导语 在C++11中，callable object（可调用对象） 包括传统C函数，C++成员函数，函数对象 也叫做仿函数（实现了（）运算符的类的实例），lambda表达式（特殊函数对象）共4种。程序设计，特别是程序库设计时，经常需要涉及到回调，如果针对每种不同的可调用对象单独进行声明类型，代码将会非常散乱，也不灵活。

**（1）普通函数**

这种函数定义比较简单，一般声明在一个文件开头。

**（2）类成员函数**

在一个类class中定义的函数一般称为类的方法，分为成员方法和静态方法，区别是成员方法的参数列表中隐含着类this指针。

**（3）仿函数**

仿函数是使用类来模拟函数调用行为，我们只要重载一个类的operator()方法，即可像调用一个函数一样调用类。这种方式用得比较少。

**（4）函数指针**

顾名思义，函数指针可以理解为指向函数的指针。可以将函数名赋值给相同类型的函数指针，通过调用函数指针实现调用函数。

# C++11 function

介绍：**类模版std::function是一种通用、多态的函数封装。**std::function可以对任何可以调用的实体进行封装，这些目标实体包括普通函数、Lambda表达式、函数指针、以及其它函数对象等。

std::function对象是对C++中现有的可调用实体的一种类型安全的包裹（我们知道像函数指针这类可调用实体，是类型不安全的）。   
 通常std::function是一个函数对象类，它包装其它任意的函数对象，被包装的函数对象具有类型为T1, …,TN的N个参数，并且返回一个可转换到R类型的值。std::function使用 模板转换构造函数接收被包装的函数对象；特别是，闭包类型可以隐式地转换为std::function。   
 也就是说，通过std::function对C++中各种可调用实体（普通函数、Lambda表达式、函数指针、以及其它函数对象等）的封装，形成一个新的可调用的std::function对象；让我们不再纠结那么多的可调用实体。一切变的简单粗暴。

语法：包含#include <functional>

声明一个返回值为int，参数为两个int的可调用对象类型

std::function<int(int, int)> Func;

C++11 function对象示例（使用场景）:





其中需要注意对于类成员函数，因为类成员函数包含this指针参数，所以单独使用std::function是不够的，还需要结合使用std::bind函数绑定this指针以及参数列表。

function对象好处

std::function实现了一套类型消除机制，可以统一处理不同的函数对象类型。以前我们使用函数指针来完成这些；现在我们可以使用更安全的std::function来完成这些任务。

# std::bind绑定器

bind是这样一种机制，它可以预先把指定可调用实体的某些参数绑定到已有的变量，产生一个新的可调用实体（*bind就是函数适配器，C++中有三类适配器，分别是容器适配器，迭代器适配器和函数适配器，函数适配器是用来让一个函数对象表现出另外一种类型的函数对象的特征。因为，许多情况下，我们所持有的函数对象或普通函数的参数个数或是返回值类型并不是我们想要的，这时候就需要函数适配器来为我们的函数进行适配*）。绑定之后的结果可以使用std::function进行保存，并延迟调用到任何需要的时候。一般来讲，它主要有两大作用：

（1）将可调用对象与其参数一起绑定成为一个仿函数

（2）将多元可调用对象转换成为1元或是（n-1）元调用对象，既只是绑定部分参数

代码实例（使用方法）：



从上面的代码可以看到，bind能够在绑定时候就同时绑定一部分参数，未提供的参数则使用占位符表示，然后在运行时传入实际的参数值。

**资料（复习）**

**函数指针**

定义：指向函数地址的指针变量

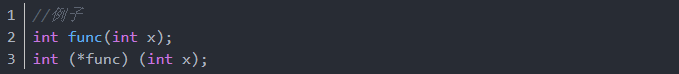
在C编译时，每一个函数都有一个入口地址，那么这个指向这个函数的函数指针便指向这个地址。

