1. **Explicit用法**

首先C++中explicit关键字只能用于修饰只有一个参数的类构造函数，它的作用是表明该构造函数是明显的，而非隐式的，跟它相对应的另一个关键字是implicit，意思是隐藏的，类构造函数默认情况下即为implicit（隐式）。explicit关键字的作用就是防止类构造函数的隐式自动转换. explicit关键字只对有一个参数的类构造函数有效, 如果类构造函数参数大于或等于两个时, 是不会产生隐式转换的, 所以explicit关键字也就无效了. 但是, 也有一个例外, 就是当除了第一个参数以外的其他参数都有默认值的时候, explicit关键字依然有效, 此时, 当调用构造函数时只传入一个参数, 等效于只有一个参数的类构造函数,

**class** CxString  // 没有使用explicit关键字的类声明, 即默认为隐式声明

{

**public**:

**char** \*\_pstr;

**int** \_size;

    CxString(**int** size)

    {

        \_size = size;                // string的预设大小

        \_pstr = malloc(size + 1);    // 分配string的内存

        memset(\_pstr, 0, size + 1);

    }

    CxString(**const** **char** \*p)

    {

**int** size = strlen(p);

        \_pstr = malloc(size + 1);    // 分配string的内存

        strcpy(\_pstr, p);            // 复制字符串

        \_size = strlen(\_pstr);

    }

    // 析构函数这里不讨论, 省略...

};

    // 下面是调用:

    CxString string1(24);     // 这样是OK的, 为CxString预分配24字节的大小的内存

    CxString string2 = 10;    // 这样是OK的, 为CxString预分配10字节的大小的内存

    CxString string3;         // 这样是不行的, 因为没有默认构造函数, 错误为: “CxString”: 没有合适的默认构造函数可用

    CxString string4("aaaa"); // 这样是OK的

    CxString string5 = "bbb"; // 这样也是OK的, 调用的是CxString(const char \*p)

    CxString string6 = 'c';   // 这样也是OK的, 其实调用的是CxString(int size), 且size等于'c'的ascii码

    string1 = 2;              // 这样也是OK的, 为CxString预分配2字节的大小的内存

    string2 = 3;              // 这样也是OK的, 为CxString预分配3字节的大小的内存

    string3 = string1;        // 这样也是OK的, 至少编译是没问题的, 但是如果析构函数里用free释放\_pstr内存指针的时候可能会报错, 完整的代码必须重载运算符"=", 并在其中处理内存释放

1. **拷贝构造函数和赋值运算符**

拷贝构造函数被用来以同类型对象初始化新对象，而赋值运算符被用来从另一个同类型对象中拷贝其值到自我对象。总体来说拷贝构造函数是用于新对象，赋值运算符用于老对象。

拷贝构造函数是一个尤其重要的函数，它定义了一个对象如何passed by value(以值传递)。

1. **条款1：视C++为一个语言联邦**

如今的C++已经是一个多重泛型编程语言(multiparadigm programming language),一个同时支持过程形式(procedural)、面向对象形式(object-oriented)、函数形式(functional)、泛型形式(generic)、元编程形式(metaprogramming)的语言。

为了理解C++下面主要有4点：

**C**。

说到底C++仍是以C为基础的。区块(blocks)、语句(statements)、预处理器(preprocessor)、内置数据类型(built-in data types)、数组(arrays)、指针(pointers)等统统来自C。许多时候C++对问题的解法其实不过就是较高级的C解法，但当你以C++内的C成分工作时，高效的编程守则映照出C语言的局限。

**Object-Oriented C++**。

这部分就是C with classes所诉求的：classes(包括构造函数和析构函数)，封装(encapsulation)、继承(inheritance)、多态(polymorphism)、virtual函数(动态绑定)……等等。

**Tamplate C++**

这是C++的泛型编程(generic programming)部分，也是大多数程序员经验最少的部分。Template相关考虑与设计已经弥漫整个C++，良好的编程守则中“惟teamplate适用”的特殊条款并不罕见。

**STL。**

STL是个template程序库，它是非常特殊的一个。它对容器(containers)、迭代器(iterators)、算法(algorithms)以及函数对象(function objects)的规约有极佳的紧密配合与协调

**Tips：高效编程守则视状况而变化，取决你使用的C++的那一部分。**

1. **条款2：尽量以const、enum、inline替换#define**

例 #define ASPECT\_RATIO 1.653

记号名称ASPECT\_RATIO也许从未被编译器看见；也许在编译器开始处理源码之前它就被预处理器移走。于是记号名称ASPECT\_RATIO可能没进入符号表，当你在追踪它时浪费时间。这时用const常量替换就能在编译时期进行类型检查，相对安全。

当用const常量替换#define时，有两种情况值得注意，第一是定义常量指针(const pointers)。由于定义式通常被放在头文件内，因此有必要将指针申明为const。第二个注意的是class专属常量。为了将常量的作用域限制于class内，你必须让它成为class的一个成员(memeber)；为确保此常量至多只有一个实体，你必须让它成为一个static成员。

另一个常见的#define误用情况是以它实现宏(macros)。宏看起来像函数，但不会招致函数调用(function call)带来的额外开销。

例 #define CALL\_WITH\_MAX(a,b) f((a)>(b)?(a):(b))

无论何时当你写出这种宏，你必须记住为宏中的所有实参加上小括号，否则某些人在表达式中调用这个宏时可能会遭遇麻烦。但纵使为所有的实参加上小括号，仍然会发生一些不可思议的事情：

int a = 5,b = 0;

CALL\_WITH\_MAX(++a,b); //a被累加两次

CALL\_WITH\_MAX(++a,b+10) //a被累加一次

为了防止这种悬而未决的问题,你可以获得宏带来的效率以及一般函数的所有可预料的行为和类型安全性(type safety)-----只要你写出template inline函数。

有了consts、enums和inlines，我们对预处理器(特别是#define)的需求降低了，但并非完全消除。#include仍然是必须品，而#ifdef/#ifndef也继续扮演控制编译的重要角色。目前还不到预处理器全面隐退的时候，但你应该明确地给予它更长更频繁的假期。

**注意**：对于单纯常量，最好以const对象或enums替换#define

对于形似函数的宏(macros)，最好改用inline函数替换#defines。

**5.条款3：尽可能使用const**

const的一件奇妙事情，它允许你定义一个语义约束,编译器会强制实施这项约束。关键字const有多种用途，它可以在classes外部修饰global或namespace作用域中的常量。或修饰文件、函数、或区块作用域(block scope)中被声明为static的对象。你也可以用classes内的staic和non-static成员变量。面对指针，你可以指出指针自身、指针所指物，或者两个都是const：

char greeting[] = “Hello”

char\* p = greeting; //non-const pointer,non-const data

const char\* p = greeting; //non-const pointer,const data

char\* const p = greeting; //const pointer,non-const data

const char\* const p = greeting; //const pointer,const data

const到底修饰变量还是指针只需关注const出现的位置，如果const出现在星号的左边，表示被指物为常量