int m\_id {0};

std::vector<int> m\_codes;

//√ 在构造函数中初始化非静态成员变量

Student():m\_codes{-1,-2}

{

...

}

};

//√ 初始化返回值

pair<string, string> getInfo(string key,string value)

{

return {key, value};

}

int main()

{

int lastCnt{ }; //√ 初始值为0

int curCnt{1}; //√ 初始值为1

int totalCnt = {2}; // 初始值为2(不推荐)

double res = double {1.2}; //初始值为1.2(不推荐)

//√ 初始值string

String code { '1', '2'};

//√ 嵌套列表初始化

[map](http://en.cppreference.com/w/cpp/container/map)<int, [string](http://en.cppreference.com/w/cpp/string/basic_string)> map

{

{ 1, "a" },

{ 2, { 'a', 'b', 'c' } },

{ 3, code }

};

//√ 绑定右值引用到临时int

int&& arr {1};

}

1. 避免与函数声明发生混淆

当直接使用括号初始化时，可能会写出下面有歧义的代码。

错误举例:

//× 1.被认为函数声明

Student wang();

//× 2.某些情况下也会被认为是函数声明

Student wang(type);

上例的注释1处这行代码是错误，直观上看，好像是一个初始化语句，显式调用Student的默认构造函数。但实际上这仅仅是一个函数声明，起不到初始化的作用，编译器不会抛出任何警告或错误。

上例的注释2处这行代码在会有歧义，当type恰好是一个类型名时，即使当前作用域中有一个变量名也是type，代码2还是会被认为是一个函数声明。

使用初始化列表可以完全避免上面的问题，初始化列表永远不会被认为是函数声明。

正确举例:

//1.√ 调用默认构造函数

Student wang{};

//2.√ 编译器自动查找最合适的构造函数

Student wang{type};

这里为了保证初始化的一致性，避免()初始化可能产生的问题，公司规定：

1. 调用默认构造函数，始终使用{}。
2. 多个参数初始化，始终使用{}。
3. 需要调用特定的构造函数时，才需要使用（）初始化。
4. 初始化类成员变量

类/结构体中优先使用初始化列表初始化成员变量，具体见

[十/5 鼓励使用初始化列表。](#_鼓励使用列表初始化)

1. 防止类型收窄

类型收窄是导致数据内容发生变化或者精度丢失的隐式类型转换。可能导致类型收窄的典型情况如下:

* 从浮点数隐式地转化为整型数。比如：int a = 1.2,这里a实际保存的值为1,可以视为类型收窄。
* 从高精度的浮点数转为低精度的浮点数，比如从long double隐式地转化为double，或从double转为float。如果这些转换导致精度降低，都可以视为类型收窄。
* 从整型(或者枚举)转化为较低长度的整型,比如:unsigned char = 1024，1024明显不能被一般长度为8位的unsigned char所容纳，所以也可以视为类型收窄。

C++11中，可以通过列表初始化来检查及防止类型收窄。

错误举例:

|  |  |
| --- | --- |
| int a = 1.1; | // 1 |
| float b = 1e40; | // inf |
| float c = (unsigned long long)-1; | // 1.8e19 |
|  | |
| const int x = 1024; | |
| char y = x; | // ‘0\’ |

上面的代码虽然都可以编译通过，但是由于发生了类型收窄，往往会导致一些隐藏的bug。

列表初始化的类型检查与类型转换将更严格，这种写法不允许有精度损失的类型转换，像下面的写法会直接导致编译无法通过，更加安全。

正确举例:

|  |
| --- |
| int a {1.1}; |
| float b {1e40}; |
| float c {(unsigned long long)-1}; |
|  |
| const int x = 1024; |
| char y {x}; |

1. 代码简洁优美

在数组，STL容器，泛型模块或者其它需要多个参数才能初始化的类型中，初始化列表可以带来极大的方便，使得代码优美简洁。

先看一个不用列表初始化的例子，初始化一个vector。

不好例子:

//简单的初始化，需要5行代码进行初始化

vector<int> ids;

for(int i = 0;i<4;++i)

{

ids.push\_back(i);

}

上例中，若使用列表初始化，只需要一行代码，简洁直观。

较好例子:

vector<int> ids{1,2,3,4};

再看一些列表初始化简化代码的例子。

举例:

struct Rectangle

{

public:

Rectangle( int width,int height );

draw( Rectangle );

private:

int m\_width { 0 };

int m\_height { 0 };

}

// 列表初始化创建对象

Rectangle rect {2, 1};

// 列表初始化在传参方面的便利

rect .draw({2,1});

对于聚合类型，初始化列表可以支持的很好，但是对于其它类型，有些注意点需要特别留意:

1. 类型含有自定义构造函数时，此时初始化列表将调用它对应的构造函数。如果{}内的参数没有与之匹配的构造函数，编译将出错。

举例:

class Pos

{

public:

pos(int x, int y);//含有自定义构造函数

private:

int m\_x { 0 };

int m\_y { 0 };

int m\_z { 0 };

}

//× 无法使用列表初始化每个成员变量,编译会出错

Pos pos { 1, 1, 1 };

//√ 调用自定义的构造函数赋值

Pos pos { 1, 1 };

1. 私有(private)或保护(protected)的非静态成员无法使用初始化列表。

错误举例:

class Pos

{

public:

int m\_x{ 0 };

int m\_y{ 0 };

private:

int m\_z{ 0 };

}

//× 含私有的非静态成员变量

Pos pos { 1, 1, 1 };

## 避免变量的不确定值

变量声明后不进行显式的初始化, 如（T object）的形式，这时编译器会进行默认初始化，默认初始化可能会产生不确定的值，不确定的值往往会引发隐藏bug，这种情况应当避免，产生不确定值有如下几种场合:

错误举例:

//Point属于POD类型

struct Point

{

int m\_x;

int m\_y;

}

//Rectangle具有用户自定义的构造函数，不属于POD类型

struct Rectangle

{

int m\_cx;

int m\_cy;

int m\_w;

int m\_h;

Rectangle():

m\_cx(-1),

m\_cy(-1)

{

}

}

【内置类型, 默认初始化后的值不确定】

int cnt; //cnt的值不确定

【POD类型, 默认初始化后成员的值不确定】

Point pt; //这里的m\_x和m\_y的值不确定

【非POD类型, 在构造函数(包括初始化列表)中没有进行

初始化成员的值不确定】

// m\_cx和ｍ\_cy的值为-1,m\_w和ｍ\_h的值不确定

Rectangle rect;

【数组类型, 数组的每一个成员进行默认初始化，值不确定】

double ratios[9];//每一个成员的值都不确定

在上例中可以看出，在类/结构体中可能在构造函数中遗漏了成员变量的初始化导致不确定（虽然在有时能够正确，但是不代表每次能够得到正确的值）。而且，大部分构造函数的实现在cpp文件中，当后续在头文件中增加成员变量时，可能在构造函数中忘记对该变量进行初始化。

在C++11中，可以在成员变量声明后使用{}进行初始化，这种初始化方式早于构造函数执行，也就是说只要在构造函数中没有进行另外的初始化工作，成员变量将按照{}中设定的值进行初始化。这可以有效避免变量的确定值，属于通用的初始化方式。

正确举例:

struct Point

{

int m\_x{0};

int m\_y{0};

}

struct Rectangle

{

int m\_cx{0};

int m\_cy{0};

int m\_w{0};

int m\_h{0};

Rectangle():

m\_cx(-1),

m\_cy(-1)

{

}

}

int cnt{ -1 }; //cnt的值为-1

Point pt; //m\_x和m\_y的值为0

Rectangle rect; //m\_x和m\_y的值为-1,m\_cx和m\_cy的值为0

double ratios[9]{-1,-1,-1};//前面3个元素为-1，后面的元素为0

注意1:

如同T object的形式的初始化叫做默认初始化，会导致不确定的值。但不包括static T object的形式，这种形式的初始化叫做零初始化，不会产生不确定值。

注意2:

上例中，虽然数组的长度为9,而{}初始化的值只有3个，但是这样写并不会出错，最终效果是前面3个元素为-1，后面剩余的元素都是按照零初始化的方式值为0。这里关于A[m]数组的长度与{}初始化值的数量之间n的具体关系如下:

* m = n 时，最理想的情况，A[m]严格按照{}中的值进行初始化。这种过程叫值初始化。
* n = 0 时，编译正常，A[m]所有的元素按照零初始化方式的初始化(如int的值为0，bool的值为false)。
* m < n时，编译出错。
* m > n时, A[m]前n个元素为值初始化，剩余元素为零初始化。虽然这样情况在语法上没有问题，但是为了可阅读性还是尽量避免这种情况。

注意3:

虽然建议类/结构体中的成员变量在声明的时候就是用列表初始化进行初始化，但是对于某些有效率要求的场合(比如在Pad检查中)，可以接受尽量在构造函数中对部分成员变量赋值，因为减小一次赋值可以提升效率。

## 空指针使用nullptr

在良好的C++编程习惯中，声明一个变量的同时，总是需要记得在何时的代码位置将其初始化，对于指针类型的变量尤其如此，未初始化的指针通常会引发一些难以调式的问题。

在C++11之前，使用最多的方式是NULL, 但是NULL有一个问题，在C语言中，NULL定义为void\*，也允许定义为0，但是在C++中，NULL必须是0。

#ifndef NULL

#ifdef \_\_cplusplus

#define NULL 0 //C++

#else

#define NULL ((void \*)0) //C语言

#endif

#endif

这样的定义，一不留神就会引起误解。

错误举例:

void func(char\*)

{

....

}

void func(int)

{

....

}

void main( )

{

func(NULL);//func(int)

return 0;

}

由于我们经常习惯于用NULL表示空指针，所以上面的代码容易被误认为是调用第一个func，但实际上，由于NULL在C++中是０，所以调用到第二个。虽然这个问题可以通过修改代码来解决，但是为了避免用户使用上的错误，有些编译器做了比较激进的改进，典型的如g++编译器，一旦遇到二义性就停止编译并向用户报告错误，虽然在一定程度上缓解了二义性带来的麻烦，但也会带来代码移植性的限制。

在C++11中，出于兼容性的考虑，NULL的二义性并没有消除，但是可以使用nullptr作为新的解决方案。nullptr是一个所谓"指针空值类型"常量，不被视为任何整型，例如，上面的代码，一定会调用func(char\*)。

正确举例:

void func(char\*)

{

....

}

void func(int)

{

....

}

void main( )

{

func(nullptr);//func(char\*)

return 0;

}

综上，使nullptr代替NULL表示空指针。

举例:

|  |  |
| --- | --- |
| int\* pNumber { nullptr }; | // √ |
| int\* m\_pNumber { nullptr }； | // √ |
| int\* g\_pNumber { nullptr }； | // √ |
| int\* pNumber ； | // × |
| int\* pNumber {NULL}； | // × |
| int\* pNumber {0x0}； | // × |

## 类型转换

C风格的强制类型转换很简单((T) expression/ T(expression))，这种强制类型转换不管表达式的值是什么类型，系统都自动将其转为赋值运算符左侧变量的类型，若不加注意，就可能产生错误。

错误举例:

//× dThres到fThres发生截断,fThres值65535.11328,丢失精度

double dThres = 65535.1123;

float fThres = (float)dThres;

//× cSum 发生了数据溢出，cSum = -1

int iSum = 65535；

char cSum= (int)iSum；

//两个不相关的类Person和Animal

class Person{…};

class Animal{…};

/\*× 把Person类型转化为Animal类型，下面语句能够编译

通过，然而因为Person和Animal是完全2种不同的类型，这

种强制转化会导致在运行时出错或者其它难以预见的错误\*/

Person\* person = new Person();

Animal\* animal = (Animal\*) person ;

对于上面的类型转化错误，编译器不会发生任何错误，只会抛出一条非安全类型转换的警告信息。程序员可能被蒙在鼓里。这与C++中一直强调的类型安全是相悖的。

C++是一种强类型的编程语言，如果希望程序顺利通过编译，就不应该试图对任何对象做任何不安全的转换操作，基于这一原则C++针对特定的目的提供了以下四种形式的强制转换。这些类型转换编译器都会根据具体的类型信息进行一些分析处理及相应的异常保护(reinterpret\_cast除外)。

1. const\_cast<T\*>(variable)

用于从一个类中去除const、volatile和unaligned。

举例:

class Student {...};

const Student\* pStud = new Student();

//× 不能将const对象指针赋值给非const对象

Student\* pTempStud = pStud;

//√ 编译通过

pTempStud = const\_cast< Student\*>(pStud );

注意:

* 虽然const\_cast可以让某些功能编译通过，但通常情况下不建议去除const属性，因为强制去除变量的const属性不仅会使原本的保护设计失效，轻则埋下隐患，重则可能导致程序崩溃，如有特殊原因必须要用请征求直属主管同意方可实施。
* C++11有最新的去除const属性的方式remove\_const，如果必须使用优先推荐它；

1. reinterpret\_cast<T\*>(variable)

C++中最为暴力的类型转换，所实现的是一个类型到一个毫不相关，完全不同的类型的映射。它是所有类型转换中最危险的，编译器直接放弃检查，效果和直接使用C风格的强制转化一样。

错误举例:

//两个不相关的类Person和Animal

class Person{…}

class Animal{…}

//把Person类型转化为Animal类型

Person\* person = new Person();

Animal\* animal = **reinterpret**\_cast<Animal\*>(person);

因为Person和Animal是不相关的类，强行转换成功的概率很小(是否成功依赖于类型内存发布情况，即使成功也很不安全，因为类型随时可能发生变动)，然而类型转换始终能够编译通过，这种强制转换会可能导致程序在运行时崩溃或者其它难以预见的错误。

在某些场合适合用reinterpret\_cast，比如像类指针与void\*的转换。

正确举例:

//生成Hash函数，通过让两个不同的值映射成不同的索引

unsigned short hash(void \*addr)

{

unsigned int val = reinterpret\_cast<unsigned int>(addr);

return (unsigned short)(val^(val >> 16));

}

int main()

{

int a[20] ;

for( int i = 0; i < 20; i++)

{

cout<< hash(a+i)<<endl;

}

}

reinterpret\_cast是所有类型转换中最危险的，只有特殊情况下才需要使用，使用前需征得直属主管同意。

1. static\_cast<T\*>(variable)

静态检查类型，包括内置类型(如int，float，bool等基本的数据类型)之间的转化，类指针之间的转换。其中类指针之间的转换包括派生类到基类指针的上行转换upcasts和基类到派生类指针的下行转换downcasts。当进行下行转换时，由于没有动态类型检查，所以是不安全的。

举例:

【内置类型的转换】

float fDividend = 21.7;

int iDividend = 7;

int divisor = 3;

cout << static\_cast<int>( fDividend )/ divisor <<endl;

cout << static\_cast<int>( iDividend )/ divisor <<endl;

输出为:

7

2

【类指针的转换】

class Bird{...};

//Student 继承Person

class Person {…};

class Student : public Person {…};

Student\* pStu = new Student();

Person\* pPerson = new Person();

//× 类型检查，Student\*无法转换为Bird\*,编译失败

Bird\* pBird = **static\_cast**< Bird\*>( pStu );

//√ 派生类转化为基类，安全

Person\* pTempPerson = **static\_cast**< Student\*>( pStu );

//× 基类转化为派生类，不安全

Student\* pTempStu = **static\_cast**< Person\*>( pPerson );

注意:

由于static\_cast进行数据转换会具备一定的安全检查，相比于reinterpret\_cast这种毫无关联类型之间的转化更加安全，但是类型检查又不是很彻底，缺乏下行转换的安全检查，所以公司规定static\_cast只准用于像int，float，double等内置类型之间的转换，对于动态对象的转换，全部使用dynamic\_cast。

1. dynamic\_cast<T\*>(variable)

只能用于指针和引用的类型转换，安全地将指针或引用根据继承关系上行(upcasts)、下行(downcasts)甚至向同一继承层次的其它类型转换。

举例:

//抽象基类Bird

class Bird

{

public:

Bird ( ) {…}

virtual ~Bird ( ) {…}

};

//Sparrow继承Bird

class Sparrow : public Bird

{

public:

Sparrow ( ) {…}

virtual ~Sparrow ( ) {…}

};

/\*基类指针转换为派生类指针。如果转换成功，返回派生类指针；

如果转换失败，返回nullptr；这里直接返回nullptr\*/

Bird\* bird = new Bird();

Sparrow\* greenSparrow= dynamic\_cast< Sparrow \*>( bird );

/\*派生类指针转换为基类指针如果转换成功，返回的是基类指针；

如果转换失败返回nullptr。这里直接返回基类指针\*/

Sparrow\* sparrow = new Sparrow();

Bird\* birdA= dynamic\_cast< Bird \*>( sparrow );

通常派生类转到基类指针式为了循环迭代方便，而派生类指针还是存在的。但在极端情况下可能不存在派生类指针，只有基类指针，所以此时需要基类指针向派生类的转换。但这种情况毕竟少见，如果真的要用请示主管后决定。

## const的用法

1. 修饰局部变量

使用const定义局部变量，表示变量不可变，可以避免在函数中对某些不应修改的变量造成可能的改动。该变量必须在声明的时候就进行初始化。

举例:

int main()

{

//const 变量必须初始化，否则会编译通不过

const int pi = 3.14159;

//可以这样初始化，编译通过

const int temp = pi+1;

//× 编译失败，不可以修改常量的值

temp ++;

...

