

Lesson00---C++入门

【本节目标】

- 1. C++关键字
- 2. 命名空间
- 3. C++输入&输出
- 4. 缺省参数
- 5. 函数重载
- 6. 引用
- 7. 内联函数
- 8. auto关键字(C++11)
- 9. 基于范围的for循环(C++11)
- 10. 指针空值---nullptr(C++11)

0. 本节知识点安排目的

C++是在C的基础之上，容纳进去了面向对象编程思想，并增加了许多有用的库，以及编程范式等。熟悉C语言之后，对C++学习有一定的帮助，本章节主要目标：

1. 补充C语言语法的不足，以及C++是如何对C语言设计不合理的地方进行优化的，比如：作用域方面、IO方面、函数方面、指针方面、宏方面等。
2. 为后续类和对象学习打基础。

1. C++关键字(C++98)

C++总计63个关键字，C语言32个关键字

ps：下面我们只是看一下C++有多少关键字，不对关键字进行具体的讲解。后面我们学到以后再细讲。

asm	do	if	return	try	continue
auto	double	inline	short	typedef	for
bool	dynamic_cast	int	signed	typeid	public
break	else	long	sizeof	typename	throw
case	enum	mutable	static	union	wchar_t
catch	explicit	namespace	static_cast	unsigned	default
char	export	new	struct	using	friend
class	extern	operator	switch	virtual	register
const	false	private	template	void	true
const_cast	float	protected	this	volatile	while
delete	goto	reinterpret_cast			

2. 命名空间

在C/C++中，变量、函数和后面要学到的类都是大量存在的，这些变量、函数和类的名称将都存在于全局作用域中，可能会导致很多冲突。使用命名空间的目的是**对标识符的名称进行本地化**，以避免命名冲突或名字污染，namespace关键字的出现就是针对这种问题的。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int rand = 10;
5
6  // C语言没办法解决类似这样的命名冲突问题，所以C++提出了namespace来解决
7  int main()
8  {
9      printf("%d\n", rand);
10     return 0;
11 }
12
13 // 编译后报错: error C2365: "rand": 重定义; 以前的定义是“函数”

```

2.1 命名空间定义

定义命名空间，需要使用到**namespace关键字**，后面跟**命名空间的名字**，然后接一对{}即可，{}中即为命名空间的成员。

```

1  // bit是命名空间的名字，一般开发中是用项目名称做命名空间名。
2  // 我们上课用的是bit，大家下去以后自己练习用自己名字缩写即可，如张三: zs
3  // 1. 正常的命名空间定义
4  namespace bit
5  {

```

```

6      // 命名空间中可以定义变量/函数/类型
7      int rand = 10;
8
9      int Add(int left, int right)
10     {
11         return left + right;
12     }
13
14     struct Node
15     {
16         struct Node* next;
17         int val;
18     };
19 }
20
21 //2. 命名空间可以嵌套
22 // test.cpp
23 namespace N1
24 {
25     int a;
26     int b;
27     int Add(int left, int right)
28     {
29         return left + right;
30     }
31
32     namespace N2
33     {
34         int c;
35         int d;
36         int Sub(int left, int right)
37         {
38             return left - right;
39         }
40     }
41 }
42
43 //3. 同一个工程中允许存在多个相同名称的命名空间,编译器最后会合成同一个命名空间中。
44 // ps: 一个工程中的test.h和上面test.cpp中两个N1会被合并成一个
45
46 // test.h
47 namespace N1
48 {
49     int Mul(int left, int right)
50     {
51         return left * right;
52     }
53 }

```

注意：一个命名空间就定义了一个新的作用域，命名空间中的所有内容都局限于该命名空间中

2.2 命名空间使用

命名空间中成员该如何使用呢？比如：

```
1 namespace bit
2 {
3     // 命名空间中可以定义变量/函数/类型
4     int a = 0;
5     int b = 1;
6
7     int Add(int left, int right)
8     {
9         return left + right;
10    }
11
12    struct Node
13    {
14        struct Node* next;
15        int val;
16    };
17 }
18
19 int main()
20 {
21     // 编译报错: error C2065: "a": 未声明的标识符
22     printf("%d\n", a);
23     return 0;
24 }
```

命名空间的使用有三种方式：

- 加命名空间名称及作用域限定符

```
1 int main()
2 {
3     printf("%d\n", N::a);
4     return 0;
5 }
```

- 使用using将命名空间中某个成员引入

```
1 using N::b;
2 int main()
3 {
4     printf("%d\n", N::a);
5     printf("%d\n", b);
6     return 0;
7 }
```

- 使用using namespace 命名空间名称 引入

```

1 using namespace N;
2 int main()
3 {
4     printf("%d\n", N::a);
5     printf("%d\n", b);
6     Add(10, 20);
7     return 0;
8 }

```

3. C++输入&输出

新生儿会以自己独特的方式向这个崭新的世界打招呼，C++刚出来后，也算是一个新事物，



那C++是否也应该向这个美好的世界来声问候呢？我们来看下C++是如何来实现问候的。

