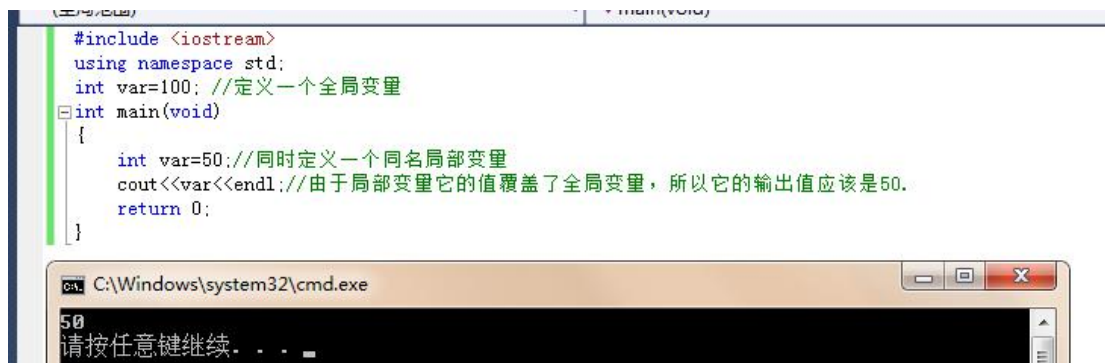


本章目标

➤ 【1】域运算符

(1) 域运算符是一种新增的运算符，用两个冒号来表示“::”。

对与局部变量同名的全局变量进行访问。接下来我们举个例子来说明这点：



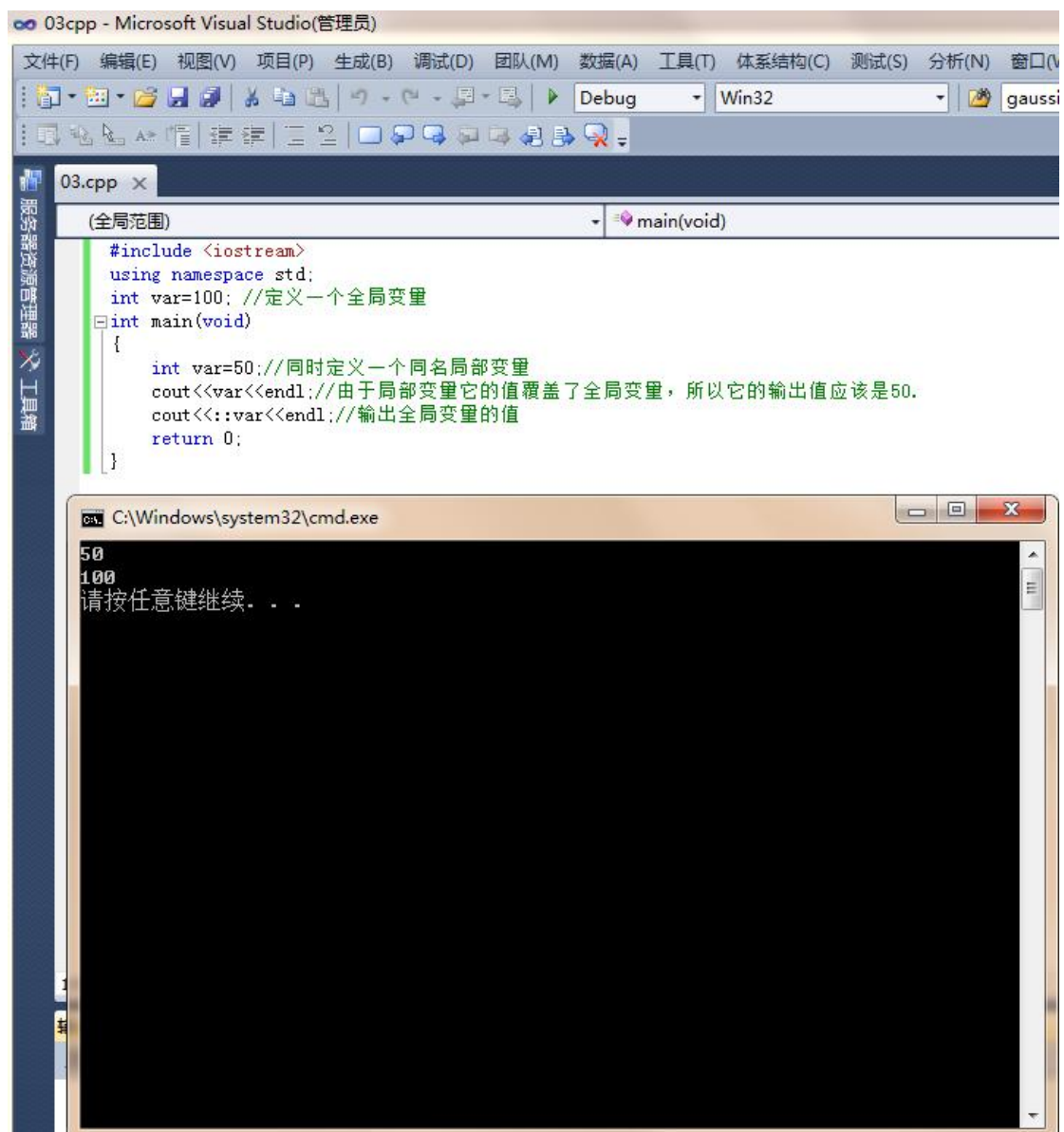
The screenshot shows a C++ IDE with a code editor and a command prompt window. The code in the editor defines a global variable 'var' with the value 100 and a local variable 'var' with the value 50 inside the 'main' function. The local variable's value is printed to the console. The command prompt window shows the output '50' and a prompt to press any key to continue.

```
#include <iostream>
using namespace std;
int var=100; //定义一个全局变量
int main(void)
{
    int var=50; //同时定义一个同名局部变量
    cout<<var<<endl; //由于局部变量它的值覆盖了全局变量，所以它的输出值应该是50.
    return 0;
}
```

