2020/05/13_C++_第7课_常成员函数、初始化列表、静态成员

笔记本: C++
创建时间: 2020/5/13 星期三 15:37
作者: ileemi
标签: 常成员函数, 初始化列表, 静态成员函数和静态数据成员

常成员函数

代码示例:

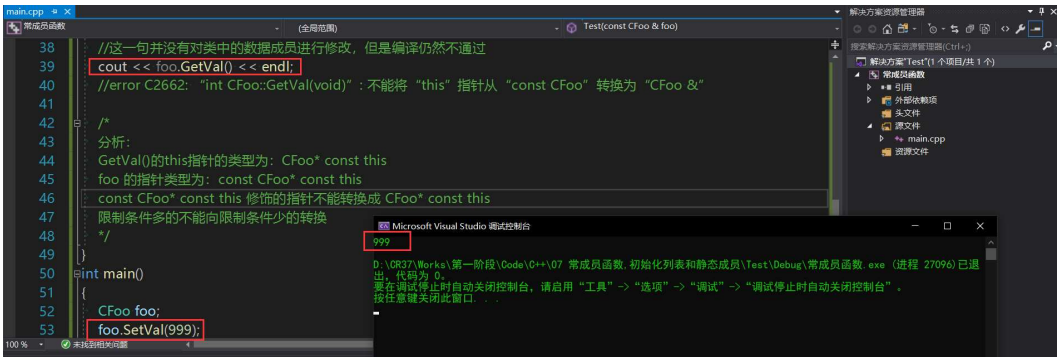
```
#include <iostream>
using namespace std;

class CFoo
{
public:
    int GetVal()
    {
            return m_nVal;
    }
    void SetVal(int nVal)
    {
            m_nVal = nVal;
    }
private:
    int m_nVal;
};

/*
使用const后，这里的形参foo不能被修改，
foo.SetVal(999); 对其进行了修改，编译报错

分析:
this 指针的类型：className* const this (CFoo* const this)
this 本身不能被修改，其指向的成员可以被修改
```

```
对象定义为：const CFoo& foo类型
当该对象在调用成员函数的时候，其this指针的类型为：const CFoo* const
this

*/
void Test(const CFoo& foo)
{
    //foo.SetVal(999);
    //error C2662: “void CFoo::SetVal(int)”: 不能将“this”指针
从“const CFoo”转换为“CFoo &”

    //这一句并没有对类中的数据成员进行修改，但是编译仍然不通过
    cout << foo.GetVal() << endl;
    //error C2662: “int CFoo::GetVal(void)”: 不能将“this”指针
从“const CFoo”转换为“CFoo &”

    /*
    分析：
    GetVal()的this指针的类型为：CFoo* const this
    foo 的指针类型为：const CFoo* const this
    const CFoo* const this 修饰的指针不能转换成 CFoo* const this
    限制条件多的不能向限制条件少的转换
    */
}

int main()
{
    CFoo foo;
    foo.SetVal(999);
    Test(foo);
    return 0;
}
```

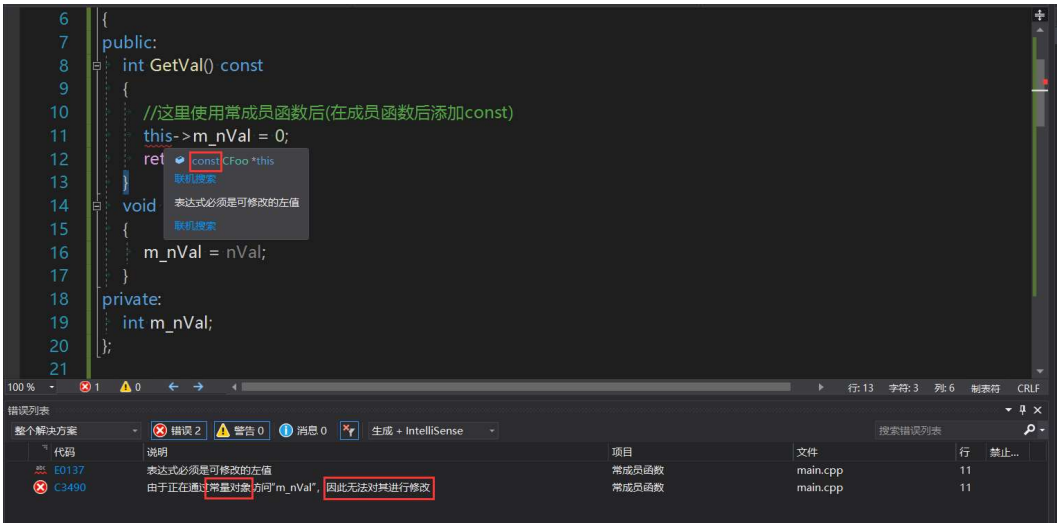
为了解决上面的问题C++新增了**常成员函数**
使用方法，在成员函数后面添加关键字 const
声明样式为：返回类型 <类标识符::>函数名称(参数表) const

```
public:
    int GetVal() const
    {
            return m_nVal;
    }
```