```

1 #include<iostream>
2 // std是C++标准库的命名空间名，C++将标准库的定义实现都放到这个命名空间中
3 using namespace std;
4
5 int main()
6 {
7     cout<<"Hello world!!!"<<endl;
8     return 0;
9 }

```

说明：

1. 使用**cout标准输出对象(控制台)**和**cin标准输入对象(键盘)**时，必须**包含< iostream >头文件**以及按命名空间使用方法使用std。
2. cout和cin是全局的流对象，endl是特殊的C++符号，表示换行输出，他们都包含在包含< iostream >头文件中。
3. <<是**流插入运算符**，>>是**流提取运算符**。
4. 使用C++输入输出更方便，不需要像printf/scanf输入输出时那样，需要手动控制格式。C++的输入输出可以自动识别变量类型。
5. 实际上cout和cin分别是ostream和istream类型的对象，>>和<<也涉及运算符重载等知识，这些知识我们后续才会学习，所以我们这里只是简单学习他们的使用。后面我们还有一个章节更深入的学习IO流用法及原理。

注意：早期标准库将所有功能在全局域中实现，声明在.h后缀的头文件中，使用时只需包含对应头文件即可，后来将其实现在std命名空间下，为了和C头文件区分，也为了正确使用命名空间，规定C++头文件不带.h；旧编译器(vc 6.0)中还支持<iostream.h>格式，后续编译器已不支持，因此**推荐使用<iostream>+std**的方式。

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a;
7      double b;
8      char c;
9
10     // 可以自动识别变量的类型
11     cin>>a;
12     cin>>b>>c;
13
14     cout<<a<<endl;
15     cout<<b<<" "<<c<<endl;
16     return 0;
17 }
18 // ps: 关于cout和cin还有很多更复杂的用法, 比如控制浮点数输出精度, 控制整形输出进制格式等等。因为C++兼容C语言的用法, 这些又用得不是很多, 我们这里就不展开学习了。后续如果有需要, 我们再配合文档学习。

```

std命名空间的使用惯例:

std是C++标准库的命名空间, 如何展开std使用更合理呢?

1. 在日常练习中, 建议直接using namespace std即可, 这样就很方便。
2. using namespace std展开, 标准库就全部暴露出来了, 如果我们定义跟库重名的类型/对象/函数, 就存在冲突问题。该问题在日常练习中很少出现, 但是项目开发中代码较多、规模大, 就容易出现。所以建议在项目开发中使用, 像std::cout这样使用时指定命名空间 + using std::cout展开常用的库对象/类型等方式。

4. 缺省参数

4.1 缺省参数概念

缺省参数是**声明或定义函数时**为函数的**参数指定一个缺省值**。在调用该函数时, 如果没有指定实参则采用该形参的缺省值, 否则使用指定的实参。

```

1  void Func(int a = 0)
2  {
3      cout<<a<<endl;
4  }
5
6  int main()
7  {
8      Func();    // 没有传参时, 使用参数的默认值
9      Func(10); // 传参时, 使用指定的实参
10
11     return 0;
12 }

```

4.2 缺省参数分类

• 全缺省参数

```
1 void Func(int a = 10, int b = 20, int c = 30)
2 {
3     cout<<"a = "<<a<<endl;
4     cout<<"b = "<<b<<endl;
5     cout<<"c = "<<c<<endl;
6 }
```

• 半缺省参数

```
1 void Func(int a, int b = 10, int c = 20)
2 {
3     cout<<"a = "<<a<<endl;
4     cout<<"b = "<<b<<endl;
5     cout<<"c = "<<c<<endl;
6 }
```

注意:

1. 半缺省参数必须从右往左依次来给出，不能间隔着给
2. 缺省参数不能在函数声明和定义中同时出现

```
1 //a.h
2 void Func(int a = 10);
3
4 // a.cpp
5 void Func(int a = 20)
6 {}
7
8 // 注意：如果生命与定义位置同时出现，恰巧两个位置提供的值不同，那编译器就无法确定到底该用那个缺省值。
```

3. 缺省值必须是常量或者全局变量
4. C语言不支持（编译器不支持）

5. 函数重载

自然语言中，一个词可以有多重含义，人们可以通过上下文来判断该词真实的含义，即该词被重载了。

比如：以前有一个笑话，国有两个体育项目大家根本不用看，也不用担心。一个是乒乓球，一个是男足。前者是“谁也赢不了！”，后者是“谁也赢不了！”