C:\Windows\system32\cmd.exe

50
请按任意键继续. . .

那么我们如何访问全局变量呢？可以用运算符来访问，用两个“::”。



(2) 可以用于表示类的成员，它将在关于类的一节中详细介绍。

➤ 【2】new、delete 运算符

(1) new 运算符可以用于创建堆空间，类似 C 语言的 malloc()。

(2) 如果成功的话，返回新分配的内存的首地址。失败的话，一般也不会返回空指针 NULL，如果失败的话，直接抛出异常了。C++ 是支持异常的，因而我们不需要对返回的值与 NULL 进行比较，因为它直接抛出异常。

(3) 语法是这样的：

指针变量=new 数据类型; //表示创建一个数据类型的元素

指针变量=new 数据类型[长度 n]; //表示创建 n 个数据类型的元素

例如：


```
Int *p;p=new int;
```

```
Char *pStr=new char[50];
```

接下来我们举个例子：

我们申请了一个空间，整形占四个字节。内容是随机的，不确定的。


```
#include <iostream>
using namespace std;
int main(void)
{
    int *p=new int;
    cout<<*p<<endl; //输出内容是随机的，不确定的。
    return 0;
}
```



可是这个值是随机的。内容是不确定的。
堆栈我们自己分配的空间，我们要自己销毁它。


(全局范围) main(void)

```
#include <iostream>
using namespace std;
int main(void)
{
    int *p=new int; //分配一个整数空间，也就是4个字节
    cout<<*p<<endl; //输出内容是随机的，不确定的。
    int *p2=new int[10]; //分配连续的10个整数空间，也就是40个字节，那么这40个字节里面的内容也是随机的
    delete p; //销毁自己创建的堆栈空间
    delete[] p2; //销毁自己创建的数组堆栈空间，注意中括号，如果不带中括号，销毁的结果是不确定的
    return 0;
}
```



如果我们分配空间的时候，需要我们指定一个初始值，我们需要加一个圆括号：

```
#include <iostream>
using namespace std;
int main(void)
{
    //int *p=new int;    //分配一个整数空间，也就是4个字节
    int *p=new int(33); //分配一个整数空间，也就是4个字节，并给定一个初始值33
    cout<<*p<<endl;    //输出内容是随机的，不确定的。
    int *p2=new int[10]; //分配连续的10个整数空间，也就是40个字节，那么这40个字节里面的内容也是随机的
    delete p;           //销毁自己创建的堆栈空间
    delete[] p2;         //销毁自己创建的数组堆栈空间，注意中括号，如果不带中括号，销毁的结果是不确定的
    return 0;
}
```



