















**基础**

* explicit构造函数

class Test

{

public:

explicit Test(int num);

};

explicit关键字阻止进行隐式类型转换，但仍可进行显式类型转换。

* 拷贝构造函数和赋值构造函数

class Widget

{

public:

Widget();

Widget(const Widget& rhs);//构造函数

Widget& operator=(const Widget& rhs);//赋值构造函数

};

**条款01:视C++为一个语言联邦**

c++语言层次:

* C.

C++仍是以c未基础。区块、语句、预处理器、内置数据类型、数组、指针

* Object-Oriented C++

类、构造与析构、封装、继承、多态、动态绑定。。。

* Template C++

模板元编程(TMP template metaprogramming)

STL

容器、迭代器、算法、函数对象

C++高效编程守则视状况变化，取决于使用C++哪一部分。

**条款02:尽量以const、enum、inline替换#define**

* inline关键字:

inline关键字更多地出现在C中，很多短小的工作函数可以达到快速执行的效果。函数声明为inline表示编译器会将inline函数粘贴到调用点出，省去了原本函数查找、调用的时间。

使用规则:

(1)不可包含复杂的结构控制语句，包含循环控制和复杂的条件控制，如while、switch等

(2)不可包含递归函数

inline函数定义在头文件。

* inline和宏函数的比较:

共同点:(1)都可以节省在函数调用方面带来的时间和空间开销。(2)实现的功能一致

区别:(1)在预编译时器，宏定义在调用处原样替换(宏展开)。在编译时期，内联函数在调用处展开，进行参数类型检查，宏定义不会进行参数类型检查。

(2)内联函数是函数，具有函数和宏定义的区别。

(3)内联函数可作为类的成员，可为类提供封装性，可以使用类的资源。

可以使用内联函数完全替代宏。

旧式编译器不允许static成员在其声明式上获得初值，可以采用将常量定义式为于实现文件内

enum hack:一个属于枚举类型(enumerated type)的是指可以权当ints被使用

class GamePlayer

{

private:

enum{NumTurns=5};

int scores[NumTruns];//数组初始化时必须明确数组长度

};

条款注意点:

* 对于单纯常量，最好以const对象或enum替换#define
* 对于形式函数的宏(macros)，最好改用inline函数替换#define

**条款03:尽可能使用const**

* STL迭代器的作用如同T\*指针，声明迭代器为const就像声明指针为const一样，即:

const vector<T>::iterator== T\* const

const\_iterator表示迭代器指向的内容不能够被改变。

* 令函数返回一个常量值，往往可以降低因客户错误而造成的意外，而又不至于放弃安全性和高效性。
* const 成员函数:不能更改类中的非静态成员变量(bitwise constness)

虽然有了const成员函数的修改限定，但若const成员函数对外部返回类内的成员变量指针或引用(返回值不加const限定的话)，仍然可以在类外部间接修改类的数据。因此有了logical constness:const成员函数可以修改对象内的某些bit,但只有在客户端侦察不出来的情况下。

* mutable:使用该关键字声明的变量可以在const成员函数中被修改。

条款注意点:

* 将某些东西声明为const可帮助编译器侦测出错误用法。const可被施加于任何作用域内的对象、函数参数、函数返回类型、成员函数本体。
* 编译器强制实施bitwise constness,但编写程序时应该使用logical constness。
* 当const和non-const成员函数有着实质等价的实现时，令non-const版本调用const版本可以避免代码重复。

**条款04:确定对象被使用前已被初始化**

* 在类中的构造函数进行的是初始赋值，使用初始化列表才是进行初始化的动作。

初始化列表进行初始化时，总是以成员声明次序被初始化，即使它们在初始化列表中处于不同的次序。

* C++对定义于不同编译单元内的non-local-static对象的初始化顺序无明确的定义。

条款注意点:

* 为内置型对象进行手工初始化，因为c++不保证初始化它们。
* 构造函数最好使初始化列表，而不要在构造函数本体内使用赋值操作。初始化列表列出的成员变量，其排列次序应该和它们在class中的声明次序相同。
* 为避免”跨编译单元之初始化次序”问题，以local static 对象替换 non-local static对象。

**条款05:了解c++默默编写并调用哪些函数**

* 父类如果是虚析构函数，则子类默认的虚构函数也是虚析构函数。
* 父类的拷贝构造函数或赋值构造函数声明为private,子类则不会生成对应的函数。

条款注意点:

* 编译器可以暗自为class创建default构造函数、copy 构造函数、copy assignment操作符以及析构函数。

**条款06:若不想使用编译器自动生成的函数，就该明确拒绝**

条款注意点:

