# 形参列表里有一个void表示什么？

**显式的**定义空形参列表，这是为了向C兼容而保留的特性，如：

**void** f2(**void**){ /\* ... \*/ }

# 函数的返回类型不能是什么？

不能是 数组 和 函数，但可以是指向它们的指针。

# 若外层有与局部变量同名的变量，会发生什么？

在该局部变量的作用于内，会隐藏外部同名的变量。

# 什么是自动对象？

对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义所在的语句时创建该变量，当到达定义末尾时销毁它， 我们把只存在于块执行期间的对象称为**自动对象**。

# 静态局部变量的初始化有什么特点？

1 ) 静态局部变量只在程序的执行路径第一次经过的时候进行初始化一次；

2 ) 静态局部变量如果没有显式初始化，会执行值初始化，内置类型将被自动初始化为 0；

# 静态局部变量 的生存周期

静态局部变量将存在到**程序结束**

# 静态局部变量存放在哪里？

初始化数据段 或 未初始化数据段

# 如何写函数的声明？

函数原型 加上 分号，如：

**void** print(vector<**int**>::const\_iterator beg, vector<**int**>::const\_iterator end);

# 函数的声明次数

类似于变量，函数可以声明多次，但只能定义一次

# 什么是分离式编译？

一个项目由若干个源文件共同实现，而每个源文件(**.cpp**)单独编译成目标文件(**.obj**)，最后将所有目标文件连接起来形成单一的可执行文件(.out 或 **.exe**)的过程。

# C++ 有哪几种参数传递方式？

传值调用 和 传引用调用

# 在传参时，传引用和传指针有何异同？

**相同点：**

都可以修改指向对象的值

**不同点：**

传指针和传值一样，都是拷贝给了另一个对象，拷贝完成之后，它们是两个不同的对象，只不过它们有着相同的值，改变一个不会影响另一个。

在C++中，推荐使用使用传引用，而不是传指针。

# 传引用有何好处？

效率高，因为它避免了拷贝

# 传引用的时候需要注意什么？

如果不希望改变引用对象的值，应该声明为const引用

# 为什么要将函数不会改变形参设为const引用？有什么好处？

1 ) 会给调用者误导，即函数可以修改它的实参值；

2 ) 限制了函数能接受的类型

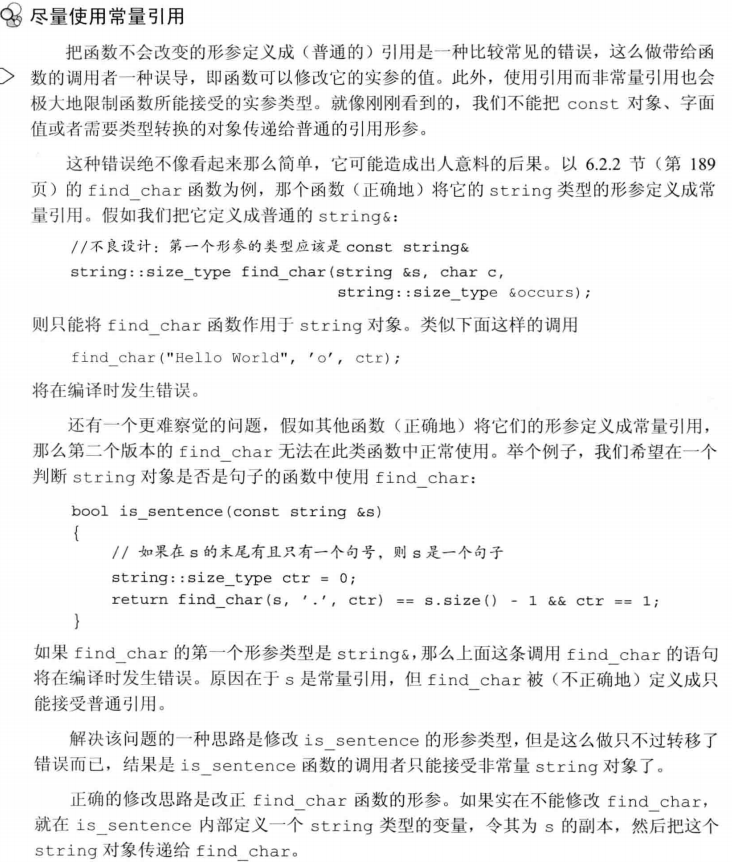
我们不能把const对象、字面值、或需要类型转换的对象传给普通的引用实参，比如：

1. **int** func(**int** & a);  // func接受的是普通的int引用
2. **int** b = 1;
3. **const** **int** &c = b;  // 常量int引用
4. func(c) //错误：不能用const int&（常量引用）初始化int &

上面之所以会报错，这很好理解，这就有点像可以用非const变量初始化const变量，但是不能用const变量初始化非const变量一样：

1. **const** **int** ci = 1024;
2. **int** i = 2048;
3. **const** **int** &r1 = ci; // 正确: r1 和 ci都是 const
4. **int** &r2 = ci;   // 错误: 非常量引用不能引用常量对象
5. **const** **int** &r3 = i;  // ok:

3 ) 避免函数调用时的一种特殊的情况：



# 以下声明有何区别？

* 1. **void** print(**const** **int**\*);
  2. **void** print(**const** **int**[]);
  3. **void** print(**const** **int**[10]);

