

<C++> 一、入门

在学习完 C 语言的基础上，继续开始 C++ 的学习。

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。

C++ 兼容 C 语言语法

C/C++

```
1 #include<stdio.h>
2 int main()
3 {
4     printf("hello world\n");
5     return 0;
6 }
7
8 #include <iostream>
9 using namespace std;
10 int main()
11 {
12     cout << "hello world" << endl;    //hello world
13     printf("hello world\n");    //hello world
14     return 0;
15 }
```

1.C++ 关键字(C++98)

C++ 总计 63 个关键字，C 语言 32 个关键字

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			

这些关键字没必要记住，以后在学习的过程中，遇到一个学一个即可

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。

C/C++

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 // 我们自己定义和库里面的名字冲突
4 // 项目组，多个人之间定义的名字冲突
5 int rand = 10; // rand为C语言stdlib里面的函数名，不能定义
6
7 // 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
8 int main()
9 {
10     printf("%d\n", rand);
11
12     return 0;
13 }
14 // 编译后报错：error C2365: "rand": 重定义；以前的定义是“函数”
```

域：全局域和局部域

C/C++

```
1 //域
2 //局部域/全局域：1、使用 2、生命周期
3 int a = 2;
4
5 void f1()
6 {
7     int a = 0;
8     printf("%d\n", a); //0
9     printf("%d\n", ::a); // 2 ::域作用限定符,表示全局域
10 }
11
12 int main()
13 {
14     printf("%d\n", a); //2
15     f1();
16
17     return 0;
18 }
```

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 // phw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 //1. 正常的命名空间定义
3 namespace phw
4 {
5     // 命名空间中可以定义变量/函数/类型
6     int rand = 10;
7     int Add(int left, int right)
8     {
9         return left + right;
10    }
11    struct Node
12    {
13        struct Node* next;
14        int val;
15    };
16 }
17
18 //2. 命名空间可以嵌套
19 // test.cpp
20 namespace N1
21 {
22     int a;
23     int b;
24     int Add(int left, int right)
25     {
26         return left + right;
27     }
28     namespace N2
29     {
30         int c;
31         int d;
32         int Sub(int left, int right)
33         {
34             return left - right;
35         }
36     }
37 }
38
39 //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。
```

```

40 //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
41 // test.h
42 namespace N1
43 {
44     int Mul(int left, int right)
45     {
46         return left * right;
47     }
48 }

```

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 命名空间的使用

C/C++

```

1 namespace phw
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     int Add(int left, int right)
7     {
8         return left + right;
9     }
10    struct Node
11    {
12        struct Node* next;
13        int val;
14    };
15 }
16
17 int main()
18 {
19     // 编译报错：error C2065: "a": 未声明的标识符
20     printf("%d\n", a);
21     return 0;
22 }

```

命名空间的使用有三种方式:

C/C++

```
1  //1.加命名空间名称及作用域限定符
2  int main()
3  {
4      printf("%d\n", phw::a);
5      return 0;
6  }
7
8  //2.使用using将命名空间中某个成员引入
9  using phw::b;
10 int main()
11 {
12     printf("%d\n", phw::a);
13     printf("%d\n", b);
14     return 0;
15 }
16
17 //3.使用using namespace 命名空间名称引入
18 using namespace phw;
19 int main()
20 {
21     printf("%d\n", N::a);
22     printf("%d\n", b);
23     Add(10, 20);
24     return 0;
25 }
```

3.C++ 输入 & 输出

C/C++

```
1 #include<iostream>
2 using namespace std;
3 // 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
4 int main()
5 {
6     // << 流插入 endl会清楚缓冲区
7     cout << "Hello World" << endl;
8     cout << "Hello World" << '\n';
9     return 0;
10 }
```

说明:

使用 cout 和 cin，必须包含#include<iostream> 文件

cout 和 cin 可以自动识别变量类型

<< 是流插入运算符，>> 是流提取运算符

实际上 cout 和 cin 分别是 ostream 和 istream 类型的对象

C++ 的头文件不带 .h

C++ 的格式化输入输出比较麻烦，推荐使用 C 语言的 printf，scanf 进行格式化输入输出

C/C++

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int a;
6     double b;
7     char c;
8
9     // 可以自动识别变量的类型
10    cin >> a;
11    cin >> b >> c;
12
13    cout << a << endl;
14    cout << b << " " << c << endl;
15    return 0;
16 }
```

std 命名空间的使用惯例：

1. 在日常练习中，使用 `using namespace std` 即可
2. `using namespace std` 展开后，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。

C/C++

```
1 #include <iostream>
2 using namespace std;
3 // 实际开发的项目工程
4 // 1、指定命名空间访问
5 // 3、常用部分展开
6
7 // 小的程序，日常练习，不太会出现冲突
8 // 2、全局展开. 一般情况，不建议全局展开的。
9
10 // 常用展开
11 using std::cout;
12 using std::endl;
```

4. 缺省参数

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。

C/C++

```
1 //全缺省
2 void Func(int a = 10, int b = 20, int c = 30)
3 {
4     cout << "a = " << a << endl;
5     cout << "b = " << b << endl;
6     cout << "c = " << c << endl;
7     cout << endl;
8 }
9
10 int main()
11 {
12     // 使用缺省值，必须从右往左连续使用
13     Func(1, 2, 3);    // 传参时，使用指定的实参
14     Func(1, 2);
15     Func(1);
16     Func();           // 没有传参时，使用参数的默认值
17
18     //Func(, 2, );
19     //Func(, , 3);
20
21     return 0;
22 }
```

```
using namespace std;
//全缺省
void Func(int a = 10, int b = 20, int c = 30)
{
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << endl;
}

int main()
{
    // 使用缺省值，必须从右往左连续使用
    Func(1, 2, 3); //1 2 3
    Func(1, 2);    //1 2 30
    Func(1);       //1 20 30
    Func();        //10 20 30
}
```

Microsoft Visual Studio 调试 × + ▾

```
a = 1
b = 2
c = 3
```

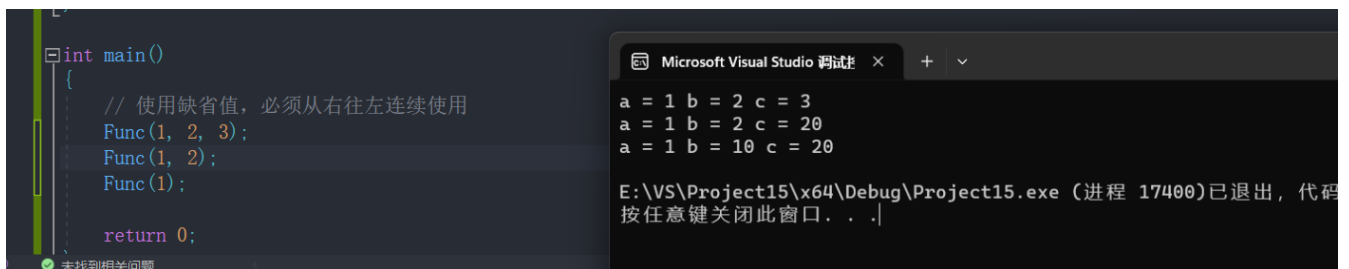
```
a = 1
b = 2
c = 30
```

```
a = 1
b = 20
c = 30
```

```
a = 10
b = 20
c = 30
```

C/C++

```
1 // 半缺省
2 // 必须从右往左连续缺省
3 void Func(int a, int b = 10, int c = 20)
4 {
5     cout << "a = " << a << " ";
6     cout << "b = " << b << " ";
7     cout << "c = " << c << " ";
8     cout << endl;
9 }
10
11 int main()
12 {
13     // 使用缺省值，必须从右往左连续使用
14     Func(1, 2, 3);
15     Func(1, 2);
16     Func(1);
17
18     return 0;
19 }
```



总结：

- 半缺省参数必须从右往左依次来给出，不能间隔着给
- 缺省参数不能在函数声明和定义中同时出现
- 缺省值必须是常量或者全局变量
- C 语言不支持（编译器不支持）

C/C++

