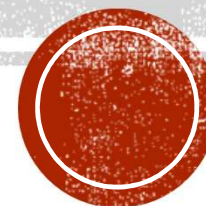


C++ 高质量编程

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn



运行正确的代码 != 好的代码

```
class Solution{    public:bool isMatch
(const char *s, const char*p) {int n= 0, m =
0; while(*(s + m)!='\0' )m ++;while (*(p + n) !=
'\0')n++ ;vector < vector<bool >> dp(m + 1, vector
< bool>(n + 1, false) );for (int j = n;j >= 0; j--)for
( int i = m ;i >= 0; i--) { if (j < n && p [j] == '*'
)continue;if (j == n) {if (i == m){dp[ i][j] = true
;}}else if (i == m) {if (p[j + 1] == '*')dp[ i]
[j] = dp[i][j + 2];} else {if (j <n- 1 &&p[j +
1] == '*' && dp[i][j + 2])dp[i] [j] = true
;if (p[j] == '.' || p[j]== s[i]) {if (
dp[i + 1][j + 1]) dp[i][j]= true;
if ( p [j + 1] == '*' && dp[i +
1] [j]) dp[i][j] = true
; } } } return dp[0]
[ 0] ; } } ;
```



高质量代码

- 别人能看懂（可维护）
- 别人能改进（可扩展）
- 别人能信任（安全性）

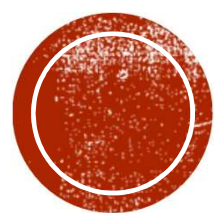


大纲

- 良好的可维护性
 - 统一代码风格
 - 命名规范
 - `const` 关键词
 - 注释
 - 传值和引用
 - 资源的释放
 - 编程规范
 - 其他可维护性
- 良好的可扩展性
 - 代码抽象
 - 设计模式

- 安全性
 - 输入判断
 - 异常处理
 - 路径的写法
- 其他





良好的可维护性



统一代码风格

项目合作中编程风格要统一

```
void Foo(int m, bool n)
{
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            int sum = i + j;
            printf("%d + %d = %d", i, j, sum);
        }
    }
}
```

逗号后面空一格

无论几条语句，for, if, while要加{ }

一行只写一条语句

运算符前后空一格

代码要缩进对齐



命名规范

- 有描述性的名字 `int a, b, c; int nDayOfDate, nMonthOfDate, nYearOfDate;`
- 风格要统一 `int size_of_nodes, currentPosition;`
- 尽量避免缩写 `ImageProcessor ip;`
- 避免过于相似的命名 `filename1, filename2 configFilename, inputFilename`
- 通过命名空间来避免冲突
- 经典命名规则
 - 匈牙利命名法: 变量名=属性+类型+对象描述, 例如 `m_pCar, g_nMaxSize`
 - 骆驼命名法: 首字母小写, 其它单词首字母大写, 例如 `createObject, run`
 - 帕斯卡命名法: 每个单词首字母大写, 例如 `BeginSend, Start`



推荐的命名习惯

- 类型（类/结构体）帕斯卡命名
 - `class StreamReader`
 - `struct RequestContext`
- 变量名匈牙利命名
 - 指针类型的成员变量 `m_pObject`
 - 整型的全局变量 `g_nAge`
 - 静态字符串局部变量 `s_zHostName,`
- 函数名帕斯卡命名或骆驼命名（但要一致）
 - `GetInstance()`
 - `isRunning()`
 - `Verb + Noun`
- 常量 全大写
 - `MAX_LENGTH`

属性前缀	说明
无	局部变量
m_	类的成员变量
sm_	类的静态成员变量
s_	静态变量(静态局部变量)
g_	全局变量
sg_	静态全局变量

类型前缀	说明
n	整型和位域变量 (number)
e	枚举型变量 (enumeration)
c	字符型变量 (char)
b	布尔型变量 (bool)
f	浮点型变量
p	指针型变量和迭代
pfn	函数指针