它们没有区别，因为数组有两个特性：

1 ) 不允许拷贝数组，**原因：**数组名是一个地址常量，其值和第一个元素的地址值相同，不可修改。，所以他们不能直接赋值：

1. **int** a[] = {0, 1, 2}; // array of three ints
2. **int** a2[] = a; // 错误: 不能拷贝数组，因此不能用一个数组初始化另一个数组；
3. a2 = a;  // 错误: 不能拷贝数组，因此不能将数组直接赋给另一个数组  ；

2 ) 使用数组时，通常会将其转换成指针。

因此对于上面几个函数原型：

1. **void** print(**const** **int**\*);
2. **void** print(**const** **int**[]);  // 函数的意图是作用于一个数组
3. **void** print(**const** **int**[10]); // 这里的维度表示我们期望数组含有多少元素，但是不一定

尽管这几个函数原型表现形式不同，但它们都是等价的：

① 每个函数的唯一形参都是 **const** **int**\* ；

② 当编译器处理对print函数的调用时，只检查传入的参数是否为 **const** **int**\*

# 使用数组引用形参时需要注意什么？

* 1. **void** print(**int** (&arr)[10])  // 注意！维度是类型的一部分
  2. {
  3. **for** (auto elem : arr)
     1. cout << elem << endl;
  4. }

1 ) 需要注意数组名两端的括号：

1. void f(**int** &arr[10])  // 错误: 将arr声明成了引用的数组
2. void f(**int** (&arr)[10]) // 正确: arr 引用了 具有10个int的数组

# 使用数组引用形参有什么缺点？有什么办法解决呢？

**缺点**：通用性太差，因为维度(length)也是形参类型的一部分，所以只能接受大小为length 的数组：

1. **int** i = 0, j[2] = {0, 1};
2. **int** k[10] = {0,1,2,3,4,5,6,7,8,9};
3. print(j); // 错误: 实参的大小为2，print
4. print(k); // 正确: 数组k的大小为10

**解决办法**：用函数模板

# 如何传递多维数组？

假设有一个遍历二维数组的函数，我们可以这么写：

1 )  **void** print\_values\_one(**int** (\*matrix)[5], **int** i) // \*a两边的括号必不可少！

注意 **int** (\*matrix)[5] 和 **int** \* matrix [5] 的区别：

1. **int** \* matrix [5] // matrix是一个数组，里面含有5个int
2. **int** (\*matrix)[5] // matrix是一个指针，指向 含有5个int的数组

2 ) **void** print\_values\_two(**int** matrix [][5], **int** i)

当且仅当matrix作为函数的参数时，才可以这样声明：

1. **int** sum2(**int** ar[][], **int** rows); // 错误的声明
2. **int** sum2(**int** ar[][4], **int** rows); // 有效声明
3. **int** sum2(**int** ar[3][4], **int** rows); // 有效声明， 但是3将被忽略

一般而言， 声明一个指向N维数组的指针时， 只能省略最左边方括号中的值：

**int** sum4d(**int** ar[][12][20][30], **int** rows);

因为第1对方括号只用于表明这是一个指针， 而其他的方括号则用于描述指针所指向数据对象的类型。下面的声明与该声明等价：

1. **int** sum4d(**int** (\*ar)[12][20][30], **int** rows); // ar是一个指针

3 ) **void** print\_values\_three(**int** \*\* matrix, **int** i, **int** j)

下面是这几个函数的实现：

1. /\*\*
2. \* @param 1:a 是int[5]类型的指针
3. \* @param 2:i 边界
4. \*/
5. **void** print\_values\_one(**int** (\*a)[5], **int** i)
6. {
7. **for**(**int** m = 0; m < i; ++ m)
8. {
9. **for**(**int** n = 0; n < 5; ++n)
10. {
11. cout << a[m][n] << " ";
12. }
13. cout << endl;
14. }
15. }
17. /\*\*
18. \* @param 1:a 元素是二维数组,每行是5个元素的一维数组
19. \* @param 2:i 边界
20. \*/
21. **void** print\_values\_two(**int** a[][5], **int** i)
22. {
23. **for**(**int** m = 0; m < i; ++ m)
24. {
25. **for**(**int** n = 0; n < 5; ++n)
26. {
27. cout << a[m][n] << " ";
28. }
29. cout << endl;
30. }
31. }
33. /\*\*
34. \* @param 1:a 是一个指向 int\* 类型的指针
35. \* @param 2:i 边界
36. \* @param 3:j 边界
37. \*/
38. **void** print\_values\_three(**int** \*\* a, **int** i, **int** j)
39. {
40. **for**(**int** m = 0; m < i; ++ m)
41. {
42. **for**(**int** n = 0; n < j; ++n)
43. {
44. cout << \*( \*(a + m) + n)<< " ";
45. }
46. cout << endl;
47. }
48. }

