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)

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。

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. 普通的命名空间
namespace N1  // N1为命名空间的名称
{
    // 命名空间中的内容，既可以定义变量，也可以定义函数
    int a;
    int Add(int left, int right)
    {
        return left + right;
    }
}

//2. 命名空间可以嵌套
namespace N2
{
    int a;
    int b;
    int Add(int left, int right)
    {
        return left + right;
    }

    namespace N3
    {
        int c;
        int d;
        int Sub(int left, int right)
        {
            return left - right;
        }
    }
}

//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。
namespace N1
{
    int Mul(int left, int right)
    {
        return left * right;
    }
}
```

> 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：

```
namespace N
{
    int a = 10;
    int b = 20;
    int Add(int left, int right)
    {
        return left + right;
    }

    int Sub(int left, int right)
    {
        return left - right;
    }
}

int main()
{
    printf("%d\n", a);  // 该语句编译出错，无法识别a
    return 0;
}
```

> 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

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

- 使用using将命名空间中成员引入

```
using N::b;
int main()
{
    printf("%d\n", N::a);
    printf("%d\n", b);
    return 0;
}
```

- 使用using namespace 命名空间名称引入

```
using namespce N;
int main()
{
    printf("%d\n", N::a);
    printf("%d\n", b);
    Add(10, 20);
    return 0;
}
```

3. C++输入&输出

新生婴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。

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

int main()
{
    cout<<"Hello world!!!"<<endl;
    return 0;
}
```

说明：

1. 使用**cout标准输出(控制台)**和**cin标准输入(键盘)**时，必须**包含< iostream >头文件**以及std标准命名空间。

   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。

2. 使用C++输入输出更方便，不需增加数据格式控制，比如：整形--%d，字符--%c

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

int main()
{
    int a;

    double b;
```

```
    char c;

    cin>>a;
    cin>>b>>c;

    cout<<a<<endl;
    cout<<b<<"  "<<c<<endl;
    return 0;
}
```

4. 缺省参数

大家知道什么是备胎吗？

C++中**函数的参数**也可以配备胎。

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**参数指定一个默认值**。在调用该函数时，如果没有指定实参则采用该默认值，否则使用指定的实参。

```
void TestFunc(int a = 0)
{
    cout<<a<<endl;
}

int main()
{
    TestFunc();      // 没有传参时，使用参数的默认值
    TestFunc(10);    // 传参时，使用指定的实参
}
```

4.2 缺省参数分类

- **全缺省参数**

```
void TestFunc(int a = 10, int b = 20, int c = 30)
{
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;
}
```

- **半缺省参数**

```
void TestFunc(int a, int b = 10, int c = 20)
{
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    cout<<"c = "<<c<<endl;
}
```

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

```
//a.h
void TestFunc(int a = 10);

// a.c
void TestFunc(int a = 20)
{}

// 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 顺序)必须不同**，常用来处理实现功能类似数据类型不同的问题

```
int Add(int left, int right)
{
```

```
    return left+right;
}

double Add(double left, double right)
{
   return left+right;
}

long Add(long left, long right)
{
    return left+right;
}

int main()
{
    Add(10, 20);
    Add(10.0, 20.0);
    Add(10L, 20L);

    return 0;
}
```

