C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B)

STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90)

STL 源码剖笔记(四)

STL 源码剖析笔记(四)

Posted by 敬方 on July 6, 2019

> *2019-08-04 21:47:52*

第七章 仿函数

仿函数也叫对象函数；在STL提供的各种算法中，都允许用户自定义相关算法。以结果的false或者true来进行相关的排序操作，用来执行函数的就是仿函数。一般步骤是先设计一个函数，在将函数的相关指针指向函数对应的结果。

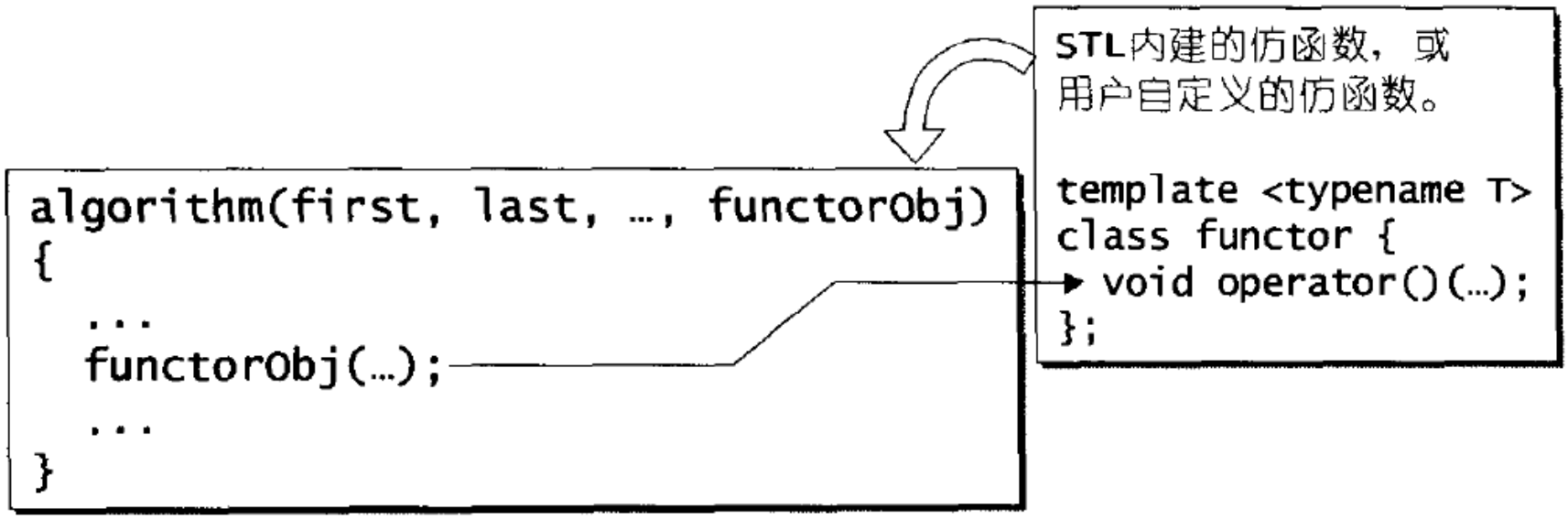


图 7-1 STL 仿函数与 STL 算法之间的关系

仿函数的分类：

- 操作数的个数：一元和二元仿函数
- 功能划分:算术运算、关系运算、逻辑运算。

使用STL内建的仿函数，都必须含有头文件。

7.2 可配接(Adaptable)的关键

- unary_function 用来呈现一元函数的参数型别和返回值类型。

- binary_function 用来呈现二元函数的参数型别和返回值类型。

```
template <class Arg1,class Arg2,class Result>

struct binary_function
{
    typdef Arg1 first_argument_type;
    typdef Arg2 second_argument_type;
    typdef Result result_type;
};
```

7.3 算术类(Arithmetic)仿函数

- 加法：plus
- 减法：minus
- 乘法:multiplies
- 除法：divides
- 膜取(modulus):modulus
- 否定(negation):negate

```
template <class T>
struct plus:public binary_function<T,T,T>
{
    T operator()(const T& x,const T& y) const {return x+y;}
};
```