* 为驳回编译器自动提供的机能，可将相应的成员函数声明为private并且不予实现(在调用时会产生无定义的链接错误)。使用Uncopyable这样的父类也是一种做法。

class Uncopyable

{

protected://允许子类构造和析构

Uncopyable(){}

~Uncopyable(){}

private://只有声明，没有定义，防止子类拷贝。如果有定义的话，子类的成员函数或友元函数仍可进行访问

Uncopyable(const Uncopyable&);

Uncopyable& operator=(const Uncopyable&);

};

**条款07:为多态基类声明virtual析构函数**

条款注意点:

* polymorphic(带多态性质的）的基类应该声明一个虚析构函数。如果类中带有任何的虚函数，这个类就应该拥有一个虚析构函数。
* 类的设计目的如果不是作为基类使用，或者不是为了具备多态性(polymorphically),就不该声明为虚析构函数。

**条款08:别让异常逃离析构函数**

* 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该能够捕获任何异常，然后不传播或结束程序。
* 如果客户需要对某个操作函数运行期间抛出的异常做出反映，那么class应该提供一个普通函数(而非在析构函数)执行该操作。

**条款09:绝不在构造和析构过程中调用virtual函数**

* 在构造和析构期间不要调用virtual函数，因为这类函数调用从不下降至derived class(比起当前执行的构造函数和析构函数那层)。

**条款10:令operator=返回一个reference to \*this**

* 令赋值(assignment)操作符返回一个reference to \*this.

**条款11:在operator=中处理“自我赋值”**

* 确保当对象自我赋值时operator=有良好的行为。其中的技术包括比较”来源对象”和“目标对象”的地址、精心周到的语句顺序以及copy-and-swap

来源对象:

Widget& Widget::operator=(const Widget& rhs)

{

if(this==&rhs)

return \*this;

delete pb;

pb=new Bitmap(\*rhs.pb);

return \*this;

}

精心周到的语句:

Widget& Widget::operator=(const Widget& rhs)

{

Bitmap\* pOrig=pb;//记住原有的

pb=new Bitmap(\*rhs.pb);//创建过程中可能抛出异常

delete pOrig;//创建正常后再删除旧有的

return \*this;

}

copy-and-swap

Widget& Widget::operator=(Widget rhs)//by value,是被拷贝对象的一份拷贝

{

swap(rhs);//void swap(Widget& rhs);

return \*this;

}

* 确定任何函数如果操作一个以上的对象，而其中多个对象是同一个对象时，其行为能然正确。

**条款12:复制对象时勿忘其每一个成分**

* Copying函数(拷贝构造函数、拷贝赋值函数)应该确保复制对象内的所有成员变量及所有父类的成员变量。

class Customer

{

public:

Customer(const Customer& rhs);

Customer& operator=(const Customer& rhs);

private:

*std*::*string* strVal;

*std*::*string* strVal2;

};

class DerivedCustomer :public Customer

{

public:

DerivedCustomer(const DerivedCustomer& rhs);

DerivedCustomer& operator=(const DerivedCustomer& rhs);

private:

*std*::*string* str;

};

DerivedCustomer::DerivedCustomer(const DerivedCustomer& rhs)

:Customer(rhs),//调用父类的拷贝构造函数

str("hello,world")

{

}

DerivedCustomer& DerivedCustomer::operator=(const DerivedCustomer& rhs)

{

Customer::operator=(rhs);//调用父类的赋值拷贝函数

str = "hello,world";

}

* 不要尝试以某个copying函数实现另外一个copying函数。应该将共同机能放进第三个函数中，并由两个copying函数共同使用。

**条款13:以对象管理资源**

* 以对象管理资源的两个关键想法:

获得资源后立刻放进管理对象中。

管理对象运用析构函数确保资源被释放。

* 为防止资源泄露，使用RAII(Resource Acquision Is Initialization)对象，它们在构造函数中获得资源并在析构函数中释放资源。
* 两个常被使用的RAII 类shared\_ptr和auto\_ptr。前者是较佳选择，因为其copy行为比较直观。auto\_ptr复制动作会使被复制中的资源变为null. shared\_ptr和auto\_ptr使用资源时都是使用delete的形式，所以不能进行对数组资源的管理(数组资源的释放需要使用delete[])。

**条款14:在资源管理类中小心copying行为**

* 处理RAII对象被复制的方法:

禁止复制，将copying操作声明为private

对底层资源使用“引用计数法”

复制底部资源

转移底部资源的拥有权

* 复制RAII对象必须一并复制它所管理的资源，所以资源的copying行为决定RAII对象的copying行为。
* 普片而常见的RAII class copying行为是:禁止复制、使用引用计数法。不过其他行为也可能被实现。

**条款15:在资源管理类中提供对原始资源的访问**

* APIs往往要求访问原始资源(raw resources),所以每一个RAII class应该提供一个取得其所管理的资源的办法。(operator-> 或 operator\* 做指针或引用时间接使用原生资源，或提供显示get()).
* 对原始资源的访问可能经由显示转换或隐式转换。一般而言显示转换比较安全，但隐式转换对客户会比较方便。

