2020/05/07_C++_第3课_命名空间、函数重载、内联函数

笔记本:	C++
创建时间:	2020/5/7 星期四 14:59
作者:	ileemi
标签:	函数重载, 命名空间, 内联函数

命名空间

作用：解决C语言的命名冲突问题。

C语言中可使用：项目_模块_功能名
缺点：名字过长，缩写会带来可读性的损失
C语言作用域：块作用域、局部（函数）作用域，文件作用域（静态函数），全局作用域。

C++：命名空间
命名空间的三种用法：
1、在main函数外自定义一个名为TEST的命名空间，里面可以做写在全局作用域下做的事情(声明函数，定义变量、结构体等)，在main()函数中使用自定义的命名空间里的数据时需要指出使用的是哪里的数值，可以在使用前使用 `using namespace xxx` xxx为自定义的作用域名字

```
#include <iostream>
using namespace std;
namespace TEST // TEST：作用域名字
{
	//所有在全局作用域里面做的事情，都可以在命名空间中做
	int g_nNumber = 999;
	void Test()
	{
		cout << "Hello World" << endl;
	}
	struct tagTest
	{
		int nNum;
		float fltNum;
	};
```

```
}
using namespace TEST;                    //相当于一个开关，将TEST里面定
义的 "名字"拉到当前作用域中

void Foo()
{
        cout << g_nNumber << endl;
        Test();
        tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
}

int main()
{
        cout << g_nNumber << endl;
        Test();
        tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
        Foo();
}
```

2、可以使用 :: 对自定义的命名空间里的数值，函数，等进行操作，示例：

```
#include <iostream>
using namespace std;
namespace TEST // TEST：作用域名字
{
        //所有在全局作用域里面做的事情，都可以在命名空间中做
        int g_nNumber = 999;
        void Test()
        {
                cout << "Hello World" << endl;
        }
        struct tagTest
        {
                int nNum;
                float fltNum;
        };
}
//using namespace TEST;
//相当于一个开关，将TEST里面定义的 "名字"拉到当前作用域中
void Foo()
{
```

```
        cout << TEST::g_nNumber << endl;
        TEST::Test();
        TEST::tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
}

int main()
{
        cout << TEST::g_nNumber << endl;
        TEST::Test();
        TEST::tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
        Foo();
}
```

3、使用命名空间中的指定数据时，可以将对应的数据，函数等拉到当前作用域中，示例：

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

namespace TEST // TEST：作用域名字
{
        //所有在全局作用域里面做的事情，都可以在命名空间中做
        int g_nNumber = 999;
        void Test()
        {
                cout << "Hello World" << endl;
        }
        struct tagTest
        {
                int nNum;
                float fltNum;
        };
}
//使用命名空间中的指定数据时，可以将对应的数据，函数等拉到当前作用域中
using TEST::g_nNumber;
using TEST::Test;
int main()
{
        cout << g_nNumber << endl;
        cout << g_nNumber << endl;
        cout << g_nNumber << endl;
```

```
        Test();
}
```

4、可以将同一个命名空间进行拆分，示例：

```
#include <iostream>
using namespace std;
namespace TEST // TEST：作用域名字
{
        //所有在全局作用域里面做的事情，都可以在命名空间中做
        int g_nNumber = 999;
        void Test()
        {
                cout << "Hello World" << endl;
        }
}
namespace TEST // 命名空间拆分
{
        struct tagTest
        {
                int nNum;
                float fltNum;
        };
}
using namespace TEST;                    //相当于一个开关，将TEST里面定义的 "名字"拉到当前作用域中
void Foo()
{
        cout << g_nNumber << endl;
        Test();
        tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
}
int main()
{
        cout << g_nNumber << endl;
        Test();
        tagTest test = {
                999, 6.18
        };
        cout << test.nNum << " " << test.fltNum << endl;
        Foo();
}
```

5、命名空间的嵌套（命名空间内可以定义命名空间），示例：

```
#include <iostream>
using namespace std;
namespace TEST // TEST：作用域名字
{
	namespace TEST2
	{
		void Test()
		{
			cout << "Hello World" << endl;
		}
	}
}
using namespace TEST::TEST2;		//休要注意层级
int main()
{
	Test();
	return 0;
}
```

6、命名空间可以取别名，示例：

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

namespace TEST
{
	int g_nNumber = 999;
}
//给TEST命名空间取个另外的名字，名为ShowData，可以像使用TEST一样使用
namespace ShowData = TEST;
using namespace ShowData;

int main()
{
	cout << g_nNumber << endl;
	return 0;
}
```