}

1. 修饰指针

指针可以被修饰为常量。const修饰指针时既可以用来修饰指针指向的内容，也可以用来修饰指针地址本身。主要有以下3种情况:

* 指向**常量实例**，不能修改被指向的对象,但可以使指针指向其它对象。

举例:

int value { 5 };

**const** int\* pValue { &value };

//× 编译不通过，不能修改指针指向对象的值

\*pValue = 6;

//√ 指向的地址可以修改

pValue = nullptr;

* 指向常量指针，不能修改地址，但是可以修改指针对象的对象。

举例:

int value { 5 };

**int\* const** pValue { &value };

//√ 编译通过，能修改指针指向对象的值

\*pValue = 6;

//× 编译不通过，不能修改地址

pValue = nullptr;

* **指针**和**实例**都定义为常量，都不能修改。

举例:

int value { 5 };

**const int\* cons**t pValue { &value };

//× 编译不通过，不能修改指针指向对象的值

\*pValue = 6;

//× 编译不通过，不能修改地址

pValue = nullptr;

1. 修饰引用

const还可以用来修饰引用。const放在类型前面和后面效果一样，都是表示引用不能被更新。

举例:

**int const &**highValue{ 5 };

**const int &**lowValue{ 1 };

//× 编译不通过，不能修改值

highValue = 20;

//× 编译不通过，不能修改值

lowValue = 0；

1. 修饰函数参数

函数参数分为输出型和输入型两类，const适用于输入型参数，被const修饰的参数不能被修改。

举例:

【参数指针指向的内容为常量】

void setStringInfo ( **const** char\* info)

{

//× 编译不通过，不能修改info的内容

strcpy(info,"context changed!");

}

【参数指针地址为常量不可修改】

void setStringInfo ( char\* **const** info)

{

//× 编译不通过，info的地址不可修改

info= new char[6];

}

【参数为引用不可修改】

void setStringInfo ( **const** string& info)

{

//× 编译不通过，info的内容不可修改

info = "context changed!";

}

1. 修饰函数返回值

使用const修饰函数返回值，返回值将不能修改，无论指针还是引用。

举例:

//const修饰返回值，不能对返回值进行任何修改

class Student

{

public:

…

const string& getName( ) { return name}

…

private:

std::string m\_name;

…

};

//修改name时，会造成编译不通过

int main()

{

Student stu {};

String& name = stu.getName();

name = “new name!”;//× 编译不通过

}

1. 修饰成员变量

const可以用来修饰成员变量，该成员变量只能类内初始化或者构造函数中使用初始化列表初始化。

举例:

class Student

{

public:

Student();

private:

//√ 类内初始化

const int m\_id{-1};

}

//√ 在构造函数中使用初始化列表初始化

Student::Student():m\_id(-1)

{

//× 编译不通过

m\_id ＝ 2;

...

}

1. 修饰成员函数

const 后置修饰的成员函数表示不允许在该函数内对它所在类的成员变量进行任何修改，一旦这么做了，编译器就会弹出错误。同时它也不能调用没有被const修饰的成员函数。

错误举例:

class Student

{

public:

…

void printDetail( ) const;

std::string getDetailString() ;

…

private:

std::string m\_name;

…

};

void Student::printDetail ( ) const

{

…

/\*× 编译不通过，const后置修饰的成员函数不能修改成

员变量\*/

m\_name = "Hello";

/\*× 编译不通过，const后置修饰的成员函数不能调用非

const的成员函数\*

std::string detailString = getDetailString();

…

}

1. 修饰常量对象

当一个对象声明为常量类型时，表示它的成员变量在其生命周期内不能修改，同时该对象也只能调用其const的成员函数。

举例:

class Student

{

…

string getName()const;

void setName(std::string name)

{

m\_name = name;

}

void printDetail( ) ;

…

private:

std::string m\_name;

…

};

int main ( )

{

const Student student{};

//× 编译不通过，const对象不能修改成员变量

student.setName(“wang”);

//× 编译不通过，const对象不能调用非const的成员函数

student.printDetail();

//√ 编译通过，可以调用const后置修饰的函数

string name = student.getName();

…

}

## 0值比较

0值在C++中比较特殊，包括几种角色:整型0、空指针nullptr、字符串结束符‘\0’、逻辑FALSE/false，不同的角色适用于不同的情形，使用的时候要特别注意。

1. 确认0是否在类型的范围内下例是简单的0值判断逻辑，却隐藏bug。

错误举例:

unsigned char count = 10;

//× count 不可能为负，所以将不能跳出循环

while ( **count >= 0** )

{

…

count --;

}

因为count总是大于等于0，所以任何值都满足while循环判断条件，想要实现原本设计的功能，只要把count的数据类型改成int就可以了。

正确举例:

int count = 10;

//√ count 可能为负，具备跳出条件

while ( **count >= 0** )

{

…

count --;

}

1. 浮点数不存在绝对精确等于0的值，浮点数的0只不过是一个十分趋近于0的值。

错误举例:

float xPosition {0.00013};

/\*× 浮点数不存在绝对精确等于0的值，所以下面的判断条

件永远不满足\*/

if( **0 == xPosition** )

{

…

}

浮点零比较正确的做法，设定一个精度，通过比较值于浮点0的绝对值差是否在该精度内来判断浮点数是否为0。

正确举例:

float xPosition {0.00013};

//设定指定精度为0.0001，这个精度是由实际的应用决定的

if(**abs(xPosition-0)<0.0001**)

{

…

}

事实上，不止是0值比较，对于浮点型变量数是否为指定的值的判断都是通过比较变量于指定值的绝对值之差是否小于指定的精度。这个精度应该是在软件中根据实际应用提前设定好的宏定义或者是常量,下面以3DAOI中判断体积和高度是否相等进行详细说明。

举例:

/\* 判断元器件的体积

curVol : 当前元件的体积

standardVol: 标准元件的体积

VOLUME\_ACCURACY: 精度的宏定义

一般元件的长宽高会高达2位整数(<99)是有可能的，那么体积

就是高达6位的整数(<99\*99\*99)，如果使用单精度浮点，精度

为7位数字，那么小数位只能控制到1位了，所以

VOLUME\_ACCURACY为0.1，\*/

#define VOLUME\_ACCURACY 0.1

if(abs(curVol – standardVol)<VOLUME\_ACCURACY)

{

…

}

/\* 判断元器件的高度

curHeight : 当前元件的高度

standardHeight: 标准元件的高度

HEIGHT\_ACCURACY: 精度的宏定义

如果以mm为单位，一般元件的高会高达2位整数(<99)，如果

使用单精度浮点，精度为7位数字，那么小数位只能控制到6

位(99.9999x)，精度可以达到是百纳米了，所以

HEIGHT\_ACCURACY为0.0001\*/

#define HEIGHT\_ACCURACY 0.0001

if(abs(curHeight–standardHeight)<HEIGHT\_ACCURACY)

{

…

}

## 尽量避免#define

宏替换只在预处理阶段起作用，由于缺少必要的类型检查，使得编译器无法检测其中的Bug，应当尽量避免，解决的方案有:

1. 使用const/enum代替#define定义常量

使用#define定义常量时，常量不会加入到符号列表中，如果代码中使用了这个常量，也许会遇到一些很不容易察觉的错误，往往很难找到问题的所在。

错误举例:

#define ASPECT\_RATIO 1.653

在上面的代码中，如果发生错误，出错信息只会涉及到1.653，而对ASPECT\_RATIO只字不提，如果 ASPECT\_RATIO是在某个不知情的头文件中定义，那么寻找1.653的出处就是大海捞针。

解决的办法就是用常量来代替宏定义:

正确举例:

const double aspectRatio = 1.653

aspectRatio 作为语言层面的常量，最终会被编译器看到，确保会进入符号列表。另外，对于浮点数常量，使用常量较#define会生产更小的目标代码，这是由于预处理器会对目标代码所有出现ASPECT\_RATIO复制出一份1.653，而aspectRatio 永远只有一份。使用常量编译效率也更高。

除了const，还可以使用enum代替#define，如下面例子。相比于const，enum强制不允许取地址，更加安全。

举例:

//修改前

#define MAX\_HEIGHT 320

#define MAX\_WIDTH 250

class Student

{

…

private:

//修改后

enum

{

MAX\_HEIGHT = 320,

MAX\_WIDTH = 250

};

}

1. 使用inline代替#define函数

使用#define定义函数需要特别小心，里面存在着诸多的陷阱。

* 要使用完备的括号。

错误举例:

/\*如下代码实现参数a和参数b的求和，但是3种定义都存在

一定风险\*/

#define ADD( a, b ) a + b;

#define ADD( a, b ) ( a + b );

#define ADD( a, b ) ( a ) + ( b );

/\*使用下面代码实现(a+b)\*(c+d)的求值\*/

ADD( a , b )\*ADD( c, d);

/\*分别采用3种宏定义后，代码展开后，只有第2种实现了

原来的意图\*/

a + b \* c + d

( a + b ) \* ( c + d )

( a) + ( b ) \* ( c ) + ( d )

/\*当采用MULTIPLE(a,c)实现乘法\*/

#define MULTIPLE( a, b ) ( a \* b );

MULTIPLE( a + b , c );

/\*展开后，得到了错误的结果\*/

a + b \* c

要避免这些问题，需要非常小心的用括号完备地保护各个宏参数。

正确举例:

#define ADD( a, b ) ( ( a ) + ( b ));

#define MULTIPLE( a, b ) ( ( a ) \* ( b ) );

* 不允许参数发生变化。

错误举例:

#define SQUARE( a ) ((a) \* (a))

int square( int a )

{

return a\*a;

}

int nValue1 = 10, nValue2 = 10;

int nSquare1 = SQUARE(nValue1++);

int nSquare2 = square(nValue2++);

输出:

nSquare1 = 110, nValue1 = 12

nSquare2 = 100, nValue1 = 11

类似的定义，却产生了不同的结果，究竟原因还是宏的字符替换问题，正如上面的示例，两处的a都被参数nValue1++替换了，所以nValue1自增操作执行了两回。为了避免这样的副作用，最简单有效的方法就是保证宏参数不发生变化。

* 用大括号将宏所定义的多条表达式括起来。

如果宏定义包含多条表达式，一定要用大括号括起来，如果没有大括号，很可能只有第一条语句被执行。

错误举例:

#define CLEAR\_CUBE\_VALUE( l , w, h )\

l = 0;\

w = 0;\

h = 0;\

for(int i = 0;i < CUBE\_COUNT; ++i)

{

//× 只有第一条变被清空

CLEAR\_CUBE\_VALUE(

cubes[i].l,

cubes[i].w,

cubes[i].h)

}

正确的做法应该用大括号将多条表达式括起来，这样能够保证所有的表达式全部执行了

正确举例:

#define CLEAR\_CUBE\_VALUE( l , w , h )\

{\

l = 0 ;\

w = 0 ;\

h = 0 ;\

}\

使用宏定义定义函数不得不把精力放在这些毫无意义的事情上，一不小心就会产生Bug，查错起来也很困难，所以应当尽量避免使用宏函数。

还好有c++中有替代方案，就是inline函数。inline函数具有函数的性质，参数传递不管是传值还是传引用，都不会对参数进行重复计算，同时对参数进行类型检查；inline函数也是在代码中做代码展开，效率上比宏逊色。同样是求和函数，使用inline可以这样实现。

正确举例:

inline int add(int a, int b)

{

return a + b;

}

由于上面是一个真实的函数，函数内部不需要为参数加括号，也不用担心参数被多次更改，遵循作用域和访问权限，也方便调式查错。

注意:

一般情况下鼓励使用inline函数代替宏函数，但是某些情况下还是必须得使用宏函数。鉴于inline的特殊性，具体见：

[二/3 内联函数](#_内联函数)**。**

对于稍微复杂一点的函数，但是又想避免函数调用的开销，只能用宏函数。

1. 类型重定义

宏定义还经常用于类型的重定义，如：

#defineDWORD unsigned int

这种方式完全可以使用typedef和using取代。

typedef unsigned int DWORD

using DWORD = unsigned int

1. 条件编译

有时候必须用#define宏定义，如防止#include重复包含头文件具体见：[二/1. 使用#include防止重复包含](#_使用#include防止重复包含)。

举例:

#ifdef setValue

cout<<"\"setValue\" is define!"<<endl;

#else

cout<<"\"setValue\" is not define!"<<endl;

#endif

## 逗号表达式

逗号表达式是从C语言继承而来的，整个逗号表达式的值是最右边表达式的值。

举例:

int a { 1 };

a = ( 1+2**,** 3+4**,** 5+6);

cout<<a<<endl; //输出11

cout<<( 1 \* 2**,** 3 \* 4**,** 5 \* 6)<<endl; //输出30

上面的例子只是为了让大家了解逗号表达式的值是最右边表达式的值，但实际使用中很少这么用，在许多情况下，使用逗号表达式的目的只是想分别得到各个表达式的值，而并非一定需要得到和使用整个逗号表达式的值，如下面的例子。

举例1:

#include <stdio.h>

int main ()

{

/\*并没有用到整个逗号表达式的值，仅仅是用于连续声明

及初始化\*/

int a = 10, b = 20, c= 30;

//注意，=的优先级高于‘,’，所以输出为20

int temp = b, c;

printf("%i\n", temp );

//注意，括号的优先级高于=，所以输出为30

i = (b, c);

printf("%i\n", i);

}

举例2:

/\*在元编程中，无法使用if判断，巧妙的结合了C++的SFINAE

机制以及逗号表达式，实现了只要Check成功，就返回true，否

则就返回false的逻辑。\*/

template<typename U>

static auto Check(int)->decltype(

std::declval<U>().member(std::declval<Args>()…)**,**

std::true\_type());

template<typename U>

static std::false\_type Check(…);

# 内存管理

## new/delete、new[]/delete[]

使用C++时禁止使用malloc/free来管理内存，请使用new/delete和new[]/delete[], 并且new和delete、new[]和delete[]必须成对使用，否则会导致内存泄漏。

内置类型会自动析构，所以在针对内置数据类型时，释放内存使用delete或delete[]的效果是一样的（虽然可以这么用，但是公司要求数组还是要必须使用delete[]，从而培养良好习惯。）

对于非内置数据类型时一定要用delete[]来释放。

举例:

【对于内置类型】

int\* pArray = new int[10];

......