5.1 函数重载概念

函数重载：是函数的一种特殊情况，C++允许在**同一作用域中**声明几个功能类似的**同名函数**，这些同名函数的**形参列表(参数个数 或 类型 或 类型顺序)不同**，常用来处理实现功能类似数据类型不同的问题。

```
1  #include<iostream>
2  using namespace std;
3
4  // 1、参数类型不同
5  int Add(int left, int right)
6  {
7      cout << "int Add(int left, int right)" << endl;
8
9      return left + right;
10 }
11
12 double Add(double left, double right)
13 {
14     cout << "double Add(double left, double right)" << endl;
15
16     return left + right;
17 }
18
19 // 2、参数个数不同
20 void f()
21 {
22     cout << "f()" << endl;
23 }
24
25 void f(int a)
26 {
27     cout << "f(int a)" << endl;
28 }
29
30 // 3、参数类型顺序不同
31 void f(int a, char b)
32 {
33     cout << "f(int a, char b)" << endl;
34 }
35
36 void f(char b, int a)
37 {
38     cout << "f(char b, int a)" << endl;
39 }
40
41 int main()
42 {
43     Add(10, 20);
44     Add(10.1, 20.2);
45
46     f();
47     f(10);
48
49     f(10, 'a');
50     f('a', 10);
```



```

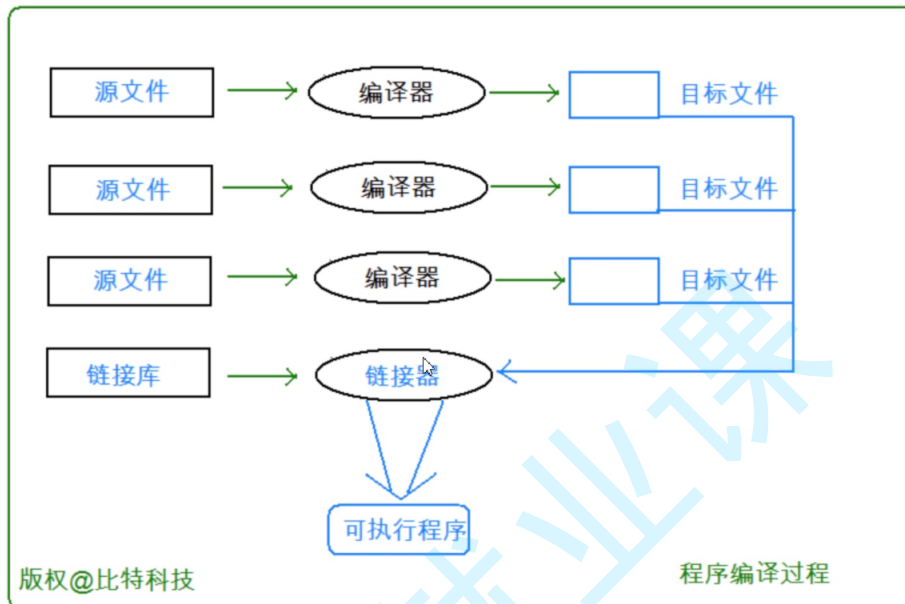
51
52     return 0;
53 }

```

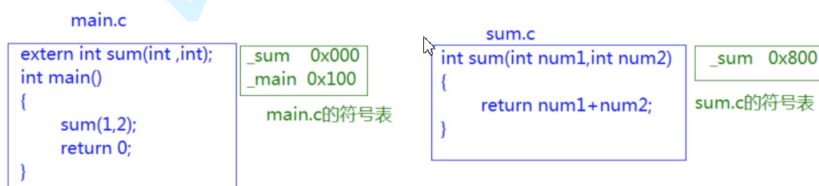
5.2 C++支持函数重载的原理--名字修饰(name Mangling)

为什么C++支持函数重载，而C语言不支持函数重载呢？

在C/C++中，一个程序要运行起来，需要经历以下几个阶段：预处理、编译、汇编、链接。



	预编译阶段 (*i) 预处理指令	编译 (*s) 语法分析 词法分析 语义分析 符号汇总	汇编 (生成可重定位目标文件*.o) 形成符号表 汇编指令->二进制指令----->test.o	链接
test.c				1.合并段表 2.符号表的合并和符号表的重定位
sum.c		----->sum.o	
隔离编译，一起链接				



1. 实际项目通常是由多个头文件和多个源文件构成，而通过C语言阶段学习的编译链接，我们可以知道，【当前a.cpp中调用了b.cpp中定义的Add函数时】，编译后链接前，a.o的目标文件中没有Add的函数地址，因为Add是在b.cpp中定义的，所以Add的地址在b.o中。那么怎么办呢？
2. 所以链接阶段就是专门处理这种问题，**链接器看到a.o调用Add，但是没有Add的地址，就会到b.o的符号表中找Add的地址，然后链接到一起。**（老师要带同学们回顾一下）
3. 那么链接时，面对Add函数，链接器会使用哪个名字去找呢？这里每个编译器都有自己的函数名修饰规则。

4. 由于Windows下vs的修饰规则过于复杂，而Linux下g++的修饰规则简单易懂，下面我们使用了g++演示了这个修饰后的名字。

5. 通过下面我们可以看出gcc的函数修饰后名字不变。而g++的函数修饰后变成【_Z+函数长度+函数名+类型首字母】。

o 采用C语言编译器编译后结果

```
int Add(int a, int b)
{
    return a + b;
}

void func(int a, double b, int* p)
{}

int main()
{
    Add(1, 2);
    func(1, 2, 0);

    return 0;
}
```

```
[xjh@localhost ~]$ ls
test.c
[xjh@localhost ~]$ gcc -o testc test.c
[xjh@localhost ~]$ ls
testc test.c
[xjh@localhost ~]$ objdump -S testc

00000000004004ed <Add>:
4004ed: 55             push    %rbp
4004ee: 48 89 e5       mov     %rsp,%rbp
4004f1: 89 7d fc       mov     %edi,-0x4(%rbp)
4004f4: 89 75 f8       mov     %esi,-0x8(%rbp)
4004f7: 8b 45 f8       mov     -0x8(%rbp),%eax
4004fa: 8b 55 fc       mov     -0x4(%rbp),%edx
4004fd: 01 d0         add     %edx,%eax
4004ff: 5d             pop     %rbp
400500: c3             retq

0000000000400501 <func>:
400501: 55             push    %rbp
400502: 48 89 e5       mov     %rsp,%rbp
400505: 89 7d fc       mov     %edi,-0x4(%rbp)
400508: f2 0f 11 45 f0 movsd   %xmm0,-0x10(%rbp)
40050d: 48 89 75 e8     mov     %rsi,-0x18(%rbp)
400511: 5d             pop     %rbp
400512: c3             retq
```

