<http://blog.csdn.net/left_la/article/details/7441159>

## ****1、什么是const?****

常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。（当然，我们可以偷梁换柱进行更新：）

## ****2、为什么引入const?****

const 推出的初始目的，正是为了取代预编译指令，消除它的缺点，同时继承它的优点。

## ****3、cons有什么主要的作用?****

（1）可以定义const常量，具有不可变性。 例如：  const int Max=100; int Array[Max];

（2）便于进行类型检查，使编译器对处理内容有更多了解，消除了一些隐患。

例如： void f(const int i) { .........} 编译器就会知道i是一个常量，不允许修改。

（3）可以避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改。 同宏定义一样，可以做到不变则已，一变都变！如（1）中，如果想修改Max的内容，只需要：const int Max=you want;即可！

 （4）可以保护被修饰的东西，防止意外的修改，增强程序的健壮性。 还是上面的例子，如果在函数体内修改了i，编译器就会报错； 例如：

void f(const int i) { i=10;//error! }

（5） 为函数重载提供了一个参考。

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A { ......
2. **void** f(**int** i) {......} //一个函数
3. **void** f(**int** i) **const** {......} //上一个函数的重载 ......
4. };

（6） 可以节省空间，避免不必要的内存分配。 例如：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. #define PI 3.14159 //常量宏
2. **const** doulbe Pi=3.14159; //此时并未将Pi放入ROM中 ......
3. **double** i=Pi; //此时为Pi分配内存，以后不再分配！
4. **double** I=PI; //编译期间进行宏替换，分配内存
5. **double** j=Pi; //没有内存分配
6. **double** J=PI; //再进行宏替换，又一次分配内存！

const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝。

（7） 提高了效率。 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

## ****4、如何使用const?****

（1）修饰一般常量 一般常量是指简单类型的常量。这种常量在定义时，修饰符const可以用在类型说明符前，也可以用在类型说明符后。

例如： int const x=2; 或 const int x=2;

（2）修饰常数组 定义或说明一个常数组可采用如下格式：

int const a[5]={1, 2, 3, 4, 5};

const int a[5]={1, 2, 3, 4, 5};

（3）修饰常对象 常对象是指对象常量，定义格式如下：

class A;

const A a;

A const a;

定义常对象时，同样要进行初始化，并且该对象不能再被更新，修饰符const可以放在类名后面，也可以放在类名前面。

（4）修饰常指针

const int \*A; //const修饰指向的对象，A可变，A指向的对象不可变

int const \*A; //const修饰指向的对象，A可变，A指向的对象不可变

int \*const A; //const修饰指针A， A不可变，A指向的对象可变

const int \*const A;//指针A和A指向的对象都不可变

（5）修饰常引用 使用const修饰符也可以说明引用，被说明的引用为常引用，该引用所引用的对象不能被更新。

其定义格式如下：

const double & v;

（6）修饰函数的常参数 const修饰符也可以修饰函数的传递参数，格式如下：

void Fun(const int Var); 告诉编译器Var在函数体中的无法改变，从而防止了使用者的一些无意的或错误的修改。

（7）修饰函数的返回值： const修饰符也可以修饰函数的返回值，是返回值不可被改变，格式如下：

const int Fun1(); const MyClass Fun2();

（8）修饰类的成员函数： const修饰符也可以修饰类的成员函数，格式如下：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** ClassName {
2. **public**:
3. **int** Fun() **const**; .....
4. };

这样，在调用函数Fun时就不能修改类里面的数据

（9）在另一连接文件中引用const常量

extern const int i;//正确的引用

extern const int j=10;//错误！常量不可以被再次赋值 另外，还要注意，常量必须初始化！ 例如： const int i=5;

## ****5、几点值得讨论的地方****

### 1、const究竟意味着什么？

说了这么多，你认为const意味着什么？一种修饰符？接口抽象？一种新类型？ 也许都是，在Stroustup最初引入这个关键字时，只是为对象放入ROM做出了一种可能，对于const对象，C++既允许对其进行静态初始化，也允许对他进行动态初始化。理想的const对象应该在其构造函数完成之前都是可写的，在析够函数执行开始后也都是可写的，换句话说，const对象具有从构造函数完成到析够函数执行之前的不变性，如果违反了这条规则，结果都是未定义的！虽然我们把const放入ROM中，但这并不能够保证const的任何形式的堕落，我们后面会给出具体的办法。无论const对象被放入ROM中，还是通过存储保护机制加以保护，都只能保证，对于用户而言这个对象没有改变。换句话说，废料收集器（我们以后会详细讨论，这就一笔带过）或数据库系统对一个const的修改怎没有任何问题。