**delete** pArray; //× 虽然这样可以使用，但公司禁止这么写

**delete**[] pArray; //√

【对于非内置类型】

Box\* boxes = new Box[10];

……

**delete** boxes; //× 只有boxes[0]被释放了

**delete[]** boxes; //√

## 智能指针

智能指针能够实现了指针的自动释放，总是自动正确地销毁动态分配的对象，使用时需要引用头文件<memory>。C++11提供了3种智能指针：

* std::shared\_ptr：使用引用计数，每一个shared\_ptr的拷贝都指向相同的内存。在最后一个shared\_ptr析构的时候，内存才会被释放。
* std::unique\_ptr：不允许其他的智能指针共享其内部的指针，不允许通过赋值将一个unique\_ptr赋值给另一个unique\_ptr。
* std::weak\_ptr：用来监视shared\_ptr的，不会使引用计数+1，不管理shared\_ptr内部的指针，主要是为了监视shared\_ptr的生命周期，更像一个shared\_ptr的助手。

使用智能指针需要注意以下几方面:

1. 不能将一个原始指针直接赋值给一个智能指针。

错误举例:

std::shared\_ptr<int> pNumber = new int(1); //×

正确举例:

std::shared\_ptr<int> pNumber(new int(1)); //√

1. 指定删除器

使用shared\_ptr管理动态数组，因为shared\_ptr的默认删除器不支持数组对象。所以需要指定删除器。

举例:

//指定删除器来删除动态数组

std::shared\_ptr<int> Person(new int[10] , [] (int\* p) { **delete[ ]** p ; });

除了动态数组，有时候一个类对象也有可能需要按需要指定删除器，实现按需释放，如下例的对象池类ObjectTool。

举例:

template<typename T>

class ObjectTool

{

public:

template<typename... Args>using Constructor =

std::function<std::shared\_ptr<T>(Args...)>;

...

template<typename... Args>void init(size\_t num,

Args&&... args)

{

auto typeName= typeid(Constructor<Args>).name();

for(size\_t i = 0;i < num;++i)

{

/\*为了能够实现对象池用完后的“入池”操作，

设计专用的删除器,调用createPtr方法\*/

m\_objectMap.emplace(typeName,shared\_ptr<T>(new

T(args...),[this,constructName](T\* p)

{

return **createPtr**<T>

(string(constructName),args...);

});

}

}

/\*当m\_isNeedClear为true时，表示需要真正的删除对象，

回收内存，使用delete当m\_isNeedClear为false时，

表示仅仅把对象放入对象池以备后续的使用，

对象在内存中还是存在的\*/

template<typename T,typename... Args>

std::shared\_ptr<T> **createPtr**(std::string&

constructName,Args...args)

{

return std::shared\_ptr<T>(new

T(args...),[constructName,this](T\* t)

{

if(m\_isNeedClear)

{

delete[] t;

}

else

{

m\_objectMap.emplace(constructName,

std::shared\_ptr<T> t);

}

});

}

private:

bool m\_isNeedClear;

int m\_maxObjectNum{10};

std::multimap<std::string,std::shared\_ptr<T>> m\_objectMap;

};

1. 不要用一个原始指针初始化多个shared\_ptr。

错误举例:

class Student

{

public:

Student();

~Student();

private:

int\* m\_pID{ nullptr };

}

Student::Student()

{

m\_pID = new int[11];

}

Student::~Student()

{

/\*当同一个原始指针初始化两个智能指针时，会产生重复

析构，如果析构函数中没有进行nullptr的判断的话，

就会导致异常\*/

**delete[] m\_pID ;**

}

//将同一个原始指针pStudent分别初始化两个智能指针

Student\* pStudent = new Student();

shared\_ptr<Student> wang(**pStudent**);

shared\_ptr<Student> wu(**pStudent**);

上面的代码虽然可以编译和运行，然而会造成重复析构，如果类的析构函数没有设计好，类的动态对象在析构函数中删除前没有进行nullptr的判断，就会导致异常发生。

1. 避免循环引用

shared\_ptr最大陷阱就是循环引用，循环引用会导致内存泄露。

错误举例:

struct Driver;

struct Bus;

struct Driver

{

std::shared\_ptr<Bus> bus; //× 循环引用

…

};

struct Bus

{

std::shared\_ptr<Driver> driver; //× 循环引用

}

void testPtr()

{

//这个使用循环引用的后果是导致两个对象都不会被删除

std::shared\_ptr<Car> car(new Car());

std::shared\_ptr<Driver> driver(new Driver());

car->driver= driver;

driver->car = car;

}

weak\_ptr可以解决循环引用的问题。把上例对象中的任意一个成员变量改为weak\_ptr。

正确举例:

struct Driver;

struct Bus;

struct Bus

{

std::shared\_ptr<Driver> bus;

…

};

struct Driver

{

std::weak\_ptr<Bus> car;

}

1. 不要在函数实参中创建shared\_ptr。

因为c++的函数参数的计算顺序在不同的编译器不同的调用约定可能是不一样的，所以不要在函数实参中创建shared\_ptr。

错误举例:

function (shared\_ptr<int>(new int) , test());

在上例中，可能的过程是先new int，然后调用test，如果恰好test()发生了异常，而shared\_ptr<int>还没有被创建，则产生内存泄露。正确的做法应该是先创建智能指针。

正确举例:

shared\_ptr<int> arr(new int());

function (arr , test());

1. 通过shared\_from\_this()返回this指针。

不要将this指针作为shared\_ptr返回出来，因为this本质上是一个裸指针，因此可能会导致重复析构。

错误举例:

class Person

{

shared\_ptr<Person> getSelf()

{

return shared\_ptr<Person>(**this**);// ×

}

}

int main()

{

shared\_ptr<Person> wang( new Person() );

shared\_ptr<Person> wu = wang->getSelf();

...

}

在这个例子中，由于用同一个指针(this)构造了两个智能指针，他们之间是没有任何关系的，在离开作用域后this将会被构造的两个智能指针各自析构，导致重复析构的错误。

正确返回this的shared\_ptr做法是让目标类通过派生std:enable\_shared\_from\_this<T>类，然后使用基类的成员函数shared\_from\_this来返回this的shared\_ptr。

正确举例:

class Cat::public std::**enable\_shared\_from\_this<Cat>**

{

shared\_ptr<Cat> getSelf()

{

return shared\_from\_this();// √

}

}

1. 指针不允许复制的场合请使用unique\_ptr

在标准库容器中，为了提高效率以及支持多态性，通常在容器元素为指针类型而不是实例，一般而言，容器元素为指针会比实例快2-3倍。

使用指针作为标准库容器对象虽然可以提高效率，但是问题就是指针有被容器外部销毁的风险，会造成异常、内存泄露或者其它更严重的问题。

使用智能指针可以解决标准库容器对于指针元素的所有权问题，shared\_ptr可以实现指针元素没有引用后自动销毁，但是shared\_ptr比较复杂，运行起来也更慢，消耗的内存更多，在高效率要求的场合使用更加轻量的unique\_ptr是更好的选择。

作为c++11提供的新的指针指针，unique\_ptr的主要特点如下：

* 指针不能复制，不提供拷贝构造函数和拷贝赋值函数。
* 只有一个智能指针包含对象的所有权，在对象生命周期结束后只有一个智能指针调用对象的析构函数。
* 没有引用计数和线程安全的方法，性能比shared\_ptr要高。实验证明，unique\_ptr的性能相对于使用原始指针几乎一样。
* 支持指向一个动态数组。如:unique\_ptr<**int[]**> ptr( new int[10] );

举例:

unique\_ptr<T> myPtr(new T);

unique\_ptr<T> otherPtr = mptr; //× 编译错误

注意:

* 虽然unique\_ptr不允许复制，但是可以通过函数返回给其他的unique\_ptr，还可以用通过move来移动到其他unique\_ptr。
* unique\_ptr不能通过make\_shared方法来创建，它不允许其他的智能指针共享其内部的指针，不允许通过赋值将一个unique\_ptr赋值给另一个unique\_ptr。

关于具体何时用unique\_ptr,何时用shared\_ptr的综合例子在：

[六/3.容器中指针的管理](#_容器中指针的管理)。

1. 禁止使用auto\_ptr

auto\_ptr是C++11之前的产物，虽然也能实现自动的内存管理，但是存在重大缺陷：

* auto\_ptr对象不能作为STL容器的元素。
* 缺少对动态配置而来的数组的支持
* auto\_ptr在被复制的时候会发生所有权的转移
* 不安全，隐性的左值搬移。

鉴于auto\_ptr的种种缺点，并且C++11有了新的更好的方式替代它，所以公司禁止使用auto\_ptr。

1. 建议多使用智能指针管理第三方库

传统的第三方库调用方式存在遗忘释放内存的风险，因为在第三方库创建、释放实例很可能是两条指令，如果调用了create( )之类创建实例的方法而最终忘记了调用release（）之类释放实例的方法或者创建实例后未执行到释放前在中间过程就因为种种原因（可能是碰到异常）提前返回了，就可能会导致内存泄漏。而通过宏定义和智能指针结合起来可以实现第三方库的安全管理，避免内存泄露。

错误举例:

/\*如果忘记调用release方法或者在执行到release之前发生了

异常退出，将产生内存泄露\*/

void\* ptr = getHandle()->create();

...

ptr.release();

正确举例:

#define GUARD(p) std::shared\_ptr<void> p##p(p,[](

void\* p){p->release();});

void\* ptr = getHandle()->create();

GUAD(ptr);//安全

## 容器中指针的管理

在STL使用指针要注意遵守RAII原则，在容器中可以使用裸指针，也可以使用智能指针，特别小心的是使用原始指针时，要避免内存泄漏。

错误举例:

/\*当persons超出作用域会被销毁，但是销毁的只是指针，

delete并没有作用于new得到的对象，于是内存泄漏。\*/

void addPerson()

{

vector<**Person\***> pPersons;

int personCnt = 10000;

Person\* pPerson = new Person;

for( uint i = 0 ; i < personCnt ; ++i )

{

persons.push\_back( new Person());

}

**}//出了这个作用域，会发生内存泄露**

如果集合数量多，为了效率必须可以使用原始指针，但要记得释放；如果集合数量少，请使用智能指针来进行集合中指针类型的内存管理。

效率测试:

/\*以100W个Pad进行效率测试，在实际情况中Pad都会小于

20W个\*/

int testCnt = 1000000;

Pad\* pads[testCnt];

for(int i = 0;i < testCnt; ++i)

{

pads[i] = new Pad();

}

//以裸指针，shared\_ptr和unique\_ptr进行效率对比

vector<Pad\*> pRawPads;

vector<shared\_ptr<Pad>> sharedPads;

vector<unique\_ptr<Pad>> uniquePads;

//raw pointer

auto startClock = steady\_clock::now();

for(int i = 0;i < testCnt; ++i)

{

pRawPads.push\_back(pads[i]);

}

auto rawDuration= duration\_cast<milliseconds>(steady\_clock::

now() - startClock);

cout<<rawDuration.count()<<endl;

//shared\_ptr

auto startClock1 = steady\_clock::now();

for(int i = 0;i < testCnt; ++i)

{

sharedPads.push\_back(shared\_ptr<Pad>(pads[i]));

}

auto sharedDuration= duration\_cast<milliseconds>

(steady\_clock::now() - startClock1);

cout<<sharedDuration.count()<<endl;

// unique\_ptr

auto startClock2 = steady\_clock::now();

for(int i = 0;i < testCnt; ++i)

{

uniquePads.push\_back(unique\_ptr<Pad>(pads[i]));

}

auto uniqueDuration= duration\_cast<milliseconds>

(steady\_clock::now() - startClock2);

cout<<uniqueDuration.count()<<endl;

输出结果:

|  |  |
| --- | --- |
| 裸指针： | 6 ms |
| shared\_ptr： | 41 ms |
| unique\_ptr: | 7 ms |

从100W个Pad的测试结果上看，unique\_ptr和裸指针的效率几乎一样，而shared\_ptr的性能下降了6-7倍。对应我们行业实际的计算规模，一般检测对象小于20W个，除非确认需要共享指针或者保证线程安全，不然推荐使用unique\_ptr。

## 对象池

对象池通过一次性批量申请内存，降低了内存的申请、释放次数，从而节省了时间。对于大批量的对象而言，使用对象池方式，对性能提升效果显著。

举例:

该例子的完整版在《深入应用C++11》P238。

//实现一个对象池类ObjectPool

class ObjectPool

{

public:

//对象池的实现

template<typename T>

class objectPool : INonCopyable

{

…

//默认创建多少个对象

template<typename…Args>

void init( size\_t num, Args&&… args)

{

…

for( size\_t i = 0; i <num; ++i)

{

m\_objectMap.emplace(constructName,

shared\_ptr<T>(new T(std::forward

<Args>(args…),[this, constructName] (T\* p)

/\*删除器中不直接删除对象，而是回收到

对象池以供下次使用\*/

{

**m\_objectMap.emplace(std::move(construct**

**Name),std::shared\_ptr(p));}));**

}

}

//从对象池中获取一个对象

template<typename…Args>

std::shared\_ptr<T> get( )

{

…

for( auto it = range.first; it != range.second; ++i)

{

auto ptr = it ->second;

//每获取一个对象，从map中移除

**m\_objectMap.erase(it);**

retrun ptr;

}

return nullptr;

}

…

}

}

private:

//所有的已经批量申请好的对象放入map中

multimap<string, std::shared\_ptr<T>> **m\_objectMap**;

…

};

//数据密集型类型，对象池管理的类型

struct BigObject

{

BigObject（）{…}

BigObject ( int value) {…}

BigObject ( const int& valueOne, const int& valueTwo){…}

…

};

//测试代码

int main( )

{

ObjectPool<BigObject> pool;

pool.init(2); //初始化对象池，初始创建两个对象

auto poolOne = pool.Get( );

auto poolTwo = pool.Get( );

}

## 内存越界

申请一块内存之后，使用时如果超过申请时的内存范围，会造成内存越界，带来极大风险，请务必要避免。

举例:

int data[3] = { 1, 2, 3 }; //申请3个字节的int数组

cout << data[4] << endl; //访问第4个元素

运行结果：

-1231511328 //每次结果都不确定

此时能正常运行，但是访问了其它内存的数据，如果对这个数据进行更改，就有可能对程序中其它地方造成无法预估的恶劣影响，所以这很危险必须要时刻注意避免内存越界。

## 使用RAII技术

坚持RAII，在其**构造时获取资源**(init)，在对象生命周期中控制资源的访问，最后在对象**析构时释放资源**(release)，如果资源生命周期无法确定，请使用智能指针。