结论：在linux下，采用gcc编译完成后，函数名字的修饰没有发生改变。

o 采用C++编译器编译后结果

```
int Add(int a, int b)
{
    return a + b;
}

void func(int a, double b, int* p)
{}

int main()
{
    Add(1, 2);
    func(1, 2, 0);

    return 0;
}
```

```
[xjh@localhost ~]$ ls
testc test.cpp
[xjh@localhost ~]$ g++ -o testcpp test.cpp
[xjh@localhost ~]$ ls
testc testcpp test.c
[xjh@localhost ~]$ objdump -S testcpp

00000000004005ad <_Z3Addii>:
4005ad: 55             push    %rbp
4005ae: 48 89 e5       mov     %rsp,%rbp
4005b1: 89 7d fc       mov     %edi,-0x4(%rbp)
4005b4: 89 75 f8       mov     %esi,-0x8(%rbp)
4005b7: 8b 45 f8       mov     -0x8(%rbp),%eax
4005ba: 8b 55 fc       mov     -0x4(%rbp),%edx
4005bd: 01 d0         add     %edx,%eax
4005bf: 5d             pop     %rbp
4005c0: c3             retq

00000000004005c1 <_Z4funcidPi>:
4005c1: 55             push    %rbp
4005c2: 48 89 e5       mov     %rsp,%rbp
4005c5: 89 7d fc       mov     %edi,-0x4(%rbp)
4005c8: f2 0f 11 45 f0 movsd   %xmm0,-0x10(%rbp)
4005cd: 48 89 75 e8     mov     %rsi,-0x18(%rbp)
4005d1: 5d             pop     %rbp
4005d2: c3             retq
```

结论：在linux下，采用g++编译完成后，函数名字的修饰发生改变，编译器将函数参数类型信息添加到修改后的名字中。

o Windows下名字修饰规则

表 3-19

函数签名	修饰后名称
int func(int)	?func@@YAHH@Z
float func(float)	?func@@YAMM@Z
int C::func(int)	?func@C@@AAEHH@Z
int C::C2::func(int)	?func@C2@C@@AAEHH@Z
int N::func(int)	?func@N@@YAHH@Z
int N::C::func(int)	?func@C@N@@AAEHH@Z

我们以 int N::C::func(int) 这个函数签名来猜测 Visual C++ 的名称修饰规则（当然，你只须大概了解这个修饰规则就可以了）。修饰后名字由“?”开头，接着是函数名由“@”符号结尾的函数名；后面跟着由“@”结尾的类名“C”和名称空间“N”，再一个“@”表示函数的名称空间结束；第一个“A”表示函数调用类型为“__cdecl”（函数调用类型我们将在第4章详细介绍），接着是函数的参数类型及返回值，由“@”结束，最后由“Z”结尾。可以看到函数名、参数的类型和名称空间都被加入了修饰后名称，这样编译器和链接器就可以区别同名但不同参数类型或名字空间的函数，而不会导致 link 的时候函数多重定义。

对比Linux会发现，windows下vs编译器对函数名字修饰规则相对复杂难懂，但道理都是类似的，我们就不做细致的研究了。

【扩展学习：C/C++函数调用约定和名字修饰规则--有兴趣好奇的同学可以看看，里面有对vs下函数名修饰规则讲解】

[C/C++的调用约定](#)

6. 通过这里就理解了C语言没办法支持重载，因为同名函数没办法区分。而C++是通过函数修饰规则来区分，只要参数不同，修饰出来的名字就不一样，就支持了重载。
7. 如果两个函数函数名和参数是一样的，返回值不同是不构成重载的，因为调用时编译器没办法区分。

5.3 extern "C"

由于C和C++编译器对函数名字修饰规则的不同，在有些场景下可能会出问题，比如：

1. C++中调用C语言实现的静态库或者动态库，反之亦然
2. 多人协同开发时，有些人擅长用C语言，有些人擅长用C++

在这种混合模式下开发，由于C和C++编译器对函数名字修饰规则不同，可能会导致链接失败，在该种场景下，就需要使用extern "C"。在函数前加extern "C"，意思是告诉编译器，将该函数按照C语言规则来编译。

下面演示一个在C++工程中使用C语言静态库的例子：

[如何创建静态库及使用](#)

```

1 // 创建一个C语言的静态库
2
3 //////////////////////////////////////////////////calc.h/////////////////////////////////
4 #pragma once
5
6 /*
7 * 注意：
8 * 在实现该库时，并不知道将来使用该静态库的工程是C语言工程还是C++工程
9 * 为了能在C/C++工程中都能使用，函数声明时需加上extern "C"
10 *
11 * __cplusplus:是C++编译器中定义的宏，即用该宏来检测是C工程还是C++工程
12 *

```

```

13  * #ifdef __cplusplus
14  extern "C"
15  {
16  #endif
17
18      // 要导出函数的声明
19
20  #ifdef __cplusplus
21  }
22  #endif
23  作用: 如果是C++工程, 编译器已经定义_cplusplus宏, 编译时该宏是可以被识别的, 被声明的函数就被
extern "C"修饰了,
24      此时C++编译就知道, 静态库中的函数是按照C的方式编译的, 这样在链接时就会按照C的方式找函
数名字
25      如果是C工程, 编译器未定义_cplusplus宏, 编译时该宏无法被识别, 则条件编译就无效, 函数就
不会被extern "C"修饰
26  */
27
28
29  #ifdef __cplusplus
30  extern "C"
31  {
32  #endif
33
34      int Add(int left, int right);
35
36      int Sub(int left, int right);
37
38  #ifdef __cplusplus
39  }
40  #endif
41
42  //////////////////////////////////////////////////calc.c////////////////////////////////////
43  #include "calc.h"
44
45  int Add(int left, int right)
46  {
47      return left + right;
48  }
49
50  int Sub(int left, int right)
51  {
52      return left - right;
53  }
54
55
56  //////////////////////////////////////////////////
57  // 创建一个c++工程, 使用上面程序编程工程的静态库
58  #include <iostream>
59  using namespace std;
60
61  #include "../Debug/calc.h"
62
63  #pragma comment(lib, "../Debug/CalcLib.lib")

