公告

昵称：chao_yu
园龄：4年11个月
粉丝：197
关注：0
+加关注

< 2010年7月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 27 | 28 | 29 | 30 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

C(12)
C++(22)
Javascript(4)
Linux开发(12)
Linux内核
QT(2)
嵌入式(1)
设计模式与数据结构(12)
网络(2)
杂类(8)

# C/C++中const关键字详解

为什么使用const？采用符号常量写出的代码更容易维护；指针常常是边读边移动，而不是边写边移动；许多函数参数是只读不写的。const最常见用途是作为数组的界和switch分情况标号(也可以用枚举符代替)，分类如下：

常变量： **const** 类型说明符 变量名

常引用： **const** 类型说明符 **&**引用名

常对象： 类名 **const** 对象名

常成员函数： 类名**::fun(**形参**) const**

常数组： 类型说明符 **const** 数组名**[**大小**]**

常指针： **const** 类型说明符***** 指针名 ， 类型说明符*** const** 指针名

首先提示的是：在常变量（**const** 类型说明符 变量名）、常引用（**const** 类型说明符 **&**引用名）、常对象（类名 **const** 对象名）、 常数组（类型说明符 **const** 数组名**[**大小**]**）， **const"** 与 **"**类型说明符**"**或**"**类名**"**（其实类名是一种自定义的类型说明符） 的位置可以互换。如：

const int a=5; 与 int const a=5; 等同

类名 const 对象名 与 const 类名 对象名 等同

用法**1**：常量

取代了C中的宏定义，声明时必须进行初始化(!c++类中则不然）。const限制了常量的使用方式，并没有描述常量应该如何分配。如果编译器知道了某const的所有使用，它甚至可以不为该const分配空间。最简单的常见情况就是常量的值在编译时已知，而且不需要分配存储。—《C++ Program Language》

用const声明的变量虽然增加了分配空间，但是可以保证类型安全。

C标准中，const定义的常量是全局的，C++中视声明位置而定。

用法**2**：指针和常量

使用指针时涉及到两个对象：该指针本身和被它所指的对象。将一个指针的声明用const"预先固定"将使那个对象而不是使这个指针成为常量。要将指针本身而不是被指对象声明为常量，必须使用声明运算符*const。

所以出现在 * 之前的const是作为基础类型的一部分：

```
char *const cp; //到char的const指针
char const *pc1; //到const char的指针
const char *pc2; //到const char的指针（后两个声明是等同的）
```

从右向左读的记忆方式：

cp is a const pointer to char. 故pc不能指向别的字符串，但可以修改其指向的字符串的内容

pc2 is a pointer to const char. 故*pc2的内容不可以改变，但pc2可以指向别的字符串

且注意：允许把非 const 对象的地址赋给指向 const 对象的指针,不允许把一个 const 对象的地址赋给一个普通的、非 const 对象的指针。

用法**3**：**const**修饰函数传入参数

将函数传入参数声明为const，以指明使用这种参数仅仅是为了效率的原因，而不是想让调用函数能够修改对象的值。同理，将指针参数声明为const，函数将不修改由这个参数所指的对象。

通常修饰指针参数和引用参数：

void Fun( const A *in); //修饰指针型传入参数

void Fun(const A &in); //修饰引用型传入参数

用法**4**：修饰函数返回值

可以阻止用户修改返回值。返回值也要相应的付给一个常量或常指针。

用法**5**：**const**修饰成员函数(c++特性)

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；

const对象的成员是不能修改的，而通过指针维护的对象确实可以修改的；

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。编译时以是否修改成员数据为依据进行检查。

具体展开来讲：

**(一).** 常量与指针

常量与指针放在一起很容易让人迷糊。对于常量指针和指针常量也不是所有的学习C/C++的人都能说清除。例如：

```
const int *m1 = new int(10);

int* const m2 = new int(20);
```

在上面的两个表达式中，最容易让人迷惑的是const到底是修饰指针还是指针指向的内存区域？其实，只要知道：const只对它左边的东西起作用，唯一的例外就是const本身就是最左边的修饰符，那么它才会对右边的东西起作用。根据这个规则来判断，m1应该是常量指针（即，不能通过m1来修改它所指向的内容。）；而m2应该是指针常量（即，不能让m2指向其他的内存模块）。由此可见：