证同元素

数值A若与该元素做OP运算，会得到A自己。加法的同证元素为0，因为任何匀速加上0仍为自己。乘法的同证元素为1，因为任何元素乘以1任然为自己

7.4 关系运算类仿函数

他们都是二元的运算函数

- 等于：equal_to
- 不等于：not_equal_to
- 大于: greater
- 大于或者等于：greater_equal
- 小于：less
- 小于或者等于:less_equal

7.5 逻辑运算类仿函数

其中And和Or为二元运算符，Not为一元运算符

- And:logical_and
- Or:logical_or
- Not:logical_not

7.6 证同(identity)、选择(select)、投射(project)

- 证同：任何数值通过此函数后，不会有任何改变，此式运用于用来指定RB-tree所需要的KeyOfValue op ， 因为set元素的键值即实值，所以采用identity

```
template <class T>
struct identity:public unary_function<T,T>
{
    const T& operator(){const T& x} const {return x;}
};
```

- 选择： 接受一个pair,传回第一元素。此式运用于用来指定RB-tree所需要的KeyOfValue op ， 因为map系以pair元素的第一个元素为其键值，所以采用select1st。
- 投射函数：传回第一参数，忽略第二参数

第8章 配接器

Adapter实际上是一种设计模式，将一个class的借口转换为另外一个class的接口。

8.1 配接器之概观与分类

- function adapter:改变仿函数(functors)接口
- container adapter:改变容器(container)接口
- iterators adapter:改变迭代器借口

辅助函数（helper function）	实际效果	实际产生的对象
bind1st(const Op& op, const T& x);	op(x, param);	binder1st<Op> (op, arg1_type(x))
bind2nd(const Op& op, const T& x);	op(param, x);	binder2nd<Op> (op, arg2_type(x))
not1(const Pred& pred);	!pred(param);	unary_negate<Pred> (pred)
not2(const Pred& pred);	!pred (param1,param2);	binary_negate<Pred> (pred)
compose1(const Op1& op1, const Op2& op2);	op1(op2(param));	unary_compose<Op1,Op2> (op1, op2)
compose2(const Op1& op1, const Op2& op2, const Op3& op3);	op1(op2(param) op3(param));	binary_compose<Op1,Op2,Op3> (op1, op2, op3)
ptr_fun (Result(*fp)(Arg));	fp(param);	pointer_to_unary_function <Arg, Result>(fp)
ptr_fun (Result(*fp)(Arg1,Arg2));	fp(param1 param2);	pointer_to_binary_function <Arg1, Arg2, Result>(fp)
mem_fun(S (T::*f)());	(param->*f)();	mem_fun_t<S,T>(f)
mem_fun(S (T::*f)() const);	(param->*f)();	const_mem_fun_t<S,T>(f)
mem_fun_ref(S (T::*f)());	(param.*f)();	mem_fun_ref_t<S,T>(f)
mem_fun_ref (S (T::*f)() const);	(param.*f)();	const_mem_fun_ref_t<S,T>(f)
mem_fun1(S (T::*f)(A));	(param->*f)(x);	mem_fun1_t<S,T,A>(f)
mem_fun1(S (T::*f)(A)const);	(param->*f)(x);	const_mem_fun1_t<S,T,A>(f)
mem_fun1_ref(S (T::*f)(A));	(param.*f)(x);	mem_fun1_ref_t<S,T,A>(f)

mem_fun1_ref (S (T::*f)(A)const);	(param.*f)(x);	const_mem_fun1_ref_t<S,T,A> (f)

- compose1 和 compose2 不在 C++ *Standard* 规范之中。
- 最后四个辅助函数在 C++ *Standard* 中已去除名称中的'1'，与其前四个辅助函数形成重载。