7、扩展：当全局作用域中的函数名和命名空间内的函数重名时，再调用时，编译器不知道使用哪个。
使用全局作用的函数(没有名称) 可以加 ::

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

```
namespace TEST // TEST: 作用域名字
{
	void Test()
	{
		cout << "TEST :: Hello World" << endl;
	}
}
void Test()
{
	cout << "Test :: Hello World" << endl;
}
using namespace TEST;
int main()
{
	//Test(); //error C2668: “Test”: 对重载函数的调用不明确

	/*
	可能是“void Test(void)”
	或	“void TEST::Test(void)”
	*/
	TEST::Test; //此时会调用TEST里面的 Test 函数
	::Test();  //此时会调用 全局作用域中的 Test 函数
}
```

函数重载

C语言中的函数名不可以重复
C++允许函数重名

函数组成：返回值、函数名、参数列表(类型、个数、位置)、调用约定

C++中构成函数重载的条件：
1、同名函数可以构成函数重载
2、参数列表（参数类型、参数个数、参数位置(这三个任意一个不同，都能构成函数重载))
3、返回值和调用约定不做考虑
4、同作用域可以构成函数重载（作用域不同，不构成函数重载）

示例1：同名函数可以构成函数重载

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

int Add(int nNum1, int nNum2)
```

```
{
		return nNum1 + nNum2;
}
float Add(int nNum1, float fltNum2)
{
		return nNum1 + fltNum2;
}
float Add(float fltNum1, float fltNum2)
{
		return fltNum1 + fltNum2;
}
int main()
{
		Add(10, 20);	//int Add(int nNum1, int nNum2);
		Add(10, 9.99f);	//float Add(int nNum1, float fltNum2);
		Add(9.275f, 22.625f);	//float Add(float fltNum1, float
fltNum2)
		return 0;
}
```

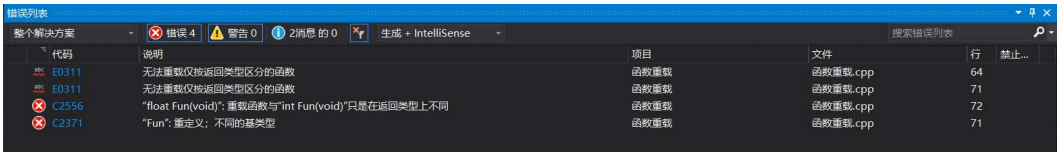
示例2：参数列表（参数类型、参数个数、参数位置(这三个任意一个不同，都能构成函数重载)）

```
//示例：函数参数类型不同
void Fun(int nNum)
{
	//当函数的参数类型不同时，可以构成函数重载
}
void Fun(float fltNum)
{

}

//示例：函数参数个数不同
void Fun(int nNum1, int nNum2)
{
	//当函数的参数个数不同时，也可以构成函数重载
}
void Fun(float fltNum)
{

}


//示例：参数位置不同
void Fun(int nNum, float fltNum)
{
```

```
    //当函数的参数的位置不同时，也可以构成函数重载
}
void Fun(float fltNum, int nNum)
{

}
```

不能构成喊出重载的条件：
返回值不同不能构成重载

```
int Fun()
{
        //无法重载仅按返回类型区分的函数
            return 999;
}
//error C2556: “float Fun(void)”: 重载函数与“int Fun(void)”只是在返回类型上不同
float Fun()
{
            //无法重载仅按返回类型区分的函数
            return 9.99f;
}
```

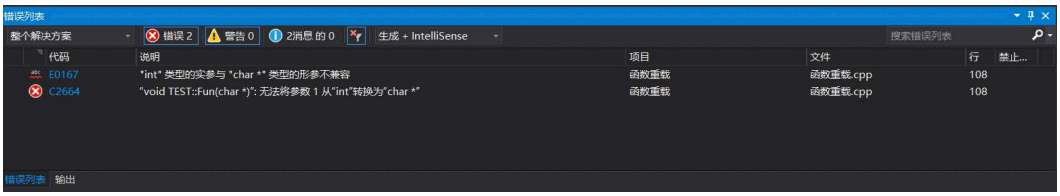


调用约定不同不能构成函数重载：