1. 对于常量指针，不能通过该指针来改变所指的内容。即，下面的操作是错误的：

```
int i = 10;

const int *pi = &i;

*pi = 100;
```

因为你在试图通过pi改变它所指向的内容。但是，并不是说该内存块中的内容不能被修改。我们仍然可以通过其他方式去修改其中的值。例如：

```
// 1: 通过i直接修改。

i = 100;

// 2:  使用另外一个指针来修改。

int *p = (int*)pi;

*p = 100;
```

实际上，在将程序载入内存的时候，会有专门的一块内存区域来存放常量。但是，上面的i本身不是常量，是存放在栈或者堆中的。我们仍然可以修改它的值。而pi不能修改指向的值应该说是编译器的一个限制。

2. 根据上面const的规则，const int *m1 = new int(10);我们也可写作：

```
int const   *m1 = new int(10);
```

这是，理由就不须作过多说明了。

3. 在函数参数中指针常量时表示不允许将该指针指向其他内容。

```
void func_02(int* const p)
```

```
{
int *pi = new int(100);
//错误！P是指针常量。不能对它赋值。
p = pi;
}
int main()
{
int* p = new int(10);
func_02(p);
delete p;
return 0;
}
```

4. 在函数参数中使用常量指针时表示在函数中不能改变指针所指向的内容。

```
void func(const int *pi)
{
//错误！不能通过pi去改变pi所指向的内容！
*pi = 100;
}
int main()
{
int* p = new int(10);
func(p);
delete p;
return 0;
}
```

我们可以使用这样的方法来防止函数调用者改变参数的值。但是，这样的限制是有限的，作为参数调用者，我们也不要试图去改变参数中的值。因此，下面的操作是在语法上是正确的，但是可能破还参数的值：

```
#include <iostream>
#include <string>
void func(const int *pi)
{
//这里相当于重新构建了一个指针，指向相同的内存区域。当然就可以通过该指针修改内存中的值了。
int* pp = (int*)pi;
```

```
*pp = 100;
}
int main()
{
using namespace std;
int* p = new int(10);
cout << "*p = " << *p << endl;
func(p);
cout << "*p = " << *p << endl;
delete p;
return 0;
}
```

**(二)**：常量与引用

常量与引用的关系稍微简单一点。因为引用就是另一个变量的别名，它本身就是一个常量。也就是说不能再让一个引用成为另外一个变量的别名，那么他们只剩下代表的内存区域是否可变。即：

```
int i = 10;
// 正确：表示不能通过该引用去修改对应的内存的内容。
const int& ri = i;
// 错误！不能这样写。
int& const rci = i;
```

由此可见，如果我们不希望函数的调用者改变参数的值。最可靠的方法应该是使用引用。下面的操作会存在编译错误：

```
void func(const int& i)
{
// 错误！不能通过i去改变它所代表的内存区域。
i = 100;
}
int main()
{
int i = 10;
func(i);
return 0;
}
```

这里已经明白了常量与指针以及常量与引用的关系。但是，有必要深入的说明以下。在系统加载程序的时候，系统会将内存分为4个区域：堆区 栈区全局区（静态）和代

码区。从这里可以看出，对于常量来说，系统没有划定专门的区域来保护其中的数据不能被更改。也就是说，使用常量的方式对数据进行保护是通过编译器作语法限制来实现的。我们仍然可以绕过编译器的限制去修改被定义为“常量”的内存区域。看下面的代码：

```
const int i = 10;

// 这里i已经被定义为常量，但是我们仍然可以通过另外的方式去修改它的值。

// 这说明把i定义为常量，实际上是防止通过i去修改所代表的内存。

int *pi = (int*) &i;
```

**(三)**：常量函数

常量函数是C++对常量的一个扩展，它很好的确保了C++中类的封装性。在C++中，为了防止类的数据成员被非法访问，将类的成员函数分成了两类，一类是常量成员函数（也被称为观察着）；另一类是非常量成员函数（也被成为变异者）。在一个函数的签名后面加上关键字const后该函数就成了常量函数。对于常量函数，最关键的不同是编译器不允许其修改类的数据成员。例如：

```
class Test

{

public:

void func() const;

private:

int intValue;

};

void Test::func() const

{

intValue = 100;

}
```