```

```

63
64 int main()
65 {
66     int ret = Add(10, 20);
67     cout << ret << endl;
68
69     ret = Sub(30, 20);
70     cout << ret << endl;
71     return 0;
72 }

```

如果在实现静态库时，函数没有使用extern "C"修饰：

```

1 #pragma once
2
3 int Add(int left, int right);
4 int Sub(int left, int right);

```

就会报无法找到xxx函数的链接错误。

```

1 1>TestCalc.cpp
2 1>TestCalc.obj : error LNK2019: 无法解析的外部符号 "int __cdecl Add(int,int)" (?
Add@@YAHHH@Z), 函数 _main 中引用了该符号
3 1>TestCalc.obj : error LNK2019: 无法解析的外部符号 "int __cdecl Sub(int,int)" (?
Sub@@YAHHH@Z), 函数 _main 中引用了该符号
4 1>D:\WorkStations\Gitee\cppLesson\TestCalc\Debug\TestCalc.exe : fatal error LNK1120: 2
个无法解析的外部命令

```

【面试题】

1. 下面两个函数能形成函数重载吗？有问题吗或者什么情况下会出问题？

```

1 void Func(int a = 10)
2 {
3     cout<<"void Func(int a = 10)"<<endl;
4 }
5
6 void Func(int a)
7 {
8     cout<<"void Func(int a)"<<endl;
9 }

```

2. C语言中为什么不能支持函数重载？
3. C++中函数重载底层是怎么处理的？
4. C++中能否将一个函数按照C的风格来编译？

6. 引用

6.1 引用概念

引用不是新定义一个变量，而是给已存在变量取了一个别名，编译器不会为引用变量开辟内存空间，它和它引用的变量共用同一块内存空间。

比如：李逵，在家称为“铁牛”，江湖上人称“黑旋风”。



类型& 引用变量名(对象名) = 引用实体;

```
1 void TestRef()
2 {
3     int a = 10;
4     int& ra = a; //<====定义引用类型
5
6     printf("%p\n", &a);
7     printf("%p\n", &ra);
8 }
```

注意：引用类型必须和引用实体是同种类型的

6.2 引用特性

1. 引用在定义时必须初始化
2. 一个变量可以有多个引用
3. 引用一旦引用一个实体，再不能引用其他实体

```
1 void TestRef()
2 {
3     int a = 10;
4     // int& ra;    // 该条语句编译时会出错
5     int& ra = a;
6     int& rra = a;
7     printf("%p %p %p\n", &a, &ra, &rra);
8 }
```

6.3 常引用

```

1 void TestConstRef()
2 {
3     const int a = 10;
4     //int& ra = a;    // 该语句编译时会出错, a为常量
5     const int& ra = a;
6     // int& b = 10;   // 该语句编译时会出错, b为常量
7     const int& b = 10;
8     double d = 12.34;
9     //int& rd = d;    // 该语句编译时会出错, 类型不同
10    const int& rd = d;
11 }

```

6.4 使用场景

1. 做参数

```

1 void Swap(int& left, int& right)
2 {
3     int temp = left;
4     left = right;
5     right = temp;
6 }

```

2. 做返回值

```

1 int& Count()
2 {
3     static int n = 0;
4     n++;
5     // ...
6     return n;
7 }

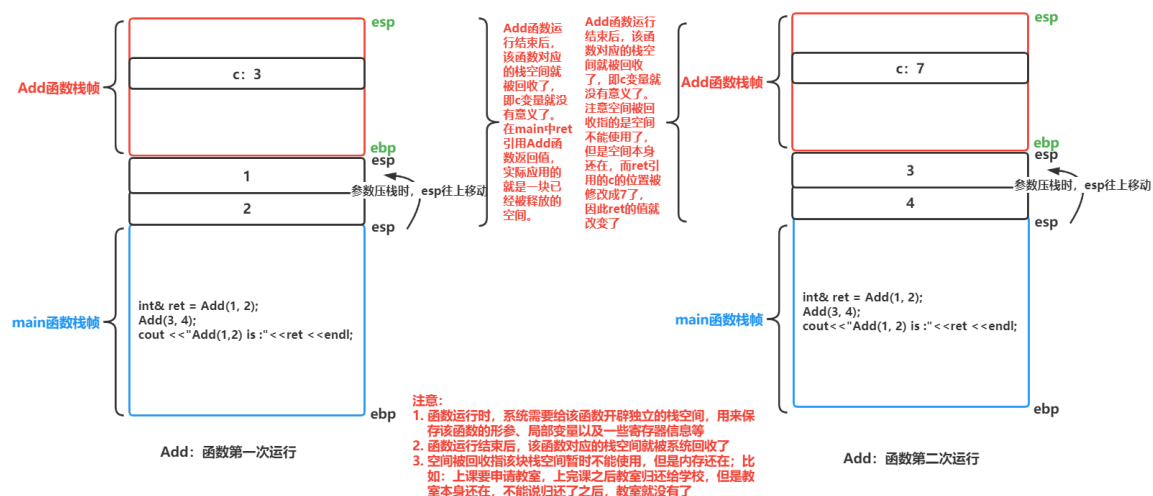
```