插播：静态变量STATIC

- 静态局部变量：
 - 它不管其所在的函数是否被调用，都将一直存在
 - 函数体内如果在定义静态变量的同时进行了初始化，则以后程序不再进行初始化操作
 - 把局部变量改变为静态变量后是改变了它的存储方式，即改变了它的生存期
- 静态全局变量：
 - 静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它
 - 外部的Static声明亦可用于声明函数
 - 把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围
- 函数的可重入性
 - 函数重复调用时，相同输入是否保证给出相同的输出



插播：静态变量STATIC

- 局部变量：生存期（时域）局部；可见性（空域）局部
- 静态局部变量：生存期（时域）**全局**；可见性（空域）局部
- 全局变量：生存期（时域）全局；可见性（空域）全局
- 静态全局变量：生存期（时域）全局；可见性（空域）**局部**（本文件内）

类的静态成员？



其它命名习惯

- 抽象类命名 I + 类型描述

```
class IProcessor
```

- 继承类命名 C + 类描述， 或 Impl + 类描述

```
class CFactory      class ImplRequestDispatcher
```

- 私有成员变量/函数以_开头或结尾

```
int count_;      void _OnTimer()
```

小心！系统头文件里将宏名、变量名、内部函数名用"_"开头



其它命名习惯

- 常量命名（以k开头，大小写混合）

```
const int kAge = 23
```

- 命名空间命名（使用小写，避免命名空间嵌套、冲突）
 - 比如，不要创建嵌套std的命名空间



合理使用CONST关键字

- 提高可读性，仅通过声明可以了解内部实现的一些特性

```
const char* doSomething(const char* input) {  
    // ...  
}
```

返回内部数据请不要修改

我不会修改你的输入

- 通过 `const` 关键字来保证特定变量不能被误改
- 在编译时发现潜在问题
- 提高编译的代码优化能力
- 提高自身对代码的了解



正确理解CONST关键字

▪ `const`^① `char*` `Foo(const`^② `char*` `const`^③ `&p)` `const`^④

▪ ① 无法直接通过返回的指针修改目标值

`const char*` `a = object.Foo(p);` // 此处无法直接赋值给 `char*`, 只能是 `const char*`

`*a = 'c';` // 编译错误

▪ ② 无法通过`*p`修改目标值

▪ ③ `p`值不能在方法内被修改

▪ ④ 修饰成员方法, 表示该方法不会修改该对象的成员变量



CONST关键字作用

在下面代码中寻找BUG

```
void f(int x, int y) {  
    if ((x==2 && y==3) || (x==1))  
        cout << 'a' << endl;  
    if ((y==x-1) && (x==-1 || y=-1))  
        cout << 'b' << endl;  
    if ((x==3) && (y==2*x))  
        cout << 'c' << endl;  
}
```

```
void f(const int x, const int y) {  
    if ((x==2 && y==3) || (x==1))  
        cout << 'a' << endl;  
    if ((y==x-1) && (x==-1 || y=-1))  
        cout << 'b' << endl;  
    if ((x==3) && (y==2*x))  
        cout << 'c' << endl;  
}
```

test.cpp: In function 'void f(int, int)': test.cpp:7:31: error:
assignment of read-only parameter 'y'



CONSTEXPR

- C++11 能够在编译时把 `constexpr` 修饰表达式直接求值，能增加程序的性能
- C++17 将 `constexpr` 这个关键字引入到 `if` 语句中，允许在代码中声明常量表达式的判断条件

```
template<typename T>
auto print_type_info(const T& t) {
    if constexpr (std::is_integral<T>::value) {
        return t + 1;
    } else {
        return t + 0.001;
    }
}

int main() {
    std::cout << print_type_info(5) << std::endl;
    std::cout << print_type_info(3.14) << std::endl;
}
```


提供“完整”的注释

- 注释帮助别人加速理解代码

```
// 查找元素所在位置，如果未找到返回-1
```

```
int FindIndex(Element ele)
```

```
// 获取指定位置的对象，如果位置非法，抛出异常
```

```
Element GetObjectByIndex(int position)
```

```
// 构造时间对象，ticks为Unix时间戳
```

```
DateTime(ulong ticks)
```