```
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。
```

5. 函数重载

C/C++

```
1 //1、参数类型不同
2 int Add(int left, int right)
3 {
4     cout << "int Add(int left, int right)" << endl;
5
6     return left + right;
7 }
8
9 // _Z3Addddd
10 double Add(double left, double right)
11 {
12     cout << "double Add(double left, double right)" << endl;
13
14     return left + right;
15 }
16
17 //2、参数个数不同
18 void fun(int a);
19 void fun(int a, int b);
20 // 参数相同，返回类型不同，不构成重载
21 void f(int a, char b)
22 {
23     cout << "f(int a, char b)" << endl;
24 }
25
26 int f(int a, char b)
27 {
28     cout << "f(int a, char b)" << endl;
29 }
30
31 //3、参数类型顺序不同
32 void f(int a, char b)
33 {
34     cout << "f(int a, char b)" << endl;
35 }
36
37 void f(char b, int a)
38 {
39     cout << "f(char b, int a)" << endl;
```

```

40 }
41
42 int main()
43 {
44     Add(1, 2);      // call _Z3Addii(0x313131310)
45     Add(1.1, 2.2); // call _Z3Adddd(0x313131320)
46
47     f(1, 'x');
48     f(1, 'x');
49
50     return 0;
51 }

```

```

18
19 // _zlvic
20 void f(int a, char b)
21 {
22     cout << "f(int a, char b)" << endl;
23 }
24
25 // _zliic
26 int f(int a, char b)
27 {
28     cout << "f(int a, char b)" << endl;
29 }
30

```

行: 10 字符: 1 制表符

解决方案

错误 1 警告 0 消息 0 生成 + IntelliSense

搜索错误列表

代码	说明	项目	文件	行
E0311	无法重载仅按返回类型区分的函数	Project15	test.cpp	26

6. 引用

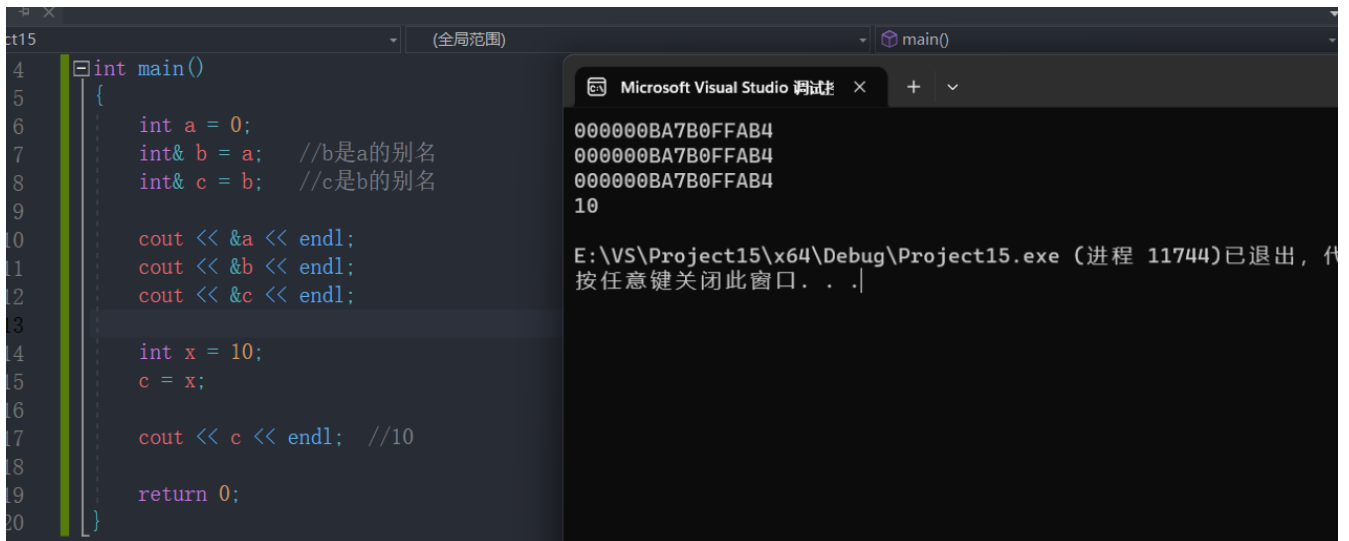
6.1 引用的概念

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。

C/C++