上面的代码中，常量函数func函数内试图去改变数据成员intValue的值，因此将在编译的时候引发异常。

当然，对于非常量的成员函数，我们可以根据需要读取或修改数据成员的值。但是，这要依赖调用函数的对象是否是常量。通常，如果我们把一个类定义为常量，我们的本意是希望他的状态（数据成员）不会被改变。那么，如果一个常量的对象调用它的非常量函数会产生什么后果呢？看下面的代码：

```
class Fred{

public:

void inspect() const;

void mutate();

};

void UserCode(Fred& changeable, const Fred& unChangeable)

{

changeable.inspect(); // 正确，非常量对象可以调用常量函数。

changeable.mutate(); // 正确，非常量对象也允许修改调用非常量成员函数修改数据成员。

unChangeable.inspect(); // 正确，常量对象只能调用常理函数。因为不希望修改对象状态。
```

unChangeable.mutate(); // 错误！常量对象的状态不能被修改，而非常量函数存在修改对象状态的可能

}

从上面的代码可以看出，由于常量对象的状态不允许被修改，因此，通过常量对象调用非常量函数时将会产生语法错误。实际上，我们知道每个成员函数都有一个隐含的指向对象本身的this指针。而常量函数则包含一个this的常量指针。如下：

void inspect(const Fred* this) const;

void mutate(Fred* this);

也就是说对于常量函数，我们不能通过this指针去修改对象对应的内存块。但是，在上面我们已经知道，这仅仅是编译器的限制，我们仍然可以绕过编译器的限制，去改变对象的状态。看下面的代码：

class Fred{

public:

void inspect() const;
private:

int intValue;

};

void Fred::inspect() const

{

cout << "At the beginning. intValue = "<< intValue << endl;

// 这里，我们根据this指针重新定义了一个指向同一块内存地址的指针。

// 通过这个新定义的指针，我们仍然可以修改对象的状态。

Fred* pFred = (Fred*)this;

pFred->intValue = 50;

cout << "Fred::inspect() called. intValue = "<< intValue << endl;

}

int main()

{

Fred fred;

fred.inspect();

return 0;

}

上面的代码说明，只要我们愿意，我们还是可以通过常量函数修改对象的状态。同理，对于常量对象，我们也可以构造另外一个指向同一块内存的指针去修改它的状态。这里就不作过多描述了。

另外，也有这样的情况，虽然我们可以绕过编译器的错误去修改类的数据成员。但是C++也允许我们在数据成员的定义前面加上mutable，以允许该成员可以在常量函数中被修改。例如：

class Fred{

```
public:
void inspect() const;
private:
mutable int intValue;
};
void Fred::inspect() const
{
intValue = 100;
}
```

但是，并不是所有的编译器都支持mutable关键字。这个时候我们上面的歪门邪道就有用了。

关于常量函数，还有一个问题是重载。

```
#include <iostream>
#include <string>
using namespace std;
class Fred{
public:
void func() const;
void func();
};
void Fred::func() const
{
cout << "const function is called."<< endl;
}
void Fred::func()
{
cout << "non-const function is called."<< endl;
}
void UserCode(Fred& fred, const Fred& cFred)
{
cout << "fred is non-const object, and the result of fred.func() is:" << endl;
fred.func();
cout << "cFred is const object, and the result of cFred.func() is:" << endl;
cFred.func();
```

```
}
int main()
{
Fred fred;
UserCode(fred, fred);
return 0;
}
```

输出结果为：

```
fred is non-const object, and the result of fred.func() is:
non-const function is called.
cFred is const object, and the result of cFred.func() is:
const function is called.
```

从上面的输出结果，我们可以看出。当存在同名同参数和返回值的常量函数和非常量函数时，具体调用哪个函数是根据调用对象是常量对像还是非常量对象来决定的。常量对象调用常量成员；非常量对象调用非常量的成员。

总之，我们需要明白常量函数是为了最大程度的保证对象的安全。通过使用常量函数，我们可以只允许必要的操作去改变对象的状态，从而防止误操作对对象状态的破坏。但是，就像上面看见的一样，这样的保护其实是有限的。关键还是在于我们开发人员要严格的遵守使用规则。另外需要注意的是常量对象不允许调用非常量的函数。这样的规定虽然很武断，但如果我们都根据原则去编写或使用类的话这样的规定也就完全可以理解了。