//隐式转换例子

#include <iostream>

#include <vector>

#include <string>

class Test

{

public:

void SaySomething(const *std*::*string*& str) const

{

*std*::*cout* << str<< *std*::*endl*;

}

};

class ImplicitClass

{

public:

ImplicitClass()

{

m\_Test = new Test;

}

~ImplicitClass()

{

if (m\_Test)

{

delete m\_Test;

m\_Test = nullptr;

}

}

operator Test\*()//隐式转换函数，称为函数对象

{

return m\_Test;

}

operator Test&()//重载隐式转换函数

{

return \*m\_Test;

}

private:

Test\* m\_Test;

};

void TestImplicitClass(const Test& testRef, Test\* testPtr)

{

testRef.SaySomething("ref...");

testPtr->SaySomething("ptr...");

}

int main()

{

ImplicitClass implicit;

TestImplicitClass(implicit, implicit);

*system*("pause");

return 0;

}

**条款16:成对使用new和delete时要采用相同的形式**

* 如果在new表达式中使用[],必须在相应的delete表达式中也是用[].相对地，如果在表达式中不使用[],一定不要在相应的delete表达式中使用[]。

**条款17：以独立语句将newed对象置入智能指针**

* 以独立语句将newed对象存储于只能指针中，如果不这样做，一旦异常被抛出，有可能导致难以察觉的资源泄漏。

//函数声明

int GetPriority();

void ProcessWidget(std::tr1::shared\_ptr<Widget> pw,int priority);

//进行调用

ProcessWidget(std::tr1::shared\_ptr<Widget>(new Widget),priority());

//如果操作序列为

new Widget->priority()->shared\_ptr构造函数

在该序列中如果priority()抛出一样就有可能造成内存泄漏

**条款18:让接口容易被使用，不易被误用**

* 好的接口容易被正确使用，不容易被误用。应该在所有的接口中努力达成这些性质。
* 促进正确使用的办法包括接口的一致性，以及与内置类型的行为兼容。
* 防止误用的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任。
* tr1::shared\_ptr支持定制型删除器(custom deleter).这可以防范跨dll资源释放问题，可被用来自动解除互斥锁(默认的shared\_ptr在引用计数为0时执行delete操作)等。

**条款19:设计class犹如设计type**

* 新type的对象应该如何被创建和删除?（operator new,operator new[],operator delete,operator delete[]）
* 对象的初始化和对象的赋值应该有怎样的差别?
* 新type的对象如果被pass by value，意味着什么?
* 什么是新type的合法值?
* 新type需要配合某个继承图系(inheritance graph)吗？
* 新type需要什么样的转换(显式类型转换、隐式类型转换)?
* 什么样的操作符和函数对新type是合理的？
* 什么样的标准函数应该驳回？(不适用编译器提供的默认拷贝函数、默认析构函数、默认构造拷贝、默认构造赋值拷贝等)
* 谁该取用新type的成员?（成员的访问级别是什么？友元类？友元函数?）
* 新type有多么一般化？(是否适合使用模板编程)
* 真的需要一个新type吗？
* class的设计就是type的设计。在定义一个新的type之前，确定考虑全面。

**条款20:宁以pass-by-reference-to-const 替换 pass-by-value**

* 尽量以pass-by-reference-to-const替换pass-by-value。前者通常比较高效，并可避免切割问题(slicing problem).
* 以上规则并不适用于内置类型，以及STL的迭代器和函数对象。对他们而言，pass-by-value往往比较恰当。

**条款21:必须返回对象时，别妄想返回其reference**

* 绝对不要返回pointer或renference指向一个local stack对象，或返回reference指向一个heap-allocated对象，或返回pointer或reference只想一个locatl static对象而有可能同时需要多个这样的对象。

**条款22:将成员变量声明为private**

* 切记将成员变量声明为private。这可赋予客户访问数据的一致性、可细微划分访问控制、允诺约束条件获得保证，并提供class作者以充分的实现弹性。
* protected并不比public更具封装性。

**条款23:宁以non-member、non-friend替换member函数**

* 从封装的角度看，friends函数对class private成员的访问权力和member函数相同，因此两者对封装的冲击力道也相同。
* 只因在意封装性而让函数称为class的non-member，并不意味它不可以是另一个class的member.
* 可以将non-member函数组织在同一命名空间中，以不同的头文件进行模块划分。
* 宁可拿non-member non-friend函数替换member函数。这样做可以增加封装性、包裹弹性(packaging flexibility)和机能扩充性。

**条款24:若所有参数皆需类型装欢，请为此才哦那个non-member函数**