错误示例:

//Student类包含了动态对象m\_favoriteCodes

class Student

{

public:

Student()

{

init();

}

~Student(){…}

void **init**(int\* codes){ m\_favoriteCodes =new int[100];}

void **release**(){delete[] m\_favoriteCodes; }

…

private:

int\* m\_favoriteCodes ;

…

};

/\*这是个错误的示例，如果实例化一个Student却忘了调用

release函数，或者程序异常退出与直接返回都将不能释放创

建在堆上的内存。\*/

int main()

{

Student student；

… //如果此处发生异常，release将得不到执行

student.release(); //手动释放创建在堆上的内存

…

}

RAII技术实现内存自动释放，可以很好的解决上述的隐患。

正确示例:

//使用RAII技术对Student进行优化

class Student

{

public:

Student()

{

init();//调用init函数，初始化成员变量

}

~Student()

{

**release**();//调用release函数，释放内存

}

void init()

{

m\_favoriteCodes = new int(100);

}

void release()

{

delete m\_favoriteCodes;

}

private:

int\* m\_favoriteCodes;

};

int main()

{

Student student；

…//即使在此处发生异常，资源还是能够释放

}//出了作用域，release能够正确得到调用。

# 面向对象

## 调用关系

调用关系包括项目、目录、文件、namespace、类、函数等之间的调用关系。合理的调用关系应该是自上而下，层次鲜明的金子塔结构。常见的调用关系如下：

1. 组合/聚合/依赖关系之间的调用

组合/聚合/依赖的对象调用其包含的被组合/被聚合/被依赖的对象。以Fov和Target举例，Fov和Target在逻辑上形成明显的从属关系，Fov属于组合/聚合/依赖对象，Target被组合/被聚合/被依赖对象。

错误举例:

【互为成员对象】

class Fov

{

private:

Target\* m\_pTarget;

…

}

class Target

{

private:

Fov\* m\_pFov;

…

}

【Target依赖于Fov】

class Target

{

public:

void update(Fov\* fov);

…

}

【Target中有Fov的局部变量】

class Target

{

public:

void update ()

{

Fov fov;

…

}

}

如果是被组合/聚合对象需要将组合/聚合对象的参数传入，请不要将组合/聚合对象整个传入(无论实例、引用还是指针)，只传入所需的数据，这些数据必须基本内置类型、能够被更细分组合/聚合的自定义类型基本内置类型或者前两者能够组成的STL容器。

举例:

class Fov

{

public:

void updateSettingFilePath(string settingFilePath)

{

m\_settingFilePath = settingFilePath;

/\* 调用一个Target的成员函数updateSettingFilePath，

同时把settingFilePath传入，这是正确的做法。

如果直接将Fov传入，就发生了Target依赖于Fov的

关系，属于下级调上级，这是错误的做法如:

updateSettingFilePath(\*this);\*/

m\_pTarget.updateSettingFilePath(settingFilePath);

…

}

private:

Target\* m\_pTarget;

string\* m\_settingFilePath;

…

}

class Target

{

public:

void updateSettingFilePath(string settingFilePath)

{

m\_settingFilePath = settingFilePath;

…

}

private:

string\* m\_settingFilePath;

…

}

备注:

如果传递的参数过多，可以考虑使用类/结构体对参数进行打包，这样可以降低类之间的耦合度，前提用于打包的类/结构体要属于能够被组合/被聚合/被依赖的类型。

特例:

如果涉及到一些特殊的设计模式，如state等，可以存在相互引用的关系，前提是征得负责人的同意。

1. 平级调用

平级之间只能是单向调用，如果双向关系必须是通过observer、messageBus之类的机制来实现，双向调用耦合是禁止。

举例:

/\*Software和Hardware属于完全平行的两个项目，没有从属关系，

但是涉及到一些参数更新的时候，两个项目会产生一些依赖关

系，此时其调用只能是单向的，一般而言，都是Software从

Hardware更新参数，Hardware从Software更新参数是不被允许

的\*/

//**×** 双向调用

class Software

{

public:

void updateParm(Hardware& hw);

…

}

class Hardware

{

public:

void updateParm(Software& sw);

…

}

//√单向调用，调用正确

class Software

{

public:

void updateParm(Hardware& hw);

…

}

class Hardware

{

//禁止硬件直接从软件更新的参数

~~public:~~

~~void updateParm(Software sw);~~

…

}

对于简单的双向关系，可以在上面再封装一层，在上层进行交互。

Linker

Hardware

Software

举例:

//√ Linker层进行数据交互，调用正确

class Linker

{

public:

void updateSwParmFromHw(){…}

void updateHwParmFromHw(){…}

private：

Hardware\* m\_hw;

Software\* m\_sw;

}

上例中的Linker其实是一个简单的observer模式，处理简单的情况，之所以说是简单情况，因为还存在下面的复杂情况:

* 超过2个类之间的交互用MessageBus。
* 虽然现在是双向2个类，但将来可能会扩展超过2个类，用MessageBus。

这时候可以选择使用MessageBus。MessageBus的举例见《深入应用C++代码优化与工程级应用》中“第12章 使用C++11开发一个对象的消息总线，P284”。

特别强调：

调用关系涉及到调用关系的设计，如果拿不准如何处理请请示开发负责人。

## 单例模式 & 控制实例的数量

单例模式保证一个类仅有一个实例，并提供一个全局访问点。实现单例常见的有2种方法：

1. 模板类实现

使用类模板是单例可以扩展到任意类型，可变参数模板让单例模式更简洁。

举例:

template <typename T>

class Singleton

{

public:

template<typename … Args> //可变参数模板

static T\* **instance**(Args&&… args)

{

if( m\_pInstance == nullptr )

{

m\_pInstance = new T(std::forward<Args>(args)…);

}

return m\_pInstance;

}

//获取单例

static T\* **getInstance**( )

{

if(m\_pInstance == nullptr)

throw std::logic\_error(“the instance is not init”);

return m\_pInstance;

}

//释放单例

static void **destroyInstance**( )

{

delete m\_pInstance;

m\_pInstance = nullptr;

}

//注意:构造函数、析构函数和拷贝功能全部禁止

private:

singleton (void);

virtual ~Singleton(void);

Singleton (const Singleton&);

Singleton& operator= (const Singleton&);

static T\* m\_pInstance;

};

//调用代码

struct Airplane{…}；

struct Bus{…};

struct Car{void fun();};

string transport = “bus”;

//调用instance创建Airplane类型单例

Singleton<Airplane>::instance(transport);

Singleton<Bus>::instance(std::move(transport);

Singleton<Car>::instance(1,3.14);

//获取单例并调用Car单例中的fun方法

Singleton<Car>::getInstance()->fun();

//释放单例

Singleton<Airplane>::destoryInstance();

Singleton<Bus>::destoryInstance();

Singleton<Car>::destoryInstance();

该方法的优点是比较泛用，可以方便的实现现有类的单例，而不需要修改代码，但是缺点需要封装Singleton这个工具类，如果需要单例的类不使用Singleton这个类的话，还是能够创建多个实例，在需要保证绝对单例的场合使用具有风险。

1. 非模板类实现

通过非模板的类设计:对具体的类进行单例设计，具体的做法是定义一个全局变量，禁用类的构造函数、析构函数和复制功能。

举例:

class GameWorld

{

public：

bool int();

void run();

//使用静态函数返回静态实例

**static** GameWorld **& getInstance**( )

{

**static** GameWorld gameWorld；

return gameWorld;

}

//注意:构造函数、析构函数和拷贝功能全部禁止

**private**:

GameWorld();

GameWorld(const CGameWorld& rhs);

virtual ~ CGameWorld (void);

GameWorld (const CGameWorld &);

GameWorld & operator= (const CGameWorld &);

…

}

//调用代码

GameWorld :: **getInstance()**.init();

GameWorld :: **getInstance()**.run();

该方法的优点是能够至始至终保证类的单例，安全性好，缺点是需要对单例的类进行特别的设计，一旦设置为单例，就再也不能创建多个实例，灵活性不够。

1. 控制实例的数量

对单例模式稍加修改，就可以实现对单例数量的控制。

举例:

class GameWorld

{

public:

GameWorld ();

~ GameWorld ();

**private**:

static size\_t m\_instanceMaxCount;//最大实例数量

static size\_t m\_currentInstanceCount;//当前实例数量

};

//构造函数

GameWorld:: GameWorld ()

{

//设定对象个数为6，数字修改成1就是单例模式

if(**m\_currentInstanceCount> m\_instanceMaxCount**)

{

//异常处理

...

}

else

{

init();

++m\_currentInstanceCount;

}

}

//析构函数

GameWorld::~ GameWorld ()

{

release();//释放单个实例内存

--m\_singletonCount;

}

## [Single Responsibility](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk) Principle单一性原则

这是所有设计规范中最重要最基本的一项规范，也是最容易被人忽视的规范。单一性主要体现在如下几个方面：

1. 类

一个类只实现某一类功能，和其他类的耦合性越小越好。

错误举例:

//Rectangle中包含了求取面积方法和在界面上绘制的方法

class Rectangle

{

public:

double calcArea();//计算该Rectangle的面积

void draw();//在界面上绘制出该矩形

...

private:

double int m\_cx;

double int m\_cx;

double int m\_width;

double int m\_height;

}

在上例中，Rectangle违背了单一性，因为对于Rectangle来说，如何在界面上绘制不是它的职责，如果界面绘制逻辑发生修改，可能导致Rectangle进行修改。

正确的做法是把draw函数从Rectangle中剥离，放入一个新类中。

正确举例:

class Rectangle

{

public:

double calcArea();//计算该Rectangle的面积

~~void draw();~~

...

private:

double int m\_cx;

double int m\_cx;

double int m\_width;

double int m\_height;

}

VisualRectangle

{

public:

void draw();

...

private:

//组合一个Rectangle类型的嵌入对象，用来保存几何数据

Rectangle m\_geoRect;

}

1. 函数

尽量避免行数过长的方法，做到功能最小化，如果有长函数就考虑对方法进行拆分，这样既能降低耦合还能灵活组合使用。

错误举例:

//包括Inspect Pads和Export Log两端长的代码

void inspectFov(Fov fov)

{

//Inspect Pads，超过了40行代码

…

…

};

//长函数

void inspectPads ()

{

//Inspect Pads,功能和inspectFov中Inspect Pads部分完全一样

…

}

上面的函数破坏了单一性，因为inspectFov和inspectPads都属于长函数，有重复的代码，修改的时候一改就是两处。

正确的做法是应该把重复的部分提取出来。

正确举例:

//对于重复代码单独提取出来作为独立的函数

void inspectPadList(list<Pad> pads)

{

//Inspect Pads

…

}

//inspectFov调用inspectPads

void inspectFov(Fov fov)

{

//Inspect Pad List

inspectPadList(fov->padList)

};

1. 继承

如果不同class中有相同功能的代码，可以提炼到一个基类中。

错误举例:

/\*BWAlg和ColorAlg的inspect中都调用了initParm函数，

该函数为各自的成员函数，并且函数内容完全一样\*/

//BWAlg有initParm函数

class BWAlg

{

void inspect()

{

initParm();

execBWAlg();

}

void initParm()

{

…

}

}

class ColorAlg

{

void inspect ()

完全一样的函数

{

initParm ();

execColorAlg ();

}

void initParm()

{

…

}

}

两个算法中都有同样的initParm方法，把该方法移到基类中，减少冗余代码，后面再扩展新的算法只需要继承基类即可。

正确举例:

//把相同部分的代码提取到基本Alg中来

class Alg

{

public:

virtual void inspect () = 0;

void initParm(){…};

}

//派生类算法只需要继承即可

class BWAlg:public Alg

{

void inspect ()

{

initParm ();

createAlg1Model();

}

}

class ColorAlg:public Alg

{

void inspect ()

{

initParm ();

createAlg1Model();

}

}

## [Liskov Substitution](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en) Principle替换原则原则

派生类必须能够替换基类，能够在不做任何变化的情况下替代基类，做到基类的真正复用。

以Pad举例，Pad有各种形状，可以通过设计多个Pad类型实现。

错误举例:

class Rectangle;

class Circle;

class RectPad

{

....

private

Rectangle\* m\_rect;

};

class CirclePad

{

....

private

Circle\* m\_circle;

};

此设计不合理，因为RectPad和CirclePad只是形状不同，并没有本质的区别，如果Pad还有别的现状，又要重新设计Pad类。

为了复用Pad类，完全可以把具体的形状用一个Shape基类指针代替，设计Rectangle和Circle两个派生类，把Shape基类指针丢到Pad类中，需要Shape的地方完全使用派生类代替。

正确举例:

//设计Shape基类

class Shape

{

public:

enum ShapeType

{

RECTANGLE,

CIRCLE

}

};

//设计Rectangle，继承Shape

class Rectangle:public Shape

{

....

};

//设计Circle，继承Shape

class Circle:public Shape

{

...

};

class Pad

{

public:

Pad(ShapeType shapeType)

{

switch(shapeType)

{

case Rectangle:

case Circle:

default:

//异常处理

...