使用常成员函数后，上面的代码就可以正常运行，结果如下：



通过下面的测试可以看出在成员函数后添加const后，this指针类型进行了修改：



总结：

1、将this指针的类型修改为const type* const this

2、常成员函数内部不能修改其成员(语法上的限制，可访问地址对其进行修改), 不能调用非常成员函数

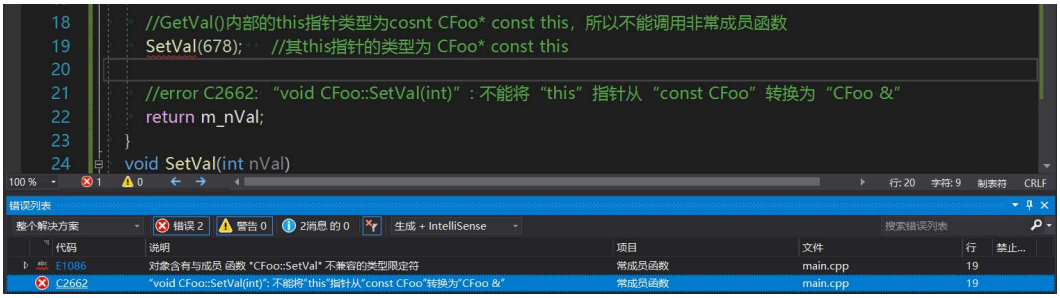
3、常成员函数是语法层限制可访问地址对其进行修改

```
//间接修改数据成员的值
*(int*)&this->m_nVal = 666;
CFoo* pTemp = (CFoo*)this;
pTemp->m_nVal = 777;
return m_nVal;
```

常成员函数内部不能调用非常成员函数：

```
int GetVal() const
{
	//这里使用常成员函数后(在成员函数后添加const)
	//this->m_nVal = 0;
	//间接修改数据成员的值
	*(int*)&this->m_nVal = 666;
	CFoo* pTemp = (CFoo*)this;
	pTemp->m_nVal = 777;
```

```
    //GetVal()内部的this指针类型为cosnt CFoo* const this，所以不能调用非常成员函数
    //SetVal(678);                    //其this指针的类型为 CFoo* const this

    //error C2662: “void CFoo::SetVal(int)”: 不能将“this”指针从“const CFoo”转换为“CFoo &”
    return m_nVal;
}
```



4、一般都用在get类的成员函数中

扩展：

1、const是函数声明的一部分，在函数的实现部分也需要加上const

2、const关键字可以重载函数名相同但是未加const关键字的函数

3、常成员函数不能用来更新类的成员变量，也不能调用类中未用const修饰的成员函数，只能调用常成员函数。即常成员函数不能更改类中的成员状态，这与const语义相符。

示例1：说明const可以重载函数，并且**实现部分也需要加const**

```
#include <iostream>
using namespace std;

class CTest
{
public:
    //默认构造函数
    CTest(int nVal = 999)
    {
        m_nVal = nVal;
        cout << "CTest::CTest" << endl;
    }
    //析构
    ~CTest()
    {
        cout << "CTest::~CTest" << endl;
    }
    void Hello()const;
    void Hello();
private:
```

```
    int m_nVal;
};

void CTest::Hello()const //this 是const
{
    cout << "Hello(const)" << endl;
}

void CTest::Hello()
{
    cout << "Hello" << endl;
}

int main()
{
    CTest test;
    test.Hello();  //Hello
    const CTest testConst;
    testConst.Hello();        //Hello(const)

    return 0;
}
```