- 使用//或者使用/**/都可以，但是要保持风格统一。//更常用
- //后一般接一个空格



案例1：一个函数

```
/**
 * 查找数组中第N大的元素
 * @param tArray:      T类型元素的向量
 * @param n:           返回的元素的顺位
 * @param tComparer:   定义了T类型元素之间的比较。当Compare(a, b) > 0, 元素a会排在b之前
 * @return:            返回按tCompare的排序方式排序的第n个元素
 * @note:              1. 若有多个第n位元素，返回任意一个。
 *                    2. 要Compare方法有传递性。
 *                    3. 如果向量长度小于n则返回排序后的最后一个元素
 */
template <class T>
T NthElement(std::vector<T>& tArray, int n, Comparer<T> tComparer)
{
    /* 该方法首先会以vector的第一个元素作为中心点，从左到右遍历
       vector所有元素，当当前位置的元素小于中心点.....*/
    .....
}
```



注释要写什么

- 函数注释
 - 每个参数是什么意思？
 - 返回结果是什么？
 - 异常情况下预期的结果是什么？
 - 调用时特别需要注意的地方
- 类注释
 - 描述类的功能和用法, 除非它的功能相当明显
 - 多线程是否安全？（静态变量）
- 文件注释
 - 版权，如Copyright (c) Some Corporation. All rights reserved。
 - 许可版本，如Apache 2.0
 - 作者、日期
 - 内容简介



TODO 注释

- 对临时的解决方案，仍然不够完美的代码使用大写**TODO**注释
 - **TODO**后圆括号内加入自己的名字、邮箱地址、bug ID
 - 在注释说明中写出待实现的功能

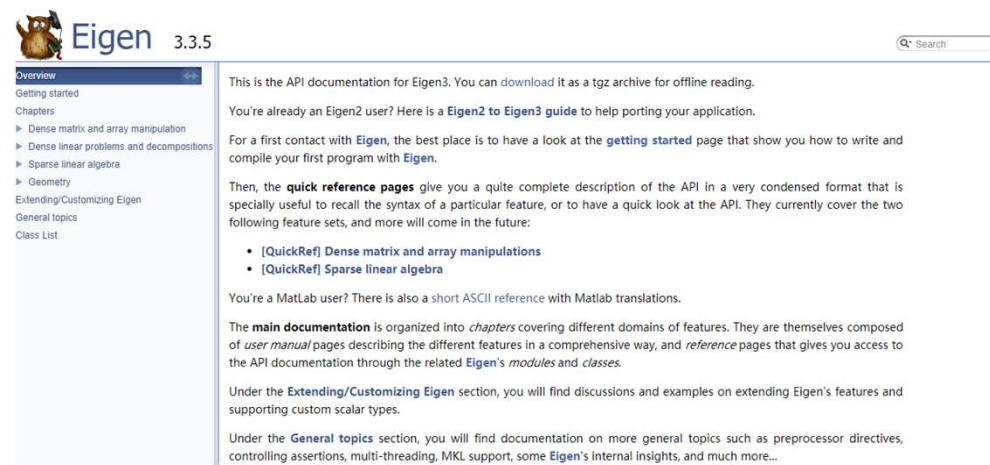
```
// TODO(wzb@mails.tsinghua.edu.cn): Update ...
```

```
// TODO(Zhibo Wang): ...
```



DOXYGEN 文档生成工具

- Doxygen 由已注释代码生成对应的文档
 - 能够生成在线的网页文档以及PDF
 - 能够提取出代码的结构，并且生成一幅代码的结构图
- 注释要符合特殊的语法规则
- Doxygen生成的文档的例子
 - Eigen文档 <http://eigen.tuxfamily.org/dox/>



案例2： 一个二分查找

```
int BinSearch(vector<int>&array, int left, int right, int want) {  
    if (left >= right) return -1;  
    int mid = left + (right - left) / 2;  
    if (array[mid] == want) return mid;  
    if (array[mid] < want) return BinSearch(array, mid + 1, right, want);  
    return BinSearch(array, left, mid, want);  
}
```