}

}

private

Shape\* m\_shape;

};

//创建形状为Rectangle，Circle的Pad

Pad rectPad = new Pad(Shape::ShapeType::Rectangle);

Pad circlePad = new Pad(Shape::ShapeType::Circle);

## [Dependency Inversion](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en) Principle依赖倒置原则

依赖倒置就是基于虚函数重写机制分离接口和实现，具体做法如下：

* 在依赖之间定义一个抽象接口，高层模块调用接口，
* 低层模块实现接口的定义，从而有效控制耦合关系，达到依赖于抽象的设计。

正确案例:

//Exporter底层抽象类

class Exporter

{

public：

virtual string readFromFilePath(string filePath) = 0;

…

}

//CSVExporter继承与Exporter

class CSVExporter: public Exporter

{

public：

virtual string readFromFilePath(string filePath) override;

…

}

//XMLExporter继承与Exporter

class XMLExporter：public Exporter

{

public：

virtual string readFromFilePath(string filePath) override;

…

}

//ExportController作为高层模块调用Exporter

class ExportController

{

public：

void setExpoter(Exporter\* exporter);

void runExport();

…

private:

Exporter\* m\_pExporter;

string m\_filePath;

}

//ExportController的runExport实现

void ExportController::runExport()

{

string exportedString = m\_pExporter->

readFromFilePath(m\_filePath);

…

}

//测试代码

ExportController control;

control.setExpoter(new CSVExporter(“wang.csv”));

control.runExport();

上面的例子中，ExportController作为高层模块，而Exporter的派生类都是低层模块，ExportController调用Exporter，ExportController不需要知道CSVExporter、XMLExporter或者其它派生类的其它派生类的细节，只要调用Exporter的接口即可。

## [Open Closed](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en) Principle开闭原则

开闭原则就是在模块设计的时候应该追求在原有代码不被修改的前提下被扩展，封装变化、降低耦合，尽量考虑接口封装机制、抽象机制和多态机制。

错误举例:

//设计一个读取Doc文件并转换成XML或者Binary的方法。

//XML转换器

class XMLConverter

{

public:

string convertDocToXML(Docment doc);

};

//Binary转换器

class BinaryConverter

{

public:

string convertDocToBinary(Docment doc);

}

//Converter的类型枚举

enum ConverterType

{

XML，

Binary

}

//Exporter，高层模块真正调用的是exportDoc方法

class DocumentExporter

{

public:

void setConvertType(ConvertType convertType);

void exportDoc(Doc doc);

private:

ConverterType m\_converterType;

}

void DocumentExporter::exportDoc(Document doc)

{

…

switch(m\_converterType)

{

case XML:

XMLConvert xmlconverter;

string xmlContent = xmlconverter.

convertDocToXML(doc);

…

case Binary:

BinaryConvert binConverter;

string binContent = binConverter. convertDocToBinary

(doc);

…

…

}

…

}

上面的代码并没有遵守开闭原则，每一次需要增加新的Converter时，DocumentExporter都必须发生修改，Enum和Switch都要进行修改，同时每一次引用他们的地方也需要接着修改。

对上面的设计，稍加修改，就可以使其满足开闭原则。

正确举例:

//设计一个读取Doc文件并转换成XML或者Binary的方法。

//设计Converter的抽象基类

class Converter

{

public:

virtual string convertDocToString(Docment doc) = 0;

};

//XML转换器

class XMLConverter: public Converter

{

public:

virtual string convertDocToString (Docment doc)override;

};

//Binary转换器

class BinaryConverter: public Converter

{

public:

virtual string convertDocToString (Docment doc)override;

};

//Exporter，高层模块真正调用的是exportDoc方法

class DocumentExporter

{

public:

void setConvert(Convert\* converter);

void exportDoc(Doc doc);

private:

Converter\* m\_converter;

}

void DocumentExporter:: exportDoc (Document doc)

{

…

string fileContent = m\_converter. convertDocToString (doc);

…

}

修改以后每次增加新的Converter时，只需要继承Converter，DocumentExporter不需要任何修改。

注意:

* 一个类不可能做到完全封闭，总是会有不可预见的修改。但是，当修改时可预见的(如下述的增加一个Converter)，需要牢牢记住遵守开闭原则。
* 万不得以需要修改时，需要特别当心，因为原来代码的修改可能导致低版本的不兼容。
* 虽然有不得不修改原有代码的情况，但是在第一次架构时还是要最大程度的追求开闭原则，代码中不应该有明显不需要的冗余，否则在后续修改时对冗余代码的判断需要花费额外更多的精力，同时删除冗余代码也违背了开闭原则。

## Interface Segregation Principle接口分离

在介绍接口分离原则之间，先重点讲下了“接口”这个概念，接口有2层意思：

对应了调用形式，提供哪些函数，函数的返回类型，函数名，形参列表等。

C#，Java等高级语言中的“接口”代表一种数据结构，因为C++并不是完全彻底的面向对象语言，在语法上没有像C#专门的interface内置关键字。

大部分情况下我们说的“接口”都是属于第一种情况(如团队工作中讨论“接口”)，当想表达第二种意思时，更倾向于描述为“抽象基类”或者“抽象接口类”这种不能被实例化的基类。这里在讨论接口分离原则中，所有的“接口”都是代表这样的意思。

设计软件通常会将一个模块分解成包含若干个子模块。当通过一个接口来扩展软件时，如果这个接口包含了过多的行为，则说明这个类的接口定义不够内聚，这种接口叫做“胖接口”或者“受污染接口”，因为它降低了系统的可扩展性。接口分离原则就是要把“胖接口”拆分成若干个最小功能的“子接口”，下面举一个“胖接口”的错误例子：

错误举例:

/\*定义一个动物接口可以有2种行为，run和swim，以期

它能够适用于更多的动物（认为人也是一种动物）\*/

//定义两种行为的接口

class IAnimal

{

public：

virtual void run () = 0;

virtual void swim() = 0;

}

//Person可以run和swim

class Person: public IAnimal

{

public:

virtual void run() override{" Person can run!";};

virtual void swim() override{" Person can swim!";};

}

//Fish只有swim行为

class Fish:public IAnimal

{

public:

virtual void run() override{"Fish can not run!";};

virtual void swim() override{"Fish can swim!";};

};

//Dog只有run行为

class Dog:public IAnimal

{

public:

virtual void run() override{" Dog can run!";};

virtual void swim() override{"Dog can not swim!";};

};

上面代码的问题就是内聚度不好，虽然Person兼容了IAnimal的所有行为，但是对于后续增加的Fish和Dog，如果想要再使用IAnimal接口，难以避免写一些Dummy方法了，这给整个软件增加了无效代码。随着IAnimal拥有的行为及派生类越来越多，项目会变得越来越不可维护。

根据接口分离原则，单独设计3个run，swim方法，做到接口的功能最小，聚合度最高。

正确举例:

//分别定义两种不同行为的两个接口

class IRun

{

public:

virtual void run()=0;

};

class ISwim

{

public:

virtual void swim()=0;

};

//Person可以run和swim

class Person: public IRun，public ISwim

{

public:

virtual void run()override{" Person can run!";};

virtual void swim()override{" Person can swim!";};

}

//Dog可以run

class Dog:public IRun

{

public:

virtual void run()override{"Dog can run!";};

};

//Dog可以swim

class Fish:public ISwim

{

public:

virtual void swim()override{"Fish can swim!";};

};

# 异常

在项目中使用异常需要注意2部分，一部分是抛出异常，另一部分是捕获异常，在底层或者中间层(非UI层)进行异常抛出或者异常捕获后再抛出，在UI层捕获处理异常。

在SSDK项目中的Exception目录中，有CustomException、IDisposal、和HandlerInfo三个类，以及一些辅助的宏定义。

* CustomException：继承自系统的异常exception，用于层层传递异常，记录了一些调用堆栈，显示信息及idispose指针
* IDisposal ：可以将中间层的类指针传递到上层进行处理，前提是对象类需要继承idisposal接口。
* HandlerInfo：用于记录和传递异常处理的Handler(以函数指针的形式)。
* 宏定义：用于方便异常代码的书写，保持单一性。

为了配合SSDK中的异常，App项目中与异常相关的有以下部分组成：

* ICanShowExMsgDlg：所有需要弹出异常对话框的都需要继承该接口，该接口主要包含一些异常到qml界面的交互逻辑，需要弹出的异常的对话框的C++文件必须继承该类。
* QCustomException：封装了底层的CustomException对象，可以直接加载到qml中使用
* ExceptionMessageBox.qml：异常对话框的控件，如果换了项目，APP发生了修改，那么这个控件也需要进行相应的修改

抛出异常主要有4种场景，具体如下：

## 只需要传递错误堆栈信息

只需要传递错误堆栈信息，显示哪里出现了异常，方便追踪 问题，如：

举例:

* 在中间层/底层

void TxtHelper::ReadFromFile(std::string filePath)

{

...

/\*发生异常的代码，在最近的地方抛出异常使用THROW\_EXCEPTION\*/

THROW\_EXCEPTION("Error occur when read from file",this);

}

* 在UI层

void readTxt()

{

try

{

//发生异常的代码

m\_texHelper.ReadFromFile("");

}

catch(CustomException& ex)

{

/\*显示调用堆栈信息，如果不想要显示，设置为false即 可\*/

ex.setStackTraceShown(true);

//仅仅是显示异常对话框用createAndShowExMsgDlg

this->createAndShowExMsgDlg(

this->m\_pEngine,

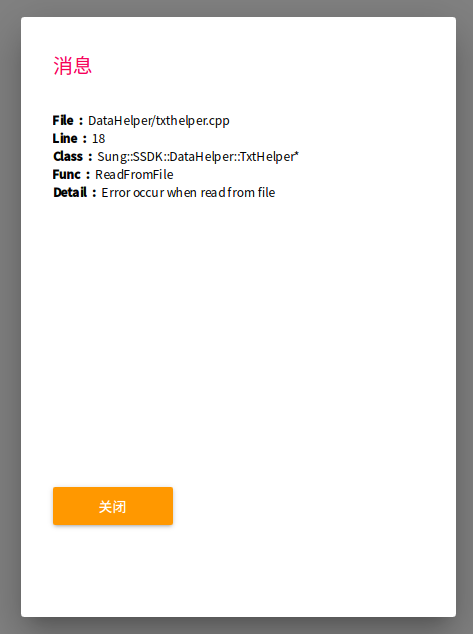
this->objectName(),

QString("rectMain"),ex);

}

}

最终效果如下:



## 在catch{}中throw异常

不确定哪里出现了问题，所以无法确定哪里应该抛异常，此时可以直接在catch部分抛出异常，如

举例:

在底层/中间层

void JsonHelper::readFromFile(const string &filePath)

{

try

{

...//发生异常的代码

}

CATCH\_THROW\_EXCEPTION(string("Read Json from file[")+filePath+string("] error!"),this);

}

使用CATCH\_THROW\_EXCEPTION相对于第一种情况只需要在底层作修改，其它部分不变，所以不再阐述。

## UI层捕获异常并定义Handlers

出现异常，并且需要进行处理，此时需要在UI层定义好Handler，供客户选择，异常界面会根据定义的Handler的种类自动生成按钮。

举例:

* 在UI层

void inspect()

{

try

{

this->m\_fov.inspect();//发生异常的代码

}

catch(CustomException& ex)

{

//需要处理的函数，这里是reponse和ignore两种

auto responseFunc = []{cout<<"response..."<<endl;};

auto ignoreFunc = []{ cout<<"ignore ..."<<endl;};

/\*中间的字符串是待翻译的字符串，是实际在界面商显示的字符串\*/

HandlerInfo responseHandlerInfo(

HandlerType::AI\_Response,tr("Response..."),

responseFunc);

HandlerInfoignoreHandlerInfo(HandlerType::Ignore,tr("Ignore..."),ignoreFunc);

ExceptionHandlers handlers;

handlers.emplace(responseHandlerInfo.handlerType(),responseHandlerInfo);

handlers.emplace(ignoreHandlerInfo.handlerType(),ignoreHandlerInfo);

//需要在异常界面中显示的文本

ex.setDisplayMsg(tr("Fov Inspect Error!"));

//带Handler的调用该函数

createAndShowExMsgDlgWithHandlers(

this->m\_pEngine,

this->objectName(),

QString("rectMain"),ex,handlers);

}

}

最终效果如下:



## 中间层/底层抛出异常的同时定义Handler

如果Handler比较关键，考虑到异常一般在中间层/底层抛出，UI层仅仅负责捕获，同时UI层的程序员并不了解异常到底应该如何处理，所以安全起见最好是把Handler只能在异常抛出的地方直接写掉。

举例:

* 在底层/中间层

void Vision::snap()

{

//定义处理的Handler

auto responseFunc = []{ cout<<"response..."<<endl; };

auto ignoreFunc = []{ cout<<"ignore ..."<<endl; };

auto retryFunc = []{ cout<<"retry ..."<<endl; };

auto abortFunc = []{ cout<<"abort ..."<<endl; };

//将处理的Handler封装到HandlerInfo中

HandlerInfo responseHandlerInfo(HandlerType::AI\_Response, string("Response..."),responseFunc);

HandlerInfo ignoreHandlerInfo(HandlerType::Ignore, string("Ignore..."),ignoreFunc);

HandlerInfo retryHandlerInfo(HandlerType::Retry, string("Retry..."),retryFunc);

HandlerInfo abortHandlerInfo(HandlerType::Abort, string("Abort..."),abortFunc);

ExceptionHandlers handlers;

handlers.emplace( responseHandlerInfo.handlerType(),responseHandlerInfo);

handlers.emplace( ignoreHandlerInfo.handlerType(),ignoreHandlerInfo);

handlers.emplace( retryHandlerInfo.handlerType(),retryHandlerInfo);

handlers.emplace( abortHandlerInfo.handlerType(),abortHandlerInfo);

//中间层，有带有Handlers的代码使 THROW\_EXCEPTION\_WITH\_HANDLERS(

"snap error!",

This,

handlers);

}

* 在底层/中间层

void snap()

{

ry

{

this->m\_vision.snap();

}

catch(CustomException& ex)

{

//需要在异常界面中显示的文本

ex.setDisplayMsg(tr("Snap Error!"));

//不需要显示调用堆栈信息，所以设置为false

ex.setStackTraceShown(false);

//设置在异常界面上显示的翻译文本

ex.setHandlerDisplayMsg(HandlerType::Ignore,tr("Ignore and go on!"));

ex.setHandlerDisplayMsg(HandlerType::AI\_Response,tr("Smart Response, then click close and retry!"));

ex.setHandlerDisplayMsg(HandlerType::Retry,tr("Retry snap Fov!"));

ex.setHandlerDisplayMsg(HandlerType::Abort,tr("Abort snap Fov,then click close!"));

this->createAndShowExMsgDlg(

this->m\_pEngine,

this->objectName(),

QString("rectMain"),ex);

}

}

最终效果如下:



## 注意事项

1. 需要弹出异常界面的c++文件，需要继承下ICanShowExMsgDlg

举例:

class MainUI:public QObject,public ICanShowExMsgDlg

1. 在上层的QCustomException定义了一个map类型的成员m\_handlerTranslatorMap，主要是用于对应翻译(翻译无法做到跨项目)，这里是一个不够自适应的地方，一旦底层的HandlerType发生修改，那么m\_handlerTranslatorMap也需要修改。

## 

# 禁止/慎用事项

## 禁止goto

goto不仅使代码流程变的错综复杂，还破坏程序的结构性，影响程序的可读性，所以禁止使用goto。

错误举例:

int main ()

{

int a = 5;

if ( 5== a )

**goto exit**; //禁止使用goto

…

**exit**: //影响阅读性，大段代码时不易找到exit

return 0;

}

## 慎用友元

友元削弱类的封装性，使private不再私有，使用友元之前务必获得主管批准。

举例1:

//Wife可以访问Husband的私有变量m\_salary,适合用友元

class Wife {…};

class Husband

{

private:

int m\_salary;

**friend** class Wife;

};

举例2:

/\*对于双目的运算符，重载为友元函数后能够接受左参数和

右参数的隐式转换，这样使用起来更方便一些\*/

class Complex

{

public:

Complex( int real = 0, int imag = 0 ): m\_real( real ),

m\_imag( imag ){}

...

friend Complex operator + ( const Complex& c1,

const Complex& c2);

friend Complex operator - ( const Complex& c1,

const Complex& c2);

...

private:

int m\_real;

int m\_imag;

};

Complex operator + (const Complex &c1, const Complex &c2)

{

return Complex ( c1.m\_real+ c2.m\_real,

c1.m\_imag+ c2.m\_imag);

}

Complex operator - (const Complex &c1, const Complex &c2)

{

return Complex ( c1.m\_real- c2.m\_real,

c1.m\_imag- c2.m\_imag);

}

## 禁止返回局部变量的引用

返回局部变量的引用时，局部变量作为函数返回值抛出后被销毁，但是引用作为其别名还存在，这造成内存泄露。

错误举例:

class Box

{

public:

【通过引用创建对象的方式】

/\*局部变量result一旦出了作用域，就自动销毁，

result只是个别名，已经没有意义\*/

**Box&** operator+ (Box& len)

{

**Box result (this->m\_length + len.m\_length);**

return **result**;