**(**四**)**：常量返回值

很多时候，我们的函数中会返回一个地址或者引用。调用这得到这个返回的地址或者引用后就可以修改所指向或者代表的对象。这个时候如果我们不希望这个函数的调用这修改这个返回的内容，就应该返回一个常量。这应该很好理解，大家可以去试试。

++++++++++++++++++++++++++++++++++++++++

**c++** 中**const**

++++++++++++++++++++++++++++++++++++++++

1. const常量，如const int max = 100;
优点：const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误（边际效应）
2. const 修饰类的数据成员。如：

```
class A
{
   const int size;
   …
}
```

const数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类声明中初始化const数据成员，因为类的对象未被创建时，编译器不知道const 数据成员的值是什么。如

```
class A
{
```

```
 const int size = 100;    //错误
 int array[size];        //错误，未知的size
}
```

const数据成员的初始化只能在类的构造函数的初始化表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现。如

```
class A
{…
 enum {size1=100, size2 = 200 };
int array1[size1];
int array2[size2];
}
```

枚举常量不会占用对象的存储空间，他们在编译时被全部求值。但是枚举常量的隐含数据类型是整数，其最大值有限，且不能表示浮点数。

3. const修饰指针的情况，见下式：

```
int b = 500;
const int* a = &              [1]
int const *a = &              [2]
int* const a = &              [3]
const int* const a = &      [4]
```

如果你能区分出上述四种情况，那么，恭喜你，你已经迈出了可喜的一步。不知道，也没关系，我们可以参考《Effectivec++》Item21上的做法，如果const位于星号的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于星号的右侧，const就是修饰指针本身，即指针本身是常量。因此，[1]和[2]的情况相同，都是指针所指向的内容为常量（const放在变量声明符的位置无关），这种情况下不允许对内容进行更改操作，如不能*a = 3；[3]为指针本身是常量，而指针所指向的内容不是常量，这种情况下不能对指针本身进行更改操作，如a++是错误的；[4]为指针本身和指向的内容均为常量。

4. const的初始化

先看一下const变量初始化的情况

1) 非指针const常量初始化的情况：A b;

```
const A a = b;
```

2) 指针const常量初始化的情况：

```
A* d = new A();
const A* c = d;
```

或者：`const A* c = new A();`

3）引用const常量初始化的情况：

```
A f;
const A& e = f;      // 这样作e只能访问声明为const的函数，而不能访问一般的成员函数；
```

[思考1]： 以下的这种赋值方法正确吗？

```
const A* c=new A();
A* e = c;
```

[思考2]： 以下的这种赋值方法正确吗？

```
A* const c = new A();
A* b = c;
```

5. 另外const 的一些强大的功能在于它在函数声明中的应用。在一个函数声明中，const可以修饰函数的返回值，或某个参数；对于成员函数，还可以修饰是整个函数。有如

下几种情况，以下会逐渐的说明用法：A&operator=(const A& a);
void fun0(const A* a );
void fun1( ) const; // fun1( ) 为类成员函数
const A fun2( );

1) 修饰参数的const，如 void fun0(const A* a ); void fun1(const A& a);
调用函数的时候，用相应的变量初始化const常量，则在函数体中，按照const所修饰的部分进行常量化，如形参为const A*a，则不能对传递进来的指针的内容进行改变，保护了原指针所指向的内容；如形参为const A&a，则不能对传递进来的引用对象进行改变，保护了原对象的属性。
[注意]：参数const通常用于参数为指针或引用的情况，且只能修饰输入参数;若输入参数采用“值传递”方式，由于函数将自动产生临时变量用于复制该参数，该参数本就不需要保护，所以不用const修饰。

[总结]对于非内部数据类型的输入参数，因该将“值传递”的方式改为“const引用传递”，目的是为了提高效率。例如，将void Func(A a)改为void Func(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void Func(int x)不应该改为void Func(const int &x)

2） 修饰返回值的const，如const A fun2( ); const A* fun3( );
这样声明了返回值后，const按照"修饰原则"进行修饰，起到相应的保护作用。const Rational operator*(const Rational& lhs, const Rational& rhs)
{
return Rational(lhs.numerator() * rhs.numerator(),
lhs.denominator() * rhs.denominator());
}