图 **8-2** 各种 function adapters 及其辅助函数，以及实际效果。

此图等于是相关源代码的接口整理。搭配源代码阅读，更得益处。

8.2 container adapter

stack和queue的底层都是使用deque构成。

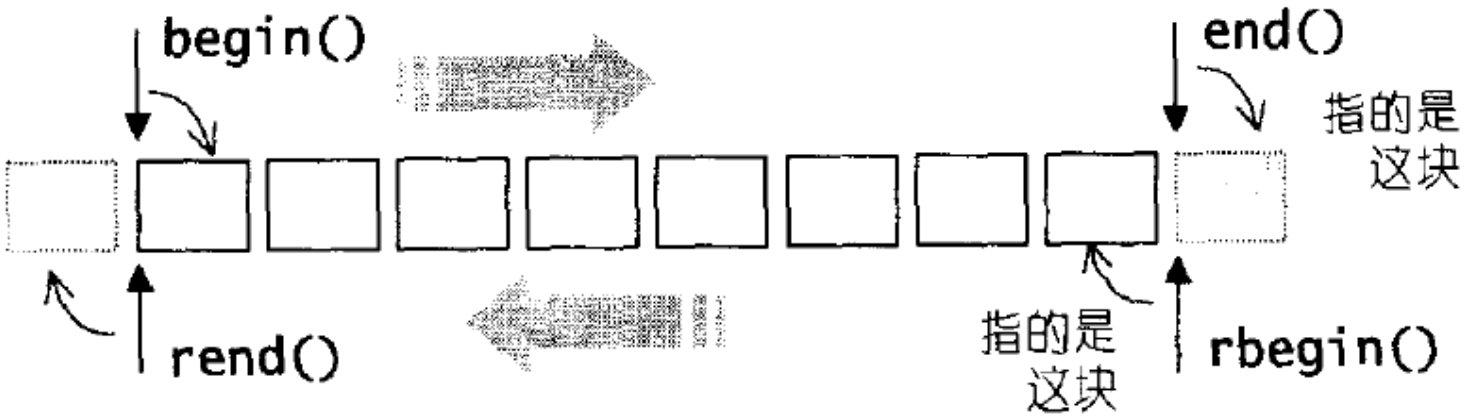
8.3 iterator adapters

insert iterator底层的调用是push_front()或者push_back()或者insert()操作函数。

8.3.2 reverse iterators

```
deque<int> id={32,26,99,1,0,1,2,3,4,0,1,2,5,3};
//32

cout <<*(id.begin())<<endl;
//3

cout <<*(id.rbegin())<<endl;
//0

cout <<*(id.end())<<endl;
//0

cout <<*(id.rend())<<endl;
//查找对应的值

deque<int>::iterator ite=find(id.begin(),id.end(),99);
reverse_iterator<deque<int>::iterator> rite(ite);
//99

cout<<*ite<<endl;
//26

cout<<*rite<<endl;
```

注意，由于迭代器区间的“前闭后开”的习惯，当迭代器被逆向方向时，虽然实体位置(真正的地址)不变，但其逻辑位置(迭代器所代表的元素)改变了(必须如此改变)：因此正向和逆向迭代器所取出的是不同的元素：




```
#typedef ...    reverse_iterator
rbegin()=reverse_iterator(end());
```

图 **8-3** 当迭代器被逆转，虽然实体位置不变，但逻辑位置必须如此改变

对逆向迭代器取值，就是将“对应之正向迭代器”后退一格而后取值。

8.3.3 stream iterators

可以将一个迭代器绑定到一格stream对象上。绑定到istream对象例如(std::cin)者，就是istream_iterator拥有输入能力，同理输出对象上有输出能力。