}

【通过new创建对象的方式】

//调用完函数之后，需要手动删除分配的内存

**Box&** operator-(Box& len)

{

Box \*result = **new Box(this->m\_length – len.m\_length);**

return **\*result**;

}

…

private:

Box m\_length;

};

注意:

除了引用，最好也不要返回局部变量的指针，原因有下：

* 增加了用户删除内存的负担。
* 内存的申请和删除不在同一模块，破坏了函数的完整性、单一性和内聚性。

特例:

在某些特定功能的操作，可以通过new动态创建的对象，然后返回其指针，但是需要经过主管同意。同时为了减少删除内存的负担，可以使用智能指针。

## 禁止编译警告

除非有特殊情况得到直属主管批准，在代码调试过程中，出现的警告信息，必须解决，哪怕一条警告。如果有些警告无法消除就要主动向直属主管汇报。

举例:

#include <iostream>

int main()

{

**int a {3}; //unused variable 'a'**

int b {5};

std::cout<<b;

return 0;

}

强烈建议：

* 采用IDE默认的配置。
* 不要放过任何一条警告信息。

## 禁止头文件using

命名空间是C++提供的一种机制，可以有效的避免函数名污染，但是禁止在头文件中使用using namespace XXX，而应该在定义时直接用全称。

举例:

//修改前版本

#include <vector>

**using namespace std; //不推荐**

class Data

{

…

private:

vector <int> m\_vecData;

};

//修改后版本

#include <vector>

class Data

{

…

private:

std::vector <int> m\_vecData;//推荐

};

## 禁止使用magic number

Magic Number，译为魔数，指没有明显的原因而出现在代码中的具体数字，魔数使代码阅读性变差，并且不易维护，所以禁止使用魔数，可以使用常量代替。

比较常见的错误就是类似于box1，box2...或者boxA,boxB，这种命名应该被坚决杜绝，除非类似循环中使用的i，j这种局部变量除外。

错误举例:

//魔数1000，看不出它的含义

for( int i = 0; i < **1000**; ++i )

{

…

}

正确举例:

const int **peopleCount** = 1000;

for( int i = 0; i < **peopleCount**; ++i )

{

…;

}

## 禁止将函数的输入参数作为工作变量

工作变量，顾名思义，就是函数内部使用的变量，将传入参数作为工作变量进行操作，可能会带来未知的隐患：函数参数在传入函数时，编译器都会为其生成一个临时拷贝，但是对这个拷贝的修改有时也会改变原始参数内容，所以禁止将函数参数作为工作变量。

错误举例:

void dataSum( uint num, int\* data, int \*sum )

{

**\*sum** = 0;

for( uint count = 0; count < num; ++count )

{

\*sum += data[count];

}

…

}

正确举例:

/\*输出参数,用局部变量先代替，最后再进行赋值\*/

void dataSum( uint num, int\* data, int \*sum )

{

int tempSum{0};

for( uint count = 0; count < sum; ++count)

{

tempSum += data[count];

}

\*sum = tempSum;

}

# 鼓励事项

## 鼓励优先使用静态断言调试

鼓励使用静态断言调试，静态断言可以在编译期间检测用法错误，release模式下无效，C++断言有assert、static\_assert、type\_traits,assert断言可以检查函数参数是否有效，static\_assert和type\_traits针对元编程(这一章节最后备注会有简单的介绍)特别有效。

我们鼓励使用static\_assert静态断言。

举例:

template<class T>

T average( const std::vector<T>& data )

{

**static\_assert**( const std::is\_arithmetic<T>::value ,

“Type parameter for average( ) must be arithmetic”);

T sum { 0 };

for( auto& value : data )

sum += value;

return sum/data.size( );

}

## 

备注:

这里简单介绍下元编程，元编程程序的执行完全是在编译期，通过将计算从运行期转移至编译期，在结果程序启动之前做尽可能多的工作，最终获得速度更快的程序。也就是说模板元编程的优势在于:

* 以编译耗时为代价换来卓越的运行期性能。
* 提供编译期类型检查。

## 鼓励优先使用标准库

STL背后蕴含的是一种新的程序设计思想-泛型化设计，被证明了是具有工业级别的强度的佳作，无愧于“标准”二字。我们应优先使用标准库中的各种现有的数据结构、算法开发我们自己的程序。其中需要注意几点：

* 由于STL涵盖的范围相当大，所以发生命名冲突的可能性也大。STL委员会设计了将某些头文件的“.h”去掉来避免冲突。所以要include正确的头文件。

举例:

//标准化以前的头文件

#include<iostream**.h**>

//标准化之后的标准头文件

#include<iostream>

* 鼓励使用标准库算法，效率更高、不易出错、可维护性更好。

不佳示例:

class Person

{

public:

void work();

…

};

vector< Person\* > pPersons{ new Person() };

**for**( iter = persons.begin() ; iter != persons.end() ; ++iter)

{

(\*iter) -> work();

}

较好举例:

**for\_each**(

persons.begin(),

persons.end(),

mem\_fun\_ref(&Person::work)

)；

* 优先使用STL，在STL无法实现的前提下，第三方通用库中(不包括OpenCV之类专业库)只能使用boost()。

## 鼓励优先使用前缀操作符

在逻辑上允许的情况下，请使用前缀操作符，原因是由于前缀操作符省去了临时对象的构造，因此它在效率上优于后缀操作，例如：++i、--i。

举例:

for( uint i = 0; i < count; **++i**)

{

…

}

## 鼓励函数内局部变量离第一次使用越近越好

局部变量的构造和析构是不可避免的开销，所以建议在需要使用局部变量时再去定义，即局部变量定义离第一次使用越近越好。

不佳举例:

string getname( const std::string& str )

{

//此处变量过早定义,如果str为空,name变量将不被使用

string **name**;

…

**name** = “temp variant is first used!”;

…

return name;

}

较好举例:

std::string getName(const std::string& str)

{

…

string **name**= “temp variant is first used!”;

…

return name;

}

特例:

虽然原则上定义变量能晚则晚，还要结合具体的应用场景及效率要求，对于效率有要求的场合，如果变量的提前声明和定义能够带来效率上的提升，则这条鼓励事项可以不用遵守。

//不佳示例，每次循环都创建了一个实例，没必要也影响效率

int count = 1000;

for( uint i = 0; i < count ; ++ i )

{

FileHelper helper; //对象被创建1000次

helper.writeToFile(i,“C:\\temp.txt”);

}

//较好示例，只要创建一次实例就可以实现完全一样的功能

FileHelper helper;

for( uint i = 0; i < 1000; ++ i )

{

helper.writeToFile(i,“C:\\temp.txt”);

}

## 鼓励使用列表初始化

类成员初始化时，鼓励优先考虑初始化列表，并且初始化顺序与类成员变量声明顺序一致，这样能提高执行效率。

举例:

class Box

{

public:

Box( int length, int height, int width) :

m\_length( length ),

m\_height( height ),

m\_width( width )

…

private:

int m\_length{0};

int m\_height{0};

int m\_width{0};

}

## 鼓励优化循环

在C++的循环语句中，for语句是使用频率最高的，在多重循环中，应该讲最长的循环放在最内层，最短的循环放在最外层，提高速率。

举例:

int rowCnt = 100;

int colCnt = 1000;

for( int i = 0; i < rowCnt ; ++i)

{

//循环次数多的放在内循环

for( int j = 0; j < colCnt ; ++j )

{

…

}

…

}

在逻辑分支中，在switch...case.../if....else if...else...语句中，按照他们发生的相对频率来排序，把最可能发生的情况放在第一位，最不可能的放在最后。

举例:

int daysPermonth{0};

…

switch( daysPermonth )

{

case 31: //每月31天的频率最高

…;

break;

case 30: //其次每月30天频率较高

…;

break;

case 28:

…;

break;

case 29:

…;

}

## 鼓励优先使用STL循环算法代替手写循环

使用STL循环算法代替手写循环，相比于手写循环，STL算法通常效率更高，不易出错，可维护性更好，并且STL算法非常丰富，涉及面也较为广泛，所以很多本需要我们用手写循环来实现的任务可以通过STL算法调用来实现。所以鼓励采用STL算法，多用算法调用，少用手写循环。

举例:

class Person

{

public:

void work();

…

};

vector< Person\* > pPersons;

...

//手写循环版

auto iter = pPersons.begin();

while( persons.end() != iter )

{

(\*iter) -> work();

iter++;

}

//算法调用版

for\_each(container.begin(),

container.end(),

mem\_fun\_ref(&container::doSomething));

虽然算法相较于手写循环有一定的优势，但手写循环在一些情形下还是值得采用的，特别是一些简单的循环，如:

举例:

vector<int> items;

int itemCnt = 10;

for( size\_t i = 0; i < itemCnt ; i++)

{

items.push\_back(i\*i);

}

## 鼓励使用表驱动法

当if和switch语句过多时，使用表驱动能够使代码简化，提高可阅读性，并且提高性能。

不佳示例:

int getMonthDays( int month)

{

int days;

//使用switch语句也是比较繁多

if( 1 == month ) { days = 31;}

else if( 2 == month ) {days = 28;}

else if( 3 == month ) {days = 31;}

else if( 4 == month ) {days = 30;}

else if( 5 == month ) {days = 31;}

else if( 6 == month ) {days = 30;}

else if( 7 == month ) {days = 31;}

else if( 8 == month ) {days = 31;}

else if( 9 == month ) {days = 30;}

else if( 10 == month ) {days = 31;}

else if( 11 == month ) {days = 30;}

else if( 12 == month ) {days = 31;}

return days;

}

较好示例:

//使用表驱动法对上面的例子进行优化

static int monthDays[12] =

{31,28,31,30,31,30,31,31,30,31,30,31};

int getMonthDays( int month)

{

return monthDays[(month - 1)];

}

表驱动法在第三方异常信息抛出时非常有用。如(C#代码)

举例:

/\*GTS所有错误信息公用String数组，在用表驱动法显示具体

错误信息时，所有指令返回值需+7后再来对应相应的字符串\*/

static string[] m\_errorMessages =

{

//相当于-7

"Without Response from Motion Card ! \n ",

//相当于-6

"Motion Card Open is Failure! \n ",

/\*相当于-5到-2，GTS卡这些返回值无意义\*/

"NA \n 无意义信息",

"NA \n 无意义信息",

"NA \n 无意义信息",

"NA \n 无意义信息",

//相当于-1

"Communication between PC and Motion Card is Failure \n ",

//相当于0

"Instruction executed successfully! \n ",

//相当于1

"Instruction executed wrong! \n 指令执行失败!",

//相当于2

"License is not available! \n ",

/\*相当于3到6，GTS卡这些返回值无意义\*/

"N/A \n 无意义信息",

"N/A \n 无意义信息",

"N/A \n 无意义信息",

"N/A \n 无意义信息",

//相当于7

"Instruction Parameter Error！\n "};

}

//如何调用

protected void GtsMotorOff(string ownerID)

{

lock (\_staticLock)

{

short sRtn = 0;

if(0 !=(sRtn=GT\_AxisOff(\_motor\_AxisNum)))

throw new Exception((String.Format("Motor: {0};

Axis No.:{1};

Owner:{2};\n Diabling When Disable

Motor: {3}",

\_motor\_Name,

\_motor\_AxisNum, ownerID,

**\_errorMessages[sRtn + 7]))**;

}

}

# 代码版面

## 注释

1. 类注释

一般一个文件只有一个类，请在最初创建类时，在类前面加上注释，如作者、日期、版本、类和note的相关描述等。当在类里面增删函数、成员变量时，在note里面追加一条新的注释记录，注释解释说明具体修改内容；且每次发生这种重大事项时都要进一步追加。

|  |  |
| --- | --- |
| /\*\* | |
| \* @brief | 类的简要说明 |
| \* @tparam | 如果是类模板，带上模板参数的说明  (注意使用tparam，普通的参数使用param表示，模板参数加上前缀t) |
| 必要的话，可以对类的作用、背景、注意事项等进一步展开说明 | |
|  | |
| \* @ author | 最初的作者  后续版本信息，包括版本号，修改时间和修改人 |
| \* @ version | 版本号 日期 作者 note： |
| \*/ | |

举例:

|  |  |
| --- | --- |
| /\*\* | |
| \* @ brief  \* | n-dimensional dense array class  n can only be 1,2,3 |
| \* @ tparam  \* | T: Type of element  T can only be unsigned char, uint, float and double |
| \* |  |
| \* @ author | wangfei |
| \* @ **version** | 1.00 Oct,18th,2016 wf note: create it |
| \* | 1.01 Oct,19th,2016 smm note: delete test function |
| \*/ | |
| template<typename T>  class Mat  {  …  } ; | |

在类注释中，需要特别注意别忘了写version，这是类注释和函数注释的最大区别。

1. 函数注释

函数注释写在函数声明/定义的前面，注释包含功能、输入/输出参数、返回值等，如果函数内部被修改，务必在函数注释里面添加修改信息。

|  |  |
| --- | --- |
| /\*\* | |
| \*@brief | 函数的简要说明 |
| \*@**t**param | 如果是函数模板，带上模板参数的说明  (注意使用**t**param，普通的参数使用param表示，模板参数加上前缀t) |
| \*@param | 参数名称：参数说明  (特别注意：对于明显的很容易就看出含义的参数可以不注释，通常这类参数要么命名比较规范使人一眼便知，要么属于常用术语，无需展开解释，新人如果拿不定注意一定要请示主管) |
| \*@return | 返回值说明 |
|  | |
| 必要的话，可以对函数的作用、背景、注意事项等进一步展开说明\*/ | |

举例:

|  |  |
| --- | --- |
| /\*\* | |
| \*@brief | read roi data from source data |
| \*@tparam | T: type of data |
| \*@param | srcData: source data, a 1d array |
| \*@param | srcWidth: width of source data |
| \*@param | srcHeight: height of source data |
| \*@param  \* | xStartROI: left position of ROI, must be in the range of [0, srcWidth - widthROI] |
| \*@param  \* | yStartROI: top position of ROI, must be in the range of [0, srcHeight- heigthROI] |
| \*@param  \* | widthROI: with of ROI, must be in the range of [1, srcWidth] |
| \*@param  \* | heigthROI: heigth of ROI, must be in the range of [1, srcHeight] |
| \*@return | roi data |
| \*/ | |
| template<typename T>  T\* readDataFromROI(  T\* srcData,  int srcWith,  int srcHeight,  int xStartROI,  int yStartROI.  int widthROI,  int heightROI); | |

1. 区块注释

标准的C++没有类似与C#的区块(Region)注释的功能，无法做到可以折叠/展开一段注释的代码，这对于查看、理解代码带来不便。为了增加代码的可读性，使用如下的方法进行(一段代码，包括类中的多个功能相近的成员变量/函数，函数内连续的几行代码)注释。

|  |
| --- |
| //>>>-------------------------------------------------------------------------------  (这里大约有80个“-”,如果代码有缩进，去掉缩进的符位数，同一层级的“-”数量要保证一致) |
| //说明或/\*说明\*/  具体的代码  //<<<------------------------------------------------------------------------------ |
|  |

举例:

【属于同一功能类成员】

|  |
| --- |
| //>>>-------------------------------------------------------------------------------  // related to exception message  //orginal exception message  string m\_srcExMsg{“”};  //full stack trace message,it contains detail call information  string m\_stackTraceMsg{“”};  //<<<------------------------------------------------------------------------------- |
| 【在长函数内部对于同一功能的代码的注释】  void inspectFov()  {  //>>>-------------------------------------------------------------------------  /\*1.Analyze element，using the traditional 2d methods to image analysis\*/  ...  //<<<-------------------------------------------------------------------------  //>>>-------------------------------------------------------------------------  /\*2.calcuate phase, is the necessary step to 3d calculation\*/  ...  //<<<-------------------------------------------------------------------------  //>>>-------------------------------------------------------------------------  /\*3.inspect normal target , get their area、volume and height\*/  ...  //<<<-------------------------------------------------------------------------  //>>>-------------------------------------------------------------------------  //4.export image and result  ...  //<<<-------------------------------------------------------------------------  } |

在函数中使用区块注释一般用于复杂函数，这些函数一般行数较多，逻辑比较复杂。对于这些复杂函数一般首选将函数拆分优化，有些函数（在上层代码比较常见）实在没办法后请示主管，征得同意后再进行区块注释。

1. 单行注释

函数内部或者类内部需要解释(比较简短)的地方，使用 //。

举例:

int m\_imgMaxWidth//max width of image

1. 多行注释

* 注释待删除的代码

待删除的代码需要加PDCAAP：please delete completely after a period标志。

|  |  |
| --- | --- |
| //>>>-------------------------------------------------------------------------------  //修订者+修订日期+PDCAAP+必要的说明 | |
| //... | 被删除的代码 |
| //<<<------------------------------------------------------------------------------- | |

举例:

|  |
| --- |
| void inspect()  {  //>>>-------------------------------------------------------  /\*wf161018 PDCAAP because inspectTargets already  contains following 4 steps, so following is no need, can  delete after rime checked!\*/  //inspectFiducialMarks();  //inspectPads();  //inspectBadMarks();  //inspectTrackTraces();  //<<<-------------------------------------------------------  inspectTargets();  } |

* 修改了多行代码后的注释

|  |  |
| --- | --- |
| //>>>-------------------------------------------------------------------------------  //修订者+修订日期+必要的说明 | |
| ... | 被修改的代码 |
| //<<<------------------------------------------------------------------------------- | |

举例:

unsigned char\* getGray8DataFromBgr24Data(

unsigned char\* bgr24Data,

int imgWidth,

int imgHeight)

{

unsigned char\* grayData = new unsigned char

[ imgWidth \* imgHeight];

//>>>----------------------------------------------------------------------

/\*zl170220 gray ratio of RGB is changed from a fixed

value (1/3) to const variable\*/

double const rRatio = 1/3 ;

double const bRatio = 1/3 ;

double const gRatio = 1/3 ;

for( int i = 0; i< imgWidth \* imgHeight; ++i )

{

unsigned char gray =

rRatio \* bgr24Data[3 \* i] +

bRatio \* bgr24Data[3 \* i + 1] +

gRatio \* bgr24Data[3 \* i + 2] ;

grayData[ i ] = gray ;

}

//<<<----------------------------------------------------------------------

inspectTargets();

}

* 单行代码被增/减时，需要在这行上面加上修订者+修订日期+必要的说明。

举例:

void init()

{

/\*zl160910 add initSystemSetting function system setting

must be inited first\*/

initSystemSetting();

initRecommendedParm();

...

}

1. 右括号后面的注释

原则上一个“{}”对不应该超过一屏(在默认显示比例的情况下,一般为50行)，如果超过一屏的，优先进行拆分，如果拆分不了的话，在右大括号后面加注释。

举例:

int maxCnt = 1000;

if(curCnt < maxCnt)

{

//以下省略大量代码

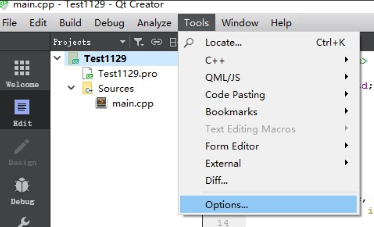
...

**}**//if (curCnt < maxCnt)

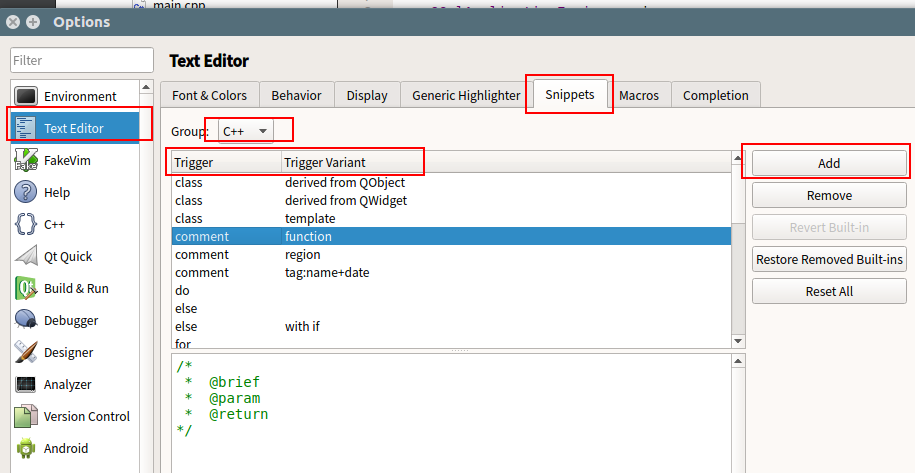
1. 快捷注释

Qt Creator自带快捷注释功能，具体操作步骤如下:

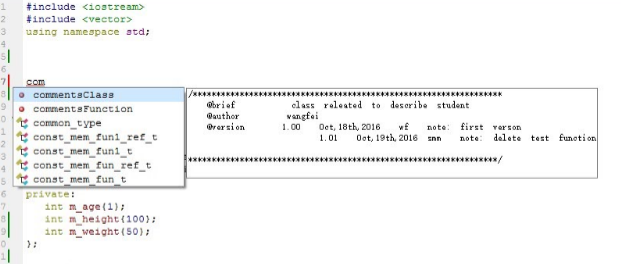
* 在Tools选择Options



* 在Text editor中选择Snippets,Group选择C++，点击Add，在Trigger设定触发的文本，Trigger Variant设定为触发后的提示文本。



* 设定好注释的内容后，只需要在代码编辑界面输入Trigger设定文本，一般在完成输入第3或4个字母后(所以需要保证手动配置的Snippets的Trigger单词数量大于3个)，Qt会弹出相应的提示项，该提示的文本就是前面设定的Trigger Variant，如果确认输入只需要选择对应项按下回车，如果想要忽略则放弃选择，忽略选项继续输入代码即可。



根据前面章节的注释规范，Add指定的Snippets，具体见《Snippets模版》（格式后续可能会发生更新，所以以最新的《Snippets模版》为主）。

## 代码格式

1. 避免一行代码过长

当一行代码过长超出一屏时，就会严重影响阅读，所以一行代码尽量确保能在一屏内读完，按照一般文本编辑器的显示设置，80列（从第一个字符开始计算）为一屏，因此一行代码不要超过80列，如果存在较长的语句，请将其分成多行内容来书写。

有些时候即使一行的代码没有超过80列，但是为了更好的阅读性还是要及时换行，主要有以下情况：

举例:

【逻辑较长语句、循环/判断语句】

if( person.m\_age <= 15

**&&** person.m\_age >= 10

**&&** person.m\_age != 12 )

上例中:

* 以较低优先级操作符(上例中就是&&)为界,操作符放在行首
* 换行的操作符和上一行的变量在同一列(&&和上一行的person在同一列)

【函数形参】

void convertImgData(

char\* imgData,

int imgWidth,

int imgHeight,

int convertAlgType，

int sourceImgType,

int destImgType)

{…}

函数参数大于5个时，必须进行换行，参数一行一个，或者把关联度很高的同类参数放在一行，如上面的imgWidth和imgHeight。

【函数调用作为参数】

for\_each( Container.begin(),

Container.end()

mem\_fun\_ref(&ContainerElement::DoSomething));

对于**url地址**、**长单词**、**文件名**等为了方便粘贴而必须放在一行的场合，一行可以超过80列的限制。

1. 代码缩进和对齐

代码缩进可以让我们很容易地对代码进行分块。所以程序块要采用缩进风格来编写。

* 缩进字符为4个，使用空格缩进，而不是TAB键，因为使用空格缩进能够保证所有的编辑器显示效果都一样，而TAB在不同的编辑器下可能缩进的字符数可能是不一样的（如记事本和QtCreator）。
* 对于QTCreator，自动格式化的快捷键为选中区域后，Ctrl+I。
* 程序块的“{”和“}”应独占一行并且位于同一列。

不佳示例

if（…）{ //×

…

}

if（…） //×

{

…

}

较好示例

if（…）//√

{

…

}

1. 空行

空格和空行能够使版本更加的层次更加清晰，有便于阅读和理解。常用的场合有：

* 不同的成员类型加空行。
* 不同的访问修饰符区域加空行。
* 不同的模块的成员之间加空行。
* 同一个文件中不同的类加空行。

可以参见“[四/12.类成员](#_类成员)”中的“3)声明次序”一节。

举例:

【在IOHelper.hpp不同的类加空行】

class FileHelper

{

…

}

**// blank line**

class DirctoryHelper

{

…

}

【函数中不同的功能区块之间加空行】

void exportResult()

{

//>>>-----------------------------------------------------------------------

//export inspection result of pad

for (int i = 0; i < m\_pads.size(); ++i)

{

...

}

//<<<-----------------------------------------------------------------------

**// blank line**

//>>>-----------------------------------------------------------------------

//export all image of fov

for (int i = 0; i < m\_fovs.size(); ++i)

{

...

}

//<<<-----------------------------------------------------------------------

…

}

1. 空格

当两个以上的关键字、变量、常量进行操作时，在它们之间的操作符前面、后面加空格。

不佳示例:

vector**<** Person **\*** **>** pPersons;

for**(**int i=0;i<count;++i**)**

{

Person**\*** pPerson=new Person **(** i **)**;

pPerson **->** init **( )** ;

pPersons**.**push\_back**(** pPerson **)**;

}

较好示例:

vector**<**Person**\*>** pPersons;

for **(**int i = 0;i < count; ++i**)**

{

Person**\*** pPerson = new Person**(**i**)**;

pPerson**->**init**()**;

pPersons**.**push\_back**(**pPerson**)**;

}

注意:

* 遇到“,”，后面加空格；
* 遇到“;”，如果其不是一行的结束符，后面加空格；
* if、for、while、catch等关键字加空格后再加“(”；
* 比较、赋值、算术、逻辑、位和双目操作符前后加空格;
* “!”、 “~”、 “++”、 “--”、 “&”、“->”，“.”，“\*”等单目操作符可以不加空格;
* “(”和“)”前后不加空格;
* “<”和“>”前后不加空格;

1. 初始化列表

构造函数初始化列表按四格缩进并排几行。

举例:

Student::Student( int var )

: m\_name,

: m\_age

{

…

}

# QML

## QML文件 & Component使用注意事项

1. 关于非QML的文件命名见“[一/ 1. filename 文件命名](#_file name 文件命名)”。
2. QML文件一律使用‘.qml’后缀。

* 如果qml文件使被当做Component来使用的，请务必使用大写字母开头，使用驼峰命名法，没有下划线。

举例:

MyComponent.qml

Background.qml

* 如果qml文件不需要当做Component使用，作为页面直接加载，则全部小写，不需要下划线连接。

举例:

main.qml //主QML

mapviewer.qml //QML动态加载的文件

1. QML位置

QML既可以定义在独立的QML文件中，也可以嵌入到其他的QML文档中。

* 如果一个Component比较小且只在某个QML文档中使用，或者一个Component从逻辑上看从属于某个QML文档，那么就采用嵌入的方式来定义该Component。

举例:

//Rectangle中定义colorComponent

Rectangle {

…

Component {

id: colorComponent;

Rectangle {

…

}

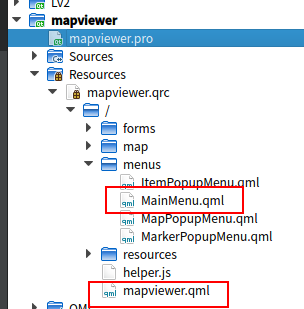
}

}

* 如果一个qml被多个文件调用，或者将来可能被多个文件调用，那么这个qml应该作为独立文件存在，按照逻辑归属放置到项目中对应的目录中，qml文件相对路径在qrc文件中配置，在需要调用的地方使用import qml所在目录。

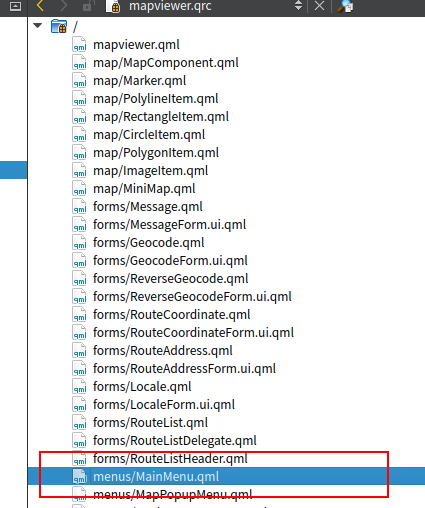
以QtCreator自带的mapviewer例子，mapviewer调用到menus目录中的MainMenu.qml。

举例:



实现MainMenu.qml调用分以下几个步骤：

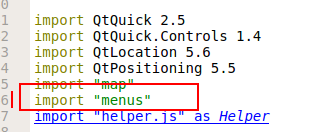
1. 在qrc中配置qml文件的相对路径，有两种方式:
   1. 在qtcreator中Editor中编辑



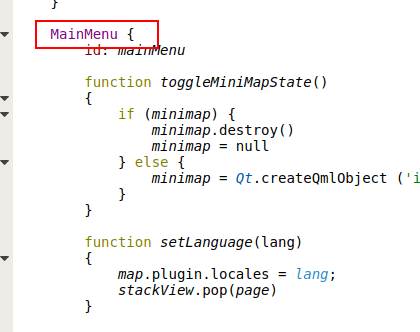
* 1. 使用文本编辑器(如gedit, sublime等)编辑qrc文件。



1. mapviewer.qml开头 import MainMenu所在目录。



1. mapviewer.qml中直接调用MainMenu即可。



## Class、Struct & Enum

1. QML中Class和Struct，命名时每个单词首字母大写，单词之间直接连接无需下划线。

举例:

class KeyBackQuit{

…

}

class ChartBackground{

…

}

1. 枚举类型采用首字母大写、枚举值则采取单词全部  
   大写，词与词之间采用“\_”连接

举例:

enum UrlTableError{

OK = 0,

OUT\_OF\_MEM= 1

}；

## Property属性

1. QML中的属性分为以下几种：

* 基本类型

小写字母开始，驼峰命名法

举例:

implicitWidth

* 使用id属性

一个对象的id是唯一的，id属性的值，首字符必须是小写，驼峰命名法。

举例:

Text {

id: txtColor;

…

}

注意:

对于一个QML元素，如果上下文中没有任何平级的同类型元素，并且没有被其它地方调用的话，该id元素可以省略。

//此Text中的id属性可以省略

Item{

Text{

~~id:txtItem;~~//id属性可以省略

text: “item”;

…

}

}

如果一个QML元素有id属性的话，一定要放在属性的第一位。

//× id属性没有排在属性的第一位

Item{

Text{

text: “item”;

