[**const的用法，特别是用在函数前面与后面的区别！**](https://www.cnblogs.com/azbane/p/7266747.html)

const的用法，特别是用在函数后面

在普通的非 const成员函数中，this的类型是一个指向类类型的 const指针。可以改变this所指向的值，但不能改变 this所保存的地址。

在 const成员函数中，this的类型是一个指向 const类类型对象的 const指针。既不能改变 this所指向的对象，也不能改变 this所保存的地址。

关键字：Const，Const函数，Const变量，函数后面的Const

看到const关键字，C++程序员首先想到的可能是const常量。这可不是良好的条件反射。如果只知道用const定义常量，那么相当于把火药仅用于制作鞭炮。const更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。

const 是constant的缩写，“恒定不变”的意思。被const修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。所以很多C++程序设计书籍建议：“Useconst whenever you need”。

     1.用const修饰函数的参数

如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加const修饰，否则该参数将失去输出功能。const只能修饰输入参数：

如果输入参数采用“指针传递”，那么加const修饰可以防止意外地改动该指针，起到保护作用。

例如StringCopy函数：

void StringCopy(char\*strDestination, const char \*strSource);

其中strSource是输入参数，strDestination是输出参数。给strSource加上const修饰后，如果函数体内的语句试图改动strSource的内容，编译器将指出错误。

如果输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加const修饰。

例如不要将函数voidFunc1(int x) 写成voidFunc1(const int x)。同理不要将函数voidFunc2(A a) 写成voidFunc2(const A a)。其中A为用户自定义的数据类型。

对于非内部数据类型的参数而言，象voidFunc(A a) 这样声明的函数注定效率比较底。因为函数体内将产生A类型的临时对象用于复制参数a，而临时对象的构造、复制、析构过程都将消耗时间。

为了提高效率，可以将函数声明改为voidFunc(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临时对象。但是函数voidFunc(A &a) 存在一个缺点：

“引用传递”有可能改变参数a，这是我们不期望的。解决这个问题很容易，加const修饰即可，因此函数最终成为voidFunc(const A &a)。

以此类推，是否应将voidFunc(int x) 改写为voidFunc(const int&x)，以便提高效率？完全没有必要，因为内部数据类型的参数不存在构造、析构的过程，而复制也非常快，“值传递”和“引用传递”的效率几乎相当。

问题是如此的缠绵，我只好将“const&”修饰输入参数的用法总结一下。

对于非内部数据类型的输入参数，应该将“值传递”的方式改为“const引用传递”，目的是提高效率。例如将voidFunc(A a) 改为voidFunc(const A &a)。

对于内部数据类型的输入参数，不要将“值传递”的方式改为“const引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如voidFunc(int x) 不应该改为voidFunc(const int &x)。

     2用const修饰函数的返回值

如果给以“指针传递”方式的函数返回值加const修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加const修饰的同类型指针。例如函数

constchar \* GetString(void);

如下语句将出现编译错误：

char\*str = GetString();

正确的用法是

constchar \*str =GetString();

如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加const修饰没有任何价值。

例如不要把函数intGetInt(void) 写成constint GetInt(void)。

同理不要把函数AGetA(void) 写成constA GetA(void)，其中A为用户自定义的数据类型。

如果返回值不是内部数据类型，将函数AGetA(void) 改写为constA &GetA(void)的确能提高效率。但此时千万千万要小心，一定要搞清楚函数究竟是想返回一个对象的“拷贝”还是仅返回“别名”就可以了，否则程序会出错。

函数返回值采用“引用传递”的场合并不多，这种方式一般只出现在类的赋值函数中，目的是为了实现链式表达。

例如：

classA

{

A & operate = (const A &other); // 赋值函数

};

Aa, b, c; // a, b, c 为A的对象

a= b = c; // 正常的链式赋值

(a= b) = c; // 不正常的链式赋值，但合法

**如果将赋值函数的返回值加const修饰，那么该返回值的内容不允许被改动。上例中，语句a= b = c 仍然正确，但是语句(a= b) = c 则是非法的。**

3const 成员函数

任何不会修改数据成员的函数都应该声明为const类型。如果在编写const成员函数时，不慎修改了数据成员，或者调用了其它非const成员函数，编译器将指出错误，这无疑会提高程序的健壮性。以下程序中，类stack的成员函数GetCount仅用于计数，从逻辑上讲GetCount应当为const函数。编译器将指出GetCount函数中的错误。

classStack

{

public:

void Push(int elem);

int Pop(void);

intGetCount(void) const; // const 成员函数

private:

intm\_num;

int m\_data[100];

};

int Stack::GetCount(void)const

{

++ m\_num; // 编译错误，企图修改数据成员m\_num

Pop();// 编译错误，企图调用非const函数

returnm\_num;

}

const 成员函数的声明看起来怪怪的：const关键字只能放在函数声明的尾部，大概是因为其它地方都已经被占用了。

关于Const函数的几点规则：

a.const对象只能访问const成员函数,而非const对象可以访问任意的成员函数,包括const成员函数.

b.const对象的成员是不可修改的,然而const对象通过指针维护的对象却是可以修改的.

c.const成员函数不可以修改对象的数据,不管对象是否具有const性质.它在编译时,以是否修改成员数据为依据,进行检查.

e.然而加上mutable修饰符的数据成员,对于任何情况下通过任何手段都可修改,自然此时的const成员函数是可以修改它的

补充：

标题:const放在后面有什么意思？

--------------------------------------------------------------------------------

一个函数

AcGePoint3dstartPoint() const;

const放在后面跟前面有区别么

==>

准确的说const是修饰this指向的对象的

譬如，我们定义了

classA{

public:

f(int);

};

这里f函数其实有两个参数，第一个是A\*const this, 另一个才是int类型的参数

如果我们不想f函数改变参数的值，可以把函数原型改为f(constint),但如果我们不允许f改变this指向的对象呢？因为this是隐含参数，const没法直接修饰它，就加在函数的后面了，表示this的类型是constA \*constthis。

const修饰\*this是本质，至于说“表示该成员函数不会修改类的数据。否则会编译报错”之类的说法只是一个现象，根源就是因为\*this是const类型的