```
void __cdecl Foo()
{
    //error C2373: “Foo”: 重定义；不同的类型修饰符
}
void __stdcall Foo()
{
    //error C2373: “Foo”: 重定义；不同的类型修饰符
}
void __fastcall Foo()
{
    //error C2373: “Foo”: 重定义；不同的类型修饰符
}
```

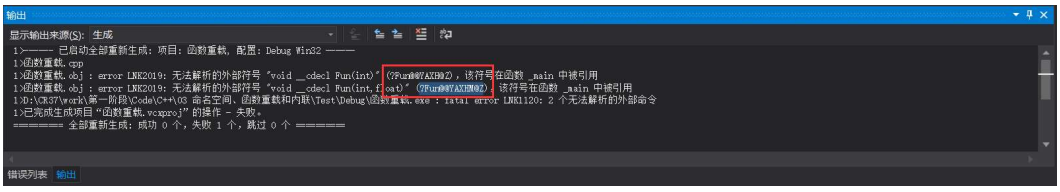
作用域不同，同名函数不会构成重载

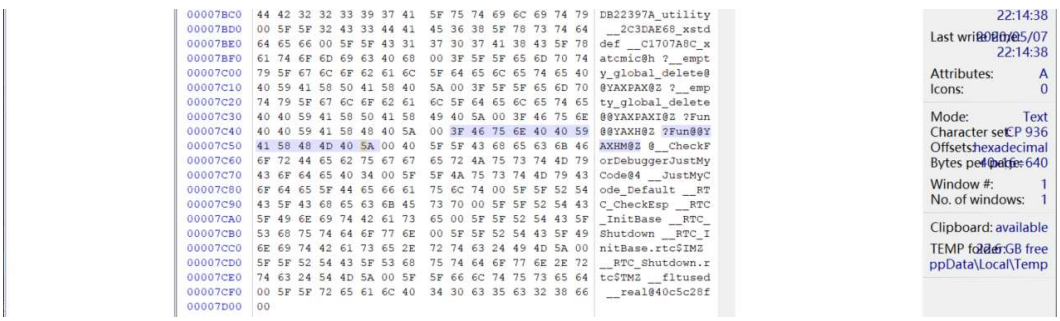
```
void Fun(int nVal)
{
}
namespace TEST
{
		void Fun(char* szVal)
		{
		}
		void Test()
		{
			Fun(5);//error C2664: “void CR37::Foo(char *)”: 无法将参数 1 从“int”转换为“char *”
		}
}
```



C++可以做到函数重载是由于C++在处理函数的时候，会对函数没进行名称粉碎后，粉碎后的保存了函数的作用域、返回值，函数名、参数类型

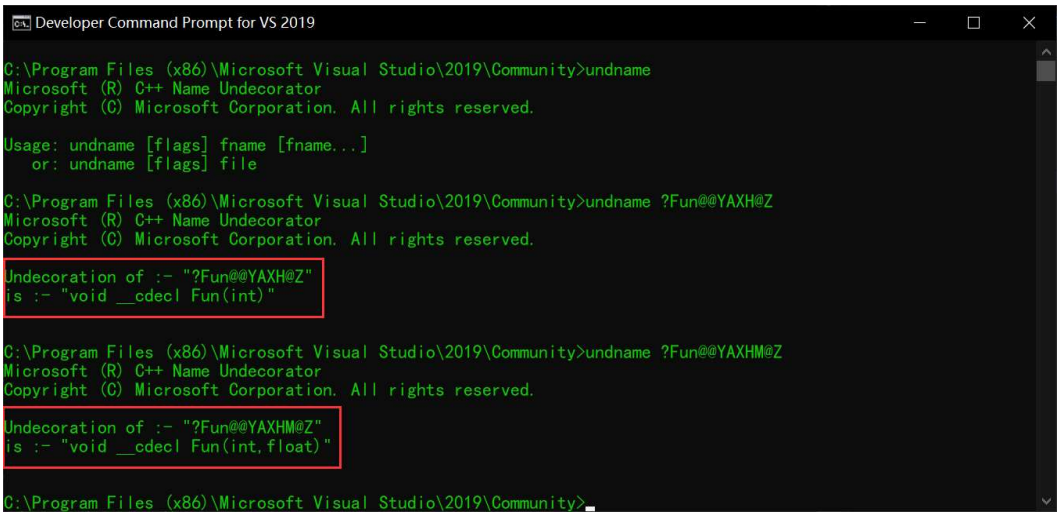
名称粉碎还原工具：undname（打开VS的开发人员命令行）





使用VS2019自带的Developer Command Prompt for VS 2019

C语言不支持函数重载的原因：C语言的命名粉碎默认只有函数名



扩展

C++文件怎样调用C语言的函数，可以使用关键字：extern "C"

```
//告诉c++，在找这个函数的实现的时候，不用使用c++的名称粉碎规则，而是使用c的名称粉碎规则
extern "C" int Add_nn(int nNum1, int nNum2);
//使用extern "C"之后，就不能进行函数重载
extern "C" int Add_nn(int nNum1, float fltNum)
{
}

int main()
{
    Add_nn(999, 6.18);
}
```

坑一：二义性，函数重载调用的时候，类型精确匹配(可以通过类型强转达到目的)