```
1 int main()
2 {
3     int a = 0;
4     int& b = a;    //b是a的别名
5     int& c = b;    //c是b的别名
6
7     cout << &a << endl;
8     cout << &b << endl;
9     cout << &c << endl;
10
11     int x = 10;
12     c = x;
13
14     cout << c << endl;    //10
15
16     return 0;
17 }
```



```
Microsoft Visual Studio 调试
000000BA7B0FFAB4
000000BA7B0FFAB4
000000BA7B0FFAB4
10
E:\VS\Project15\x64\Debug\Project15.exe (进程 11744)已退出，代
按任意键关闭此窗口。 . . |
```

注意：引用类型必须和引用实体是同种类型的

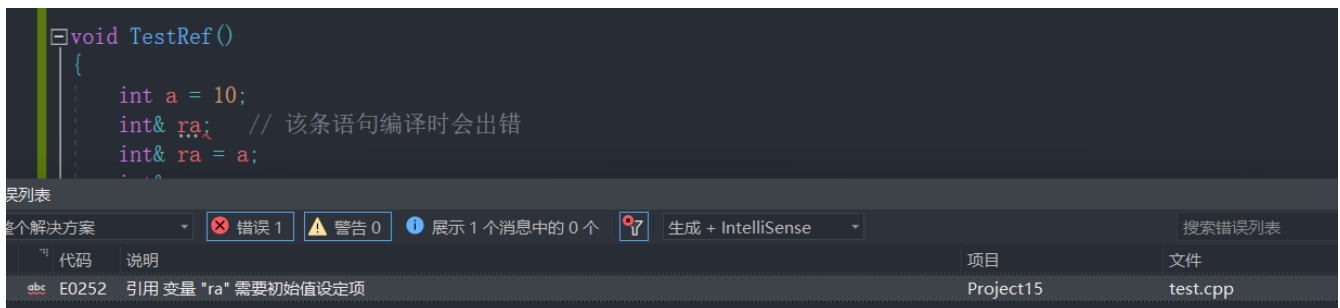
6.2 引用的特性

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

C/C++

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



6.3 常引用

C/C++

```
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;    //常引用，用来引用常变量或者变量
6     int& b = 10;    // 该语句编译时会出错，10为常量
7     const int& b = 10;    //常引用，用来引用常量
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;    //d赋值会产生临时变量，这个变量具有常属性，所以可以引用
11 }
```

```
3
4 void TestConstRef()
5 {
6     const int a = 10;
7     int& ra = a; // 该语句编译时会出错, a为常量
8     const int& ra = a; //常引用, 用来引用常变量或者变量
9     int& b = 10; // 该语句编译时会出错, 10为常量
10    const int& b = 10; //常引用, 用来引用常量
11    double d = 12.34;
12    int& rd = d; // 该语句编译时会出错, 类型不同
13    const int& rd = d; //d赋值会产生临时变量, 这个变量具有常属性, 所以可以引用
14 }
```

错误列表

整个解决方案 错误 3 警告 0 消息 0 生成 + IntelliSense 搜索错误列表

代码	说明	项目	文件	行
E0433	将 "int &" 类型的引用绑定到 "const int" 类型的初始值设定项时, 限定符被丢弃	Project15	test.cpp	7
E0461	非常量引用的初始值必须为左值	Project15	test.cpp	9
E0434	无法用 "double" 类型的值初始化 "int &" 类型的引用(非常量限定)	Project15	test.cpp	12

6.4 使用场景

1. 做参数:

C/C++