对于10M个数字做查询，占用内存890MB



对于10M个数字做查询，占用内存41MB



引用/值传递

- 什么时候用引用传递
 - (大型)对象

```
BYTE* ExtractFeature(const Image& srcImage)  
Complex& operator=(const Complex& rhs)
```

- 需要改变参数(对象)本身
`void Swap(int& a, int& b)`
- 多态继承(动态多态性)



值传递的陷阱

- 当值传递发生时，会调用参数声明类型的拷贝构造函数

```
class BaseClass
{
public:
    BaseClass()
    {
        std::cout << "BaseClass constructed"
                    << std::endl;
    };
    ~BaseClass(){};

    BaseClass(const BaseClass& rhs)
    {
        std::cout << "BaseClass called"
                    << std::endl;
    }

    virtual void Mew()
    {
        std::cout << "BaseClass mew"
                    << std::endl;
    }
};
```

```
class DerivedClass : public BaseClass
{
public:
    DerivedClass()
    {
        std::cout << "DerivedClass constructed"
                    << std::endl;
    };
    virtual ~DerivedClass(){};

    DerivedClass(const DerivedClass& rhs) :
    BaseClass(rhs)
    {
        std::cout << "DerivedClass called"
                    << std::endl;
    }

    virtual void Mew()
    {
        std::cout << "DerivedClass mew"
                    << std::endl;
    }
};
```

```
void Foo(BaseClass& object)
```

```
void Foo(BaseClass object)
{
    object.Mew();
}

int _tmain(int argc, _TCHAR* argv[])
{
    DerivedClass derivedObject;
    Foo(derivedObject);
}
```

OUTPUT:
DerivedClass constructed
BaseClass called
BaseClass mew



RAII

- Resource Acquisition Is Initialization
- 这是C++中的一个重要的概念：
 - 当一个对象初始化时，该对象在其构造函数中应当已经获取了对应的资源
 - 当离开这个对象的作用域时，该对象在其析构函数中应当释放掉其使用的所有资源



案例3：日志记录

```
void output(const char* fileName, char* message)
{
    FILE* fout = fopen(fileName, "a");
    if (fout == nullptr || message == nullptr)
    {
        return;
    }

    fwrite(message, sizeof(char), strlen(message), fout);
    fclose(fout);
}
```



符合RAII的代码

```
class FileObj {  
public:  
    FILE* ptr;  
    FileObj(char* name) : ptr(fopen(name, "r")) {}  
    ~FileObj() { fclose(ptr); }  
};
```



及时释放资源

- 有哪些资源？
 - 内存、文件、线程、网络、锁
- 所有的创建都应该有对应的释放
 - `new` / `delete`
 - `socket.connect()`; / `socket.close()`;
 - `fopen` / `fclose`
- 特别注意，异常情况往往是出现泄露的原因
- 尽量延迟变量的声明及初始化
- 合理使用智能指针解决内存释放问题
 - 将指针装在一个类的对象里，在对象被析构时尝试释放指针指向的地址

```
int Foo()  
{  
    unique_ptr<char[]> pBuffer(new char[1024]);  
    // Do something  
    return 0;  
}
```

我的例子：

```
Cimg* a = new Cimg (width, height);  
a=cvClone(b);  
Release(*a);  
Release(*b);
```



CVCOPY 与 CVCLONE

opencv 中的 `cvCopy` 函数与 `cvClone` 函数都可以用做图像的复制

- `cvCopy` 使用时需要先手动开一段内存并作为输入，该函数会把要进行复制的图像拷贝到这段内存中
- `cvClone` 使用时则会自动开一段内存并进行复制，之后返回这段内存。循环使用时易引发内存泄漏

智能指针

- `std::unique_ptr`
 - 独占其所拥有的资源
 - 离开该变量的作用域，资源释放
- `std::shared_ptr`
 - 可以与其他`shared_ptr`共享资源
 - 当资源无指针指向，资源释放
- 在现代C++用，总是使用智能指针，代替`new/delete`