图 8-4 所示的是 copy() 和 istream_iterator 共同合作的例子。

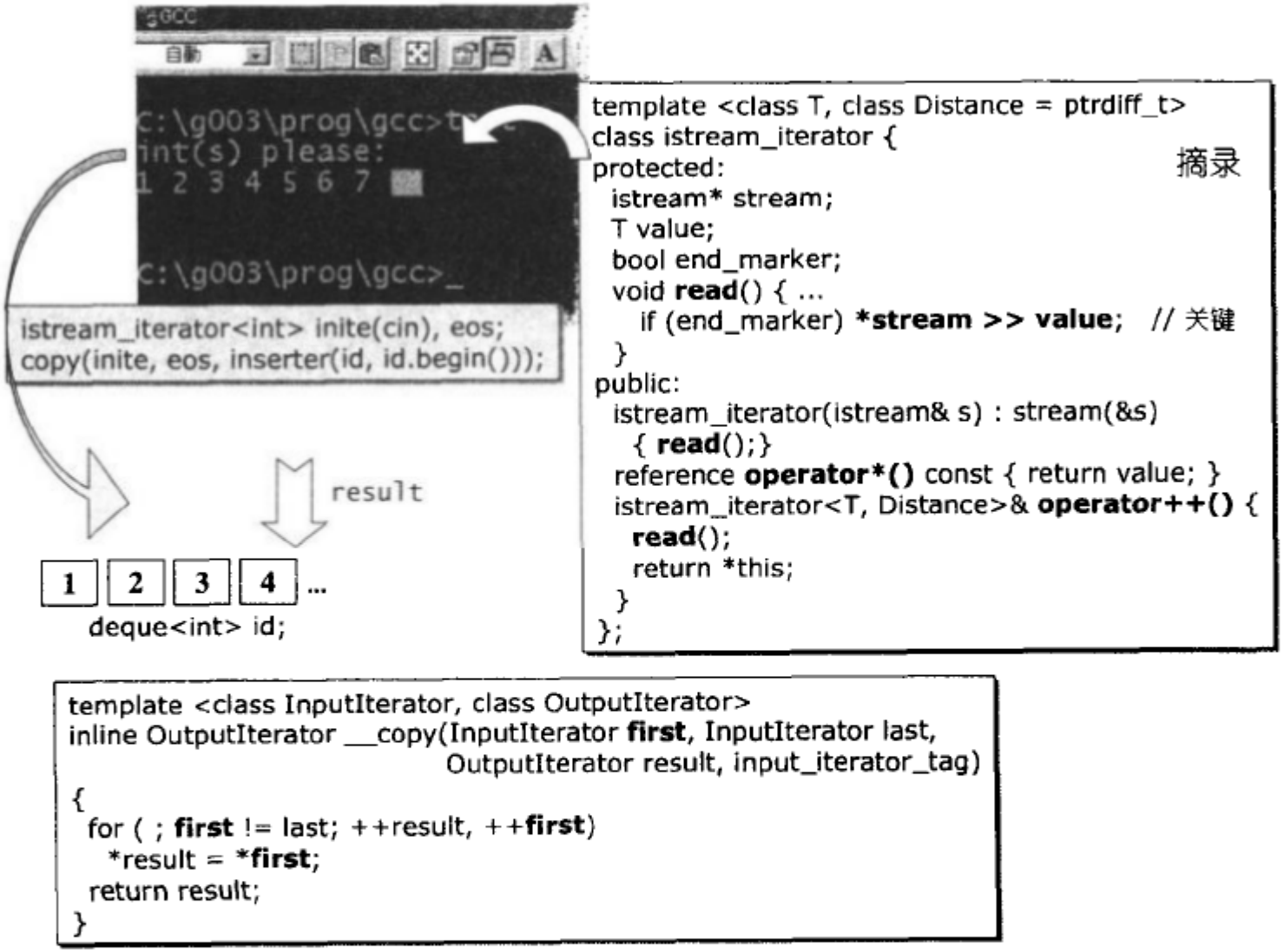


图 **8-4** copy() 和 istream_iterator 合作。屏幕画面下方的浅蓝色方块是客户端程序代码，程序流程将停在其中第一行，等待用户从 cin 输入（因为右侧的

istream_iterator 源代码告诉我们，一产生 istream_iterator 对象，便会调用 read()，执行 *stream >> value，而 *stream 在本例就是 cin）。用户输入数值后，copy 算法（源代码见最下方块内容）将该数值插入到容器之内，然后执行 istream_iterator 的 operator++ 操作——这又再次引发 istream_iterator::read()，准备读入下一笔资料…