# 处理含有可变形参的函数

**方法一：initializer\_list 形参**

**1 ) 何时使用initializer\_list 形参？**

① 实参数量未知；

② 全部实参类型相同；

**2 ) initializer\_list是什么类型？**

和vector一样，它是一种模板类型。

**3 ) initializer\_list有何特点？**

① initializer\_list对象中的元素永远是常量值，我们无法修改initializer\_list对象中元素。

② 拷贝或赋值一个initializer\_list对象不会拷贝列表中的元素，其实只是引用而已，原始列表和副本共享元素。

**4 )用 initializer\_list 写一个参数不定的错误输出函数，另一个形参是ErrCode**

1. **void** error\_msg(ErrCode e, initializer\_list<string> il)
2. {
3. cout << e.msg() << ": ";
4. **for** (**const** auto &elem : il)
5. cout << elem << " " ;
6. cout << endl;
7. }

**方法二：省略符形参**

省略符形参是为了方便C++程序访问某些特殊的C代码而设置。

省略符形参只能出现在形参列表的最后一个位置，它的形式无非下面两种：

1. **void** foo(parm\_list, ...);
2. **void** foo(...);

# 函数在返回值的时候需要注意什么？

不要返回局部对象的引用或指针，因为函数结束后，它所占的空间都被释放了，访问它们将引发未定义的行为。

# 函数什么时候返回左值？ 返回的左值有什么用？

**1 ) 函数什么时候返回左值？**

调用一个 **返引用的函数** 将得到左值（因为引用返回的是变量本身，而不是对象的副本），其它返回类型得到右值。

**2 ) 如何使用函数返回的左值？**

我们可以给返回的左值赋值

# 如何使用函数返回的左值？

* 1. // 注意！形参类型、返回值类型都是引用！
  2. **char** &get\_val(string &str, string::size\_type ix)
  3. {
  4. **return** str[ix]; // get\_val assumes the given index is valid
  5. }
  6. **int** main()
  7. {
  8. string s("a value");
  9. cout << s << endl; // prints a value
  10. get\_val(s, 0) = 'A'; // changes s[0] to A
  11. cout << s << endl; // prints A value
  12. **return** 0;
  13. }

将函数调用写在赋值语句的左边看起来可能怪怪的，但是这其实没什么特别的：

因为get\_val函数返回的是引用，因此调用后返回的是左值，既然它返回的是左值，那么意味着它可以和其它左值一样出现在赋值语句的左侧。

# 列表初始化 返回值

**背景：**

在C++11之前，如果我们想要返回一组数据，我们必须在子函数中构造一个对应的容器，借助容器来进行返回。

1. vector<**int**> process()
2. {
3. vector<**int**> v={1,2,3,4}
4. **return** v;
5. }

**在新标准下，我们可以直接返回字面值，该字面值会用于容器的构造，而无需我们自己去构造：**

1. vector<**int**> process()
2. {
3. **return** {1,2,3,4};
4. }

C++ primer的例子：

* 1. vector<string> process()
  2. {
  3. // . . .
  4. // expected and actual are strings
  5. **if** (expected.empty())
  6. **return** {}; // return an empty vector
  7. **else** **if** (expected == actual)
  8. **return** {"functionX", "okay"}; // return list-initialized vector
  9. **else**
  10. **return** {"functionX", expected, actual};
  11. }

# 主函数main的返回值可以省略吗？

可以的，如果控制流程到了main函数的结尾 且没有return语句，**编译器将隐式的插入一条返回0的return语句。**

# 函数返回类型为void就一定要有return语句吗？

是的，输了main函数，因为**编译器将隐式的插入一条返回0的return语句。**

# 什么函数不能调用自己？

main函数。

# C++可以返回数组吗？

和C语言一样，C++也不能返回指针，因为数组不能被拷贝。

# 有其它办法返回数组吗？

可以返回数组的指针和引用。

# 声明一个 接受一个int参数，返回指向10个int的指针 的函数

和typedef定义数组别名一样，数组维度得放在函数名字的后面，两端的括号也不能少：

**typedef定义数组别名**：

**int** (\*arra)[10];

**一个int参数，返回指向10个int的指针 的函数**：

**int** (\*func(**int** i))[10];

# 用typedef简化 int (\*func(int i))[10];

* 1. **typedef** **int** arrT[10];  // 或: using arrT = int[10];
  2. arrT \* func(**int** i);

# 还有办法可以简化函数返回数组的指针？

如果我们知道 函数返回的指针 指向哪个数组，那么就可以使用decltype关键字来声明返回类型，比如：

1. **int** odd[] = {1,3,5,7,9};
2. **int** even[] = {0,2,4,6,8};
3. // 返回一个指针，该指针指向含有5个整数的数组
4. decltype(odd) \*arrPtr(**int** i)
5. {
6. **return** (i % 2) ? &odd : &even; // returns a pointer to the array
7. }

**需要注意的是**，decltype关键字并不负责把数组类型转换成对应的指针，所以decltype的结果是个数组，若想要表示arrPtr函数返回数组的指针，则需要在函数声明的时候加上一个 \*

# 重载、重写、重定义有什么区别？

把它们放在一起比较好理解。

**一、重载（overload）**

指函数名相同，但是它的参数表列个数或顺序，类型不同。但是不能靠返回类型来判断。

（1）相同的范围（在同一个作用域中） ；

（2）函数名字相同；

（3）形参列表不同；

（4）virtual 关键字可有可无。

（5）返回值可以不同；

**二、重写（也称为覆盖 override）**

是指派生类重新定义基类的虚函数，特征是：

（1）不在同一个作用域（分别位于派生类与基类） ；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 virtual 关键字，不能有 static 。

