bind

概念

`bind`函数可以看作是一个通用的函数适配器，它接受一个调用对象，并生成一个新调用对象来“适应”原来调用对象的参数列表

`bind`的一般形式为：

```
auto new_callable = bind(old_callable, arg_list)
```

- `new_callable`是生成的新的调用对象
- `old_callable`是原来的调用对象
- `arg_list`是给新调用对象`new_callable`的参数列表，其中包含形如`_n`(n为正整数)的占位符，表示原先调用对象上此位置的参数映射给新调用对象第**n**个位置

算法适配

我们知道`lambda`表达式可以用于哪些只用的到一两次的地方，但是针对多次重复调用的功能，最好还是改成函数形式。但有些情况下，算法所支持的回调函数的参数与函数接口不统一

```
/*
在字符串数组中查找字符串长度为sz的字符串
*/

// lambda版本
find_if(words.begin(), words.end(), [=](string & str) { return str.length() == sz; });

// lambda表达式转函数
bool cmp_size(string &str, size_t sz)
{
    return str.length() == sz;
}
find_if(words.begin(), words.end(), cmp_size);  // error C2198: “_Pr”: 用于调用的参数太少
```

此时，`find_if`函数只向回调函数`cmp_size`提供一个参数，此函数又需要两个参数，故编译失败。而`lambda`表达式可以捕获当前所在作用域的变量，所以变相的向其传递了两个参数，普通函数则不能

想解决此问题，需要`bind`函数来完成函数的绑定工作

```
// 将cmp_size绑定到new_cmp上
auto new_cmp = bind(cmp_size, placeholders::_1, sz);
find_if(words.begin(), words.end(), new_cmp);
```

- `placeholders::_1`就是占位符，,此位置是`arg_list`的第一个位置，表示旧调用对象`cmp_size`第一个参数的位置，映射到新调用对象`new_cmp`的第一个参数

- `sz`表示第二个参数绑定到此变量上
- 调用`new_cmp(words[0])`就代表`cmp_size(words[0], sz)`

修正参数

与上例类似，假定有一函数`foo`，接受**5**个参数，调用如下`bind`

```
auto bar = bind(foo, a, b, _2, c, _1);
```

则`foo`函数的**第一**、**第二**和**第四**个参数会绑定到变量`a`、`b`和`c`，**第三**和**第五**个参数会绑定到`bar`的**第二**和**第一**参数上

即调用`bar(X, Y)`等价于`foo(a, b, Y, c, X)`

参数重排序

假定有函数`foo`接受两个参数，将这两个参数位置对调

```
auto bar = bind(foo, _2, _1);
```

`foo`的**第一**个参数在`bar`的**第二个**参数位置上，`foo`的**第二个**参数在`bar`的**第一个**参数位置上

即调用`bar(X, Y)`等价于`foo(Y, X)`

绑定引用参数

被绑定的参数是通过拷贝才到新调用对象的（同`lambda`类似），对于引用来说是无法拷贝过去，需利用`ref`来返回一个引用或者`cref`返回一个`const`引用来完成

```
// 打印str并以split_ch分割
ostream & print(ostream &os, string &str, char split_ch)
{
    return os << str << split_ch;
}

// c为本作用域下的变量
for_each(words.begin(), words.end(), bind(print, ref(cout), placeholders::_1, c));
```