这意味着我们分配了整数空间 4 个字节，并存放了整数 33.而这里数组是没法初始化的，如果需要初始化，需要通过循环语句逐个赋值。

Delete 运算符，可以释放堆空间。与 C 语言中的 Free 相对应。

➤ 语法：

delete 指针变量;

Delete [] 指针变量; //这里需要注意的是，new 的时候有中括号，delete 的时候也要有中括号。否则的话，释放的结果是不确定的。

例如

delete p;

delete [] pStr;

另外一个需要注意的是，中括号是分配的连续 n 个数据类型的空间，而圆括号是对一个空间进行初始化。

接下来我们看一下 new、delete 运算符做了几件事情。



New 做了两件事情，C 语言的 malloc 只做了一件事情，只做内存的分配，这边的内存分配相当于 operator new。

内存分配和调用构造函数的两个操作我们叫做 new operator（new 操作符）。

实际上 new 有三种用法：

- (1) new operator //分配内存，并且调用构造函数
- (2) Operator new //只分配内存，不调用构造函数
- (3) Placement new //不分配内存，紧急只是调用了拷贝构造函数。

关于 `operator new` 和 `placement new` 我们只做简单的了解，后面我们再做详细的讲解。我们今天只讲解 `new operator`，也就是分配内存和调用构造函数的用法。当然我们还看不到构造函数的调用，因为我们还没讲到类。我们需要知道的就是 `new operator` (`new` 运算符)。

➤ **Delete 释放一个对象**

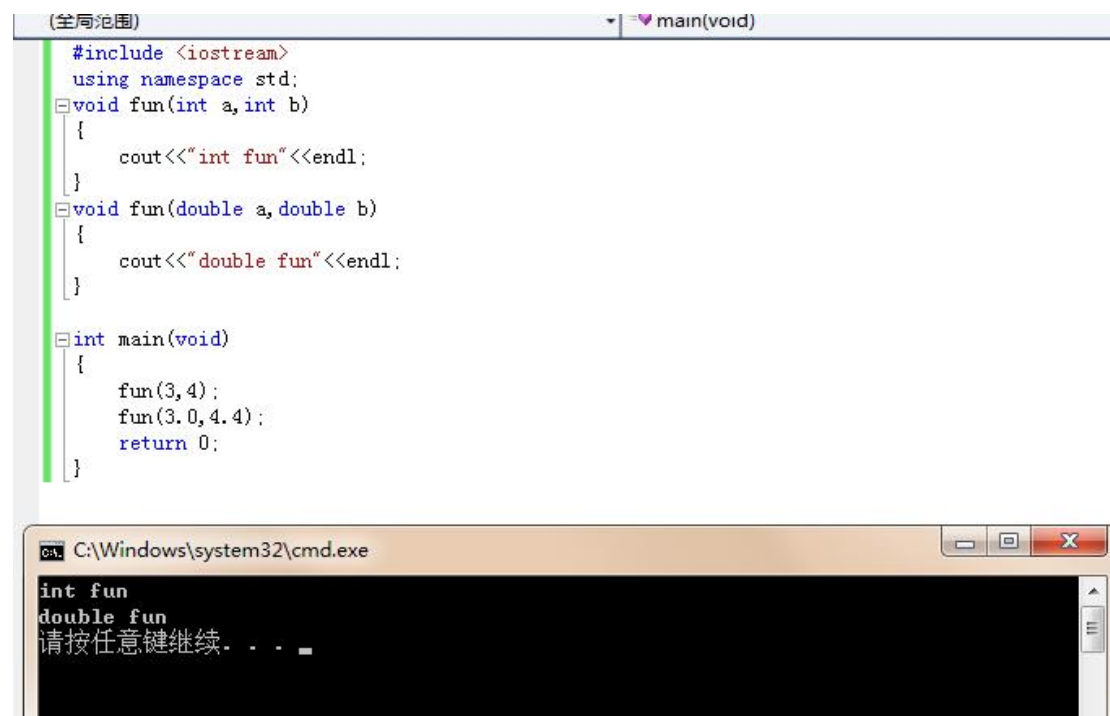
- (1) 调用析构函数
- (2) 释放内存 (`operator delete`)