函数指针主要由以下两方面的用途：

调用函数和用作函数参数。

#### 函数指针的声明方法

函数返回值类型 +（指针变量名）+（形参列表）

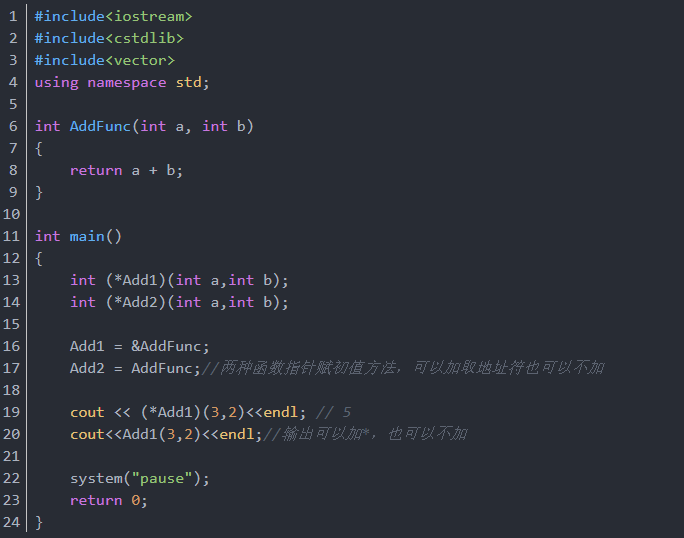


ps：注意前面的那个(\*func)中括号是必要的，

这会告诉编译器我们声明的是函数指针 而

不是声明一个具有返回型为指针的函数

使用例子



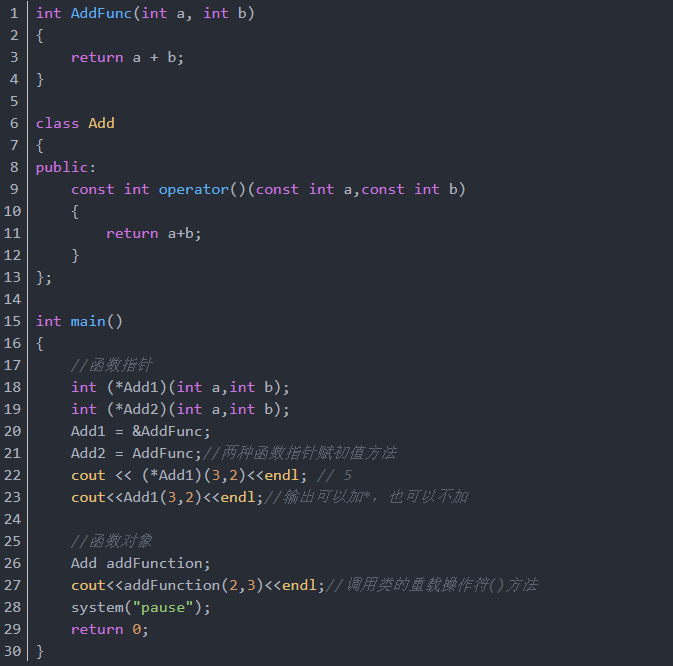
两种赋值方法：函数名是一个地址，可以将他赋值给一个指向函数的指针。前面加了&符号其意义是一样的。比如定义一个数组arr[]，arr表示这个数组的首地址，但&arr同样表示他的首地址。这些都是设计语言时这样规定的

两种输出方法：比如你定义一个字符串指针char \*str, 输出str 他并不是输出了地址，而是输出存在该地址的值，所有这些设计都是为了方便使用，而不是为了符合同一个规则。