返回值用const修饰可以防止允许这样的操作发生:Rational a,b;
Radional c;
(a*b) = c;

一般用const修饰返回值为对象本身（非引用和指针）的情况多用于二目操作符重载函数并产生新对象的时候。
**[**总结**]**

1．一般情况下，函数的返回值为某个对象时，如果将其声明为const时，多用于操作符的重载。通常，不建议用const修饰函数的返回值类型为某个对象或对某个对象引用的情况。原因如下：如果返回值为某个对象为const（const A test = A实例）或某个对象的引用为const（const A& test = A实例），则返回值具有const属性，则返回实例只能访问类A中的公有（保护）数据成员和const成员函数，并且不允许对其进行赋值操作，这在一般情况下很少用到。

2．如果给采用“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。如：

const char * GetString(void);

如下语句将出现编译错误：

char *str=GetString();

正确的用法是：

const char *str=GetString();

3.　　函数返回值采用“引用传递”的场合不多，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。如：

class A

{…

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

}
A a,b,c;　　　　//a,b,c为A的对象

…

```
a=b=c;          //正常

(a=b)=c;        //不正常，但是合法
```

若负值函数的返回值加const修饰，那么该返回值的内容不允许修改，上例中a=b=c依然正确。(a=b)=c就不正确了。
[思考3]： 这样定义赋值操作符重载函数可以吗？

```
const A& operator=(const A& a);
```

6. 类成员函数中const的使用
一般放在函数体后，形如：void fun() const;
任何不会修改数据成员的函数都因该声明为const类型。如果在编写const成员函数时，不慎修改了数据成员，或者调用了其他非const成员函数，编译器将报错，这大大提高了程序的健壮性。如：

```
class Stack

{

 public:

     void Push(int elem);

     int Pop(void);

     int GetCount(void) const;   //const 成员函数

 private:

     int m_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

  Return m_num;

}
```

7. 使用const的一些建议

1) 要大胆的使用const，这将给你带来无尽的益处，但前提是你必须搞清楚原委；
2) 要避免最一般的赋值操作错误，如将const变量赋值，具体可见思考题；
3) 在参数中使用const应该使用引用或指针，而不是一般的对象实例，原因同上；
4) const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；
5) 不要轻易的将函数的返回值类型定为const;
6) 除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;

[思考题答案]
1) 这种方法不正确，因为声明指针的目的是为了对其指向的内容进行改变，而声明的指针e指向的是一个常量，所以不正确；
2) 这种方法正确，因为声明指针所指向的内容可变；
3) 这种做法不正确；
在const A::operator=(const A& a)中，参数列表中的const的用法正确，而当这样连续赋值的时侯，问题就出现了：

A a,b,c:
(a=b)=c;
因为a.operator=(b)的返回值是对a的const引用，不能再将c赋值给const常量。

++++++++++++++++++++++++++++++++++++++++

**const** 在**c**和**c++**中的区别 http://tech.e800.com.cn/articles/2009/722/1248229886744_1.html

++++++++++++++++++++++++++++++++++++++++

1. C++中的const正常情况下是看成编译期的常量,编译器并不为const分配空间,只是在编译的时候将期值保存在名字表中,并在适当的时候折合在代码中.所以,以下代码:

```
using namespace std;
int main()
{
const int a = 1;
const int b = 2;
int array[ a + b ] = {0};
for (int i = 0; i < sizeof array / sizeof *array; i++)
{
cout << array << endl;
}
}
```

在可以通过编译,并且正常运行.但稍加修改后,放在C编译器中,便会出现错误:

```
int main()
{
int i;
const int a = 1;
const int b = 2;
int array[ a + b ] = {0};
for (i = 0; i < sizeof array / sizeof *array; i++)
{
printf("%d",array);
}
}
```

