陈伟 2016.3

dll的创建和使用

使用C++语言创建dll并使用该dll的演示、说明

附2

**Dll的创建和使用**

[第一部分 创建C风格的dll 4](#_Toc7087)

[步骤1：创建dll工程 4](#_Toc20290)

[步骤2：文件改名 5](#_Toc11233)

[步骤3：拷贝文件内容 6](#_Toc21484)

[步骤4：从工程中删掉文件main.cpp,main.h 7](#_Toc30278)

[步骤5：继续修改文件mydll.h和mydll.cpp 7](#_Toc826)

[步骤6：编译并生成dll文件 8](#_Toc30217)

[第二部分 以间接调用的方式使用.dll 10](#_Toc12171)

[步骤1：新建一个控制台工程(UseDll.cbp). 10](#_Toc28774)

[步骤2：修改main.cpp。 11](#_Toc15390)

[步骤3：编译，生成UseDll.exe 12](#_Toc27342)

[步骤4：拷贝mydll.dll到UseDll.exe所在目录 13](#_Toc13091)

[步骤5：执行UseDll.exe,输出： 13](#_Toc32555)

[间接调用Dll的好处 14](#_Toc1387)

[第三部分 以直接调用的方式使用.dll 15](#_Toc26449)

[步骤1：改写main.cpp 15](#_Toc2210)

[步骤2：添加mydll.dll的静态导出库到UseDll工程中 16](#_Toc8407)

[步骤3：编译并执行，输出 17](#_Toc12333)

[直接调用的好处 18](#_Toc21752)

[Dll的优势 18](#_Toc7717)

[调用约定的说明 18](#_Toc19491)

[第四部分 创建C++风格的dll（本地方法） 19](#_Toc21770)

[步骤1：在mydll.h中新增导出的类 19](#_Toc13156)

[步骤2：在mydll.cpp中实现Cat类 19](#_Toc22555)

[步骤3：编译MyDll，生成mydll.dll，mydll.a 21](#_Toc27091)

[第五部分 使用本地风格的C++ DLL 22](#_Toc14209)

[步骤1：修改UseDll中的main.cpp 22](#_Toc32419)

[步骤2：将mydll.a添加到UseDll工程的BuildOptions中 23](#_Toc11943)

[步骤3：编译UseDll工程，生成UseDll.exe 23](#_Toc3213)

[步骤4：将MyDll.dll拷贝到UseDll.exe目录下 23](#_Toc28276)

[步骤5：执行UseDll.exe，输出 23](#_Toc17831)

[本地风格的不足 24](#_Toc14919)

[第六部分 创建推荐风格的C++ DLL 26](#_Toc23275)

[步骤1：修改mydll.h，创建Cat的接口类ICat 26](#_Toc30180)

[步骤2：新建类Cat 27](#_Toc1271)

[步骤3：实现createCat函数 29](#_Toc29620)

[步骤4：编译，生成mydll.dll 31](#_Toc24527)

[第七部分 使用推荐风格的C++ DLL 32](#_Toc3581)

[步骤1：修改UseDll中的 main.cpp 32](#_Toc6565)

[步骤2：将第六部分生成的libmydll.a添加到UseDll的BuildOptions中 33](#_Toc11017)

[步骤3：编译UseDll工程，生成useDll.exe 34](#_Toc13559)

[步骤4：将第六部分生成的mydll.dll拷贝到UseDll.exe所在文件目录 34](#_Toc2898)

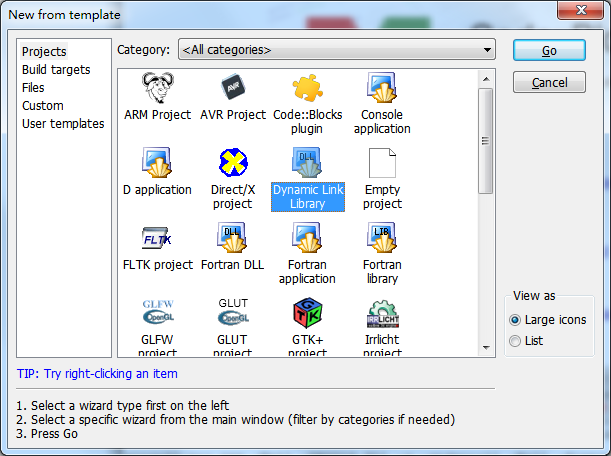
[步骤5：执行，输出 35](#_Toc24704)

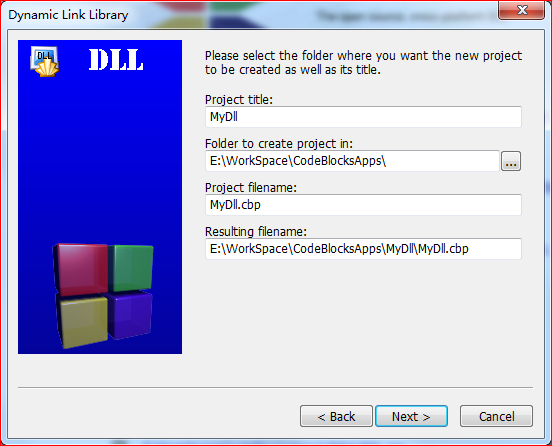
[推荐风格的好处： 35](#_Toc31270)