```
1 // 输出型参数
2 // 形参的改变，影响实参
3 void swap(int* p1, int* p2)
4 {
5     int tmp = *p1;
6     *p1 = *p2;
7     *p2 = tmp;
8 }
9
10 void swap(int& r1, int& r2)
11 {
12     int tmp = r1;
13     r1 = r2;
14     r2 = tmp;
15 }
16
17 int main()
18 {
19     int a = 0, b = 1;
20     swap(&a, &b);
21     swap(a, b);
22
23     return 0;
24 }
```

2. 做返回值

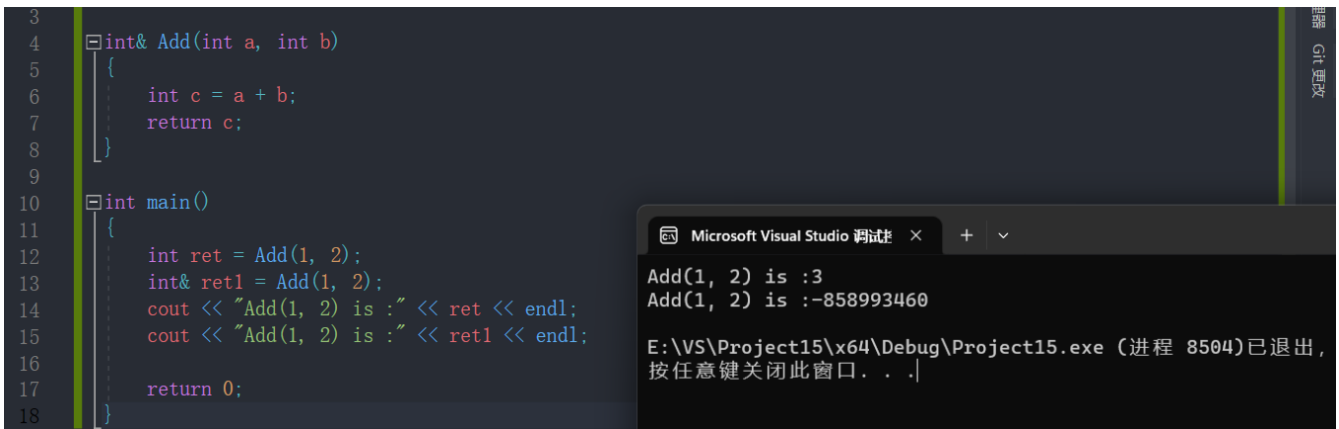
C/C++

```
1 int& Count()
2 {
3     static int n = 0;
4     n++;
5     // ...
6     return n;
7 }
8 //返回的是n的别名
```

下面代码输出什么结果？为什么？

C/C++

```
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    int& ret1 = Add(1, 2);
11    cout << "Add(1, 2) is :" << ret << endl;
12    cout << "Add(1, 2) is :" << ret1 << endl;
13
14    return 0;
15 }
```