下面代码输出什么结果? 为什么?

```

1 int& Add(int a, int b)
2 {
3     int c = a + b;
4     return c;
5 }
6
7 int main()
8 {
9     int& ret = Add(1, 2);
10    Add(3, 4);
11    cout << "Add(1, 2) is : "<< ret << endl;
12    return 0;
13 }

```



注意：如果函数返回时，出了函数作用域，如果返回对象还在(还没还给系统)，则可以使用引用返回，如果已经还给系统了，则必须使用传值返回。

6.5 传值、传引用效率比较

以值作为参数或者返回值类型，在传参和返回期间，函数不会直接传递实参或者将变量本身直接返回，而是传递实参或者返回变量的一份临时的拷贝，因此用值作为参数或者返回值类型，效率是非常低下的，尤其是当参数或者返回值类型非常大时，效率就更低。

```

1  #include <time.h>
2  struct A{ int a[10000]; };
3  void TestFunc1(A a){}
4
5  void TestFunc2(A& a){}
6
7  void TestRefAndValue()
8  {
9      A a;
10     // 以值作为函数参数
11     size_t begin1 = clock();
12     for (size_t i = 0; i < 10000; ++i)
13         TestFunc1(a);
14     size_t end1 = clock();
15
16     // 以引用作为函数参数
17     size_t begin2 = clock();
18     for (size_t i = 0; i < 10000; ++i)
19         TestFunc2(a);
20     size_t end2 = clock();
21
22     // 分别计算两个函数运行结束后的时间
23     cout << "TestFunc1(A)-time:" << end1 - begin1 << endl;
24     cout << "TestFunc2(A&)-time:" << end2 - begin2 << endl;
25 }

```


6.5.2 值和引用的作为返回值类型的性能比较

```
1  #include <time.h>
2  struct A{ int a[10000]; };
3
4  A a;
5  // 值返回
6  A TestFunc1() { return a;}
7  // 引用返回
8  A& TestFunc2(){ return a;}
9
10 void TestReturnByRefOrValue()
11 {
12     // 以值作为函数的返回值类型
13     size_t begin1 = clock();
14     for (size_t i = 0; i < 100000; ++i)
15         TestFunc1();
16     size_t end1 = clock();
17
18     // 以引用作为函数的返回值类型
19     size_t begin2 = clock();
20     for (size_t i = 0; i < 100000; ++i)
21         TestFunc2();
22     size_t end2 = clock();
23
24     // 计算两个函数运算完成之后的时间
25     cout << "TestFunc1 time:" << end1 - begin1 << endl;
26     cout << "TestFunc2 time:" << end2 - begin2 << endl;
27 }
```

通过上述代码的比较，发现传值和指针在作为传参以及返回值类型上效率相差很大。

6.6 引用和指针的区别

在语法概念上引用就是一个别名，没有独立空间，和其引用实体共用同一块空间。

```
1  int main()
2  {
3      int a = 10;
4      int& ra = a;
5
6      cout<<"a = "<<&a<<endl;
7      cout<<"ra = "<<&ra<<endl;
8
9      return 0;
10 }
```

在底层实现上实际是有空间的，因为引用是按照指针方式来实现的。

```

1  int main()
2  {
3      int a = 10;
4
5      int& ra = a;
6      ra = 20;
7
8      int* pa = &a;
9      *pa = 20;
10
11     return 0;
12 }

```

我们来看下引用和指针的汇编代码对比：

int a = 10;	int a = 10;
mov dword ptr [a], 0Ah	mov dword ptr [a], 0Ah
int& ra = a;	int* pa = &a;
lea eax, [a]	lea eax, [a]
mov dword ptr [ra], eax	mov dword ptr [pa], eax
ra = 20;	*pa = 20;
mov eax, dword ptr [ra]	mov eax, dword ptr [pa]
mov dword ptr [eax], 14h	mov dword ptr [eax], 14h

引用和指针的不同点：

1. 引用概念上定义一个变量的别名，指针存储一个变量地址。
2. 引用在定义时必须初始化，指针没有要求
3. 引用在初始化时引用一个实体后，就**不能再引用其他实体**，而指针可以在任何时候指向任何一个同类型实体
4. **没有NULL引用**，但有NULL指针
5. 在sizeof中含义不同：引用结果为引用类型的大小，但指针始终是地址空间所占字节个数(32位平台下占4个字节)
6. 引用自加即引用的实体增加1，指针自加即指针向后偏移一个类型的大小
7. **有多级指针，但是没有多级引用**
8. 访问实体方式不同，**指针需要显式解引用**，引用编译器自己处理
9. **引用比指针使用起来相对更安全**

7. 内联函数

7.1 概念

以**inline**修饰的函数叫做内联函数，编译时C++编译器会在调用内联函数的地方展开，没有函数调用建立栈帧的开销，内联函数提升程序运行的效率。

<pre> int Add(int left, int right) { return left + right; } int main() { int ret = 0; ret = Add(1, 2); return 0; } </pre>	<pre> int ret = 0; mov dword ptr [ret], 0 ret = Add(1, 2); push 2 push 1 call Add @12C107Dh add esp, 8 mov dword ptr [ret], eax </pre>
--	---