（5）返回值相同（或是协变），否则报错；

（6）重写函数的访问修饰符可以不同。尽管 virtual 是 private 的，派生类中重写改写为 public,protected 也是可以的

**三、重定义（也称为隐藏，Overwrite）**

（1）不在同一个作用域（分别位于派生类与基类） ；

（2）函数名字相同；

（3）返回值可以不同；

（4）参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载以及覆盖混淆） 。

（5）参数相同，但是基类函数没有 virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆） 。

# 对于重载的函数，怎么确定调用的具体是哪个函数呢？

编译器会根据 传递的实参类型 判断想要的是哪个函数。

# 为什么需要重载？

更加方便程序员，几个功能相近，但是接受参数不一样的函数可以取同样的名字，这样就不需要记那么多名字了。

# 什么函数不能重载？

main函数不能重载。

# 仅仅是返回类型不同，形参列表都一样，算重载吗？

不算的，如果仅仅是返回类型不同，形参列表都一样，那么第二个函数的声明是错误的。

# 有时候两个形参列表看起来是不一样的，但实际上是一样的

* 1. // 忽略了形参的名字，但实际上是一样的
  2. Record lookup(**const** Account &acct);
  3. Record lookup(**const** Account&); // parameter names are ignored
  4. // 只是用typedef取了个别名而已，实际上形参是一样的。
  5. **typedef** Phone Telno;
  6. Record lookup(**const** Phone&);
  7. Record lookup(**const** Telno&); // Telno and Phone are the same type

# const对重载有什么影响？

**顶层const对重载没有影响**：

一个 **拥有**顶层const的形参 无法和另一个 **没有**顶层const的形参 区分开来：

1. Record lookup(Phone);
2. Record lookup(**const** Phone); //重复声明了Record lookup(Phone)
4. Record lookup(Phone\*);
5. Record lookup(Phone\* **const**); //虽然是指针，但这是顶层const，因此重复声明了Record lookup(Phone\*)

**底层const会影响重载**：

如果形参是指针或引用，则 **通过区分它们指向的是常量对象还是非常量对象** 可以实现函数的重载：

1. Record lookup(Account&); //函数作用于Account的引用
2. Record lookup(**const** Account); //因为是底层const，所以是新函数，作用于常量引用
4. Record lookup(Account\*); //新函数，作用于指向Account的指针
5. Record lookup(**const** Account\*); //新函数，作用于指向常量的指针

# 用 Account\* ptr 调用下面两个函数，编译器会怎么处理？

* 1. Record lookup(Account\*); //新函数，作用于指向Account的指针
  2. Record lookup(**const** Account\*); //新函数，作用于指向常量的指针

虽然两个函数都可以匹配成功(因为可以用非常量对象来初始化常量对象)，但编译器会优先选择最佳匹配的那个，即Record lookup(**const** Account\*)

# const\_cast和重载

对于下面这个函数，实参和返回类型都const string的引用，我们可以用两个非常量的string来调用这个函数，但是返回的也是const string的引用：

1. //  实参和返回类型都const string的引用
2. **const** string &shorterString(**const** string &s1, **const** string &s2)
3. {
4. **return** s1.size() <= s2.size() ? s1 : s2;
5. }

利用const\_cast函数，重载shorterString函数，实参为非const string，最后也返回一个非常量string的引用：

1. // 重载shorterString，形参都是非常量
2. string &shorterString(string &s1, string &s2)
3. {
4. // 利用const\_cast将实参转换为const对象，然后调用另一个shorterString函数
5. auto &r = shorterString(**const\_cast**<**const** string&>(s1),
6. **const\_cast**<**const** string&>(s2));
7. // 因为r是通过auto得到的，所以r的类型也是const string & ， 因此通过const\_cast<string&>去掉底层const，返回一个非常量string对象
8. **return** **const\_cast**<string&>(r);
9. }

# fooBar函数内的 print("Value:")调用为什么报错？

* 1. string read();
  2. **void** print(**const** string &);
  3. **void** print(**double**);//重载print函数
  4. **void** fooBar(**int** ival)
  5. {
  6. **bool** read = **false**; //新作用域：隐藏了外层的read函数
  7. string s = read(); //错误：read是一个布尔值，而非函数
  8. //不好的习惯：通常来说，在局部作用域中声明函数不是一个好的选择
  9. **void** print(**int**); //新作用域：隐藏了之前的print
  10. print("Value:"); //错误：void print(const string &)被隐藏掉了
  11. print(ival); //正确：当前print(int)可见
  12. print(3.14); //正确：调用print(int);print(double)被隐藏掉了
  13. }

**print("Value:")报错的原因：**

上面print函数的调用过程：在 **void** fooBar(**int** ival) 内声明的**void** print(**int**)隐藏了fooBar函数外的两个print函数，因此在fooBar函数内只有**void** print(**int**)可见，fooBar函数外的两个print函数都被隐藏了。

因此print("Value:")调用的是**void** print(**int**)函数，而字面值"Value:")是无法转换为**int**类型的，所以print("Value:")会报错