# 第一部分 创建C风格的dll

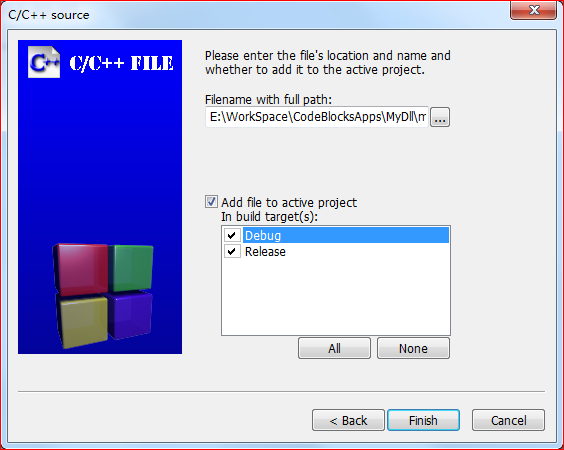
下面以CodeBlocks为例演示说明创建和使用dll，其它开发环境类似

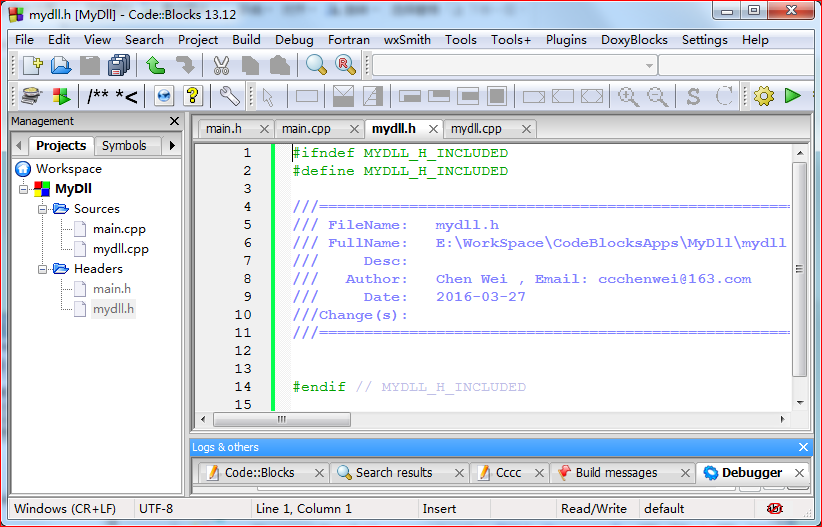
步骤1：创建dll工程

创建dll工程MyDll.cpp  
 



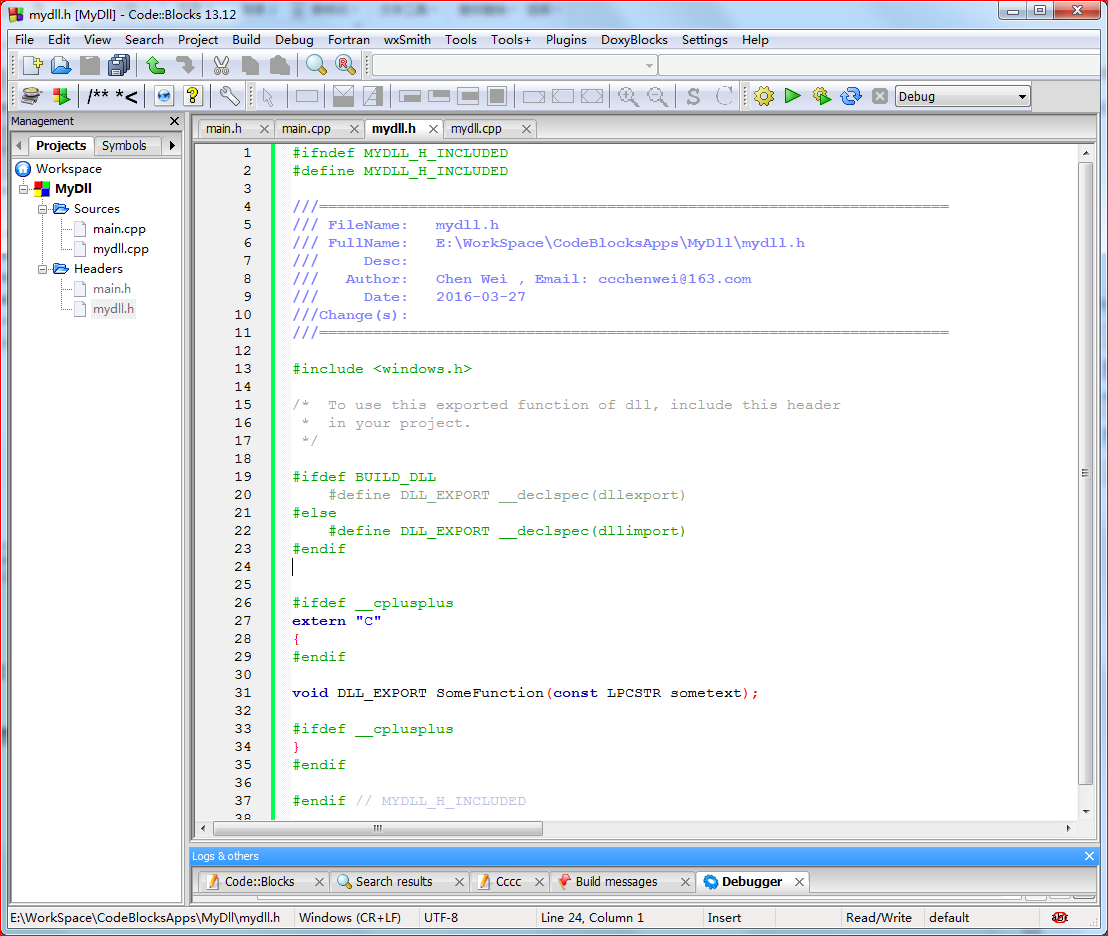
## 步骤2：文件改名

新建两个文件mydll.h和mydll.cpp  


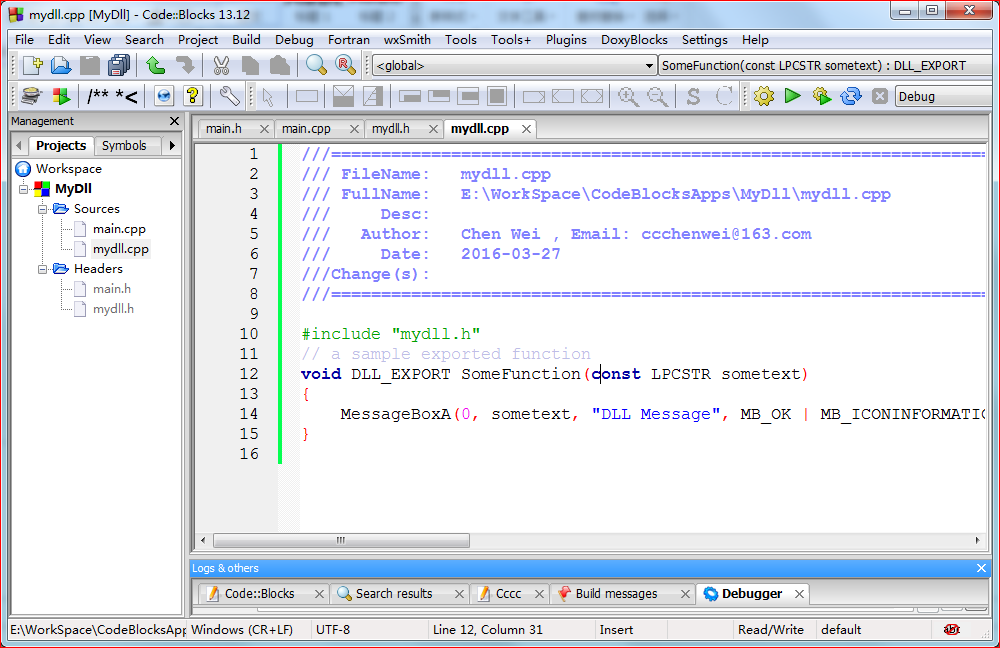


## 步骤3：拷贝文件内容

将main.h中的核心内容拷贝到mydll.h中



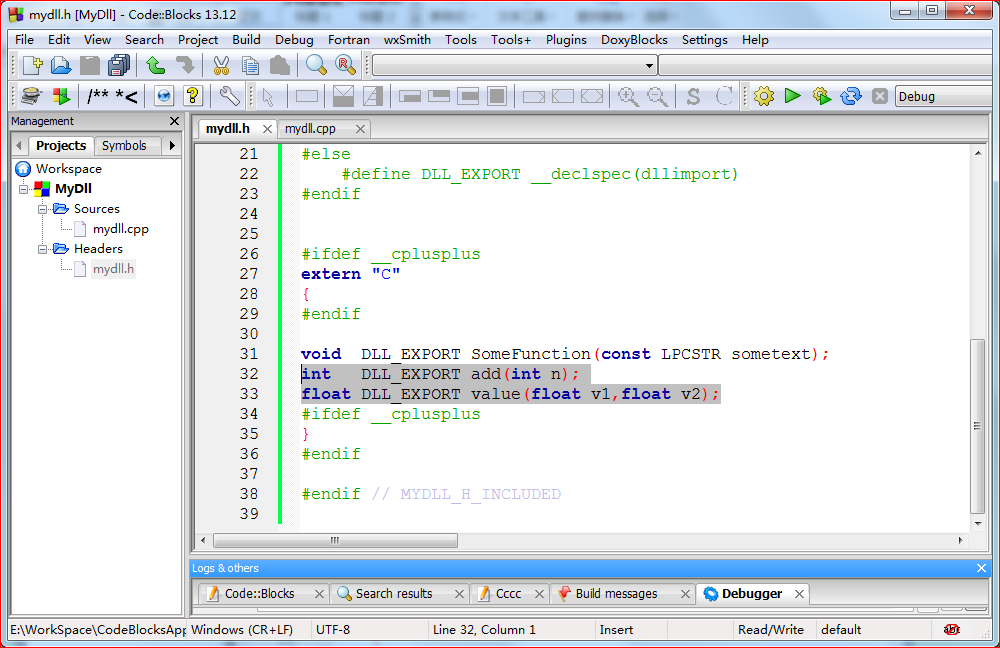
再将main.cpp中的部分内容拷贝到mydll.cpp中，并更改include文件名 。( 函数DllMain的缺省实现也可以去掉 )



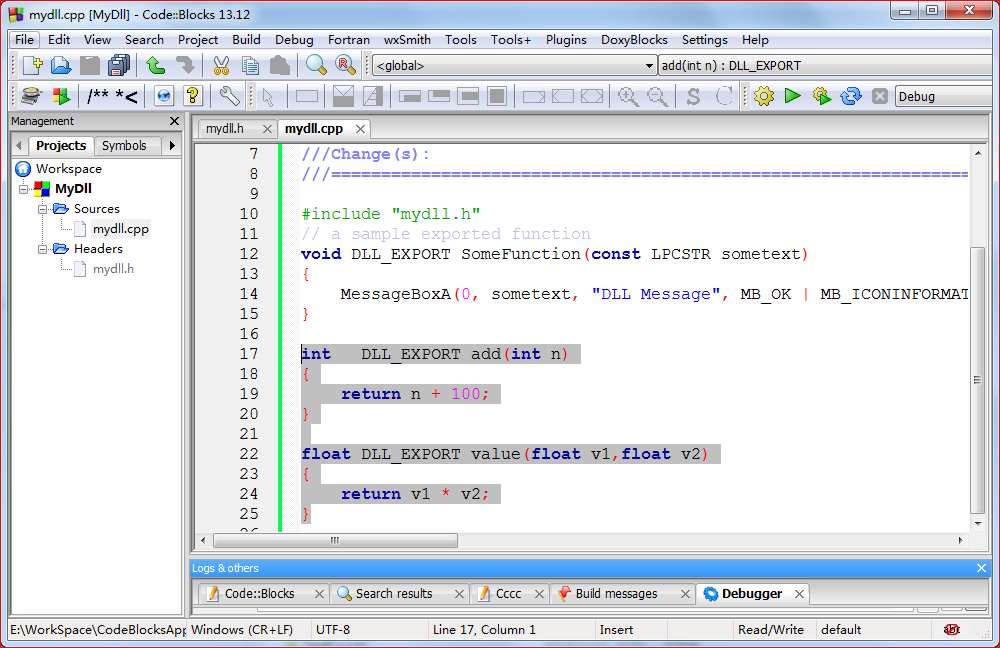
## 步骤4：从工程中删掉文件main.cpp,main.h

## 步骤5：继续修改文件mydll.h和mydll.cpp

在mydll.h中添加两个函数 , add和value



在mydll.cpp中给出这两个函数的实现。

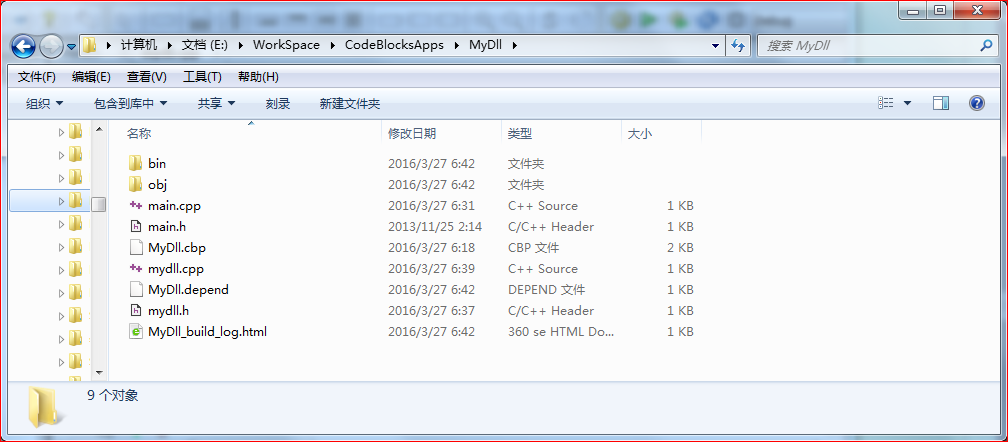


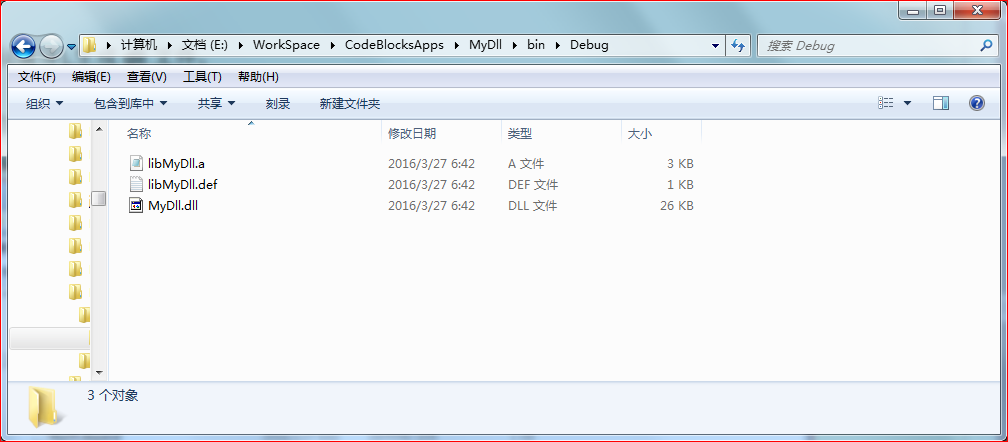
## 步骤6：编译并生成dll文件

点击齿轮图标(编译),生成dll文件。

打开我的电脑或资源管理器，可以查看生成的文件目录。

E:\WorkSpace\CodeBlocksApps\MyDll目录下(本演示的工程目录):