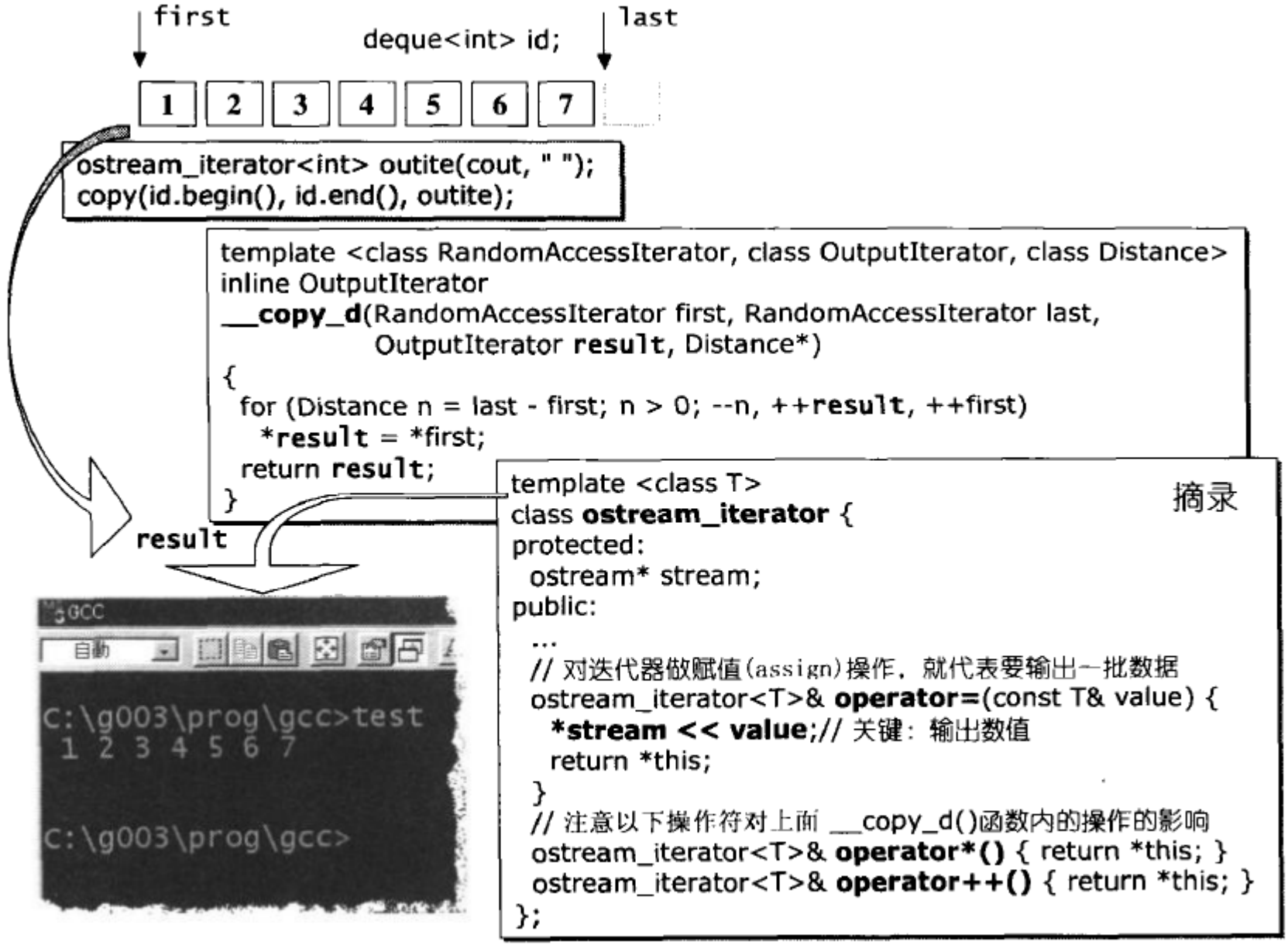


图 **8-5** copy() 和 ostream_iterator 合作。见内文说明。

8.4 function adapters

容器是以class templates完成，算法是以function templates完成，仿函数是一种将operator()重载的class template,迭代器则是一种将operator++和operator*等指针习惯性常行为重载的class template。