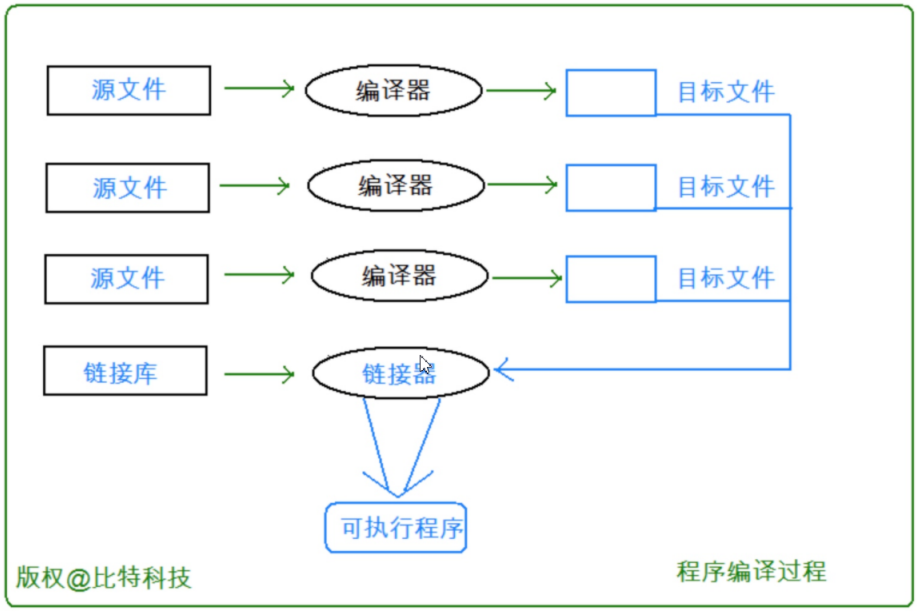
下面两个函数属于函数重载吗?

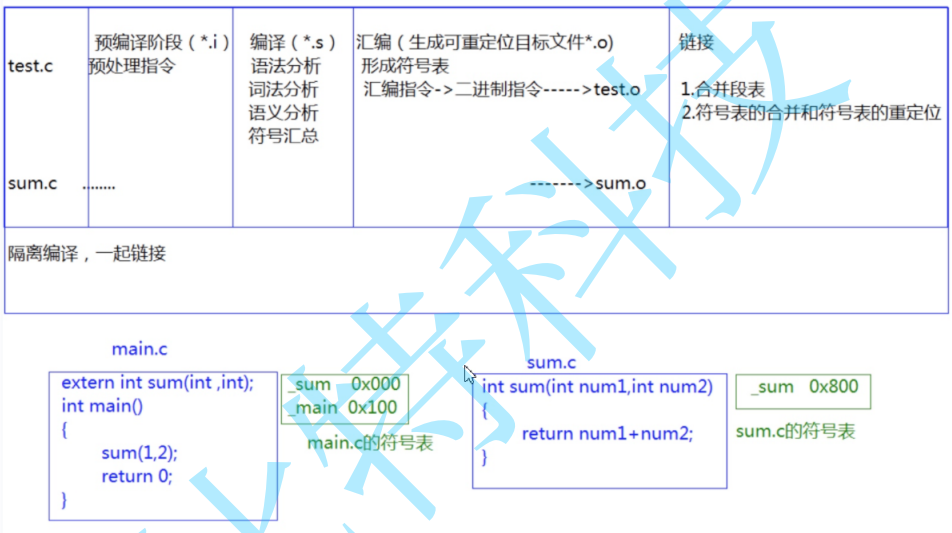
```
short Add(short left, short right)
{
   return left+right;
}

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

5.2 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。**





1. 实际我们的项目通常是由多个头文件和多个源文件构成，而通过我们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。**(如果同学们忘记了上面过程，咋们老师要带同学们回顾一下)
3. 那么链接时，面对Add函数，连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gcc的修饰规则简单易懂，下面我们使用了gcc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】。**
    - **采用C语言编译器编译后结果**

```c
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。**

- **采用C++编译器编译后结果**

```c
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.cpp
[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。**

- **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C++编译器对函数名字修饰非常诡异**，但道理都是一样的。

【扩展学习：C/C++函数调用约定和名字修饰规则】

C++函数重载

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. 另外我们也理解了，为什么函数重载要求参数不同！而跟返回值没关系。

5.3 extern “C”

有时候在C++工程中可能需要**将某些函数按照C的风格来编译，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**。比如：tcmalloc是google用C++实现的一个项目，他提供tcmallc()和tcfree两个接口来使用，但如果是C项目就没办法使用，那么他就使用extern “C”来解决。

```
extern "C" int Add(int left, int right);

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

链接时报错：error LNK2019: 无法解析的外部符号_**Add**，该符号在函数 _main 中被引用

【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？

```cpp
void TestFunc(int a = 10)
{
   cout<<"void TestFunc(int)"<<endl;
}

void TestFunc(int a)
{
   cout<<"void TestFunc(int)"<<endl;
}
```