E:\WorkSpace\CodeBlocksApps\MyDll\bin\Debug:(若选择Release版，则Release子目录) 

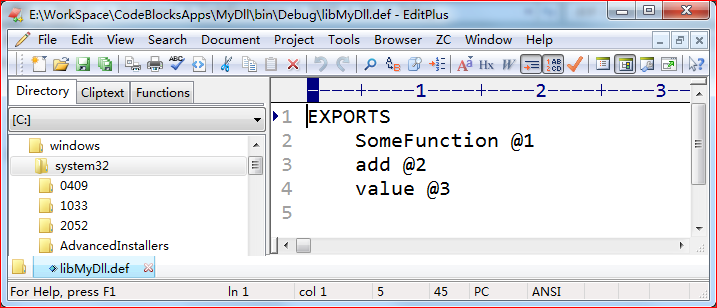
至此：我们就得到了四个有用的文件。

Mydll.h: 头文件。编写调用该dll的exe工程时，要用到。

Mydll.dll ：目标dll文件，要随调用该dll的exe文件一起发布。

libmydll.a: 含有mydll.dll中导出函数的地址信息的静态库。在编译调用方的工程时，在链接(Link)阶段, 可能会用到。在使用其他平台和其他编译器时，该文件的扩展名可能为.lib.

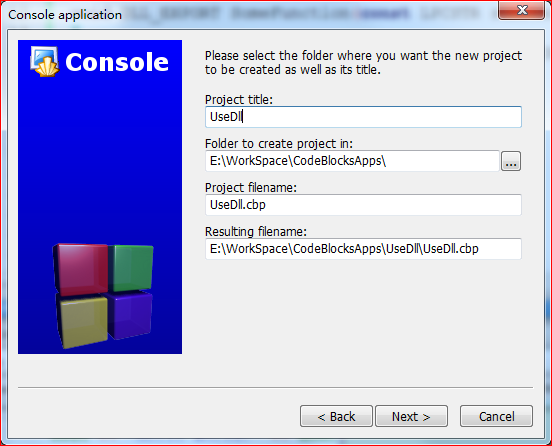
libMydll.def ：导出函数的定义文件(文本文件)，内容如下：

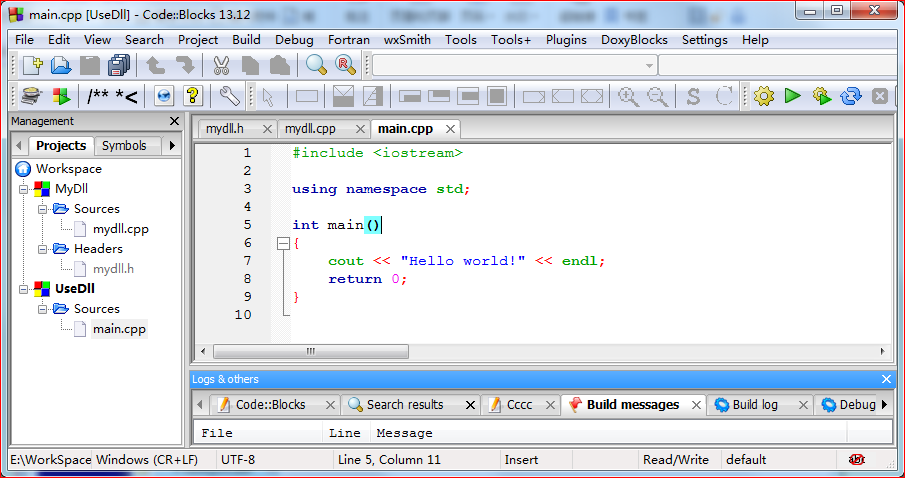
****

# 第二部分 以间接调用的方式使用.dll

（使用上边创建的mydll.dll)

## 步骤1：新建一个控制台工程(UseDll.cbp).





## 步骤2：修改main.cpp。

//main.cpp

#include <iostream>

using namespace std;

#include "..\\MyDll\\mydll.h"

int main()

{

typedef void (\*FuncTypeSome)(LPCSTR);

typedef int (\*FuncTypeAdd)(int);

typedef float (\*FuncTypeVal)(float,float);

HINSTANCE hDll = LoadLibrary("MyDll.dll");

if (hDll==nullptr) {

cout<<"Load Library Failed."<<endl;

return 1;

}

FuncTypeSome some = (FuncTypeSome)GetProcAddress(hDll,"SomeFunction");

FuncTypeAdd addFunc = (FuncTypeAdd)GetProcAddress(hDll,"add");

FuncTypeVal valFunc = (FuncTypeVal)GetProcAddress(hDll,"value");

some("Call SomeFunction.");

cout<<"add(88)="<<addFunc(88)<<endl;

cout<<"value(2.5,1.2)="<<valFunc(2.5,1.2)<<endl;

FreeLibrary(hDll);

cout << "Hello world!" << endl;

return 0;

}

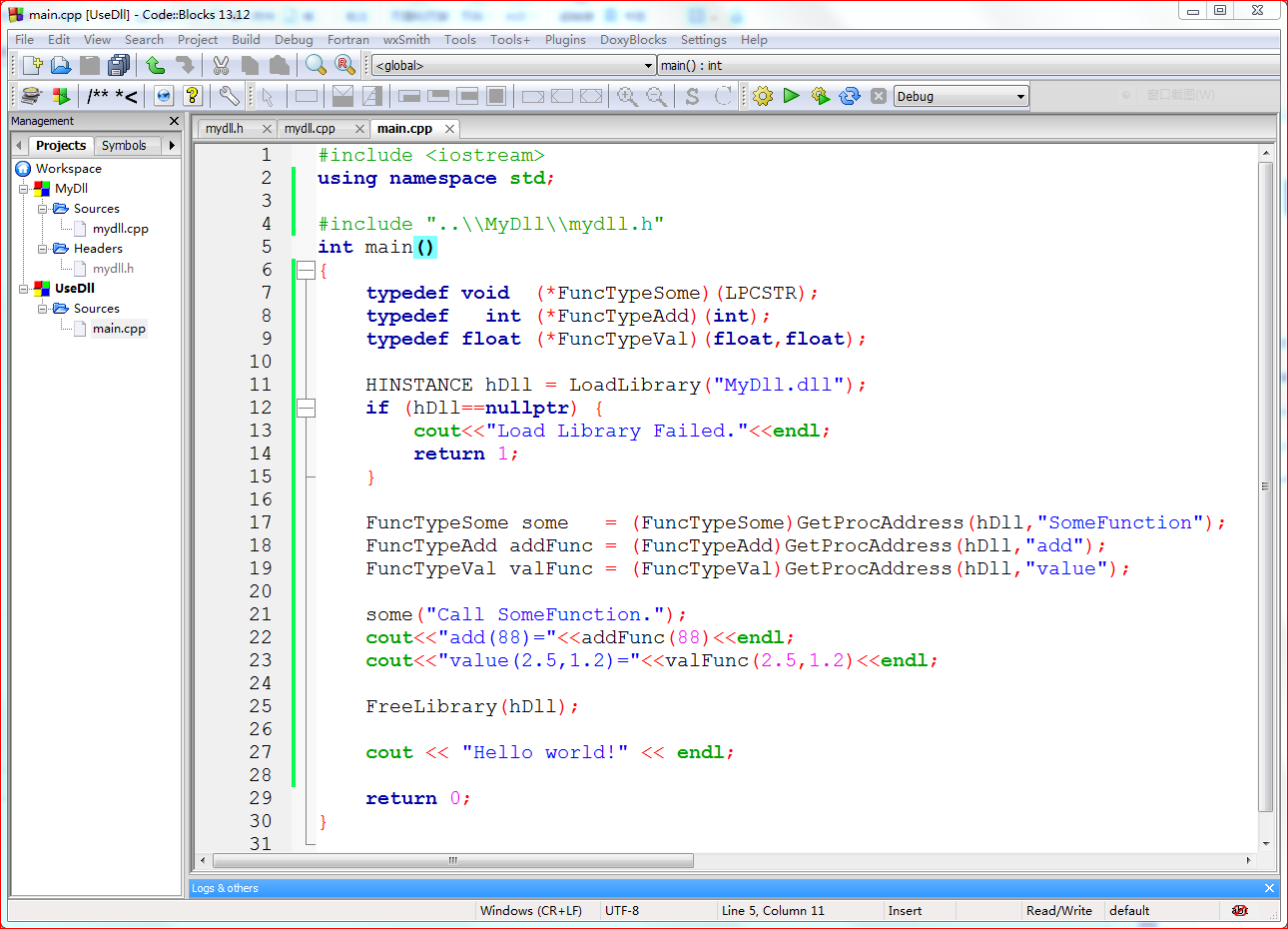
这部分说明：

LoadLibrary()是Window API函数，加载库文件，返回一个指针（也称句柄）。

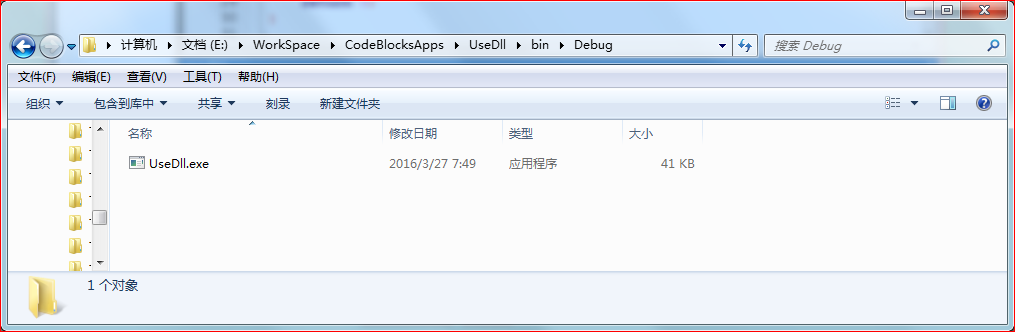
typedef void (\*FuncType)(const LPCSTR),是声明类型名FuncType，表示FuncType的每一个变量都是形如void XXXX(const LPCSTR)的某个函数。