示例2：用例子说明常成员函数不能更改类的数据成员，也不能调用非常成员函数，但是可以调用常成员函数，非常成员函数可以调用常成员函数。

```
#include <iostream>
using namespace std;

//用例子说明常成员函数不能更改类的变量，也不能调用非常成员函数，
//但是可以调用常成员函数。非常成员函数可以调用常成员函数
class CTest
{
public:
                CTest();
                CTest(int nNum1, int nNum2);

                int Sum()const;
                int Sum();

                void HelloWorld()const;
                void Hello();

private:
                int m_nNum1;
                int m_nNum2;
};
```

```
CTest::CTest()
{
		cout << "CTest::CTest" << endl;
}

CTest::CTest(int nNum1, int nNum2)
{
		this->m_nNum1 = nNum1;
		this->m_nNum2 = nNum2;
}

int CTest::Sum()const
{
		//常成员函数不能更改类的数据成员的值

		//this->m_nNum1 = this->m_nNum1 + 10;
		//error C3490: 由于正在通过常量对象访问“m_nNum1”，因此无法对其进行修改
		//this->m_nNum2 = this->m_nNum2 + 20;
		//error C3490: 由于正在通过常量对象访问“m_nNum2”，因此无法对其进行修改

		//Hello();                                          //常成员函数内部不能调用非常成员函数
		//rror C2662: “void CTest::Hello(void)”: 不能将“this”指针从“const CTest”转换为“CTest &”
		HelloWorld();  //常成员函数内部可以调用常成员函数
		return m_nNum1 + m_nNum2;
}

int CTest::Sum()
{
		this->m_nNum1 = this->m_nNum1 + 10;
		this->m_nNum2 = this->m_nNum2 + 20;
		HelloWorld();//非常成员函数内部可以调用常成员函数
		Hello();                //非常成员函数内部可以调用非常成员函数

		return m_nNum1 + m_nNum2;
}

void CTest::HelloWorld()const
{
		cout << "void CTest::Hello()const" << endl;
}
```

```
void CTest::Hello()
{
		cout << "void CTest::Hello()" << endl;
}

int main()
{
		CTest test;
		test.Sum();
		test.Hello();

		//在示例化类对象的时候在前面添加const，可以通过对象调用类中的常成员函数
		const CTest testConst;
		testConst.HelloWorld();
		testConst.Sum();
		return 0;
}
```

初始化列表

问题： 怎样对类中的数据成员进行初始化?

```
#include <iostream>
using namespace std;

class CFoo
{
public:
   CFoo()
   {
         //m_nVal = 999;   赋值语句
   }
private:
   //const int m_nVal = 999;        //11 14新语法可以直接对其进行初始化
   const int m_nVal;          // 怎样对类中的数据成员进行初始化?
};

int main()
{
    CFoo foo;
    return 0;
}
```

为此C++提供了一个新语法：初始化列表
使用方法(示例)：

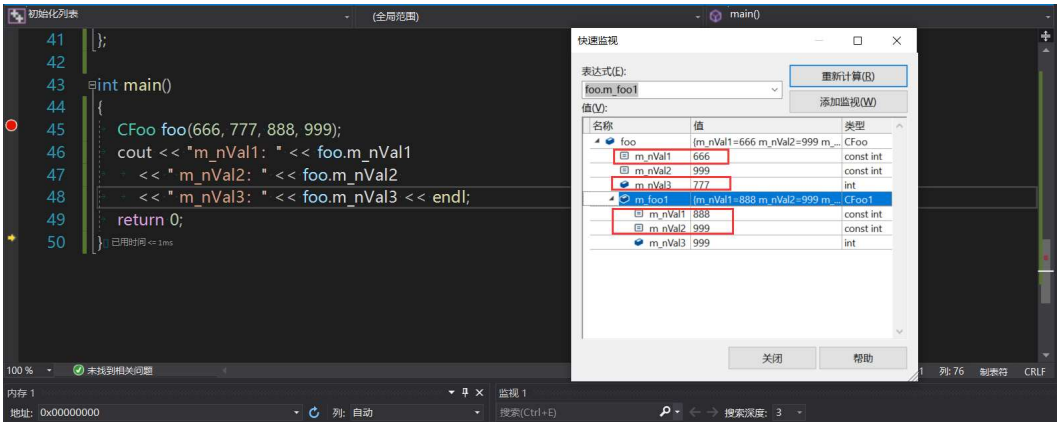
```
在构造函数后面添加 ':' 号，紧接着数据成员+(初始化的值)
public:
    CFoo(int nVal, int nVal2) : m_nVal1(nVal), m_nVal2(999),
m_nVal3(nVal2)
    {
     //DOTO
    }
public:
    const int m_nVal1;
    const int m_nVal2;
    int m_nVal3;
int main()
{
    CFoo foo(666);
    cout << "m_nVal1: " << foo.m_nVal1
        << " m_nVal2: " << foo.m_nVal2
        << "m_nVal3: " << foo.m_nVal3 << endl;
}
```