```
3
4 int& Add(int a, int b)
5 {
6     int c = a + b;
7     return c;
8 }
9
10 int main()
11 {
12     int ret = Add(1, 2);
13     int& ret1 = Add(1, 2);
14     cout << "Add(1, 2) is :" << ret << endl;
15     cout << "Add(1, 2) is :" << ret1 << endl;
16
17     return 0;
18 }
```

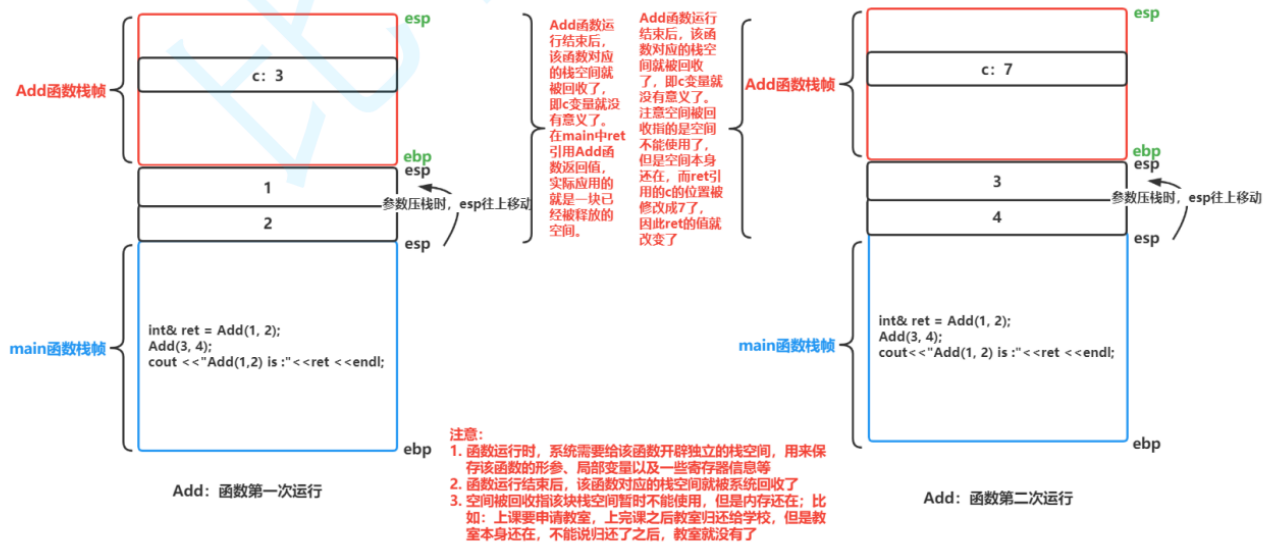
Microsoft Visual Studio 调试

Add(1, 2) is :3
Add(1, 2) is :-858993460

E:\VS\Project15\x64\Debug\Project15.exe (进程 8504)已退出，
按任意键关闭此窗口。 . . .|

ret是变量接收，Add函数返回了c的引用，进行类型转换，产生临时变量，赋值给ret，结果3

ret1是引用接收，Add函数返回了c的引用，Add函数结束后，函数栈帧被销毁，c也被销毁，ret1的别名就是随机值



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。

C/C++

```
1 #include <time.h>
2 #include<iostream>
3 using namespace std;
4 struct A { int a[10000]; };
5 A a;
6 // 值返回
7 A TestFunc1() { return a; }
8 // 引用返回
9 A& TestFunc2() { return a; }
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     // 以引用作为函数的返回值类型
18     size_t begin2 = clock();
19     for (size_t i = 0; i < 100000; ++i)
20         TestFunc2();
21     size_t end2 = clock();
22     // 计算两个函数运算完成之后的时间
23     cout << "TestFunc1 time:" << end1 - begin1 << endl;
24     cout << "TestFunc2 time:" << end2 - begin2 << endl;
25 }
26
27 int main()
28 {
29     TestReturnByRefOrValue();
30     return 0;
31 }
```

```
size_t end2 = clock();
// 计算两个函数运算完成之后的时间
cout << "TestFunc1 time:" << end1 - begin1 << endl;
cout << "TestFunc2 time:" << end2 - begin2 << endl;
}
```

```
TestFunc1 time:222
TestFunc2 time:0
```

```
E:\VS\Project15\x64\Debug\Project15.exe (进程 1
按任意键关闭此窗口. . .)
```

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。

C/C++

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

```
int main()
{
    int a = 10;
    int& ra = a;
    cout << "&a = " << &a << endl;
    cout << "&ra = " << &ra << endl;
    return 0;
}
```

Microsoft Visual Studio 调试

&a = 0000002DCE5AF9D4
&ra = 0000002DCE5AF9D4

E:\VS\Project15\x64\Debug\Project15.exe
按任意键关闭此窗口. . .|

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。

C/C++

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

我们来看下引用和指针的汇编代码对比：

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

发现汇编指令是一模一样的，说明引用的底层是指针实现的

总结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

6.7 引用的权限放大缩小

指针和引用，赋值 / 初始化 权限可以缩小，但是不能放大

C/C++