GetProcAddress也是Window API函数，取得库中指定名字的函数入口地址。

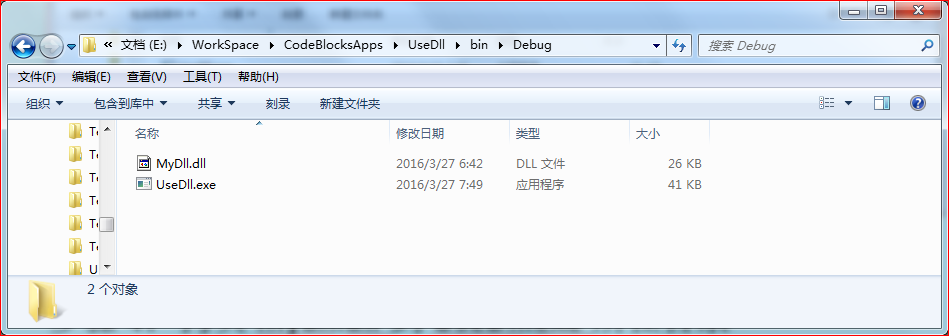
如图：



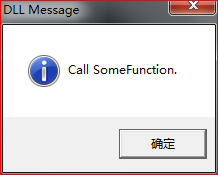
## 步骤3：编译，生成UseDll.exe

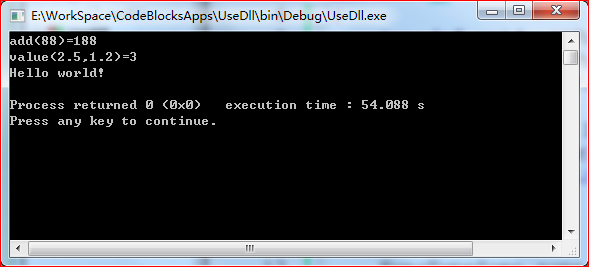


## 步骤4：拷贝mydll.dll到UseDll.exe所在目录



## 步骤5：执行UseDll.exe,输出：





## 间接调用Dll的好处

程序员可以在程序中(如 UseDll.exe)中，动态地(或者称运行时期)决定具体调用哪个dll中的的哪个函数。例如，plug\_in(插件)就是一个dll，通常就需要这种方式决定调用哪个插件中的函数。

# 第三部分 以直接调用的方式使用.dll

(使用上边创建的mydll.dll)

## 步骤1：改写main.cpp

(注释掉部分，改用原始名字调用，最后代码如下)

// main.cpp

#include <iostream>

using namespace std;

#include "..\\MyDll\\mydll.h"

int main()

{

/\*\*

typedef void (\*FuncTypeSome)(LPCSTR);

typedef int (\*FuncTypeAdd)(int);

typedef float (\*FuncTypeVal)(float,float);

HINSTANCE hDll = LoadLibrary("MyDll.dll");

if (hDll==nullptr) {

cout<<"Load Library Failed."<<endl;

return 1;

}

FuncTypeSome some = (FuncTypeSome)GetProcAddress(hDll,"SomeFunction");

FuncTypeAdd addFunc = (FuncTypeAdd)GetProcAddress(hDll,"add");

FuncTypeVal valFunc = (FuncTypeVal)GetProcAddress(hDll,"value");

some("Call SomeFunction.");

cout<<"add(88)="<<addFunc(88)<<endl;

cout<<"value(2.5,1.2)="<<valFunc(2.5,1.2)<<endl;

FreeLibrary(hDll);

\*\*/

SomeFunction("Call SomeFunction");

cout<<"add(88)="<<add(88)<<endl; // mydll.h中的原始名字

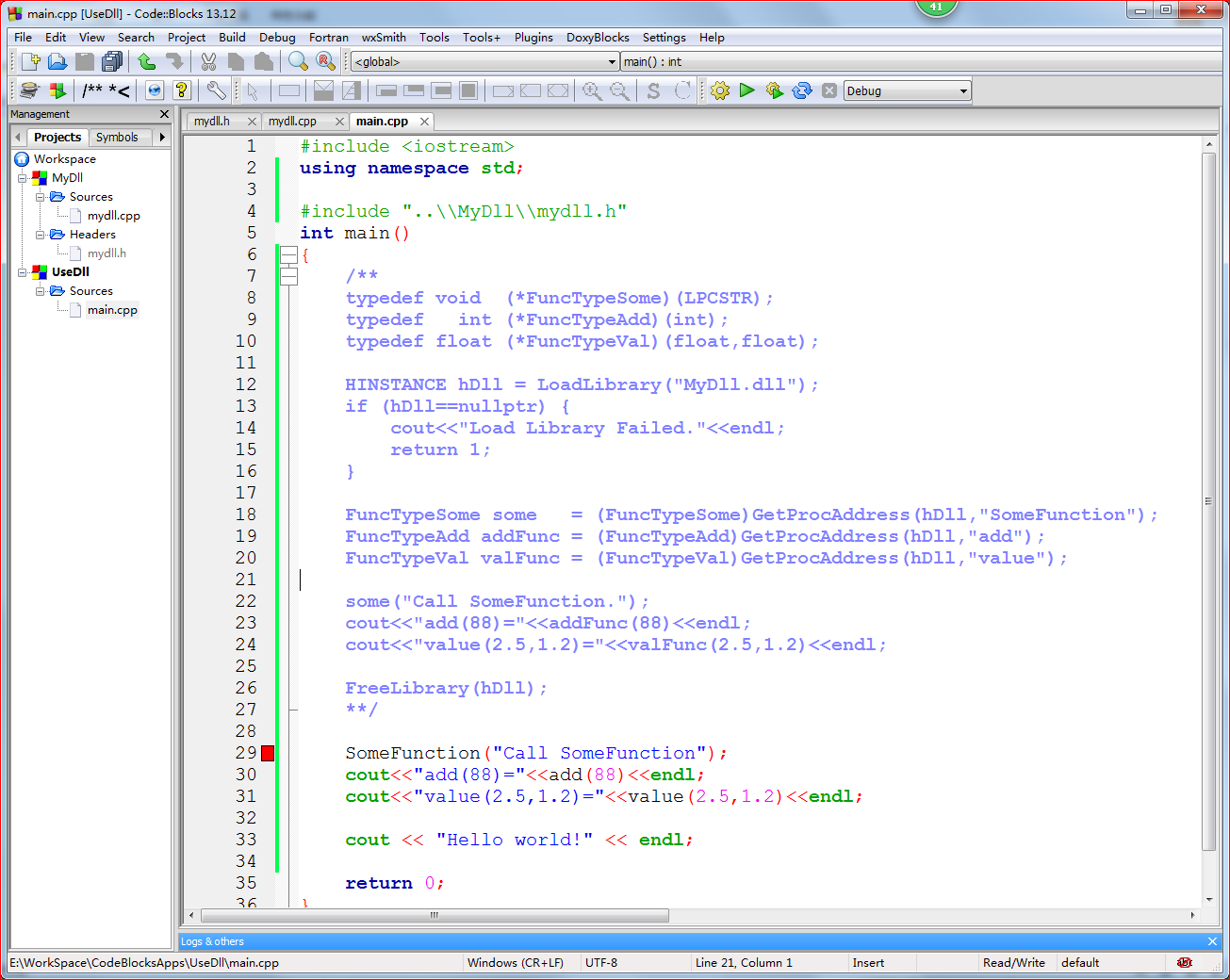
cout<<"value(2.5,1.2)="<<value(2.5,1.2)<<endl; // mydll.h中的原始名字

cout << "Hello world!" << endl;

return 0;