```
void Fun(int nNum)
{
}

void Fun(float fltNum)
```

```
{
}

int main()
{
    //Fun(6.18); //二义性
}
```

坑二: 默认参对函数重载也是有影响的

```
void Fun(int nNum, float fltNum = 6.18f)
{
}

void Fun(int fltNum)
{
}

int main()
{
    Fun(999); //二义性 error C2668: “Fun”: 对重载函数的调用不明确
}
```

内联

函数调用过程

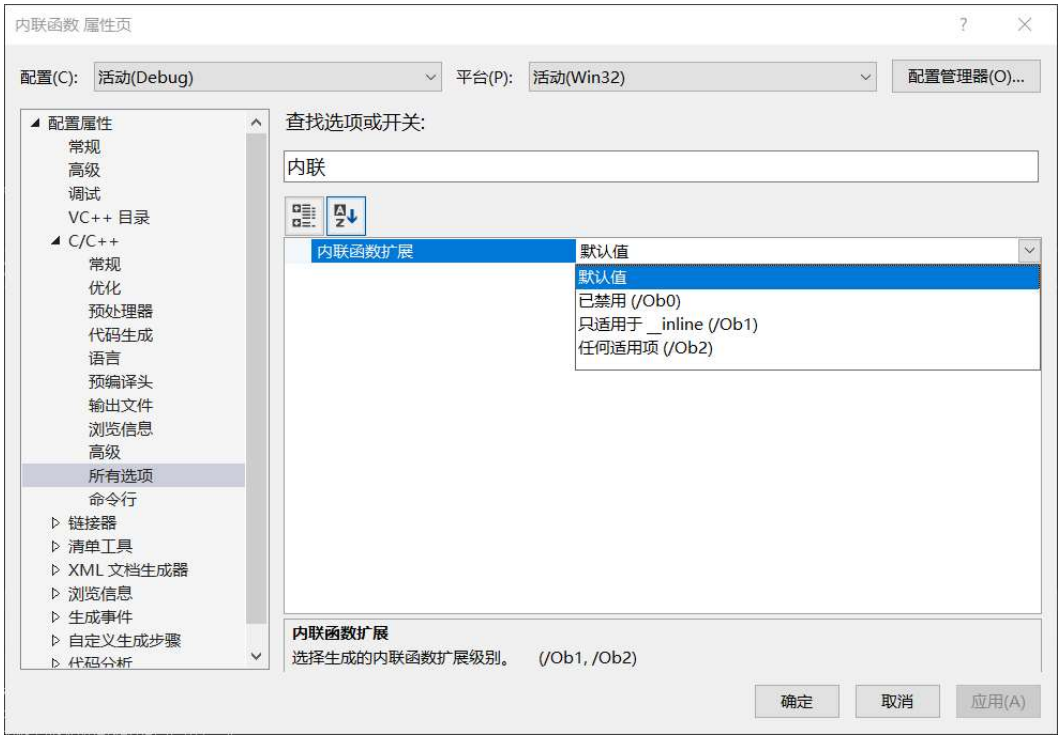
- 参数入栈
- 返回地址
- 调用函数
- 分配栈空间（局部变量）
- 执行函数体
- 撤销栈空间
- 回到原函数
- 参数出栈

短函数: 效率低，有类型检查，方便调试　　体积小 -- 时间换空间
宏　: 效率高但是其没有类型检查，不方便调试　体积大 -- 空间换时间

短函数和宏各有各的优缺点，C++提供 内联函数

内联函数：函数会像宏一样在调用点展开, 同时保留函数的类型检查
debug版不会展开, 方便调试

通过修改相应的属性，设置函数是否进行内联：



inline是建议编译器进行内联展开, 但是是否能展开成功, 看编译器

C++中支持内联函数，其目的是为了提高函数的执行效率，用关键字 inline 放在函数定义(注意是定义而非声明，下文继续讲到)的前面即可将函数指定为内联函数，内联函数通常就是将它在程序中的每个调用点上“内联地”展开，假设我们将 Max 定义为内联函数

```
inline int Max(int nNum1, int nNum2)
{
    return nNum1 > nNum2 ? nNum1 : nNum2;
}
```

调用时：cout << Max(666, 999) << endl;

在编译阶段，编译器会将其展开为：cout << (nNum1 > nNum2 ? nNum1 : nNum2) << endl;

从而消除了把 Max 写成函数的额外执行开销

内联函数是文件作用域, 内联函数声明和实现一般都应该放到头文件中
关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用

C++ inline函数是一种“用于实现的关键字”，而不是一种“用于声明的关键字”