**如果把print("Value:")放在外面，结果将不一样：**

1. **void** print(**const** string&);
2. **void** print(**double**);
3. **void** print(**int**);
4. **void** fooBar2(**int** ival)
5. {
6. print("value");   //调用print(const string&)
7. print(ival);       //调用print(int)
8. print(3.14);    //调用print(double)
9. }

# 类型检查 和 名字查找 谁先进行？

在C++中，名字查找先发生。

# 声明 带默认形参的函数时需要注意什么？

一旦某个形参被赋予了默认值，那它后面的所有形参都必须提供默认值

# 调用 带默认形参的函数时需要注意什么？

**默认实参的填补规则**：

默认实参负责填补函数调用缺少的 **尾部实参。**也就是说，如果调用带默认实参的函数时，只能省略尾部的实参：

① 如果第1个实参你省略了，那么后面的所有实参都必须省略；

② 如果第2个实参你提供了非默认值，那么第1个实参你也必须提供非默认值；

下面来看几个例子：

1. **typedef** string::size\_type sz;
2. string screen(sz ht = 24, sz wid = 80, **char** backgrnd = ' ');
3. string window;
4. window = screen();  // equivalent to screen(24,80,' ')
5. window = screen(66); // equivalent to screen(66,80,' ')
6. window = screen(66, 256);  // screen(66,256,' ')
7. window = screen(66, 256, '#');  // screen(66,256,'#')
8. window = screen(, , '?');  // 错误: can omit only trailing arguments
9. window = screen('?');  // calls screen('?',80,' ')

对于 window = screen('?')：

这个调用时合法的，但是却和意图不符吗，它调用的是 screen('?',80,' ')，因为参数'?'填补的是第一个参数 ht，'?'是一个char类型，他将被强制转换为类型sz，即string::size\_type。

# 默认实参和多次声明

① 函数可以多次声明，但在同一作用域内，一个形参只能赋予一次默认实参；

② 也就是说函数后续的声明只能为之前哪些没有默认值的形参添加默认实参，而且该形参右侧所有形参都必须有默认值。

1. // no default for the height or width parameters
2. string screen(sz, sz, **char** = ' ');
3. string screen(sz, sz, **char** = '\*'); // 错误: 重复声明
4. string screen(sz = 24, sz = 80, **char**); // 正确: 为前两个参数添加，默认实参，且最后一个参数前面已经有默认实参了(添加默认实参的形参的右侧所有形参都有默认值)

# 默认实参的初始值

**注意，局部变量不能作为默认实参！**（因为默认实参是在编译时确定默认实参地址，在运行时取默认实参的值，而局部变量是自动变量，地址在运行时才能确定）

**只要表达式的类型能转换为性擦所需的类型，该表达式就能作为默认实参**

* 1. //  wd, def, and ht 的声明必须要出现在函数之外，也就是说不能为局部变量
  2. sz wd = 80;
  3. **char** def = ' ';
  4. sz ht();
  5. string screen(sz = ht(), sz = wd, **char** = def);
  6. string window = screen(); // calls screen(ht(), 80, ' ')
  7. **void** f2()
  8. {
  9. def = '\*'; // 变量def是全局变量，所以改变了默认实参的值
  10. sz wd = 100; // wd隐藏了外部同名的变量wd，但它没有改变函数screen()的默认值，因为它是局部变量
  11. window = screen(); //def被改变了， 等同于调用screen(ht(), 80, '\*')
  12. }

**问题：**为什么 window = screen() 调用的是screen(ht(), 80, '\*')，而不是screen(ht(), 100, '\*')，或者说： 为什么 sz wd = 100没有改变screen()函数的默认参数值，而def = '\*';却**改变了**？

因为带默认参数的函数有如下特性：

① 用作实参的名字在函数声明所在的作用域内解析；

② 求值过程发生在函数调用时；

**1 ) sz wd = 100没有改变screen()函数的默认参数值的原因：**

screen()函数声明在f2函数的外面，因此看不到声明sz wd = 100，它找到的还是和它同一作用于的声明sz wd = 80，因此sz wd = 100没有改变screen()函数的默认参数值。

**2 ) def = '\*'改变了screen()函数的默认参数值的原因：**

def的声明是在外部，也就是说def和screen()函数属于同一作用域，而在函数f2中修改的也是全局变量def；

因为默认实参的求值过程发生在函数调用时，因此调用screen()函数时，def的值已经变成了 '\*'，因此def = '\*'改变了screen()函数的默认参数值。

# 哪些变量不能作为函数的默认实参？

① 局部变量（自动对象）不能作为函数的默认实参（因为默认实参是在编译时确定默认实参地址，在运行时取默认实参的值，而局部变量是自动变量，地址在运行时才能确定）

② 类的非静态成员不能作为成员函数的默认实参

# 为什么局部变量不能作为函数的默认实参？

因为默认实参是在编译时确定默认实参地址，在运行时取默认实参的值，而局部变量是自动变量，地址在运行时才能确定。

# 内联函数有什么优点？为什么？

比普通的函数调用开销小，因为内联函数就是编译时把函数的定义替换到调用的，节省了函数调用开销。

# 内联函数为了提高性能，付出的代价是什么

内联通过把要调用的代码直接嵌入其中，省去一次函数调用的开销，这意味着代码会比较臃肿，这也是内联函数不宜过长的原因。

# 内联函数的原理

简单理解，内联函数就是编译时把函数的定义替换到调用的位置：

1. **inline** **int** Add(**int** a, **int** b)
2. {
3. **return** a + b;
4. }
5. **int** main()
6. {
7. **int** num1 = 1;
8. **int** num2 = 2;
9. **int** myNum = Add(num1, num2);
10. }