【3】重载

- **重载是**:相同的作用域，函数的名称相同，而参数不同的，我们把他们叫做重载 `overload`
- 函数的重载又称为函数的多态性，这种多态性我们称为**静态的多态性**。而不是动态的多态，动态的多态是表现为派生类。这些函数的入口地址实在编译的时候就已经确定的，静态的多态实在编译的时候确定函数的入口地址（**静态多态也成为静态联编**），而动态的多态是在运行时确定函数的入口地址（**动态联编**）。静态的多态可以通过函数的重载来实现，**动态的多态是发生在基类和派生类之间，是通过虚函数来实现的**。这一点后面在讲解。
- **函数重载的不同形式**
 - (1) 形参的数量不同
 - (2) 形参的类型不同
 - (3) 形参的顺序不同
 - (4) 形参的数量和类型都不同。

总而言之就是参数不同。加下来我们举个例来说明这点：

它的第一个要求是作用域是相同的，我们都定义两个全局的函数（函数名相同，但是参数类型不同）：



The screenshot shows a C++ IDE with a code editor and a command prompt window. The code editor displays the following code:

```
#include <iostream>
using namespace std;
void fun(int a,int b)
{
    cout<<"int fun"<<endl;
}
void fun(double a,double b)
{
    cout<<"double fun"<<endl;
}
int main(void)
{
    fun(3,4);
    fun(3.0,4.4);
    return 0;
}
```

The command prompt window shows the output of the program:

```
int fun
double fun
请按任意键继续. . .
```

- 调用重载函数的时候，编译器通过检查实际参数的个数、类型和顺序来确定相应的被调函数。

下面是一些合法的重载的例子：

```
Int abs(int i);
Long abs(long l);
Double abs(double d);
```

下面是非法的重载的例子：.

```
Int abs(int i);
Long abs(int i);
Void abs(int i);
```

//如果返回类型不同，而函数名相同，形参也相同，则是不合法的，编译器会报“语法错误”。

➤ 【4】Name mangling 与 extern “C”

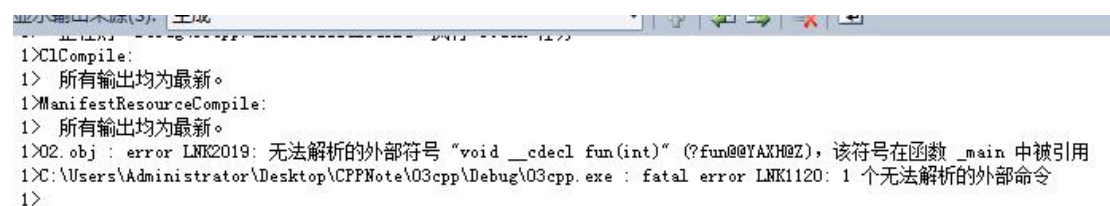
Name mangling 这里把它翻译为名字改编。为什么需要 name mangling 呢？是支持重载我们引入的一种技术。

也就是说如前面的例子一样：两个重载的函数名字是一样的，实际上系统会对名字进行改编，编译器看到的不是 fun()，而是其他的名字。那么到底是什么名字呢？

```
#include <iostream>
using namespace std;
void fun(int a, int b)
{
    cout<<"int fun"<<endl;
}
void fun(double a, double b)
{
    cout<<"double fun"<<endl;
}
void fun(int a);
int main(void)
{
    fun(3, 4);
    fun(3.0, 4.4);
    return 0;
}
```

三个函数都构成了重载，第三个函数只有声明，没有实现。

所以调用的时候会出错：



```
1>ClCompile:
1> 所有输出均为最新。
1>ManifestResourceCompile:
1> 所有输出均为最新。
1>O2.obj : error LNK2019: 无法解析的外部符号 "void __cdecl fun(int)" (?fun@@YAXH@Z), 该符号在函数 _main 中被引用
1>C:\Users\Administrator\Desktop\CPPNote\03cpp\Debug\03cpp.exe : fatal error LNK1120: 1 个无法解析的外部命令
1>
```