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;

```cpp
void TestRef()
{
   int a = 10;
   int& ra = a;//<====定义引用类型

   printf("%p\n", &a);
   printf("%p\n", &ra);
}
```

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

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

6.3 常引用

```
void TestConstRef()
{
   const int a = 10;
   //int& ra = a;   // 该语句编译时会出错，a为常量
   const int& ra = a;
   // int& b = 10;  // 该语句编译时会出错，b为常量
   const int& b = 10;
   double d = 12.34;
   //int& rd = d;  // 该语句编译时会出错，类型不同
   const int& rd = d;
}
```

6.4 使用场景

1. **做参数**

```
void Swap(int& left, int& right)
{
   int temp = left;
   left = right;
   right = temp;
}
```

2. **做返回值**

```
int& Count()
{
   static int n = 0;
   n++;
   // ...
   return n;
}
```

下面代码输出什么结果？为什么？

```cpp
int& Add(int a, int b)
{
    int c = a + b;
    return c;
}

int main()
{
    int& ret = Add(1, 2);
    Add(3, 4);
    cout << "Add(1, 2) is :"<< ret <<endl;
    return 0;
}
```

注意：**如果函数返回时，出了函数作用域，如果返回对象还未还给系统，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。

```cpp
#include <time.h>
struct A{ int a[10000]; };
void TestFunc1(A a){}

void TestFunc2(A& a){}

void TestRefAndValue()
{
    A a;
    // 以值作为函数参数
    size_t begin1 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc1(a);
    size_t end1 = clock();

    // 以引用作为函数参数
    size_t begin2 = clock();
    for (size_t i = 0; i < 10000; ++i)
        TestFunc2(a);
    size_t end2 = clock();

    // 分别计算两个函数运行结束后的时间
    cout << "TestFunc1(A)-time:" << end1 - begin1 << endl;
    cout << "TestFunc2(A&)-time:" << end2 - begin2 << endl;
}
```

6.5.2 值和引用的作为返回值类型的性能比较

```cpp
#include <time.h>
struct A{ int a[10000]; };

A a;
// 值返回
A TestFunc1() { return a;}
// 引用返回
A& TestFunc2(){ return a;}

void TestReturnByRefOrValue()
{
    // 以值作为函数的返回值类型
    size_t begin1 = clock();
    for (size_t i = 0; i < 100000; ++i)
        TestFunc1();
    size_t end1 = clock();

    // 以引用作为函数的返回值类型
    size_t begin2 = clock();
    for (size_t i = 0; i < 100000; ++i)
        TestFunc2();
    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;
}
```

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。

```cpp
int main()
{
    int a = 10;
    int& ra = a;

    cout<<"&a = "<<&a<<endl;
    cout<<"&ra = "<<&ra<<endl;

    return 0;
}
```

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。

```cpp
int main()
{
    int a = 10;

    int& ra = a;
    ra = 20;

    int* pa = &a;
    *pa = 20;

    return 0;
}
```

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：

指针更强大，更危险，更复杂
引用相对局限一些，更安全，更简单

1. **引用**在定义时**必须初始化**，指针没有要求
2. **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
3. **没有NULL引用**，但有NULL指针
4. **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)
5. 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
6. **有多级指针，但是没有多级引用**
7. 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**
8. **引用比指针使用起来相对更安全**