上面的代码，内联之后大概就是

1. **int** main()
2. {
3. **int** num1 = 1;
4. **int** num2 = 2;
5. **int** myNum = num1 + num2; // 内联函数被替换了
6. }

这意味内联函数可以节省很多开销

# 都说内联函数比普通函数开销小，这个开销具体是什么呢？

都知道C/C++的内存分为常量区、代码区、静态全局区、栈区、堆，其中函数代码存放在代码区，函数调用就发生在栈区里面，每次调用的时候会把当前函数的相关内容压入到栈里面处理寄存器相关的数据信息（所谓没有地址的右值很多情况就是指通过寄存器存储的数据）。然后，调用地址指向我们要执行的函数位置，开始处理函数内部的指令进行计算，当函数执行结束后，要弹出相关数据，处理栈内数据以及寄存器数据，这个过程也就是所谓的**“函数调用开销”**。

# 内联函数（inline）和宏（#define）有何区别

**处理方式：** 宏做的是简单的字符串替换（注意是字符串的替换，不是其他类型参数的替换），而函数的参数的传递，参数是有数据类型的，可以是各种各样的类型。

**是否用类型检查**：内联函数要做参数类型检查，而宏不需要（仅仅是替换），从而内联函数相比宏而言更加安全

**处理的时期：**宏在编译之前（预编译）进行，即先用宏体替换宏名，然后再编译的，而函数显然是编译之后，在执行时，才调用的。因此，宏占用的是编译的时间，而函数占用的是执行时的时间。

**空间占用：**宏的参数是不占内存空间的，因为只是做字符串的替换，而函数调用时的参数传递则是具体变量之间的信息传递，形参作为函数的局部变量，显然是占用内存的。

**开销：**函数的调用是需要付出一定的时空开销的，因为系统在调用函数时，要保留现场，然后转入被调用函数去执行，调用完，再返回主调函数，此时再恢复现场，这些操作，显然在宏中是没有的。

**建议：尽量使用inline函数替换宏定义**

# 使用内联函数时，应注意什么？

1）内联函数的定义性声明应该出现在对该函数的第一次调用之前。

　　2）内联函数首先是函数，函数的很多性质都适用于内联函数，如内联函数可以重载。

　　3）在内联函数中不允许使用循环语句和switch结果，带有异常接口声明的函数也不能声明为内联函数。

4) 只将规模很小（一般5个语句一下）而使用频繁的函数声明为内联函数。因为在函数规模很小的情况下，函数调用的时间开销 可能相当于甚至超过 执行函数本身的时间，把它定义为内联函数，可大大减少程序运行时间。

# 内联函数有什么缺点？

内联函数其实相当于给编译器一个提示，告诉它最好把这个函数在被调用处展开，省掉一个函数调用的开销（压栈，跳转，返回），它会使代码变得臃肿。

# 什么是constexpr函数

constexpr函数是指 能用于常量表达式 的函数。它和其它函数类似，不过有几个约束条件：

① 函数的返回类型、所有的形参类型 都必须是 字面值类型；

② 函数体内 **有且仅有** 一条return语句。

③ constexpr函数体内也可以包含其他语句，只要这些语句在运行时不执行任何操作就行。例如，constexpr函数中可以有空语句、类型别名以及using声明。

# constexpr函数必须返回常量吗？

不一定：

1. //如果arg是常量表达式，则scale(arg)也是常量表达式
2. constexpr **size\_t** scale(**size\_t** cnt) {**return** new\_sz()\*cnt;}
3. // 如果arg不是常量表达式，则scale(arg)不是常量表达式
4. **int** arr[scale(2)];    //正确：scale(2)是常量表达式
5. **int** i=2;                //i不是常量表达式
6. **int** a2[scale(i)];    //错误：scale(i)不是常量表达式

如果我们用一个非常量表达式调用scale函数，比如int类型的i（上面的代码的**int** i=2），则返回值是一个非常量表达式。当把scale函数用在需要常量表达式的上下文中时，由编译器负责检查函数的结果是否符合要求。如果结合恰好不是常量表达式，编译器将发出错误信息。

# 内联函数和constexpr函数有什么和普通函数不一样的？

它们可以在程序中多次定义，但多个定义必须完全一致，所以内联函数和constexpr函数一般都放在头文件中。

# assert()本质是什么？如何关闭？

它是 预处理宏

当程序被完整地测试完毕之后，可以在编译时通过定义NDEBUG消除所有的断言。你可以

①使用-DNDEBUG编译器命令行选项

②在源文件中头文件assert.h被包含之前增加下面这个定义：

#define NDEBUG

# 编译器定义了哪些对于程序调试有帮助的名字？

这些都是静态数组，以双下划线开头和结尾：

\_\_func\_\_ 当前调试的函数的名字

\_\_FILE\_\_ 存放文件名的字符串字面值

\_\_LINE\_\_ 存放当前行号的整型字面值

\_\_TIME\_\_ 存放文件编译时间的字符串字面值

\_\_DATE\_\_ 存放文件编译日期的字符串字面值

# 调用重载函数时的匹配过程是怎样的？

对于函数调用 f(5.6) ，具体调用的是哪个函数呢？

* 1. **void** f();
  2. **void** f(**int**);
  3. **void** f(**int**, **int**);
  4. **void** f(**double**, **double** = 3.14);
  5. f(5.6); // 调用的是哪个函数呢？