可以看到无法解析外部符号，可以看到这个符号不再是 fun(int)，而是？fun@@YAXH@Z。也就是说编译器对这个数进行名字的改编。为了支持重载，需要编译器对这些重载的函数进行改编。也就是说改编之后的函数名是不同的。

➤ Extern ”C “是实现 C 和 C++的混合编程


```

#ifdef __cplusplus
extern "C"
{
#endif
...
#ifdef __cplusplus
}
#endif

```

在 `void fun(int a)` 前面加 `extern "C"` 关键词，那么运行就会出现下下面的错误提示：

```

#include <iostream>
using namespace std;
void fun(int a, int b)
{
    cout<<"int fun"<<endl;
}
void fun(double a, double b)
{
    cout<<"double fun"<<endl;
}
extern "C" void fun(int a);
int main(void)
{
    fun(3, 4);
    fun(3.0, 4.4);
    fun(1);
    return 0;
}

```

输出

显示输出来源(S): 生成

```

1>ClCompile:
1> 02.cpp
1>ManifestResourceCompile:
1> 所有输出均为最新。
1>02.obj : error LNK2019: 无法解析的外部符号 _fun, 该符号在函数 _main 中被引用
1>C:\Users\Administrator\Desktop\CPPNote\03cpp\Debug\03cpp.exe : fatal error LNK1120: 1 个无法解析的外部命令
1>
1>生成失败。
1>
1>已用时间 00:00:02.16
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====

```

也就是说加上 `extern "C"` 就不进行名字改编。按照 C 语言的方式进行解析，我们知道 C 语言是不进行名字改编。也就不可以支持函数重载。

如果我们想要 C++ 编写的一些函数，能够被 C 语言调用的话，我们就需要在这个函数之前加

上 extern “C”

```
#ifdef __cplusplus
extern "C"
{
#endif
...
#ifdef __cplusplus
}
#endif
```

也就是说我们编写了好多的函数，如果我们需要 C 语言调用的话，我们就需要加上：extern “C”

```
#include <iostream>
using namespace std;
void fun(int a,int b)
{
    cout<<"int fun"<<endl;
}
void fun(double a,double b)
{
    cout<<"double fun"<<endl;
}
extern "C" void fun(int a)
{
    cout<<"xxx"<<endl;
}
extern "C" void fun(double a)
{
    cout<<"yyy"<<endl;
}

int main(void)
{
    fun(3,4);
    fun(3.0,4.4);
    fun(1);
    return 0;
}
```

100 %

输出


显示输出来源(S): 生成

1>InitializeBuildStatus:
1> 正在对“Debug\03cpp.unsuccessfulbuild”执行 Touch 任务。
1>ClCompile:
1> 02.cpp
1>c:\users\administrator\desktop\cppnote\03cpp\03cpp\02.cpp(16): error C2733: 不允许重载函数“fun”的第二个 C 链接
1> c:\users\administrator\desktop\cppnote\03cpp\03cpp\02.cpp(15) : 参见“fun”的声明
1>
1>生成失败。
1>
1>已用时间 00:00:00.45
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====

因为两个 extern “C”的名字是一样的，并且不允许重载，所以报错。


```
#include <iostream>
using namespace std;
void fun(int a,int b)
{
    cout<<"int fun"<<endl;
}
void fun(double a,double b)
{
    cout<<"double fun"<<endl;
}
extern "C" void fun(int a)
{
    cout<<"xxx"<<endl;
}
extern "C" void fun2(double a)
{
    cout<<"yyy"<<endl;
}

int main(void)
{
    fun(3,4);
    fun(3.0,4.4);
    fun(1);
    return 0;
}
```



因为有 extern “C”，就不参与函数重载了，所以输出 “xxx”，如果我们有很多 extern “C”的话。我们可以这么编写：括起来：

```

void fun(double a, double b)
{
    cout<<"double fun"<<endl;
}

//extern "C"表示不进行名字改编
extern "C"
{
    void fun(int a)
    {
        cout<<"xxxx"<<endl;
    }
    extern "C" void fun2(double a)
    {
        cout<<"yyyy"<<endl;
    }
}

int main(void)
{
    fun(3, 4);
    fun(3.3, 4.4);
    fun(3);
    return 0;
}