如果在上述函数前增加inline关键字将其改成内联函数，在编译期间编译器会用函数体替换函数的调用。

查看方式：

1. 在release模式下，查看编译器生成的汇编代码中是否存在call Add
2. 在debug模式下，需要对编译器进行设置，否则不会展开(因为debug模式下，编译器默认不会对代码进行优化，以下给出vs2013的设置方式)



<pre> inline int Add(int left, int right) { return left + right; } int main() { int ret = 0; ret = Add(1, 2); return 0; } </pre>	<pre> int ret = 0; mov dword ptr [ret], 0 ret = Add(1, 2); mov eax, 1 add eax, 2 mov dword ptr [ret], eax </pre>
---	--

7.2 特性

1. inline是一种以空间换时间的做法，如果编译器将函数当成内联函数处理，在编译阶段，会用函数体替换函数调用，缺陷：可能会使目标文件变大，优势：少了调用开销，提高程序运行效率。

2. **inline**对于编译器而言只是一个建议，不同编译器关于**inline**实现机制可能不同，一般建议：将函数规模较小(即函数不是很长，具体没有准确的说法，取决于编译器内部实现)、**不是递归、频繁调用的函数**采用**inline**修饰，否则编译器会忽略**inline**特性。下图为《C++prime》第五版关于**inline**的建议：



内联说明只是向编译器发出的一个请求，编译器可以选择忽略这个请求。

一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数。很多编译器都不支持内联递归函数，而且一个 75 行的函数也不大可能在调用点内联地展开。

3. **inline**不建议声明和定义分离，分离会导致链接错误。因为**inline**被展开，就没有函数地址了，链接就会找不到。

```
1 // F.h
2 #include <iostream>
3 using namespace std;
4
5 inline void f(int i);
6
7 // F.cpp
8 #include "F.h"
9 void f(int i)
10 {
11     cout << i << endl;
12 }
13
14 // main.cpp
15 #include "F.h"
16 int main()
17 {
18     f(10);
19     return 0;
20 }
21
22 // 链接错误: main.obj : error LNK2019: 无法解析的外部符号 "void __cdecl f(int)" (?f@@YAXH@Z), 该符号在函数 _main 中被引用
```

【面试题】

宏的优缺点？

优点：

- 1.增强代码的复用性。
- 2.提高性能。

缺点：

- 1.不方便调试宏。（因为预编译阶段进行了替换）
- 2.导致代码可读性差，可维护性差，容易误用。

3.没有类型安全的检查。

C++有哪些技术替代宏?

1. 常量定义 换用const enum
2. 短小函数定义 换用内联函数

8. auto关键字(C++11)

8.1 类型别名思考

随着程序越来越复杂，程序中用到的类型也越来越复杂，经常体现在：

1. 类型难于拼写
2. 含义不明确导致容易出错

```
1  #include <string>
2  #include <map>
3
4  int main()
5  {
6      std::map<std::string, std::string> m{ { "apple", "苹果" }, { "orange", "橙子" },
7                                             { "pear", "梨" } };
8      std::map<std::string, std::string>::iterator it = m.begin();
9      while (it != m.end())
10     {
11         //....
12     }
13     return 0;
14 }
```

`std::map<std::string, std::string>::iterator` 是一个类型，但是该类型太长了，特别容易写错。聪明的同学可能已经想到：可以通过typedef给类型取别名，比如：

```
1  #include <string>
2  #include <map>
3
4  typedef std::map<std::string, std::string> Map;
5  int main()
6  {
7      Map m{ { "apple", "苹果" }, { "orange", "橙子" }, { "pear", "梨" } };
8      Map::iterator it = m.begin();
9      while (it != m.end())
10     {
11         //....
12     }
13     return 0;
14 }
```

使用typedef给类型取别名确实可以简化代码，但是typedef有会遇到新的难题：

```

1  typedef char* pstring;
2
3  int main()
4  {
5      const pstring p1;    // 编译成功还是失败?
6      const pstring* p2;  // 编译成功还是失败?
7      return 0;
8  }

```

在编程时，常常需要把表达式的值赋值给变量，这就要求在声明变量的时候清楚地知道表达式的类型。然而有时候要做到这点并非那么容易，因此C++11给auto赋予了新的含义。

8.2 auto简介

在早期C/C++中auto的含义是：使用**auto**修饰的变量，是具有自动存储器的局部变量，但遗憾的是一直没有人去使用它，大家可思考下为什么？

C++11中，标准委员会赋予了auto全新的含义即：**auto不再是一个存储类型指示符，而是作为一个新的类型指示符来指示编译器，auto声明的变量必须由编译器在编译时期推导而得。**

```

1  int TestAuto()
2  {
3      return 10;
4  }
5
6  int main()
7  {
8      int a = 10;
9      auto b = a;
10     auto c = 'a';
11     auto d = TestAuto();
12
13     cout << typeid(b).name() << endl;
14     cout << typeid(c).name() << endl;
15     cout << typeid(d).name() << endl;
16
17
18     //auto e; 无法通过编译，使用auto定义变量时必须对其进行初始化
19     return 0;
20 }

```

【注意】

使用auto定义变量时必须对其进行初始化，在编译阶段编译器需要根据初始化表达式来推导auto的实际类型。因此auto并非是一种“类型”的声明，而是一个类型声明时的“占位符”，编译器在编译期会将auto替换为变量实际的类型。