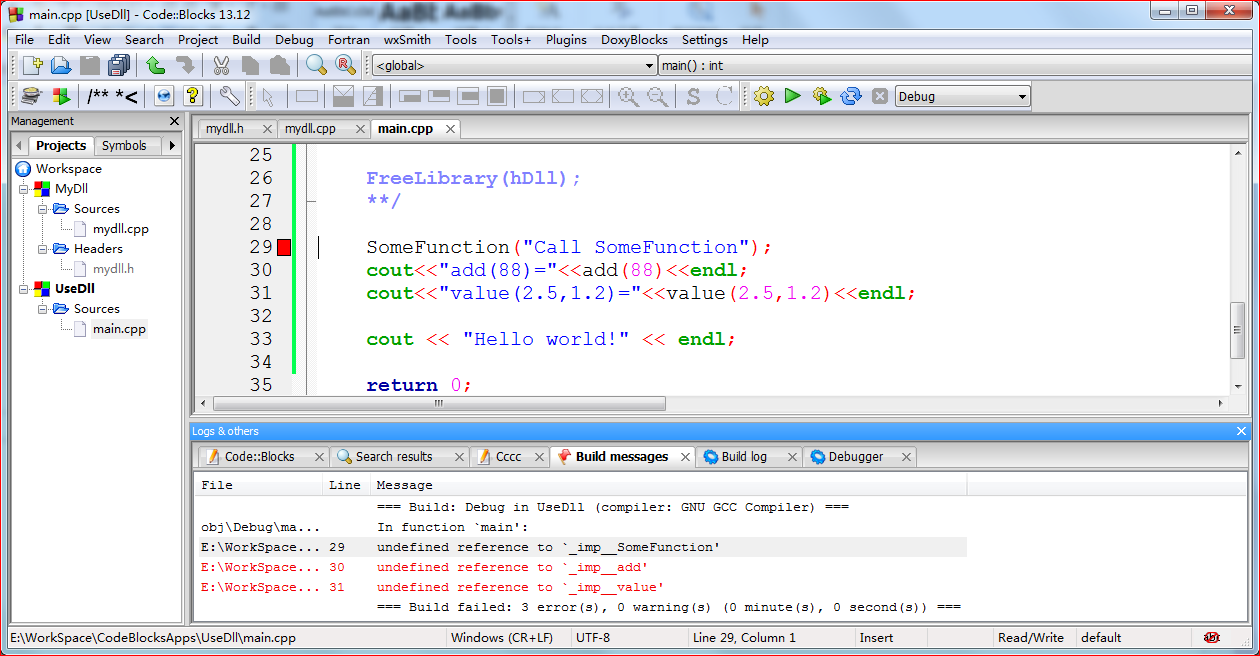
}

如图：



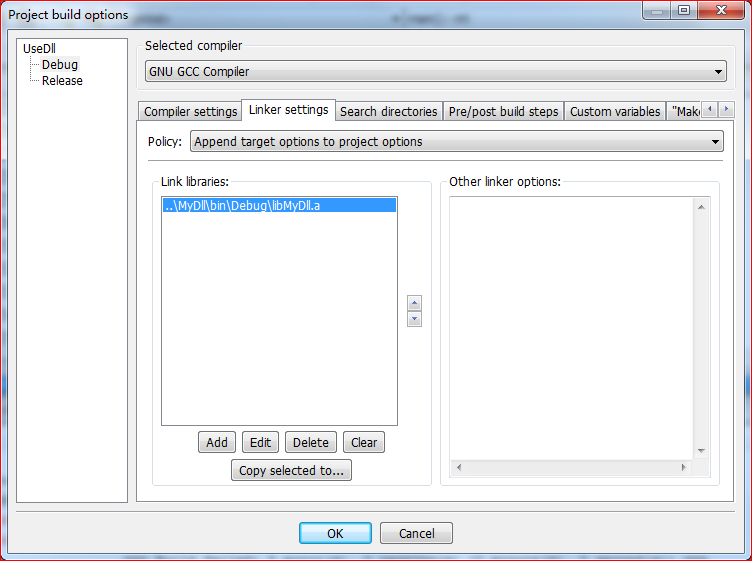
## 步骤2：添加mydll.dll的静态导出库到UseDll工程中

此时编译，会报错。如图：

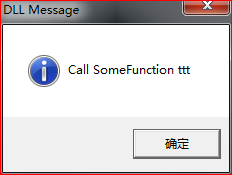


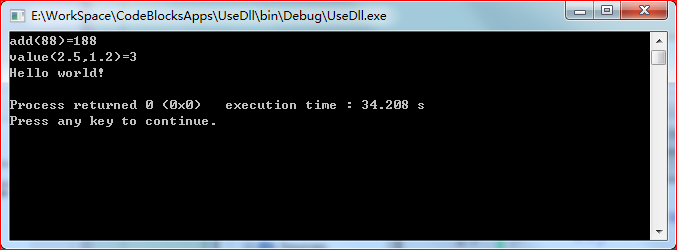
表明编译器在链接(Link)时，找不到三个函数 \_imp\_\_SomeFunction, \_imp\_\_add,\_imp\_\_value的入口地址，也就是没找到这三个函数的具体实现部分。但这三个函数的名字明显不是我们自己定义的，实际上这些名字是mydll.dll中对SomeFunction,add,value函数的内部表示名，他们的实现必定在mydll.dll中，而不会在UseDll工程中，当然就找不到了。为了告诉编译器，这些函数的入口地址，就需要用到从mydll.dll导出的存有函数入口信息的静态导出库-- libMyDll.a文件了。

打开 Project->Build Options->Linker Setting,添加LibMyDll.a库文件



## 步骤3：编译并执行，输出





## 直接调用的好处

相比间接调用更方便

## Dll的优势

1. 在UseDll.exe不动且不改变MyDll中的MyDll.h文件的情况下，只重新修改MyDll.cpp中SomeFunction、add、value的实现，并重新生成Mydll.dll, 就可以改变UseDll.exe的输出。

2. 若多个程序运行时，都需要Mydll.dll, Mydll.dll只会加载一次（内存中只有一个），各程序共享该dll，所以dll也称共享库(Shared Libray)。相对于静态库，大大节省内存消耗。

## 调用约定的说明

本Demo中，MyDll工程和UseDll工程都是在Window7平台上开发运行的，而且使用的编译器是MinGW的同一个版本，所以两个工程中函数的默认的调用约定一定是一致的，因此Usedll一定能识别MyDll中export的SomeFunction等函数名，本例中也就可以不关心具体采用那种调用约定了。(C++98默认\_\_cdecl,C++11默认\_\_stdcall).

但是若两个工程分别在不同的平台开发，使用的编译器也不一样，那就要关心所采用的调用约定了，这种情况时，通常采用\_\_stdcall。

# 第四部分 创建C++风格的dll（本地方法）

## 步骤1：在mydll.h中新增导出的类

添加一个Export的类Cat。

class DLL\_EXPORT Cat

{

public:

void setAge(int curAge);

int getAge() const;

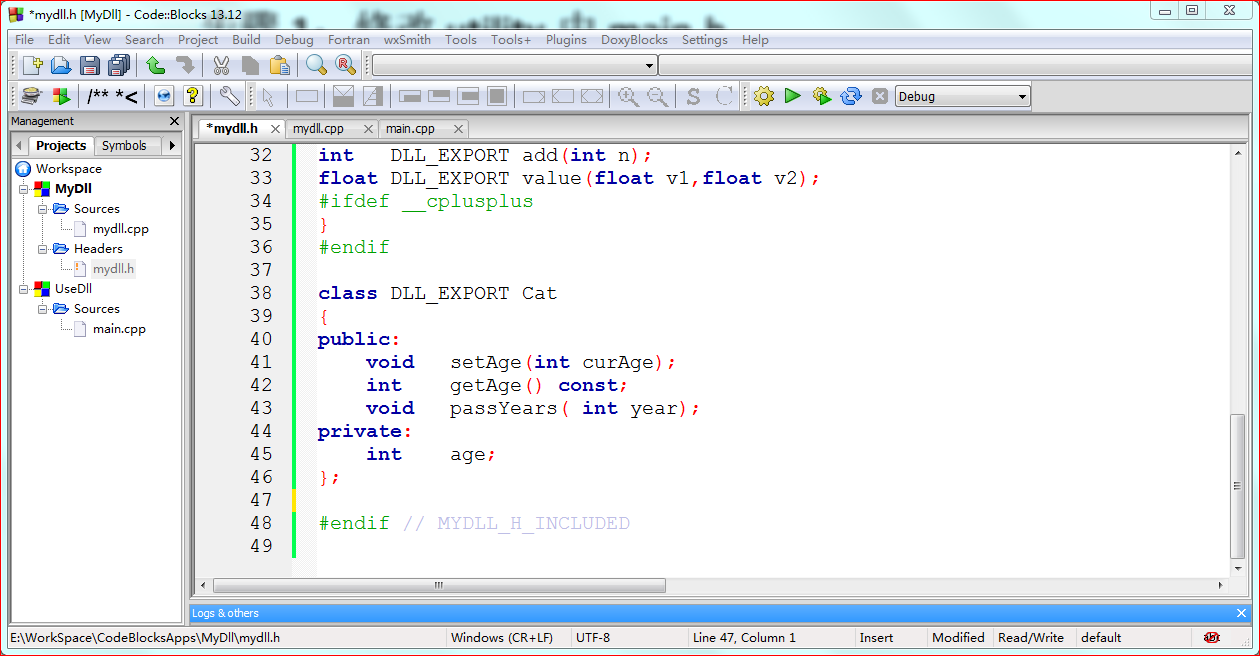
void passYears( int year);

private:

int age;

};

如图：



## 步骤2：在mydll.cpp中实现Cat类

在mydll.cpp中添加下列代码

///-------------------------------------

void Cat::setAge(int curAge)

{

age = curAge;

}

int Cat::getAge() const

{

return age;

}

void Cat::passYears( int year)

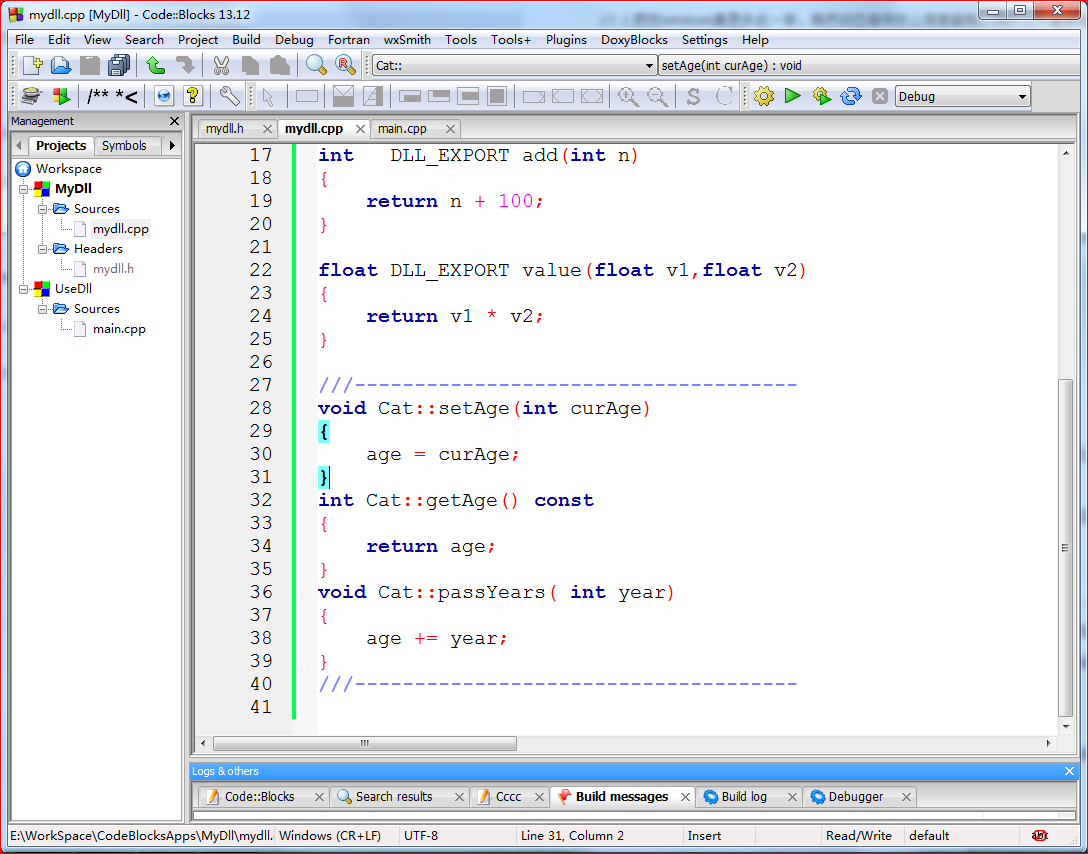
{

age += year;