第一步：确定**候选函数**，候选函数有两个特征，

① 是和被调用的函数同名，

② 是其声明在调用点可见，在上面的例子中，全部4个函数都入选了；

第二步，确定**可行函数**，在候选函数中，寻找满足如下特征的函数：

① 是形参数量和本次调用提供的实参数量相等，

② 是每个实参的类型与对应的形参类型相同，或是可以转换成形参的类型。

其中满足的是：

1. **void** f();  // 不满足，0个形参
2. **void** f(**int**);    // 满足，5.6(double)可以转换为int，且形参只有一个
3. **void** f(**int**, **int**); // 不满足，有两个实参
4. **void** f(**double**, **double** = 3.14); // 满足，虽然有两个形参，但最后一个有默认实参

第三步，寻找**最佳匹配**，实参类型和形参类型最匹配的那个那个可行函数就是最佳匹配函数。在上面中，最匹配的是**void** f(**double**, **double** = 3.14)，因为实参5.6是**double**类型。

# 调用重载函数时，若有有多个形参是最佳匹配怎么办？

如果在检查了所有实参之后没有任何一个函数脱颖而出，则该调用是错误的，编译器将报告二义性调用的信息。（如果有多个都是最佳匹配，则报错）下面来看一个例子，

* 1. **void** f();
  2. **void** f(**int**);
  3. **void** f(**int**, **int**);
  4. **void** f(**double**, **double** = 3.14);
  5. f(42, 2.56); // 调用的是哪个函数呢？

最终入围最佳匹配函数是：

1. **void** f(**int**, **int**);   // 只考虑第一个形参，它是最佳匹配
2. **void** f(**double**, **double** = 3.14);   // 只考虑第二个形参，它是最佳匹配

# 调用重载函数时的实参类型转换分级

为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级

**1 ) 精确匹配**：

实参类型与形参类型相同；

实参从数组类型或函数类型转换成对应的指针类型；

向实参添加顶层const 或从实参中删除顶层const。

**2 ) 通过const 转换实现的匹配（向实参添加底层const）**。

**3 )** 通**过整形提升（小整形向上提升成int 或unsigned）实现的转变。（即使实参是很小的整数值，有时候也会直接提升为int）**

当两个同名函数一个接受int、一个接受short，则仅当传入实参为short 时才调用short，其它类型参数都会提升为int 或unsigned。有的时候，即使是一个很小的整数值，也会直接提升了int类型

1. **void** ff(**int**);
2. **void** ff(**short**);
3. ff('a'); // 'a'(char类型) 提升成了int，所以调用的是 f(int)

**值得注意的是，所有算术类型转换的级别都一样**，例如 例如 从int向unsigned int的转换 并不比 int向double的转换 级别高，举个例子：

1. **void** manip(**long**);
2. **void** manip(**float**);
3. manip(3.14);  // 错误: 二义性调用，3.14是double类型，它向long和float转换都是同样的级别，因此报错

**4 )** 通过算数转换（算数类型之间的转换）或指针转换（转换为void \*）实现的转换。

所有算数类型转换的级别都一样，例如从int 向unsigned 和从int 向double 的转换级别相同；从double 向float 和从double 向long 的转换级别相同。

**5 )** 通过类类型转换实现的转化。

# 声明一个指针，指向bool lengthCompare(const string &, const string &);

**声明函数指针的方法**：用指针替换函数名即可（两端的括号不能少）：

1. // 声明了一个函数指针
2. **bool** (\*pf)(**const** string &, **const** string &);
3. // 声明了一个名为pf的函数，该函数返回 bool\*
4. **bool** \*pf(**const** string &, **const** string &);

# 将函数名作为一个值使用时会发生什么？

函数名自动转换成了函数指针：

1. **bool** (\*pf)(**const** string &, **const** string &);
2. pf = lengthCompare; // pf 指向函数 lengthCompare
3. pf = &lengthCompare; // 等价于pf = lengthCompare，取地址符（&）可有可无

# 声明了函数指针pf后，怎么利用它调用它指向的函数？

* 1. **bool** b1 = pf("hello", "goodbye"); // 调用lengthCompare  函数
  2. **bool** b2 = (\*pf)("hello", "goodbye"); // 等价于pf("hello", "goodbye")
  3. **bool** b3 = lengthCompare("hello", "goodbye"); // equivalent call