8.3 auto的使用细则

1. auto与指针和引用结合起来使用

用auto声明指针类型时，用auto和auto*没有任何区别，但用auto声明引用类型时则必须加&

```

1  int main()
2  {
3      int x = 10;
4      auto a = &x;
5      auto* b = &x;
6      auto& c = x;
7
8      cout << typeid(a).name() << endl;
9      cout << typeid(b).name() << endl;
10     cout << typeid(c).name() << endl;
11
12     *a = 20;
13     *b = 30;
14     c = 40;
15
16     return 0;
17 }

```

2. 在同一行定义多个变量

当在同一行声明多个变量时，这些变量必须是相同的类型，否则编译器将会报错，因为编译器实际只对第一个类型进行推导，然后用推导出来的类型定义其他变量。

```

1  void TestAuto()
2  {
3      auto a = 1, b = 2;
4      auto c = 3, d = 4.0; // 该行代码会编译失败，因为c和d的初始化表达式类型不同
5  }

```

8.3 auto不能推导的场景

1. auto不能作为函数的参数

```

1  // 此处代码编译失败，auto不能作为形参类型，因为编译器无法对a的实际类型进行推导
2  void TestAuto(auto a)
3  {}

```

2. auto不能直接用来声明数组

```

1  void TestAuto()
2  {
3      int a[] = {1, 2, 3};
4      auto b[] = {4, 5, 6};
5  }

```

3. 为了避免与C++98中的auto发生混淆，C++11只保留了auto作为类型指示符的用法

4. auto在实际中最常见的优势用法就是跟以后会讲到的C++11提供的新式for循环，还有lambda表达式等进行配合使用。

9. 基于范围的for循环(C++11)

9.1 范围for的语法

在C++98中如果要遍历一个数组，可以按照以下方式进行：

```
1 void TestFor()
2 {
3     int array[] = { 1, 2, 3, 4, 5 };
4     for (int i = 0; i < sizeof(array) / sizeof(array[0]); ++i)
5         array[i] *= 2;
6
7     for (int* p = array; p < array + sizeof(array) / sizeof(array[0]); ++p)
8         cout << *p << endl;
9 }
```

对于一个有范围的集合而言，由程序员来说明循环的范围是多余的，有时候还会容易犯错误。因此C++11中引入了基于范围的for循环。**for循环后的括号由冒号“:”分为两部分：第一部分是范围内用于迭代的变量，第二部分则表示被迭代的范围。**

```
1 void TestFor()
2 {
3     int array[] = { 1, 2, 3, 4, 5 };
4     for(auto& e : array)
5         e *= 2;
6
7     for(auto e : array)
8         cout << e << " ";
9
10    return 0;
11 }
```

注意：与普通循环类似，可以用continue来结束本次循环，也可以用break来跳出整个循环。

9.2 范围for的使用条件

1. for循环迭代的范围必须是确定的

对于数组而言，就是数组中第一个元素和最后一个元素的范围；对于类而言，应该提供begin和end的方法，begin和end就是for循环迭代的范围。

注意：以下代码就有问题，因为for的范围不确定

```
1 void TestFor(int array[])
2 {
3     for(auto& e : array)
4         cout << e << endl;
5 }
```

2. 迭代的对象要实现++和==的操作。(关于迭代器这个问题，以后会讲，现在提一下，没办法讲清楚，现在大家了解一下就可以了)

10. 指针空值nullptr(C++11)

10.1 C++98中的指针空值

在良好的C/C++编程习惯中，声明一个变量时最好给该变量一个合适的初始值，否则可能会出现不可预料的错误，比如未初始化的指针。如果一个指针没有合法的指向，我们基本都是按照如下方式对其进行初始化：

```
1 void TestPtr()
2 {
3     int* p1 = NULL;
4     int* p2 = 0;
5
6     // .....
7 }
```

NULL实际是一个宏，在传统的C头文件(stddef.h)中，可以看到如下代码：

```
1 #ifndef NULL
2 #ifdef __cplusplus
3 #define NULL    0
4 #else
5 #define NULL    ((void *)0)
6 #endif
7 #endif
```

可以看到，**NULL可能被定义为字面常量0，或者被定义为无类型指针(void*)的常量**。不论采取何种定义，在使用空值的指针时，都不可避免的会遇到一些麻烦，比如：

```
1 void f(int)
2 {
3     cout<<"f(int)"<<endl;
4 }
5
6 void f(int*)
7 {
8     cout<<"f(int*)"<<endl;
9 }
10
11 int main()
12 {
13     f(0);
14     f(NULL);
15     f((int*)NULL);
16     return 0;
17 }
```

程序本意是想通过f(NULL)调用指针版本的f(int*)函数，但是由于NULL被定义成0，因此与程序的初衷相悖。

在C++98中，字面常量0既可以是一个整形数字，也可以是无类型的指针(void*)常量，但是编译器默认情况下将其看成是一个整形常量，如果要将其按照指针方式来使用，必须对其进行强转(void *)0。

注意：

1. 在使用nullptr表示指针空值时，不需要包含头文件，因为nullptr是C++11作为新关键字引入的。
2. 在C++11中，sizeof(nullptr) 与 sizeof((void*)0)所占的字节数相同。
3. 为了提高代码的健壮性，在后续表示指针空值时建议最好使用nullptr。

比特就业课