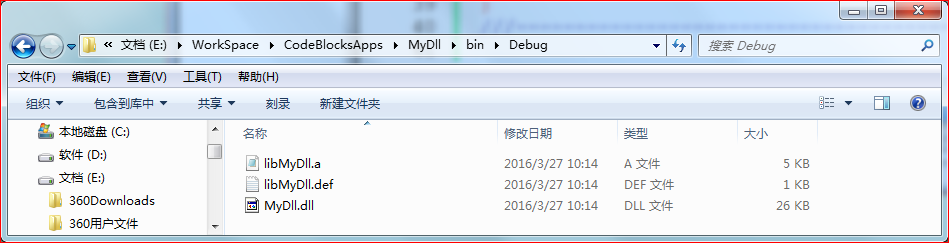
}

///-------------------------------------

如图：



## 步骤3：编译MyDll，生成mydll.dll，mydll.a



# 第五部分 使用本地风格的C++ DLL

(使用第四部分中mydll.dll中的Cat类)

## 步骤1：修改UseDll中的main.cpp

// main.cpp

#include <iostream>

using namespace std;

#include "..\\MyDll\\mydll.h"

int main()

{

Cat aCat;

aCat.setAge(10);

aCat.getAge();

cout<<"Cat's age = "<<aCat.getAge()<<endl;

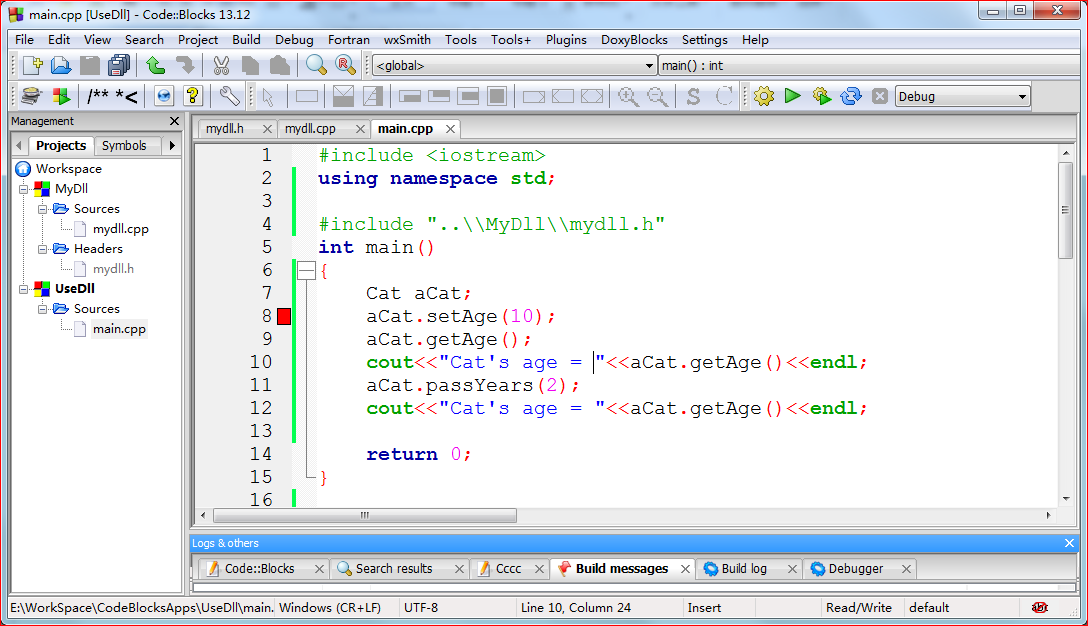
aCat.passYears(2);

cout<<"Cat's age = "<<aCat.getAge()<<endl;

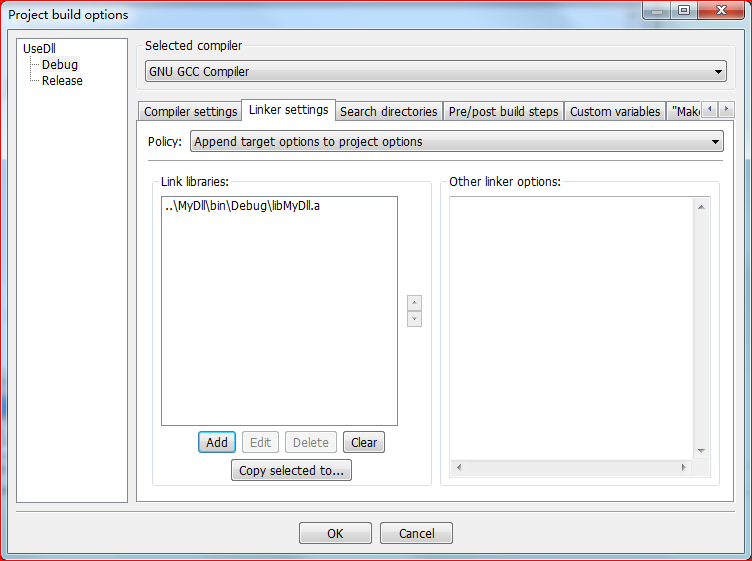
return 0;

}

如图：

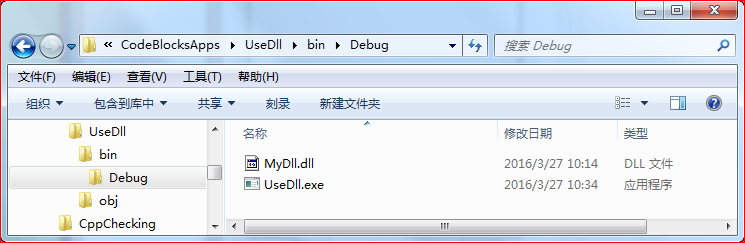


## 步骤2：将mydll.a添加到UseDll工程的BuildOptions中

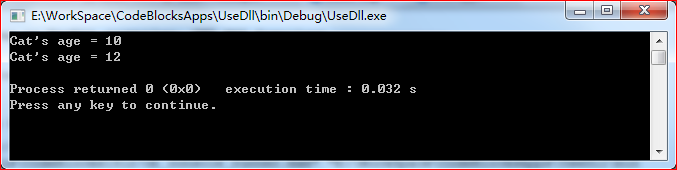


## 步骤3：编译UseDll工程，生成UseDll.exe

## 步骤4：将MyDll.dll拷贝到UseDll.exe目录下



## 步骤5：执行UseDll.exe，输出



使用完毕。

## 本地风格的不足

1.当export某个类时，如 class DLL\_EXPORT Cat { .. };, 要求必须同时export所有Cat类的基类以及Cat所有数据成员的类(内置类型除外)。

如在MyDll.h中有：   
class Base { ... };  
struct KK { ... };

class DLL\_EXPORT Cat: public Base {  
...  
private:  
 KK aKK;  
};

那么在编译UseDll工程时，则会报错或给出警告，会说XXX找不到。因此必须将MyDll.h中的内容改写为：  
class DLL\_EXPORT Base { ... };  
struct DLL\_EXPORT KK { ... };

class DLL\_EXPORT Cat: public Base {  
...  
private:  
 KK aKK;  
};

1. 由于1中的原因，export的类难以直接使用STL。
2. 考察mydll.h中的Cat类的定义：

class DLL\_EXPORT Cat

{

public:

void setAge(int curAge);

int getAge() const;

void passYears( int year);

private:

int age;

};

Cat类定义中包含了类Cat的所有细节信息，如int型数据成员age，而mydll.dll的使用者UseDll工程需要include “mydll.h”, 这样就会将MyDll中Cat类的定义细节暴露给使用者，但使用者一般是不关系细节的，也不利于mydll.dll的使用安全。

正因为这点，通常建议采用下面介绍的方法。

# 第六部分 创建推荐风格的C++ DLL

## 步骤1：修改mydll.h，创建Cat的接口类ICat

#ifndef MYDLL\_H\_INCLUDED

#define MYDLL\_H\_INCLUDED

#include <windows.h>

**#include <memory>**

#ifdef BUILD\_DLL

#define DLL\_EXPORT \_\_declspec(dllexport)

#else

#define DLL\_EXPORT \_\_declspec(dllimport)

#endif

#ifdef \_\_cplusplus

extern "C"

{

#endif

void DLL\_EXPORT SomeFunction(const LPCSTR sometext);

int DLL\_EXPORT add(int n);

float DLL\_EXPORT value(float v1,float v2);