**函数对象**

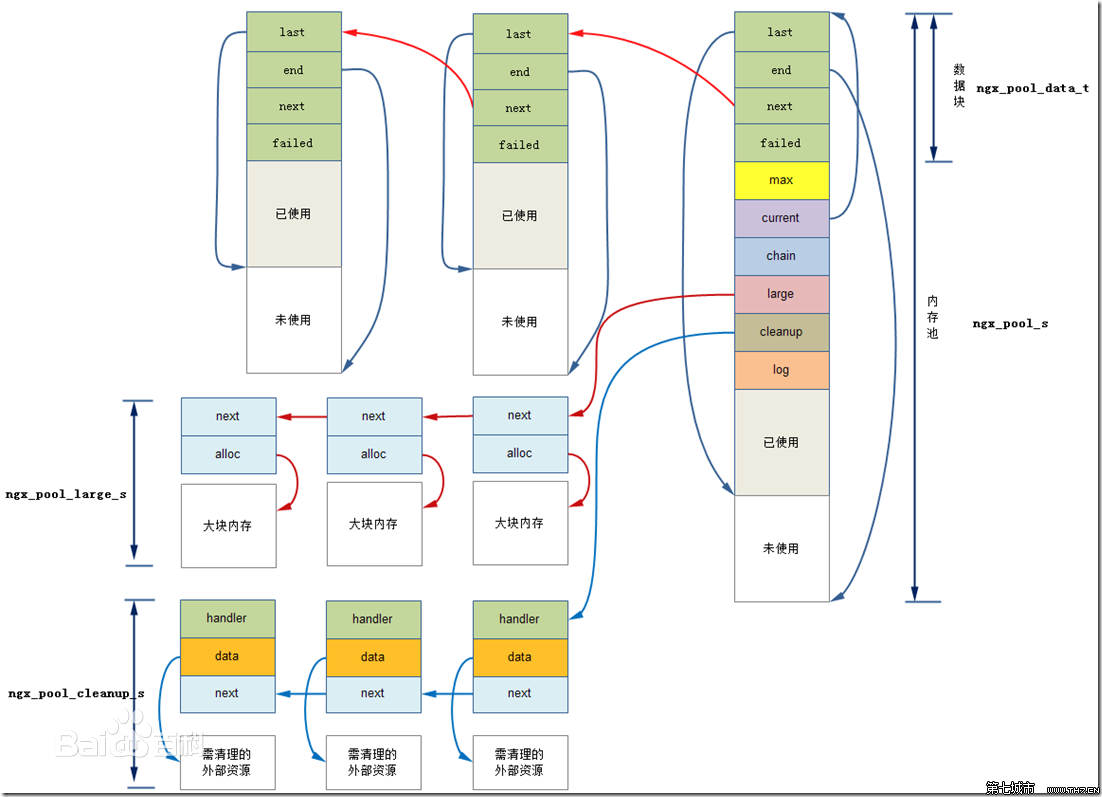
C++函数对象**实质上是操作符重载**，实现了对**()操作符**的重载。C++函数对象不是函数指针。但是，在程序代码中，**它的调用方式与函数指针一样，后面加个括号就可以了**。

函数对象例子



**函数回调**

回调函数就是一个通过[函数指针](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E6%8C%87%E9%92%88/2674905)调用的函数。如果你把函数的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304)（地址）作为[参数传递](https://baike.baidu.com/item/%E5%8F%82%E6%95%B0%E4%BC%A0%E9%80%92/9019335)给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。



**类型安全**

类型安全简单来说就是访问可以被授权访问的内存位置，类型安全的代码不会试图访问自己未被授权的内存区域。一方面，类型安全被用来形容编程语言，主要根据这门编程语言是否提供类型安全的保障机制；另一方面，类型安全也可以用来形容程序，根据这个程序是否隐含类型错误。类型安全的语言和程序之前，其实没有必然的联系。类型安全的语言，使用不当，也可能写出来类型不安全的程序；类型不安全的语言，使用得当，也可以写出非常安全的程序。

2、C的类型安全

C语言不是类型安全的语言，原因如下：

1）很多情况下，会存在类型隐式转换，比如bool自动转成int类型；

2）malloc函数返回的是void \*的空类型指针，通常需要这样的显示类型转换char\* pStr=(char\*)malloc(100\*sizeof(char))，类型匹配没有问题。但如果出现int\* pInt=(int\*)malloc(100\*sizeof(char))这样的转换，可能会带来一些问题，但C并不会提示。

当然，在有些情况下表现还是类型安全的，当从一个结构体指针转换成另一个结构体指针时，编译器会报错，除非显式转换。

3、C++的类型安全

C++也不是类型安全的语言，但远比C更具类型安全。相比于C，提供了一些安全保障机制：

1）用操作符new来申请内存，严格与对象类型匹配，而malloc是void \*；

2）函数参数为void \*的可以改写成模板，模板支持运行时检查参数类型；

3）使用const代替define来定义常量，具有类型、作用域，而不是简单的文本替换；

4）使用inline代替define来定义函数，结合函数的重载，在类型安全的前提下可以支持多种类型，如果改写成模板，会更安全；

5）提供dynamic\_cast使得转换过程更安全。

尽管如此，但如果使用空类型指针或者在两个不同类型指针间做强制转换，很可能引发类型不安全的问题。