二、函数对象包装器

这部分内容虽然属于标准库的一部分，但是从本质上来看，它却增强了 C++ 语言运行时的能力，这部分内容也相当重要，所以放到这里来进行介绍。

std::function

Lambda 表达式的本质是一个函数对象，当 Lambda 表达式的捕获列表为空时，Lambda 表达式还能够作为一个函数指针进行传递，例如：

```
#include <iostream>

using foo = void(int); // 定义函数指针, using 的使用见上一节中的别名语法
void functional(foo f) {
 f(1);
}

int main() {
 auto f = [](int value) {
  std::cout << value << std::endl;
  };
  functional(f); // 函数指针调用
  f(1); // lambda 表达式调用
  return 0;
 }
```

上面的代码给出了两种不同的调用形式，一种是将 Lambda 作为函数指针传递进行调用，而另一种则是直接调用 Lambda 表达式，在 C++11 中，统一了这些概念，将能够被调用的对象的类型，统一称之为可调用类型。而这种类型，便是通过 `std::function` 引入的。

C++11 `std::function` 是一种通用、多态的函数封装，它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，它也是对 C++中现有的可调用实体的一种类型安全的包裹（相对来说，函数指针的调用不是类型安全的），换句话说，就是函数的容器。当我们有了函数的容器之后便能够更加方便的将函数、函数指针作为对象进行处理。例如：

```
#include <functional>
#include <iostream>

int foo(int para) {
 return para;
}

int main() {
```

```
// std::function 包装了一个返回值为 int, 参数为 int 的函数
 std::function<int(int)> func = foo;

 int important = 10;
 std::function<int(int)> func2 = [&](int value) -> int {
 return 1+value+important;
 };
 std::cout << func(10) << std::endl;
 std::cout << func2(10) << std::endl;
}
```

std::bind/std::placeholder

而 `std::bind` 则是用来绑定函数调用的参数的，它解决的需求是我们有时候可能并不一定能够一次性获得调用某个函数的全部参数，通过这个函数，我们可以将部分调用参数提前绑定到函数身上成为一个新的对象，然后在参数齐全后，完成调用。例如：

```
int foo(int a, int b, int c) {
 ;
}
int main() {
 // 将参数1,2绑定到函数 foo 上，但是使用 std::placeholders::_1 来对第一个参数进行占位
 auto bindFoo = std::bind(foo, std::placeholders::_1, 1,2);
 // 这时调用 bindFoo 时，只需要提供第一个参数即可
 bindFoo(1);
}
```

提示：注意 `auto` 关键字的妙用。有时候我们可能不太熟悉一个函数的返回值类型，但是我们却可以通过 `auto` 的使用来规避这一问题的出现。