### 2、位元const V.S. 抽象const?

对于关键字const的解释有好几种方式，最常见的就是位元const 和 抽象const。下面我们看一个例子： class A { public: ...... A f(const A& a); ...... }; 如果采用抽象const进行解释，那就是f函数不会去改变所引用对象的抽象值，如果采用位元const进行解释，那就成了f函数不会去改变所引用对象的任何位元。 我们可以看到位元解释正是c++对const问题的定义，const成员函数不被允许修改它所在对象的任何一个数据成员。 为什么这样呢？因为使用位元const有2个好处： 最大的好处是可以很容易地检测到违反位元const规定的事件：编译器只用去寻找有没有对数据成员的赋值就可以了。另外，如果我们采用了位元const，那么，对于一些比较简单的const对象，我们就可以把它安全的放入ROM中，对于一些程序而言，这无疑是一个很重要的优化方式。（关于优化处理，我们到时候专门进行讨论） 当然，位元const也有缺点，要不然，抽象const也就没有产生的必要了。 首先，位元const的抽象性比抽象const的级别更低！实际上，大家都知道，一个库接口的抽象性级别越低，使用这个库就越困难。 其次，使用位元const的库接口会暴露库的一些实现细节，而这往往会带来一些负面效应。所以，在库接口和程序实现细节上，我们都应该采用抽象const。 有时，我们可能希望对const做出一些其它的解释，那么，就要注意了，目前，大多数对const的解释都是类型不安全的，这里我们就不举例子了，你可以自己考虑一下，总之，我们尽量避免对const的重新解释。

### 3、放在类内部的常量有什么限制？

看看下面这个例子：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **private**:
3. **const** **int** c3 = 7; // ???
4. **static** **int** c4 = 7; // ???
5. **static** **const** **float** c5 = 7; // ??? ......
6. };

你认为上面的3句对吗？呵呵，都不对！使用这种类内部的初始化语法的时候，常量必须是被一个常量表达式初始化的整型或枚举类型，而且必须是static和const形式。这显然是一个很严重的限制！ 那么，我们的标准委员会为什么做这样的规定呢？一般来说，类在一个头文件中被声明，而头文件被包含到许多互相调用的单元去。但是，为了避免复杂的编译器规则，C++要求每一个对象只有一个单独的定义。如果C++允许在类内部定义一个和对象一样占据内存的实体的话，这种规则就被破坏了。

### 4、如何初始化类内部的常量？

一种方法就是static 和 const 并用，在内部初始化，如上面的例子； 另一个很常见的方法就是初始化列表：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A(**int** i=0):test(i) {}
4. **private**:
5. **const** **int** i;
6. };

还有一种方式就是在外部初始化，例如：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A() {}
4. **private**:
5. **static** **const** **int** i;//注意必须是静态的！
6. };
7. **const** **int** A::i=3;

### 5、常量与数组的组合有什么特殊吗？

我们给出下面的代码： const int size[3]={10,20,50}; int array[size[2]]; 有什么问题吗？对了，编译通不过！为什么呢？ Const可以用于集合，但编译器不能把一个集合存放在它的符号表里，所以必须分配内存。在这种情况下，const意味着“不能改变的一块存储”。然而，其值在编译时不能被使用，因为编译器在编译时不需要知道存储的内容。自然，作为数组的大小就不行了：） 你再看看下面的例子：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A(**int** i=0):test[2]({1,2}) {}//你认为行吗？
4. **private**:
5. **const** **int** test[2];
6. };

vc6下编译通不过，为什么呢？ 关于这个问题，前些时间，njboy问我是怎么回事？我反问他：“你认为呢？”他想了想，给出了一下解释，大家可以看看：我们知道编译器堆初始化列表的操作是在构造函数之内，显式调用可用代码之前，初始化的次序依据数据声明的次序。初始化时机应该没有什么问题，那么就只有是编译器对数组做了什么手脚！其实做什么手脚，我也不知道，我只好对他进行猜测：编译器搜索到test发现是一个非静态的数组，于是，为他分配内存空间，这里需要注意了，它应该是一下分配完，并非先分配test[0],然后利用初始化列表初始化，再分配test[1],这就导致数组的初始化实际上是赋值！然而，常量不允许赋值，所以无法通过。 呵呵，看了这一段冠冕堂皇的话，真让我笑死了！njboy别怪我揭你短呀：）我对此的解释是这样的：C++标准有一个规定，不允许无序对象在类内部初始化，数组显然是一个无序的，所以这样的初始化是错误的！对于他，只能在类的外部进行初始化，如果想让它通过，只需要声明为静态的，然后初始化。 这里我们看到，常量与数组的组合没有什么特殊！一切都是数组惹的祸！

### 6、this指针是不是const类型的？

this指针是一个很重要的概念，那该如何理解她呢？也许这个话题太大了，那我们缩小一些：this指针是个什么类型的？这要看具体情况：如果在非const成员函数中，this指针只是一个类类型的；如果在const成员函数中，this指针是一个const类类型的；如果在volatile成员函数中,this指针就是一个volatile类类型的。

### 7、const到底是不是一个重载的参考对象？

先看一下下面的例子：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. ......
3. **void** f(**int** i) {......}//一个函数
4. **void** f(**int** i) **const** {......}//上一个函数的重载
5. ......
6. };

上面是重载是没有问题的了，那么下面的呢？

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. ......
3. **void** f(**int** i) {......}//一个函数
4. **void** f(**const** **int** i) {......}//？？？？？
5. ......
6. };