```

如果我们希望写的更好一些的话，可以写成这样：

```

void fun(double a, double b)
{
    cout<<"double fun"<<endl;
}

//extern "C"表示不进行名字改编
#ifdef __cplusplus
extern "C"
{
    #endif
void fun(int a)
{
    cout<<"xxx"<<endl;
}
extern "C" void fun2(double a)
{
    cout<<"yyy"<<endl;
}
#ifdef __cplusplus
}
#endif

int main(void)
{
    fun(3, 4);
    fun(3.3, 4.4);
    fun(3);
    return 0;
}

```



为什么我们要写`#ifdef __cplusplus`呢？因为很多时候，我们会将这段代码放到一个头文件当中，头文件实际上不给出实际的实现。只给出函数的声明。假设这个文件是 `a.h`，那么这个文件既可以被 C 语言使用，也可以被 C++ 使用。而他的实现是在 C++ 语言中提供的。也就是说 C++ 语言编写的程序，可以被 C 语言调用，只需要给他这个头文件。那么在 C++ 中也是可以用的。两者都可以使用，在 C 语言使用的时候，是不需要加上 `extern "C"` 的，因为 C 语言没有名字改编，在 C++ 中是含有 `extern "C"` 的，因为有名字改编。也就是 `extern "C"` 是实现 C 语言和 C++ 的混合编程。如果 C++ 上有定义这个宏，

```
#ifdef __cplusplus
```

我们就加上 `extern "C"` 。

//extern "C"表示不进行名字改编

```
#ifdef __cplusplus
extern "C"
{
#endif
void fun(int a);
void fun2(double a);
#ifdef __cplusplus
}
#endif
```

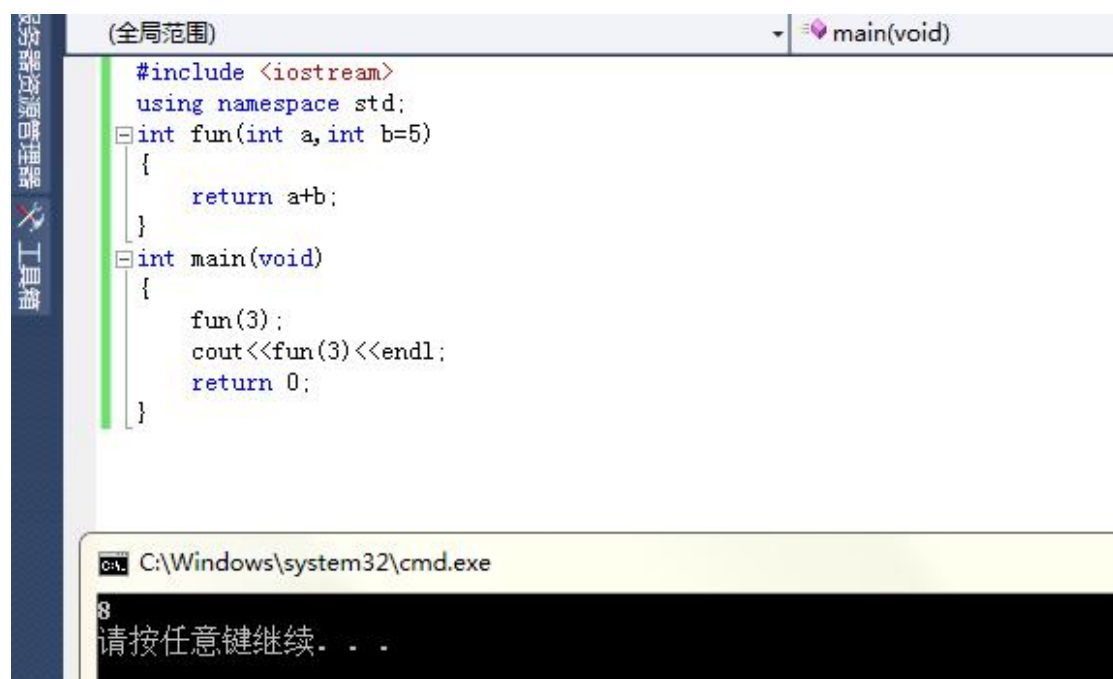
这是名字改编的作用，一个是可以支持重载，一个是支持 C 语言和 C++ 语言的混合编程。