错误消息:
c:\test1\te.c(8): error C2057: 应输入常数表达式
c:\test1\te.c(8): error C2466: 不能分配常数大小为 0 的数组
出现这种情况的原因是:在C中,const是一个不能被改变的普通变量,既然是变量,就要占用存储空间,所以编译器不知道编译时的值.而且,数组定义时的下标必须为常量.
2. 在C语言中: const int size; 这个语句是正确的，因为它被C编译器看作一个声明,指明在别的地方分配存储空间.但在C++中这样写是不正确的.C++中const默认是内部连接,如果想在C++中达到以上的效果,必须要用extern关键字.即**C++**中**,const**默认使用内部连接**.**而**C**中使用外部连接**.**
**(1)** 内连接:编译器只对正被编译的文件创建存储空间,别的文件可以使用相同的表示符或全局变量.C/C++中内连接使用static关键字指定.
**(2)** 外连接:所有被编译过的文件创建一片单独存储空间.一旦空间被创建,连接器必须解决对这片存储空间的引用.全局变量和函数使用外部连接.通过extern关键字声明,可以从其他文件访问相应的变量和函数.

```
/* C++代码  header.h */
const int test = 1;
/* C++代码  test1.cpp */
#include "header.h"
using namespace std;
int main() { cout << "in test1 :" << test << endl; }
/* C++代码 test2.cpp */
```

#include "header.h"
using namespace std;
void print() { cout << "in test2:" << test << endl;}
以上代码编译连接完全不会出问题,但如果把header.h改为:
extern const int test = 1;
在连接的时候,便会出现以下错误信息:
test2 error LNK2005: "int const test" (?test@@3HB) 已经在 test1.obj 中定义
　　因为extern关键字告诉C++编译器test会在其他地方引用,所以,C++编译器就会为test创建存储空间,不再是简单的存储在名字表里面.所以,当两个文件同时包含header.h的时候,会发生名字上的冲突.
此种情况和C中const含义相似:
/* C代码 header.h */
const int test = 1;
/* C代码 test1.c */
#include "header.h"
int main() { printf("in test1:%d\n",test); }
/* C代码 test2.c */
#include "header.h"
void print() { printf("in test2:%d\n",test); }
错误消息:
test3 fatal error LNK1169: 找到一个或多个多重定义的符号
test3 error LNK2005: _test 已经在 test1.obj 中定义

也就是说：在c++ 中const 对象默认为文件的局部变量。与其他变量不同，除非特别说明，在全局作用域声明的 const 变量是定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。通过指定 const 变更为 extern，就可以在整个程序中访问 const 对象：

```
// file_1.cc
// defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_2.cc
extern const int bufSize; // uses bufSize from file_1
// uses bufSize defined in file_1
for (int index = 0; index != bufSize; ++index)
    // ...
```

3. C++中,是否为const分配空间要看具体情况.如果加上关键字extern或者取const变量地址,则编译器就要为const分配存储空间.
4. C++中定义常量的时候不再采用define,因为define只做简单的宏替换，并不提供类型检查.

分类: C, C++



chao_yu
关注 - 0
粉丝 - 197
+加关注



« 上一篇：STL，ATL，WTL之间的联系和区别
» 下一篇：C/C++中extern关键字详解

posted on 2010-07-14 17:29 chao_yu 阅读(9365) 评论(5) 编辑 收藏

评论

不洗脸都帅 | 2011-10-19 10:57 #1楼

给力。。。

支持(0) 反对(0)

零风腾飞 | 2012-10-29 14:14 #2楼

赞，果断mark！

支持(0) 反对(0)

イケメンおっさん_汪汪 | 2013-04-27 14:47 #3楼

太好了·····

支持(0) 反对(0)

**laoshufeifei** | 2014-12-08 22:59 #4楼

太长了

支持(0) 反对(0)

塔尔西斯高原的寄居蟹 | 2014-12-29 14:43 #5楼

mark

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新**IT**新闻：

· 科学家发现轨道周期仅为2.4小时的双星系统
· 点点客定向增发 再次融资2.2亿元
· 百度大数据：国产手机小米第一
· 英特尔宣布收购Lantiq公司 补全互联家庭短板
· 盖茨抛微软股票套现4千万美元
» 更多新闻...



最新知识库文章：

· 什么是工程师文化？
· 大数据架构和模式（五）对大数据问题应用解决方案模式并选择实现它的产品
· 大数据架构和模式（四）了解用于大数据解决方案的原子模式和复合模式
· 大数据架构和模式（三）理解大数据解决方案的架构层
· 大数据架构和模式（二）如何知道一个大数据解决方案是否适合您的组织
» 更多知识库文章...