**class ICat;**

**std::shared\_ptr<ICat> DLL\_EXPORT createCat( );**

#ifdef \_\_cplusplus

}

#endif

**class ICat**

**{**

**public:**

**virtual ~ICat();**

**virtual void setAge(int curAge) = 0;**

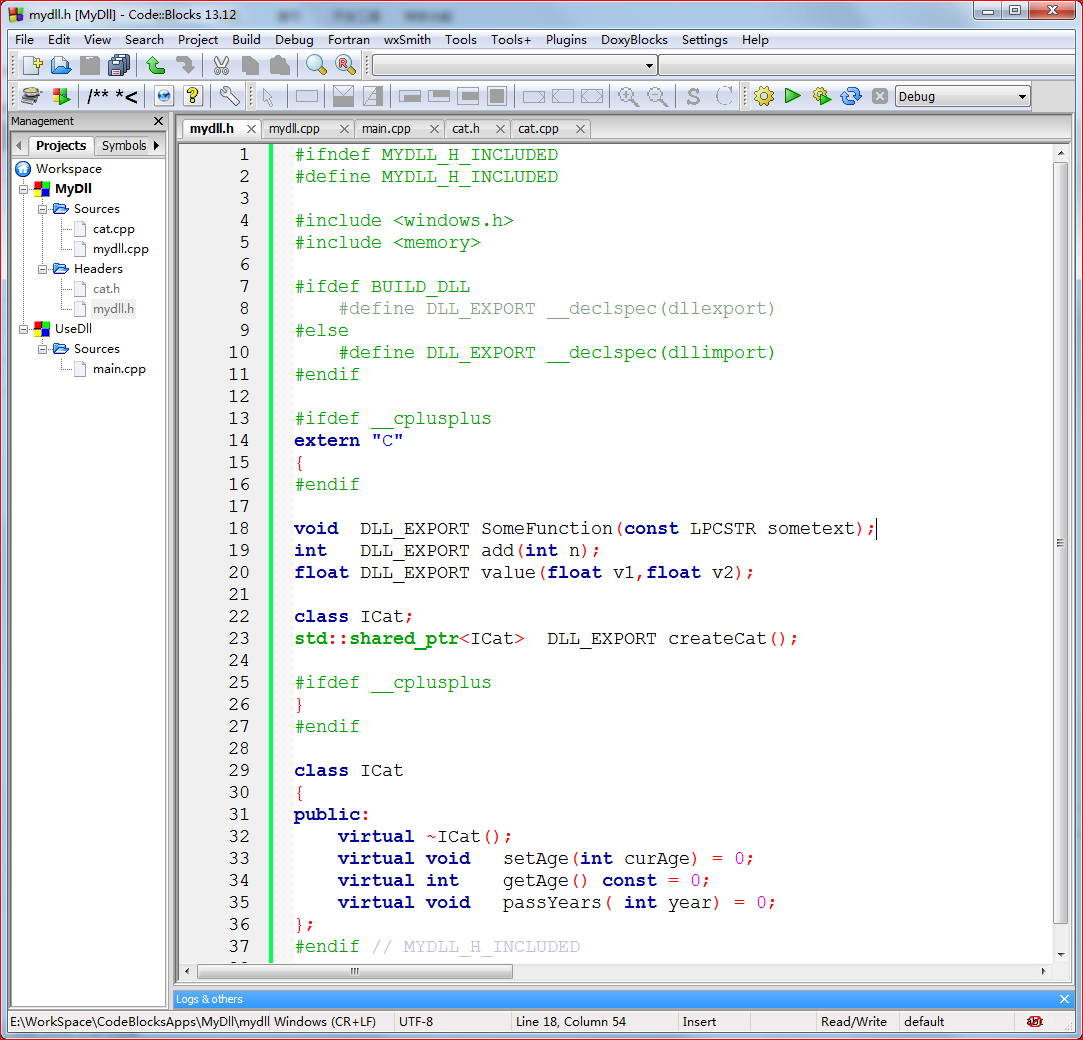
**virtual int getAge() const = 0;**

**virtual void passYears( int year) = 0;**

**};**

#endif // MYDLL\_H\_INCLUDED

如图：



说明：定义了Cat类的一个接口类ICat， 声明了创建Cat实例的创建方法：  
**std::shared\_ptr<ICat> DLL\_EXPORT createCat( );**

其中shared\_ptr是C++11中新引入的共享指针模板。使用它，可以有效防止用户“只创建，不释放”的问题，防止造成内存泄露。

## 步骤2：新建类Cat

在cat.h中定义ICat的子类Cat

//cat.h

#ifndef CAT\_H

#define CAT\_H

#include "mydll.h"

class Cat : public ICat

{

public:

virtual ~Cat( );

virtual void setAge(int curAge);

virtual int getAge() const;

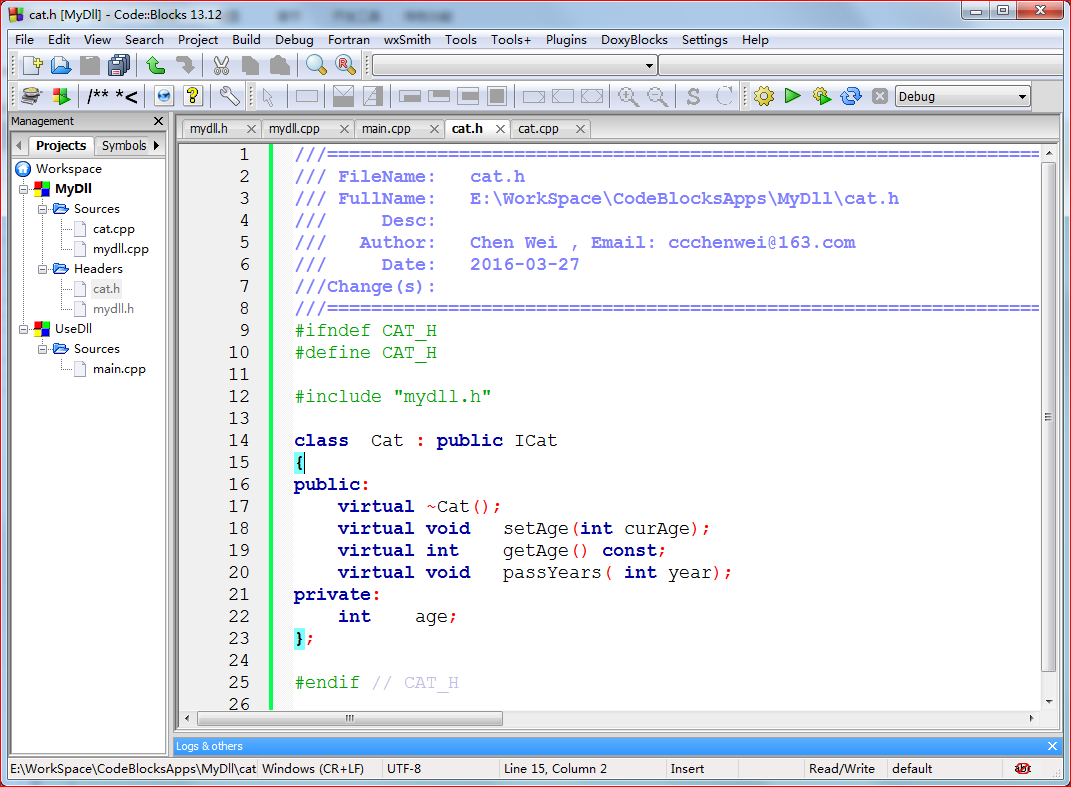
virtual void passYears( int year);

private:

int age;

};

#endif // CAT\_H



在cat.cpp中实现Cat

// cat.cpp

#include "mydll.h"

#include "cat.h"

ICat::~ICat()

{

}

Cat::~Cat()

{

}

void Cat::setAge(int curAge)

{

age = curAge;

}

int Cat::getAge() const

{

return age;

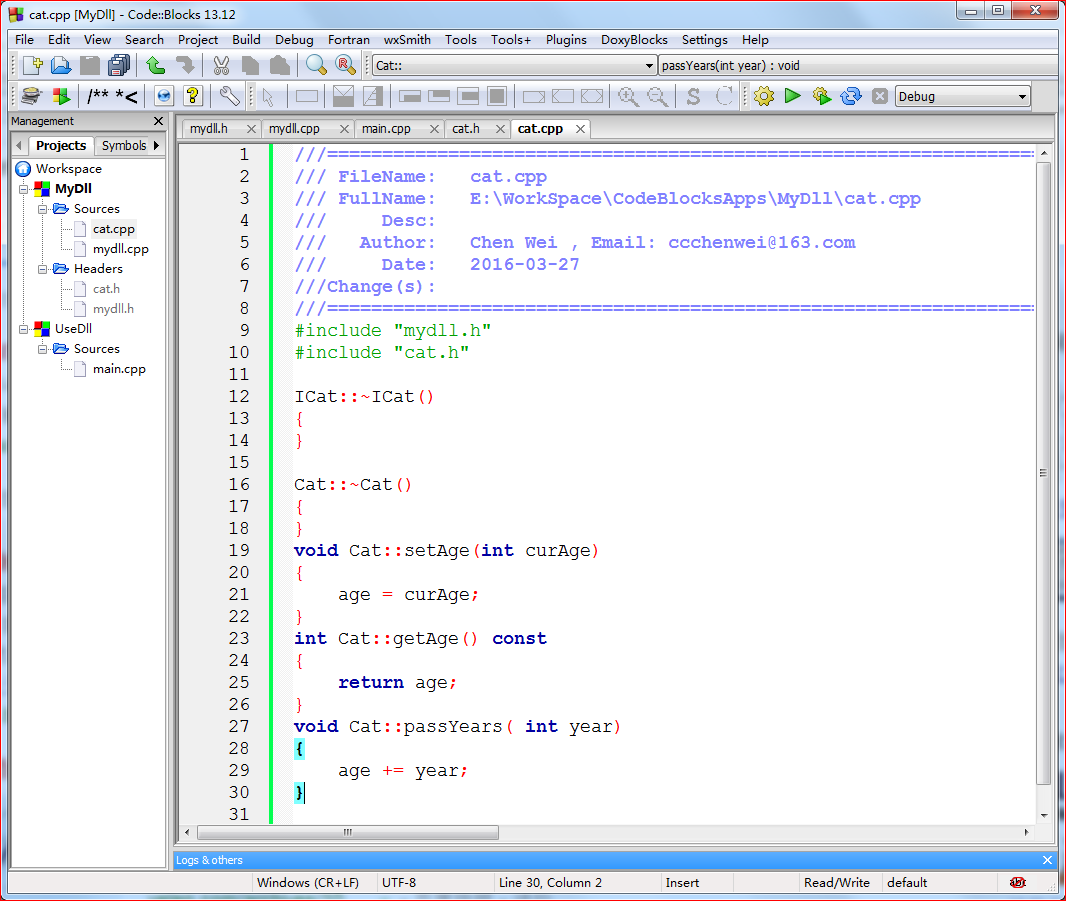
}

void Cat::passYears( int year)

{

age += year;

}



## 步骤3：实现createCat函数

在mydll.cpp或cat.cpp中均可，本例在mydll.cpp中实现

// mydll.cpp

#include "mydll.h"

#include "cat.h"

// a sample exported function

void DLL\_EXPORT SomeFunction(const LPCSTR sometext)

{

MessageBoxA(0, sometext, "DLL Message", MB\_OK | MB\_ICONINFORMATION);

}

int DLL\_EXPORT add(int n)

{

return n + 100;

}

float DLL\_EXPORT value(float v1,float v2)

{

return v1 \* v2;