➤ 【5】带默认参数的函数

(1) 函数在声明或定义的时候，我们可以给这些形参一些默认的值。

(2) 调用函数时，若没有给出实参，则按照指定的默认值进行工作。

我们来举一个例子吧：



The screenshot shows a C++ IDE with a source code editor and a command prompt window. The source code defines a function `fun` with a default parameter `b=5` and a `main` function that calls `fun(3)`. The command prompt shows the program running and displaying the output `8`.

```
(全局范围) main(void)
#include <iostream>
using namespace std;
int fun(int a, int b=5)
{
    return a+b;
}
int main(void)
{
    fun(3);
    cout<<fun(3)<<endl;
    return 0;
}
```

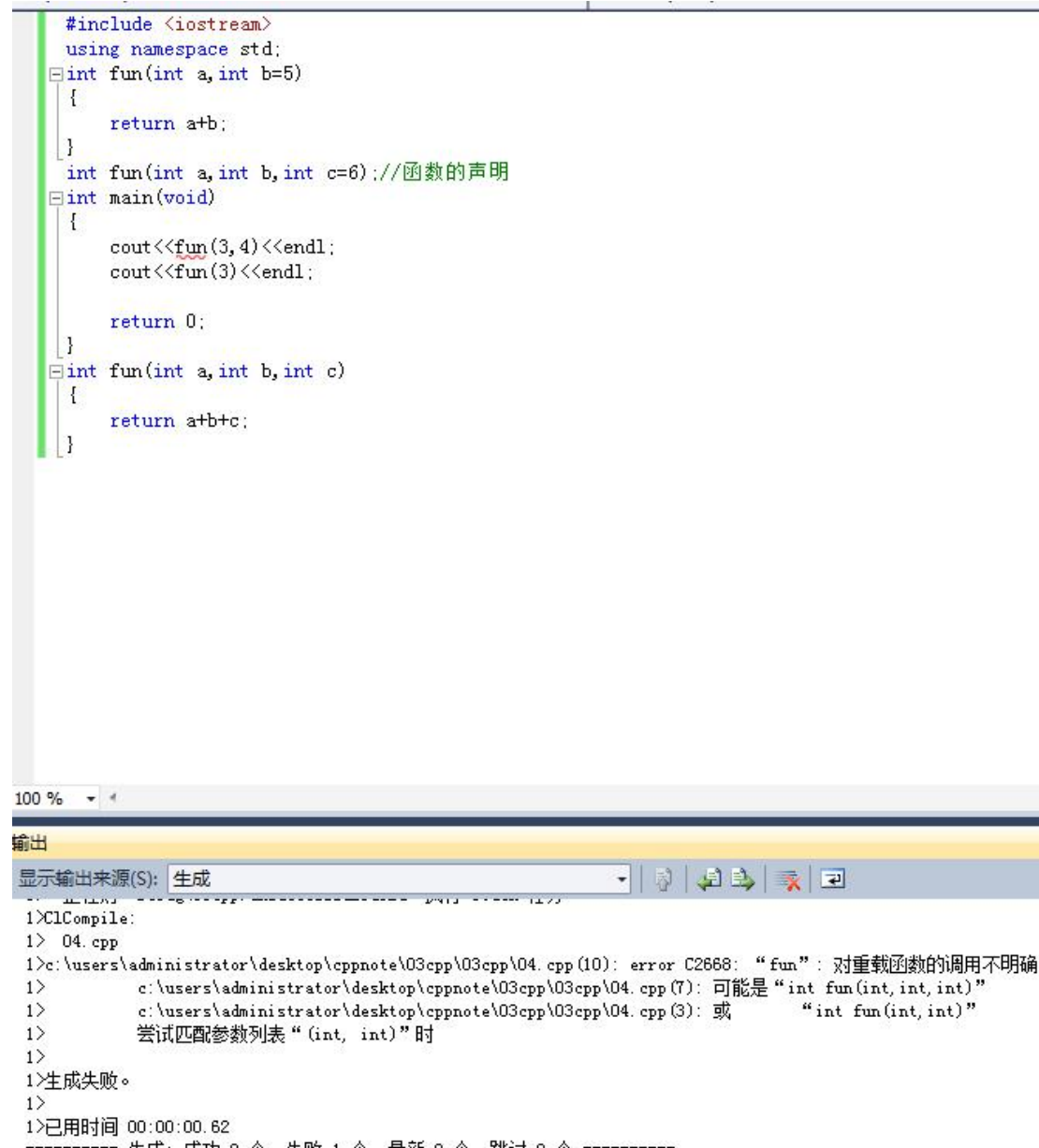
C:\Windows\system32\cmd.exe

8
请按任意键继续. . .

我们没有给出第一个参数的值，b 的值采用默认的值。

- 函数在没有声明时，在函数定义中指定形参的默认值。
- 函数在既有定义又有声明的时候，声明时指定后，定以后就不能再指定默认值。

这是什么意思呢？



```
#include <iostream>
using namespace std;
int fun(int a, int b=5)
{
    return a+b;
}
int fun(int a, int b, int c); //函数的声明
int main(void)
{
    cout<<fun(3,4)<<endl;
    cout<<fun(3)<<endl;

    return 0;
}
int fun(int a, int b, int c)
{
    return a+b+c;
}
```

输出

显示输出来源(S): 生成

```
1>C:\Compile:
1> 04.cpp
1>c:\users\administrator\desktop\cppnote\03cpp\03cpp\04.cpp (10): error C2668: "fun": 对重载函数的调用不明确
1>      c:\users\administrator\desktop\cppnote\03cpp\03cpp\04.cpp (7): 可能是 "int fun(int, int, int)"
1>      c:\users\administrator\desktop\cppnote\03cpp\03cpp\04.cpp (3): 或 "int fun(int, int)"
1>      尝试匹配参数列表 "(int, int)" 时