```
1  int Count()
2  {
3      int n = 0;
4      n++;
5
6      return n;
7  }
8
9  int main()
10 {
11     int a = 1;
12     int& b = a;
13
14     // 指针和引用，赋值/初始化 权限可以缩小，但是不能放大
15
16     // 权限放大
17     /*const int c = 2;
18     int& d = c;
19
20     const int* p1 = NULL;
21     int* p2 = p1;*/
22
23     // 权限保持
24     const int c = 2;
25     const int& d = c;
26
27     const int* p1 = NULL;
28     const int* p2 = p1;
29
30     // 权限缩小
31     int x = 1;
32     const int& y = x;
33
34     int* p3 = NULL;
35     const int* p4 = p3;
36
37     //
38     //const int m = 1;
39     //int n = m;
```

```

40
41     const int& ret = Count();
42
43     int i = 10;
44
45     cout << (double)i << endl;
46
47     double dd = i;
48
49     const double& rd = i;
50
51     return 0;
52 }


```

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)

```
inline int Add(int left, int right)
{
    return left + right;
}

int main()
{
    int ret = 0;
    ret = Add(1, 2);
    return 0;
}
```

int ret = 0;
mov dword ptr [ret], 0
ret = Add(1, 2);
mov eax, 1
add eax, 2
mov dword ptr [ret], eax



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**、**且频繁调用**的函数采用 inline 修饰，否则编译器会忽略 inline 特性。

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。

C/C++

```
1 // F.h
2 #include <iostream>
3 using namespace std;
4 inline void f(int i);
5 // F.cpp
6 #include "F.h"
7 void f(int i)
8 {
9     cout << i << endl;
10 }
11 // main.cpp
12 #include "F.h"
13 int main()
14 {
15     f(10);
16     return 0;
17 }
18 // 链接错误:main.obj : error LNK2019: 无法解析的外部符号 "void __cdecl f(int
    t)" (?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

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

C/C++

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

`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：

C/C++

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

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：

C/C++

```
1 typedef char *pstring;
2 int main()
3 {
4     const pstring p1; // 编译成功还是失败？
5     const pstring *p2; // 编译成功还是失败？
6     return 0;
7 }
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，但遗憾的是从没有人去使用它，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。

C/C++

```
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    cout << typeid(b).name() << endl;
13    cout << typeid(c).name() << endl;
14    cout << typeid(d).name() << endl;
15    //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
16    return 0;
17 }
```

【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 &

C/C++

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

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。

C/C++

```
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 }
```

8.4 auto 不能推导的场景

1. auto 不能作为函数的参数

C/C++

```
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

C/C++

```
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：

C/C++

```
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     for (int* p = array; p < array + sizeof(array) / sizeof(array[0]); ++p)
7         cout << *p << endl;
8 }
9
10 int main()
11 {
12     TestFor();
13     return 0;
14 }
```

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 int main()
2 {
3     int a = 0;
4     auto b = &a;
5     auto* c = &a;
6     auto& d = a;
7
8     int array[] = { 1,2,3,4,5,6,6,4 };
9     for (int i = 0; i < sizeof(array) / sizeof(int); ++i)
10    {
11        cout << array[i] << " ";
12    }
13    cout << endl;
14
15    // 范围for -- 语法糖
16    // 自动依次取数组中数据赋值给e对象，自动判断结束
17    for (auto& e : array)
18    {
19        e *= 2;
20        cout << e << " ";
21    }
22    cout << endl;
23
24    //for (int x : array)
25    for (auto x : array)
26    {
27        cout << x << " ";
28    }
29    cout << endl;
30
31    return 0;
32 }
```

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

C/C++

```
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

10.1 C++98 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：

C/C++

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

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：

C/C++

```
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：

C/C++

```
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`。