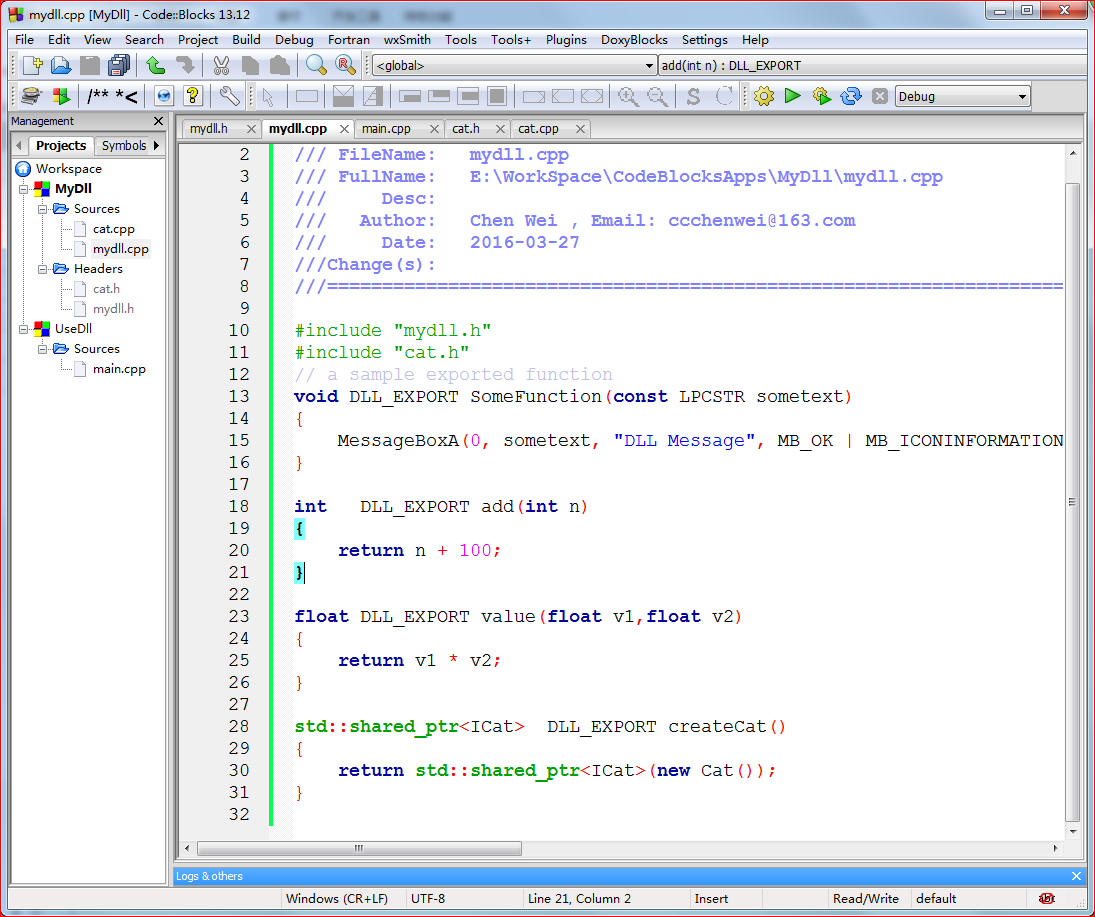
}

std::shared\_ptr<ICat> DLL\_EXPORT createCat( )

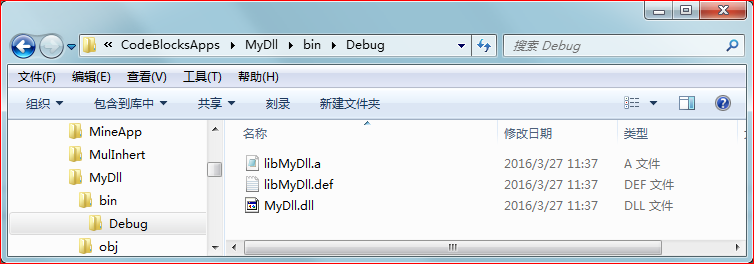
{

return std::shared\_ptr<ICat>(new Cat( ));

}



## 步骤4：编译，生成mydll.dll



# 第七部分 使用推荐风格的C++ DLL

(使用第六部分中mydll.dll中的ICat接口)

## 步骤1：修改UseDll中的 main.cpp

// main.cpp

#include <iostream>

using namespace std;

#include "..\\MyDll\\mydll.h"

int main()

{

std::shared\_ptr<ICat> spCat = createCat();

spCat->setAge(5);

cout<<"Cat's age = "<<spCat->getAge()<<endl;

spCat->setAge(7);

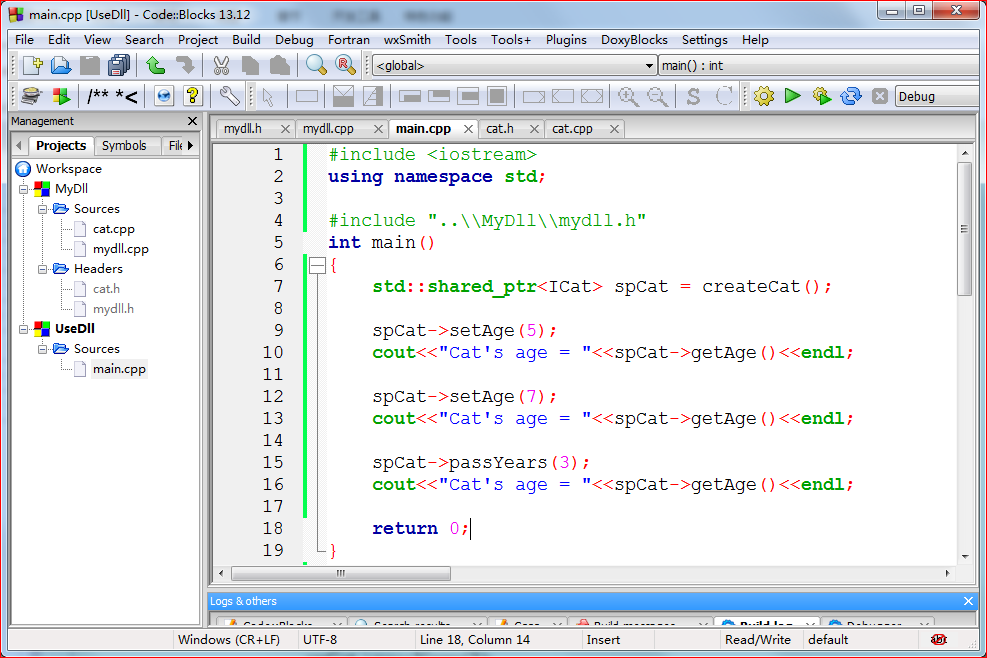
cout<<"Cat's age = "<<spCat->getAge()<<endl;

spCat->passYears(3);

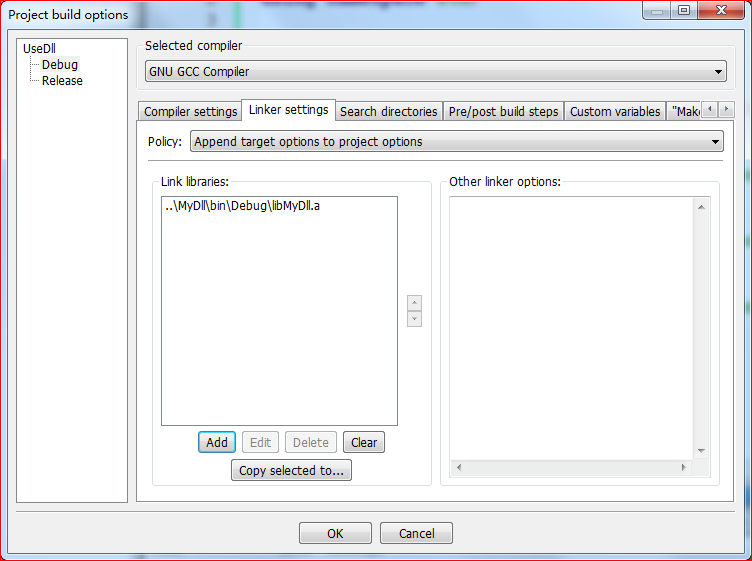
cout<<"Cat's age = "<<spCat->getAge()<<endl;

return 0;

}

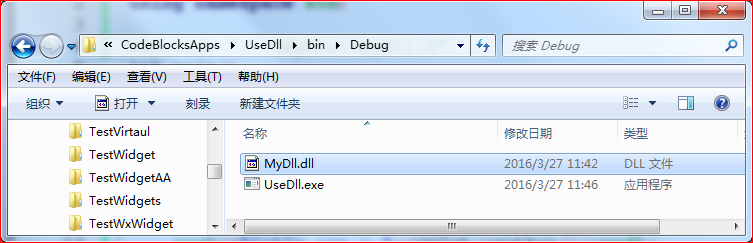


## 步骤2：将第六部分生成的libmydll.a添加到UseDll的BuildOptions中

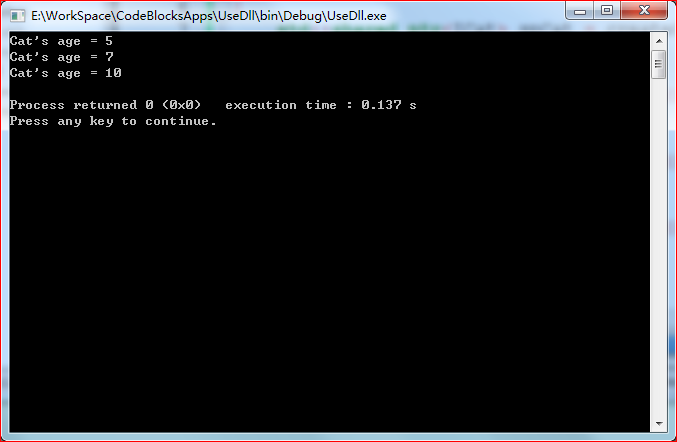


## 步骤3：编译UseDll工程，生成useDll.exe

## 步骤4：将第六部分生成的mydll.dll拷贝到UseDll.exe所在文件目录



## 步骤5：执行，输出



## 推荐风格的好处：

既能隐藏类（Cat）的细节，又能避免export相关的其它类（参考上一部分的最后说明).