1>
1>生成失败。
1>
1>已用时间 00:00:00.62
----- 生成: 成功 0 个 失败 1 个 悬空 0 个 取消 0 个 -----
```

出现了一个新问题，函数出现了二义性。

```

#include <iostream>
using namespace std;
int fun(int a, int b=5)
{
    return a+b;
}
int fun(int a, int b, int c)://函数的声明
int main(void)
{
    cout<<fun(3,4)<<endl;
    cout<<fun(3)<<endl;

    return 0;
}
int fun(int a, int b, int c)
{
    return a+b+c;
}

```

C:\Windows\system32\cmd.exe

```

7
8
请按任意键继续. . .

```

(全局范围)

fun

```

#include <iostream>
using namespace std;
int fun(int a, int b=5)
{
    return a+b;
}
int fun(int a, int b, int c)://函数的声明
int main(void)
{
    cout<<fun(3,4)<<endl;
    cout<<fun(3)<<endl;
    cout<<fun(5,4,8)<<endl;

    return 0;
}
int fun(int a, int b, int c)
{
    return a+b+c;
}

```

C:\Windows\system32\cmd.exe

```

7
8
17
请按任意键继续. . .

```

- 默认值的定义必须遵循从右到左的顺序，如果某个形参没有默认值，则左边的参数就不

能有默认值。

Void func1(int a ,double b=4.5,int c=3); //合法

Void func1(int a=1,double b,int c=3); //不合法

- 函数调用的时候，实参与形参按照从左到右的顺序进行匹配。
- 我们使用重载函数中使用形参带默认值时，可能产生二义性，前面已经遇到过了。

```
int add(int x=5, int y=6);  
int add(int x=5, int y=6, int z=7);  
int main() {  
    int sum;  
    sum= add(10,20);  
    return 0;  
}
```

```
int add(int x, int y)  
{  
    return x+y;  
}  
int add(int x, int y, int z)  
{  
    return x+y;  
}
```

-
- 比如说这个例子，那么这个函数可以调用第一个函数，也可以调用第二个函数

sum=add(10, 20) 语句产生二义性，可以认为该语句是调用第一个函数，也可以是第二个，因此编译器不能确定调用的是哪一个函数。

-