避免“野”指针

- “野”指针
 - 没有赋值的指针
 - 任何指针变量刚被创建时不会自动成为空指针，它的缺省值是随机的，它会乱指一气。
 - 被 `delete` 但没有赋为空的指针
 - 别看 `free` 和 `delete` 的名字恶狠狠的（尤其是 `delete`），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。
- 问题
 - 无法判断指针是不是有效
- 解决方法
 - `#define SAFE_DELETE(x) if (x) delete (x); (x) = nullptr;`
- 注意
 - 1: 不要返回指向栈内存的指针或引用，因为栈内存在函数结束时会被释放。
 - 2: 在使用指针进行内存操作前记得要先给指针分配一个动态内存。

注意多个指针指向同一块内存的情况，还是用智能指针吧



使用nullptr赋值空指针

- C++11定义了一个特殊的表示空指针的符号：nullptr，使用nullptr而非0或NULL

```
void fun(int) {  
    std::cout << "fun1" << std::endl;  
}  
void fun(void *) {  
    std::cout << "fun2" << std::endl;  
}  
int main(int argc, char *argv[])  
{  
    fun(NULL); // 输出 fun1 而非 fun2  
    system("pause");  
    return 0;  
}
```



插播：栈内存和堆内存

- 本质区别
 - 栈区 (`stack`) — 由编译器自动分配释放
 - 堆区 (`heap`) — 一般由程序员分配释放
 - 全局区 (静态区) (`static`) — 全局变量和静态变量
- 效率
 - 栈: 由系统自动分配, 速度较快。但程序员是无法控制的。
 - 堆: 由 `new` 分配的内存, 一般速度比较慢, 而且容易产生内存碎片, 在内存中并不连续
- 大小
 - 栈: 较小, 连续的内存区域, 由编译器决定栈的大小 (一般 1M/8M)
 - 堆: 较大, 是不连续的内存区域 (链表管理), 受限于计算机系统中有效的虚拟内存



其他编程规范

- 不要将程序输出目录直接设置为源文件目录
- 不要在.h文件中使用**using namespace**, 但尽量把自己的函数和类定义在**namespace**
- 头文件务必使用**#pragma once**或**#ifndef XXX...#define...#endif**
 - **#pragma once**由编译器提供保证, 在某些平台上可能不被支持,
 - 而第二种用法可能会因宏命名的问题而出错, 如何使用还是要看自己的需求来



其他编程规范

- 使用{}

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

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



其他编程规范

- 每行代码应有一个合理的长度

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

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



其他编程规范

- 引用头文件时使用双引号

include <> 格式：一般用于包含标准的库头文件，编译器会去系统配置的库环境变量或者用户配置的路径去搜索，而不会在项目的当前目录去查找

include "" 格式：一般用于包含用户自己编写的头文件，编译器会先在项目的当前目录查找，找不到后才会去系统配置的库环境变量和用户配置的路径去搜索

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

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

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



其他编程规范

- 变量初始化
 - 在使用变量时声明变量
 - 尽量让变量的声明和初始化在一起

```
std::vector<std::string> text;  
....  
text={"aa","bb","cc"};  
...  
for(const auto& it:text)  
{...}
```



```
std::vector<std::string> text={"aa","bb","cc"};  
for(const auto& it:text)  
{...}
```



其他编程规范

■ 成员变量的初始化

```
// Bad Idea
class MyClass
{
public:
    MyClass(int t_value)
    {
        m_value = t_value;
    }

private:
    int m_value;
};
```

```
// Good Idea
// C++'s member initializer list is unique to the language and leads to
// cleaner code and potential performance gains that other languages cannot
// match.
class MyClass
{
public:
    MyClass(int t_value)
        : m_value(t_value)
    {
    }

private:
    int m_value;
};
```

初始化列表省一次变量的构造



其他编程规范

- 当初始化参数过多时，可以用初始化参数列表进行初始化

```
class Test
{
public:
    Test(int x, int y, int z) : a(x), b(y), c(z) {} //初始化参数列表
private:
    int a;
    int b;
    int c;
};

int main() {
    Test test(0, 1, 2); //x=0, y=1, z=2
    return 0;
}
```