除了给类中的数据成员进行赋初值外，还可以对类对象成员进行赋初值

```
#include <iostream>
using namespace std;

class CFoo1
{
public:
   CFoo1(int nVal, int nVal2) :
      m_nVal1(nVal),
      m_nVal2(999),
      m_nVal3(nVal2)
   {
         //m_nVal = 999;    赋值语句
   }
public:
   //const int m_nVal = 999;         //11 14新语法可以直接对其进行初始化
   const int m_nVal1;         // 怎样对类中的数据成员进行初始化？
   const int m_nVal2;
```

```
    int m_nVal3;
};

class CFoo
{
public:
    CFoo(int nVal, int nVal2, int nValfoo1, int nValfoo2) :
      m_nVal1(nVal),
      m_nVal2(999),
      m_nVal3(nVal2),
      m_foo1(nValfoo1, nValfoo2)
    {
        //m_nVal = 999;    赋值语句
    }
public:
    //const int m_nVal = 999;          //11 14新语法可以直接对其进行初始化
    const int m_nVal1;          // 怎样对类中的数据成员进行初始化？
    const int m_nVal2;
    int m_nVal3;
    CFoo1 m_foo1;      //类对象成员
};

int main()
{
    CFoo foo(666, 777, 888, 999);
    cout << "m_nVal1: " << foo.m_nVal1

         << " m_nVal2: " << foo.m_nVal2

         << " m_nVal3: " << foo.m_nVal3 << endl;

    return 0;
}
```

快速监视：Shift + F9



结论：
1、用来给成员(常成员、非常成员、类对象成员)赋初值
2、初始化列表的时机要早于构造函数体

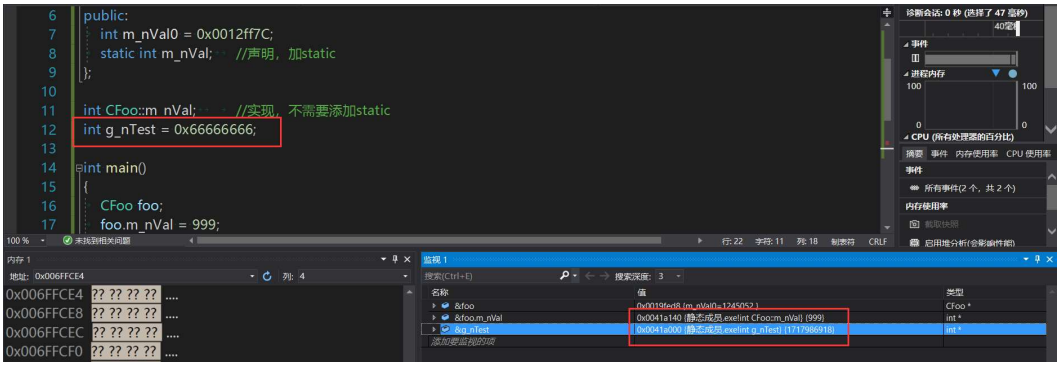
构造和析构的顺序:
构造 - 先成员, 后自己
析构 - 先自己, 再成员
多个成员受定义的顺序的影响, 了解即可,换个编译器不一定是这个结果

静态成员

设计初衷：解决C语言中全局变量的滥用问题、全局变量(不利于团队间的合作，多人开发代码不易维护)