因此链表中的指针不
能用引用给替代

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**，没有函数压栈的开销，内联函数提升程序运行的效率。

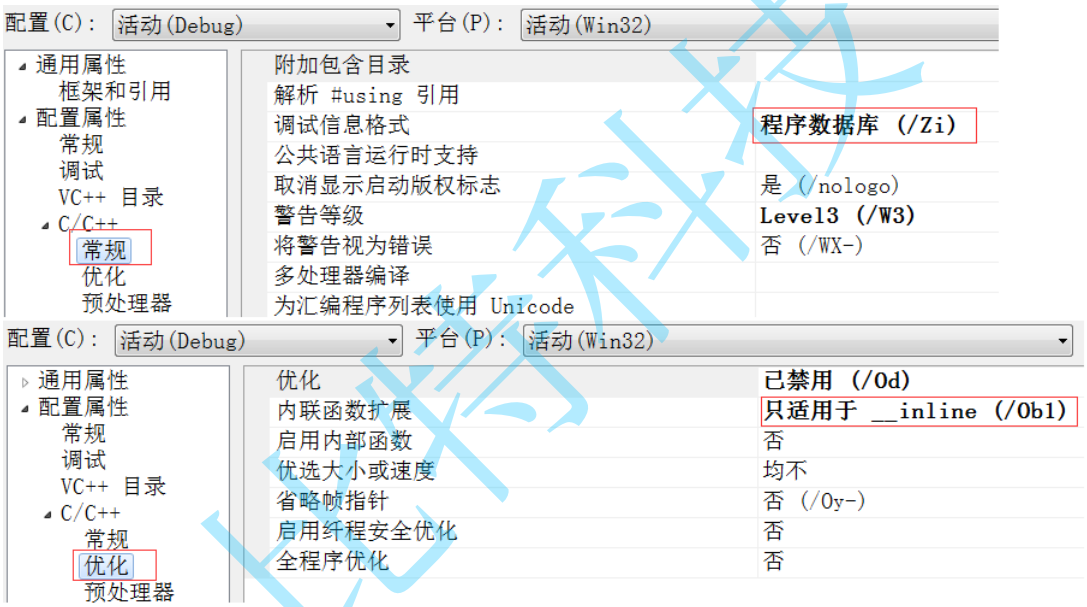
```
int Add(int left, int right)
{
    return left + right;
}

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

```
int ret = 0;
mov         dword ptr [ret],0
ret = Add(1, 2);
push        2
push        1
call        Add (12C107Dh)
add         esp,8
mov         dword ptr [ret],eax
```

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)




```
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，省去调用函数额开销。所以**代码很长**或者有**循环/递归**的函数不适宜使用作为内联函数。

2. **inline对于编译器而言只是一个建议**，编译器会自动优化，如果定义为inline的函数体内有循环/递归等等，编译器优化时会忽略掉内联。
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。

```
// F.h
#include <iostream>
using namespace std;

inline void f(int i);

// F.cpp
#include "F.h"
void f(int i)
{
    cout << i << endl;
}

// main.cpp
#include "F.h"
int main()
{
    f(10);
    return 0;
}

// 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
2. 函数定义 换用内联函数

8. auto关键字(C++11)

8.1 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。**

```
int TestAuto()
{
   return 10;
}

int main()
{
   int a = 10;
   auto b = a;
   auto c = 'a';
   auto d = TestAuto();

   cout << typeid(b).name() << endl;
   cout << typeid(c).name() << endl;
   cout << typeid(d).name() << endl;


   //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
   return 0;
}
```

> 【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。**

8.2 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&**

```
int main()
{
    int x = 10;
    auto a = &x;
    auto* b = &x;
    auto& c = x;

    cout << typeid(a).name() << endl;
    cout << typeid(b).name() << endl;

    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：

```
void TestFor()
{
    int array[] = { 1, 2, 3, 4, 5 };
    for (int i = 0; i < sizeof(array) / sizeof(array[0]); ++i)
        array[i] *= 2;

    for (int* p = array; p < array + sizeof(array)/ sizeof(array[0]); ++p)
        cout << *p << endl;
}
```

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。**

```
void TestFor()
{
    int array[] = { 1, 2, 3, 4, 5 };
    for(auto& e : array)
        e *= 2;

    for(auto e : array)
        cout << e << " ";

    return 0;
}
```

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

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

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：

```
void TestPtr()
{
    int* p1 = NULL;
    int* p2 = 0;

    // ……
}
```

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：

```
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#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：

```
void f(int)
{
    cout<<"f(int)"<<endl;
}

void f(int*)
{
    cout<<"f(int*)"<<endl;
}

int main()
{
    f(0);
    f(NULL);
    f((int*)NULL);
    return 0;
}
```

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。