function adapters也内藏了一格member object

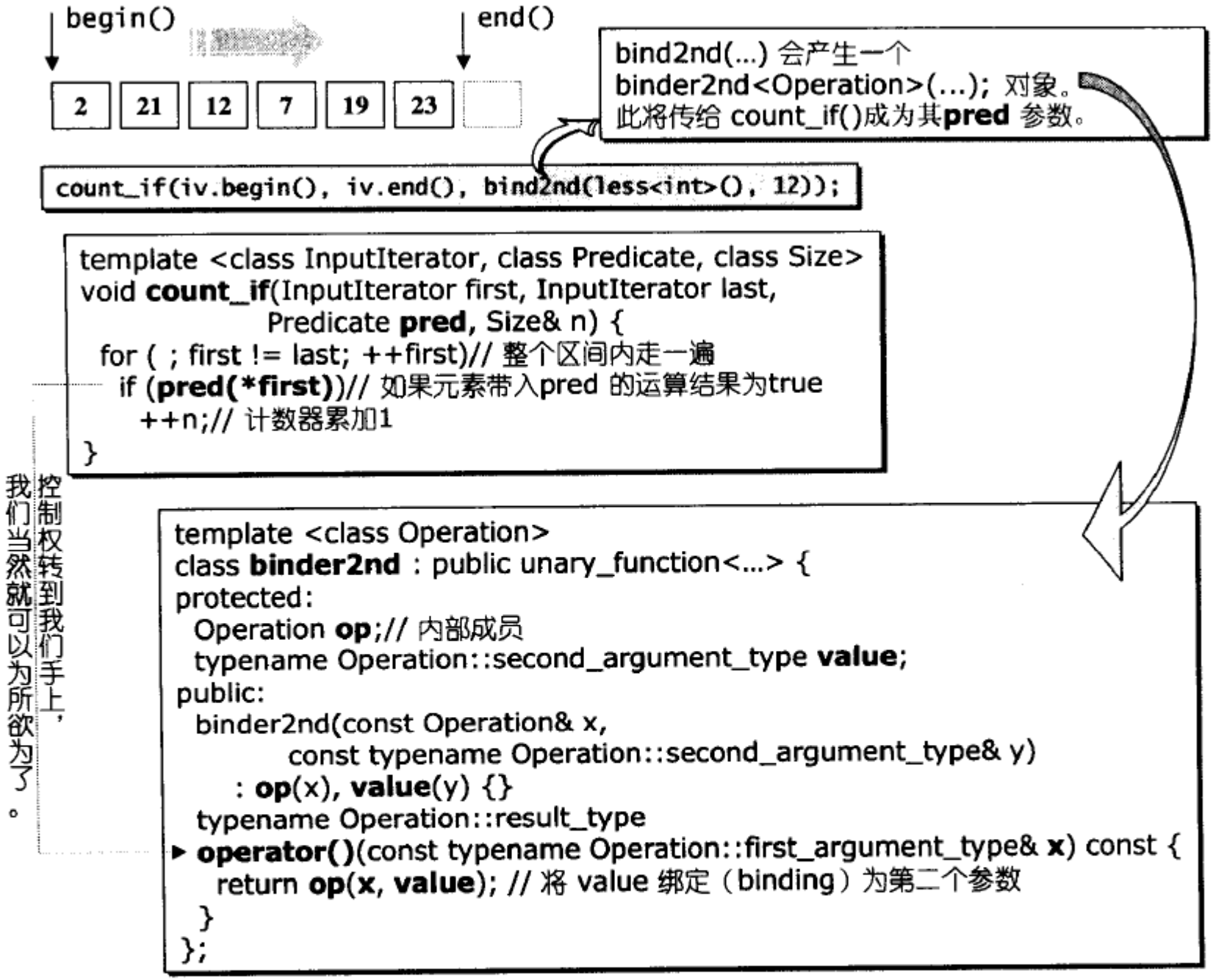


图 **8-7** 鸟瞰 `count_if()` 和 `bind2nd(less<int>(), 12))` 的搭配实例。此

图等于是相关源代码的接口整理，搭配 bind2nd() 以及 class binder2nd 源代码阅读，更得益处。图中浅色底纹方块为客户端调用 count_if() 实况；循着箭头行进，便能理解的整个合作机制。

因此function adapters多是在内部实现函数的合成与操作

8.4.4 用于函数指针： ptr_fun

这种配接器使得我们能够将一般函数当做仿函数使用。一般函数当做仿函数传给STL算法。其实质就是把一个函数指针包裹起来。

```
//当仿函数被使用时，就调用该函数指针,这里是一元函数指针的封装

template <class Arg,class Result>
class pointer_to_unary_function:public unary_function<Arg,Result>
{
protected:
    //内部成员，一个函数指针

    Result (*Ptr)(Arg);
public:
    pointer_to_unary_function(){}
    //以下constructor将函数指针记录于内部成员之中

    explicit pointer_to_unary_function(Result (*x)(Arg)):ptr(x){}
    //以下，通过函数指针指向函数

    Result operator()(Arg x) const {return ptr(x);}
};
//辅助函数，让我们得以方便运用

template <class Arg,class Result>
inline pointer_to_unary_function<Arg,Result> ptr_fun(Result (*x)(Arg))
{
    return pointer_to_unary_function<Arg,Result>(x);
}

//二元函数指针的封装

template <class Arg1,class Arg2,class Result>
class pointer_to_binary_function:public binary_function<Arg1,Arg2,Result>{
protected:
    //内部成员，一个函数指针

    Result (*ptr)(Arg1,Arg2);
public:
    pointer_to_binary_function(){}
    //将函数指针记录在内部成员之中

    explicit pointer_to_binary_function(Result (*x)(Arg1,Arg2)):ptr(x){}
```

```
    //函数指针执行函数

    Result operator()(Arg1 x,Arg2 y) const {return ptr(x,y);}
};
//辅助函数，方便函数的使用

template<class Arg1,class Arg2,class Result>
//定义返回类型

inline pointer_to_binary_function<Arg1,Arg2,Result>
ptr_fun(Result (*x)(Arg1,Arg2))
{
    return pointer_to_binary_function<Arg1,Arg2,Result>(x);
}
```