这个是错误的，编译通不过。那么是不是说明内部参数的const不予重载呢？再看下面的例子：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. ......
3. **void** f(**int**& ) {......}//一个函数
4. **void** f(**const** **int**& ) {......}//？？？？？
5. ......
6. };

这个程序是正确的，看来上面的结论是错误的。为什么会这样呢？这要涉及到接口的透明度问题。按值传递时，对用户而言，这是透明的，用户不知道函数对形参做了什么手脚，在这种情况下进行重载是没有意义的，所以规定不能重载！当指针或引用被引入时，用户就会对函数的操作有了一定的了解，不再是透明的了，这时重载是有意义的，所以规定可以重载。

### 8、什么情况下为const分配内存？

以下是我想到的可能情况，当然，有的编译器进行了优化，可能不分配内存。  A、作为非静态的类成员时；  B、用于集合时；  C、被取地址时；  D、在main函数体内部通过函数来获得值时；  E、const的 class或struct有用户定义的构造函数、析构函数或基类时；。  F、当const的长度比计算机字长还长时；  G、参数中的const；  H、使用了extern时。 不知道还有没有其他情况，欢迎高手指点：）

### 9、临时变量到底是不是常量？

很多情况下，编译器必须建立临时对象。像其他任何对象一样，它们需要存储空间而且必须被构造和删除。区别是我们从来看不到编译器负责决定它们的去留以及它们存在的细节。对于C++标准草案而言：临时对象自动地成为常量。因为我们通常接触不到临时对象，不能使用与之相关的信息，所以告诉临时对象做一些改变有可能会出错。当然，这与编译器有关，例如：vc6、vc7都对此作了扩展，所以，用临时对象做左值，编译器并没有报错。

### 10、与static搭配会不会有问题？

假设有一个类：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. ......
4. **static** **void** f() **const** { ......}
5. ......
6. };

我们发现编译器会报错，因为在这种情况下static不能够与const共存！ 为什么呢？因为static没有this指针，但是const修饰this指针，所以...

### 11、如何修改常量？

有时候我们却不得不对类内的数据进行修改，但是我们的接口却被声明了const，那该怎么处理呢？我对这个问题的看法如下： 1）标准用法：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **mutable** **class** A {
2. **public**:
3. A(**int** i=0):test(i) { }
4. **void** SetValue(**int** i)**const** { test=i; }
5. **private**:
6. **mutable** **int** test;//这里处理！
7. };

2）强制转换：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **const\_cast** **class** A {
2. **public**:
3. A(**int** i=0):test(i) { }
4. **void** SetValue(**int** i)**const** {
5. **const\_cast** (test)=i;
6. }//这里处理！
7. **private**:
8. **int** test;
9. };

3）灵活的指针：

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **int**\* **class** A {
2. **public**:
3. A(**int** i=0):test(i) { }
4. **void** SetValue(**int** i)**const** { \*test=i; }
5. **private**:
6. **int**\* test; //这里处理！
7. };

4）未定义的处理

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A(**int** i=0):test(i) { }
4. **void** SetValue(**int** i)**const** {
5. <span style="white-space:pre;"> </span>**int** \*p=(**int**\*)&test; \*p=i;
6. }//这里处理！
7. **private**:
8. **int** test;
9. };

注意，这里虽然说可以这样修改，但结果是未定义的，避免使用！

5）内部处理：this指针

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A(**int** i=0):test(i) { }
4. **void** SetValue(**int** i)**const** {
5. ((A\*)**this**)->test=i;
6. }//这里处理！
7. **private**:
8. **int** test;
9. };

6）最另类的处理：空间布局

**[cpp]** [view plain](http://blog.csdn.net/left_la/article/details/7441159) [copy](http://blog.csdn.net/left_la/article/details/7441159)

1. **class** A {
2. **public**:
3. A(**int** i=0):test(i),c('a') { }
4. **private**:
5. **char** c;
6. **const** **int** test;
7. };
8. **int** main()
9. {
10. A a(3);
11. A\* pa=&a;
12. **char**\* p=(**char**\*)pa;
13. **int**\* pi=(**int**\*)(p+4）；//利用边缘调整
14. \*pi=5;//此处改变了test的值！
15. **return** 0;
16. }

虽然我给出了6中方法，但是我只是想说明如何更改，但出了第一种用法之外，另外5种用法，我们并不提倡，不要因为我这么写了，你就这么用，否则，我真是要误人子弟了：） （12）最后我们来讨论一下常量对象的动态创建。 既然编译器可以动态初始化常量，就自然可以动态创建，例如：  const int\* pi=new const int(10);

这里要注意2点：

1）const对象必须被初始化！所以(10)是不能够少的。

2）new返回的指针必须是const类型的。 那么我们可不可以动态创建一个数组呢？ 答案是否定的，因为new内置类型的数组，不能被初始化。 这里我们忽视了数组是类类型的，同样对于类内部数组初始化我们也做出了这样的忽视，因为这涉及到数组的问题，我们以后再讨论。