# 声明重载函数的指针有什么规则？

**返回类型、形参须严格匹配，不能类型转换！**

1. **void** ff(**int**\*);
2. **void** ff(unsigned **int**);
3. **void** (\*pf1)(unsigned **int**) = ff; // 正确，指向了 ff(unsigned int)
4. **void** (\*pf2)(**int**) = ff; // 错误: 两个ff 函数没有接受int形参的，虽然 有一个接受的是unsigned int，但不能类型转换
5. **double** (\*pf3)(**int**\*) = ff; // 错误: 两个ff 函数都没有返回值，此声明返回double

# 如何将 函数指针 设为形参？

方法一：直接将 **函数类型** 作为形参，因为他会自动的转换为指向函数的指针

方法二：将 函数指针 作为形参

1. //  将 函数类型 作为形参，因为他会自动的转换为指向函数的指针
2. **void** useBigger(**const** string &s1, **const** string &s2,
3. **bool** pf(**const** string &, **const** string &));
4. // 将 函数指针 作为形参
5. **void** useBigger(**const** string &s1, **const** string &s2,
6. **bool** (\*pf)(**const** string &, **const** string &));

# 将 函数指针 设为形参会使函数显得很冗长，怎么简化？

方法一：只用typedef

1. // Func 是函数类型
2. **typedef** **bool** Func(**const** string&, **const** string&);
3. **void** useBigger(**const** string&, **const** string&, Func);
4. // FuncP 是 指向函数的指针（函数指针）
5. **typedef** **bool**(\*FuncP)(**const** string&, **const** string&);
6. **void** useBigger(**const** string&, **const** string&, FuncP);

方法二：typedef + decltype

1. //  Func2 是函数类型
2. **typedef** decltype(lengthCompare) Func2; // equivalent type
3. **void** useBigger(**const** string&, **const** string&, Func2);
4. // FuncP 和FuncP2 是 指向函数的指针（函数指针）
5. **typedef** decltype(lengthCompare) \*FuncP2; // equivalent type
6. **void** useBigger(**const** string&, **const** string&, FuncP2);

# 可以返回函数吗？有什么折中的办法吗？

不能返回函数，但是可以返回函数指针。

1. **using** F = **int**(**int**\*, **int**); // F 是函数类型，不是指针！
2. **using** PF = **int**(\*)(**int**\*, **int**); // PF 是指针类型（函数指针）
3. PF f1(**int**); // 正确: PF 是函数指针; f1 返回的是函数指针
4. F f1(**int**); // 错误: F是函数 ; f1 并不能返回函数
5. F \*f1(**int**); // 正确: 虽然F是函数，但F\* 是指针，因此fa返回的是函数指针

不使用别名，我们也可以这么写：**int** (\*f1(**int**))(**int**\*, **int**)，只不过不那么好理解，现在我们来分析一下这个声明：

# 声明一个函数，这个函数 返回 指向int func(int\*, int) 的函数指针，并 接受一个int实参

**int** (\*f1(**int**))(**int**\*, **int**)

注意！函数的指针 指向的函数 的形参列表和f1一起放在括号里，它指向的函数的返回值放在最右边，这和 返回指向数组的函数 很像：

**int** (\*func(**int** i))[10];//接受一个int参数，返回指向10个int的指针 的函数

# 如何理解下面的声明：

**int** (\*f1(**int**))(**int**\*, **int**)

我们从内到外来分析：

① f1的左边是\*，右边是(**int**)，但括号运算符比\*优先级高，因此f1有形参列表(**int**)，说明它是函数，而且他前面有指针符号\*，说明它是函数指针；

② 再看 (**int**\*, **int**)，这是函数指针f1指向的函数 的形参列表，至于它为什么写在最右边，我们来看一个返回 指向10个int的指针的函数：

**int** (\*func(**int** i))[10]; //接受一个int参数，返回指向10个int的指针 的函数

# 如何理解int (\*(\*f)(int,int))(int)  ？

还是从内到外：

① (\*f)说明f是一个函数指针；

② (\*(\*f)(**int**,**int**)) (**int**,**int**)，括号比指针符\* 级别高，因此(\*f)(**int**,**int**)意味着f是一个函数指针，它指向一个形参为两个**int**的函数，(\*(\*f)(**int**,**int**))中最前面哪个 指针符\* 意味着 函数指针f 指向的函数 的返回值也是一个指针B；

③ **int** (\*(\*f)(**int**,**int**))(**int**) 意味着 指针B 有形参列表(**int**)，也就是说指针B是一个函数指针，它接受一个int形参，并且返回一个**int**

简而言之，一个函数指针f，指向的函数有两个int形参，这个就是(\*f)(**int**,**int**)，返回的是一个函数指针B，即 **int** (\*)(**int**)。

如果用typedef简化的话，应该这么写：

1. **typedef** **int** (\*B)(**int**);  // B是返回类型
2. B (\*f)(**int**, **int**);   // 等同于 int (\*(\*f)(int,int))(int)

# 用decltype作用于函数指针类型

**有一点要注意，用decltype作用于 函数名的时候，返回的是函数类型，而不是指针类型！**

1. string::size\_type sumLength(**const** string&, **const** string&);
2. string::size\_type largerLength(**const** string&, **const** string&);
3. // 根据形参的取值，getFCN返回指向sumLength或largerLength的指针
4. decltype(sumLength) \*getFcn(**const** string &);  // decltype返回的是函数类型，而不是指针，所以要加一个“\*”