…

id:txtItem;

}

}

//√ id属性排在属性的第一位

Item{

Text{

id:txtItem;

…

}

}

如果是控件的话，控件缩写作为前缀，具体命名建议参考该章节最后一章。

* 列表属性

列表在方括号内，以逗号分隔多个元素的集合。QML内只能包含QML对象，不能包含任何基本类型的字面量（如果非要包含，需要使用var关键字）。

举例:

Item{

Children:[

Text{

text: “one”;

},

Text{

text: “two”;

},

]

}

注意:

如果列表中只有一个元素，可以省略方括号

Item{

children :Image{};

}

* 信号处理器

一般以on<Signal>这种形式。

举例:

Button{

…

onClicked:{

…

}

}

* 分组属性

在某些情况下，使用一个“.”符号将相关的属性形成一个逻辑组。

举例:

//分组属性一个一个的赋值

Text {

font.pixelSize:18;

font.bold: true;

…

}

//分组属性放在一个花括号内赋值

Text {

Font{pixelSize：12；bold: true;}

…

}

分组属性一律放在一起集中赋值，如下例的font。禁止同一分组的属性被隔开赋值。

举例:

//√ font属性放在一起

Text {

font.pixelSize:18;

font.bold: true;

…

}

//×font属性没有放在一起

Text {

font.pixelSize:18;

text: “color”;

font.bold: true;

…

}

* 附加属性

附加到一个对象上的额外属性，这里需要注意的是，附件属性放到最后集中赋值。

举例:

Item{

width:100;

height:100;

focus:true;

…

Keys.enabled:false;

}

1. 属性初始化语句可以分行书写，语句后可以不要加“;”不过公司规定全部要加上“;”从而提高阅读性。

举例:

width: 600**;**

height: 400**;**

1. 属性可以把多个属性初始化语句写在一行内，它们之间必须加“;”，但是公司禁止这么做。

举例:

width: 600；height: 400； //×

width: 600； //√

height: 400；

1. 自定义属性遵守驼峰命名法，需使用**property**关键字申明，属性名称以小写字母开头。自定义属性和QML元素本身的属性之间要加一行空格，不同类型的属性也加一空行加以区分

举例:

Rectangle{

width:100;

…

//这里加一行空格区分

property int shapeTypeIndex: 0；

…

//这里加一行空格区分

property bool isSelected: false；

}

1. 属性的赋值可以使用表达式，公司禁止无意义的表达式属性赋值。

举例:

/\*× 这里的23\*10没有任何意义，既浪费了效率，又降低

了可读性，如果想表达和其它属性的绑定关联关系，直接

用id属性\*/

width: 23\*10;

//√ 正确的写法

height: 230;

1. 如果需要对属性之间建立关联，那么直接将属性设置为其他属性即可；如果需要引用一个对象，通过对象的id来引用。

举例:

【直接通过属性关联】

Text {

width：300;

height: width;

anchors.horizontalCenter: parent.horizontalCenter;

}

【通过id属性关联】

Button{

id: btnStart;

width:300;

height: 300;

}

Button{

id: btnEnd;

width：btnStart.width;

height：btnStart.height;

}

1. 属性初始化的顺序：
2. id属性(如果有的话)
3. 数据源相关属性（如果有的话，source、componentSource…）
4. Size相关属性（width，height，implictWidth、color…）
5. 显示相关属性（text，context…）
6. 布局相关属性（Row、Column、Grid…）
7. 其它属性
8. 附加属性
9. 信号处理器
10. 自定义属性

举例:

Rectangle {

id: rectTarget; //id属性

implicitWidth: 120; //Size相关属性

implicitHeight: 50;

color: "transparent"; //显示相关属性

anchors.left: parent.left; //布局相关参数

… //其他相关

Keys.enabled: false; //附加属性

signal clicked; //信号处理器

property alias iconSource: icon.source; //自定义属性

}

注意:

对于属性赋值，“:”后加一个空格，这样方便阅读。

## Function函数

1. 对于函数的命名使用驼峰命名法，小写字母开头，单词直接不加下划线，对于方法中形参的命名同样遵守这一原则。

举例:

function removeTarget(posX,posY){

…

}

1. 在QML中，推荐使用console.assert()，当使用console.assert在表达式不成立时并不会终止程序，而是会打印出错误的代码位置。

## [Variable](javascript:void(0);)变量

1. 变量命名以小写字母开始，单词之间使用驼峰命名法。

举例:

var isSuccessful = false;

var objectName = “obj”;

var tempValue = 20.1;

1. 一个var语句可以定义多个变量，可以是不同的类型,但是公司禁止这么做,为保证阅读性,请一行只放一个类型

错误做法:

var index = 0,name = "}"; //×

正确做法:

var index = 0; //√

var name = "}"; //√

1. 禁止一个变量在使用中变更类型

举例:

var index = 0;

index = "}"; //×

index = 10; //√

1. 判断2个变量是否相等要用“===”(3个等号)，“===”表示全等，包括值相等和类型相等

举例:

var index = 5;

index ===5 //√

index ==="5" //×

1. 判断变量的类型不能用typeof而要使用instanceof。因为typeof对于引用值统一返回object，这就导致了typeof无法准确判断出引用的类型，而instanceof就没有这样的问题，它能够返回引用对象真实的类型。

举例:

var str =new String(“Hello World”);

//× 输出固定为“object”

console.log(typeof str);

//√ 输出为true

console.log(str instanceof String);

1. 如果明确知道一个QML对象的属性或信号参数的类型(除此之外是不能使用基本类型定义变量)，比如说它只可能是int，那就直接指定其类型为int(放弃var)，这会提高性能。

举例:

Rectangle{

id: rectTarget;

var width = 100; //×

int width = 100; //√

…

}

## Comment 注释

1. 和普通的c++规范不同的是，公司规定在qml中的代码块左括号一定要写在句尾。

举例:

if(!isDrawBorder)**{**

ctx.strokeStyle = shapePicker.drawedShapeColor;

}

else**{**

ctx.strokeStyle = selectedShapeColor;

}

对于有多个括号嵌套或者是大范围的代码段，需要在右括号后面加上“End of xxx”的注释，但是如果“{}”内的代码很短，一屏可以看完，}后面可以不加注释。

举例:

Rectangle {

id: rectMain;

…

TargetListModel**{**

id: lstModel;

…

**}//End of lstModel**

**}//End of rectMain**

1. 为了提高可阅读性，方便代码定位。整个QML文件可以分Region区域进行注释（需要进一步讨论）。

//>>>----------------------------------------------------------------------------------

//Related to Property

//顶层元素（Rectange、Item、Window）属性相关

…

//<<<----------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to Model

//数据Model（ListView、XmlListModel等）相关

…

//<<<----------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to UI

//子元素及其模块相关

//（根据不同的作用模块可再细分）

…

//<<<----------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to function

//元素中的函数

//（根据不同的作用可再细分）

…

//<<<----------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to Component

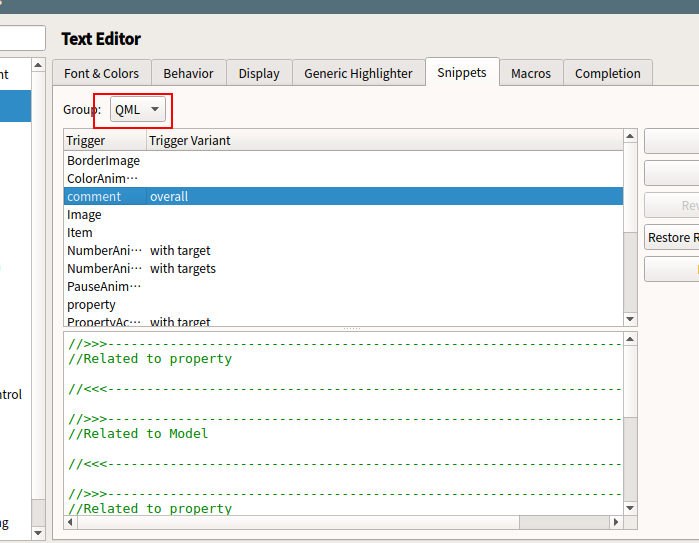
//QML调用到的Component相关

...

//>>>---------------------------------------------------------------------------------

注意:

* 一旦有>>>开头，必须要有<<<结尾，一定要成对出现，否则阅读起来很麻烦。
* 以上注释模版可写入Sinppets，设置和“[十一 / 1.注释](#_注释) / 6) 快捷注释”类似，区别是在Group中选择qml，而不是C++。



Add一个Trigger为“comment”，Trigger Variant为“overall”， 内容如下:

//>>>---------------------------------------------------------------------------------

//Related to property

//<<<---------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to Model

//<<<---------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to property

//<<<---------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to UI

//<<<---------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to function

//<<<----------------------------------------------------------------------------------

//>>>---------------------------------------------------------------------------------

//Related to component

//<<<---------------------------------------------------------------------------------

1. 对于属性的注释。

* 对于属性的注释以This property holds …，通常用于比较正式的注释,并且写在这条属性的上一行。

举例:

//**This property** **holds** whether the darkStyle is used.

property bool isDarkStyle: true;

* 对于比较简单的属性，可以直接在属性后面加“//…”。

举例:

property color mainRectDarkColor: Qt.rgba(0,0,0,0.5);

**//**color of rectMain in dark style

* 多个功能相近的属性放在一起注释，可以这样写:

举例:

//>>>----------------------------------------------------------------------------

//These properties hold color option

property color mainRectDarkColor: Qt.rgba(0,0,0,0.5);//color of rectMain in dark style

property color mainRectWhiteColor: Qt.rgba(1,1,1,0);//color of rectMain in white style

//<<<---------------------------------------------------------------------------

1. 对于子元素的注释：

对于一些重要的控件，一定要详细描述其作用

举例:

**/\***As a tempoary tool, it used in app start one time,then move data

to ListModel\*/

XmlListModel {

id: xmlModel;

…

onStatusChanged: {

…

}

}//End of xmlModel

1. 对于signal的注释

以This signal is emitted开头:

举例:

//This signal is emitted when the darkStyle is used.

void shapePicked(int shapeType);

1. 对于function的注释

以This function will 开头:

举例:

//This function will remove target which in (posX,posY)

function removeTarget(posX,posY){

…

}

## 常见QT控件缩写

缩写的原则优先级, 序号代表优先级, 优先级越高排序越前面：

1. 不超过5个字母的无需缩写。

举例:

|  |  |
| --- | --- |
| Pane | pane |
| Tab | tab |

1. 有英语标准缩写的用标准缩写。

举例:

|  |  |
| --- | --- |
| Calendar | **cdr** |
| ExclusiveGroup | exclgrp |

1. 程序员习惯缩写。

举例:

|  |  |
| --- | --- |
| MessageDialog | msgdlg |
| Drawer | drw |

1. 省略元音(a,o,e,i,u)。

举例:

|  |  |
| --- | --- |
| Dialog | dlg |
| Label | lbl |

1. 连续两个辅音为同一个字母省略其中一个。

举例:

|  |  |
| --- | --- |
| Button | btn |

1. 省略元音还很长后可以省略元音后面的辅音。

举例:

|  |  |
| --- | --- |
| TableViewColumn | tblvwcol |

1. 单词数量超过3个(含3个)可以直接使用首字母缩写。

举例:

|  |  |
| --- | --- |
| open file dialog | ofd |
| Save file dialog | sfd |

1. 类似与xxxBox这种Box结尾的常见控件，可以省略Box。

举例:

|  |  |
| --- | --- |
| CheckBox | chk |
| ComboBox | cmb |

1. 如果感觉缩写后完全无法理解其含义也可以部缩写。

举例:

|  |  |
| --- | --- |
| Stack | stack |

注意:

以上缩写规范在qml需要严格遵守，但是在C++中并不需要，在保证阅读性的前提下可以不完全遵守这些原则以换取写码效率。

1. Qt Quick常用控件的缩写:

|  |  |  |
| --- | --- | --- |
| **Module** | **UI元素全称** | **缩写** |
| Qt Quick XML List Model QML Types | XML List Mode | **xmlstmd** |
| XMLRole | **xmlrole** |
|  | ListView | lstVw |
| Qt Quick Dialogs QML Types | ColorDialog | clrdlg |
| Dialog | dlg |
| FileDialog | fldlg  ofd（打开对话框）  sfd（保存对话框） |
| FontDialog | fntdlg |
| MessageDialog | msgdlg |
| Qt Core | Timer | **timer** |
| Qt Quick Controls & Controls2 QML Types | Action | **actn** |
| ApplicationWindow | **app**wnd |
| AbstractButton | abstractbtn |
| BusyIndicator | **busyindic** |
| Button | btn |
| ButtonGroup | btngrp |
| Calendar | **cdr** |
| CheckBox | **chk** |
| CheckDelegate | chkdlgt |
| ComboBox | **cmb** |
| Container | **cntr** |
| Control | ctrl |
| Dial | **dial** |
| Drawer | **drw** |
| ExclusiveGroup | exclgrp |
| Flickable | flick |
| ItemDelegate | itmdlgt |
| Group Box | grp |
| Label | lbl |
| Menu | **menu** |
| MenuBar | **menubr** |
| MenuItem | **menuitm** |
| MenuSeparator | **menusepr** |
| ProgressBar | prg |
| Page | page |
| PageIndicator | **pgindic** |
| Pane | pane |
| Popup | **popup** |
| RadioButton | rdo |
| RadioDelegate | rdodlgt |
| RangeSlider | rngsld |
| ScrollView | svw |
| ScrollBar | vsb(垂直)  hsb(水平) |
| ScrollIndicator | **scrlindic** |
| Slider | sld |
| SpinBox | spn |
| SplitView | spltvw |
| Stack | **stack** |
| StackView | **stackvw** |
| StackViewDelegate | **stackvwd** |
| StatusBar | statbr |
| SwipeView | swpvw |
| Switch | **sw** |
| switchdlgt | swdlgt |
| Tab | tab |
| TabBar | tbbr |
| TabButton | tbbtn |
| TabView | tbvw |
| TableView | tblvw |
| TableViewColumn | tblvwcol |
| TextArea | txt |
| TextField | txtfld |
| ToolBar | tlbr |
| ToolButton | tlbtn |
| ToolTip | tltp |
| Tumbler | tmblr |
| TreeView | trvw |
| Qt Quick Layouts QML Types | ColumnLayout | clmnlyt |
| GridLayout | grdlyt |
| Layout | lyt |
| RowLayout | rowlyt |
| StackLayout | stacklyt |
| Qt Quick QML Type | Component | comp |
|  | Loader | ldr |
|  | QtObject | obj |
|  | Rectangle | rect |
|  | Canvas | canvas |
|  | Image | img |

## QtCreator 中翻译注意事项

1. 在qml文件中，需要翻译的文本一定要使用***qsTr***封装起来。

举例:

Window {

visible: true

width: 640

height: 480

title: ***qsTr***("Hello World")

1. 在c++文件中，需要翻译的文本一定要使用***tr***封装起来。

举例:

QString week[]

{

QObject::**tr**("Monday"),

QObject::**tr**("Thuesday"),

QObject::**tr**("Wednesday"),

QObject::**tr**("Thursday"),

QObject::**tr**("Friday"),

QObject::**tr**("Saturday"),

QObject::**tr**("Sunday")

};