8.4.5 用于成员函数指针：mem_fun,mem_fun_ref;

这种配接器使得我们能够将成员函数当做仿函数来进行使用，使得成员函数可以搭配各种泛型算法。

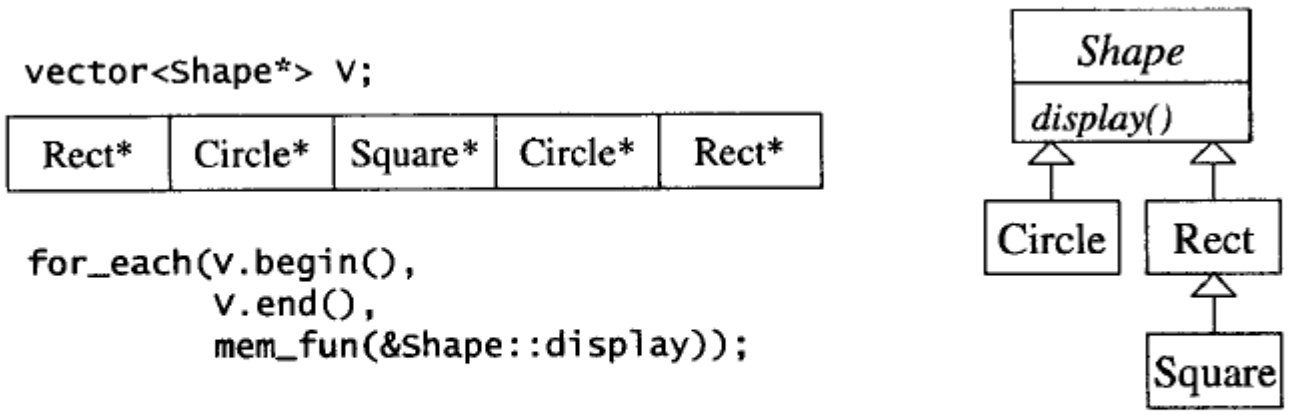


图 **8-8** 图右是类阶层体系（classes hierarchy），图左是实例所产生的容器状态。

PREVIOUS
STL 源码剖笔记(三)
(/2019/07/06/CPLUSPLUS_ANNOTATED_STL_SOURCES_03/)

NEXT
跟我一起写MAKEFILE 学习笔记
(/2019/07/06/WRITE_MAKEFILE_WITH_ME/)

FEATURED TAGS (/tags/)

C++ (/tags/#C++) 基础编程 (/tags/#%E5%9F%BA%E7%A1%80%E7%BC%96%E7%A8%8B) C/C++ (/tags/#C/C++)
后台开发 (/tags/#%E5%90%8E%E5%8F%B0%E5%BC%80%E5%8F%91) C (/tags/#C) 网络编程 (/tags/#%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
STL源码解析 (/tags/#STL%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90) Linux (/tags/#Linux)
操作系统 (/tags/#%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F) 程序设计 (/tags/#%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)
优化 (/tags/#%E4%BC%98%E5%8C%96) UML (/tags/#UML) UNIX (/tags/#UNIX) 学习笔记 (/tags/#%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0)
面试 (/tags/#%E9%9D%A2%E8%AF%95) Java (/tags/#Java) 读书笔记 (/tags/#%E8%AF%BB%E4%B9%A6%E7%AC%94%E8%AE%B0) go (/tags/#go)
阅读笔记 (/tags/#%E9%98%85%E8%AF%BB%E7%AC%94%E8%AE%B0)

FRIENDS

WY (http://zhengwuyang.com) 简书·JF (http://www.jianshu.com/u/e71990ada2fd) Apple (https://apple.com)
Apple Developer (https://developer.apple.com/)

● (https://www.facebook.com/wangpengcheng) ● (https://github.com/wangpengcheng)