其他编程规范

- 当初始化参数过多时，可以用初始化参数列表进行初始化

```
class ClassA
{
public:
    ClassA() { cout<< "construct\n"
<<endl; }
    ClassA(const ClassA&
classB){ cout<< "Copy\n" <<endl; }
//初始化参数列表
};

Class Test
{
Public:
    Test(ClassA &x,ClassA &y,int z) :
```

```
a(x), b(y), c(z) {} //初始化参数列表
```

```
Private:
    ClassA a;
    ClassA b;
    int c;
};
```

```
...
```

```
int z=0;
ClassA CA1, CA2;
Test test(CA1, CA2, z);
//这一行会输出什么?不使用初始化列呢?
//不使用引用呢?
```



其他编程规范

- 成员变量的初始化2
 - C++11中，可以尝试给每个成员变量设定初值

```
// Best Idea

// ... //

private:
    int m_value{ 0 }; // allowed
    unsigned m_value_2 { -1 }; // compile-time error, narrowing from signed to unsigned.
// ... //
```

- 使用{}进行初值的设定!
- 忘记初始化造成的问题，是程序开发中一种较难发现的bug



其他编程规范

- 谨慎处理STL的返回值类型
 - Auto转化成了std::size_t

```
std::vector<int> v1{2,3,4,5,6,7,8,9};  
std::vector<int> v2{9,8,7,6,5,4,3,2,1};  
const auto s1 = v1.size();  
const auto s2 = v2.size();  
const auto diff = s1 - s2; // diff underflows to a very large number
```



插播：AUTO

- 声明变量时，根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符

```
auto f=3.14; //double
auto z = new auto(9); // int*
auto x1 = 5, x2 = 5.0, x3='r'; //错误，必须是初始化为同一类型
```

- **auto**关键字更适用于类型冗长复杂、变量使用范围专一时，使程序更清晰易读：

```
std::vector<int> vect;
for(auto it = vect.begin(); it !=
vect.end(); ++it)
{ //it的类型是std::vector<int>::iterator
    std::cin >> *it;
}
```



插播：AUTO

- 在函数模板定义时，如果变量的类型依赖于模板参数，使用auto关键字使得在编译期确定这些类型，如：

```
template <class T, class U>void Multiply(T t, U u)
{
    auto v = t * u;
    std::cout<<v;
}
```

- auto可以推导变量的类型，但必须有初始化（=的赋值），而decltype可以推导表达式的类型，得到无初始化的变量的类型

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

其他编程规范

- 使用 `.hpp` 和 `.cpp`
 - 广泛的被各种编辑器和工具所识别
 - Vim并不能识别 `.cc` 文件
- 编程时不要混合使用 `Tab` 和 `space`
 - 不同编辑器对两者的处理不同，可能造成代码“混乱”
- 不要在 `assert()` 中放置实际运算代码
 - Release编译下，`assert`不可见

```
assert(registerSomething()); // make sure that registerSomething() returns true
```

`assert`的作用是先计算表达式 `expression`，如果其值为假（即为0），那么它先向 `stderr` 打印一条出错信息，然后通过调用 `abort` 来终止程序运行



其他编程规范

- 运算符重载要小心
 - 只有操作的意义非常清晰且毫无争议，才考虑进行运算符重载；否则，定义一个函数，并起个好名字
 - 重载时，一定要完全符合被重载运算符的语意信息
 - 用减法操作重载“+”
 - 重载一个运算符之后，要将所有其它相关的运算符也进行重载
 - + 和 +=



其他编程规范

- 避免隐式的数据转化

- 将单参数的构造函数和数据转化操作符定义为**explicit**

```
class MyClass
{
    public:
        MyClass( int num );
}
//.
MyClass obj = 10; //ok
```

```
class MyClass
{
    public:
        explicit MyClass( int num );
}
//.
MyClass obj = 10; //err
```

除非我有一个好理由允许构造函数被用于隐式类型转换，否则我会把它声明为**explicit**。

——**effective C++**