C++提出了静态成员

```
#include <iostream>
using namespace std;

class CFoo
{
public:
              static int m_nVal;    //声明，加关键字 static，不占用内存
};

int CFoo::m_nVal;   //实现，不需要添加static

int main()
{
   CFoo foo;
   foo.m_nVal = 999;    //编译不通过需要对静态成员进行实现
   //编译不通过error LNK2001: 无法解析的外部符号 "public: static int CFoo::m_nVal" (?m_nVal@CFoo@@2HA)
   //无法解析的外部符号

    cout << sizeof(CFoo) << endl;   //当前类的大小为 4
    return 0;
}
```



静态数据成员使用注意事项:

1. 声明和实现分开

- 类中声明，类外定义
- 定义要写到.cpp里，不然包含头文件会出现重定义error
- 使用类名访问

2. 有独立的内存, 放在全局数据区
3. 同全局变量的生命周期一样, 与程序本身同生共死
4. 属于类，不属于对象, 所有的对象共享

我们可以使用 static 关键字来把类成员定义为静态的。当我们声明类的成员为静态时，这意味着无论创建多少个类的对象，静态成员都只有一个副本

静态成员在类的所有对象中是共享的。如果不存在其他的初始化语句，在创建第一个对象时，所有的静态数据都会被初始化为零。我们不能把静态成员的初始化放置在类的定义中，但是可以在类的外部通过使用范围解析运算符 :: 来重新声明静态变量从而对它进行初始化，如下面的实例所示。
代码示例：

```
#include <iostream>
using namespace std;

class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
```

```
    }
  private:
    double length;     // 长度
    double breadth;    // 宽度
    double height;     // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main()
{
  Box Box1(3.3, 1.2, 1.5);    // 声明 box1
  Box Box2(8.5, 6.0, 2.0);    // 声明 box2

  // 输出对象的总数
  cout << "Total objects: " << Box::objectCount << endl;

  return 0;
}
/*
程序运行结果:
Constructor called.
Constructor called.
Total objects: 2
*/
```

静态成员函数

如果把函数成员声明为静态的，就可以把函数与类的任何特定对象独立开来。静态成员函数即使在类对象不存在的情况下也能被调用，静态函数只要使用类名加范围解析运算符 :: 就可以访问。
静态成员函数只能访问静态成员数据、其他静态成员函数和类外部的其他函数。
静态成员函数有一个类范围，他们不能访问类的 this 指针。您可以使用静态成员函数来判断类的某些对象是否已被创建。

静态成员函数与普通成员函数的区别：

- 静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
- 普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针。

静态成员函数使用注意事项:

- 类中声明，可以类中定义，也可以类外定义
- 使用类名访问

代码示例1：

```
/*
静态成员函数:
1、属于类,可以直接通过类名调用 CFoo::GetVal()
2、静态成员函数没有this指针
*/
#include <iostream>
using namespace std;
class CFoo
{
public:
   static int GetVal()
   {
       //this->m_nVal0 = 999;
       //error C2355: “this”: 只能在非静态成员函数或非静态数据成员初始值设定项的内部引用
       //m_nVal0 = 666;
       return m_nVal;
   }

private:
       int m_nVal0;
       static int m_nVal;
};

int CFoo::m_nVal = 999;

int main()
{
   //CFoo::m_nVal = 999;
   CFoo foo;
   int nVal = foo.GetVal();
   //int nVal0 = ((CFoo*)0)->GetVal();          //没有静态成员函数的时候这样访问成员函数
   int nVal1 = CFoo::GetVal();
   //可通过类名，对象访问类中的方法
return 0;
}
```

代码示例2：

```
#include <iostream>

using namespace std;
```

```
class Box
{
   public:
      static int objectCount;
      // 构造函数定义
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // 每次创建对象时增加 1
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // 长度
      double breadth;    // 宽度
      double height;     // 高度
};

// 初始化类 Box 的静态成员
int Box::objectCount = 0;

int main()
{

   // 在创建对象之前输出对象的总数
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // 声明 box1
   Box Box2(8.5, 6.0, 2.0);    // 声明 box2

   // 在创建对象之后输出对象的总数
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
/*
程序运行结果：
```

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
*/
```