Module05-05
C++ Boost：函数对象相关

C++ Boost - 函数对象相关

- 容器相关
- 字符串和文字处理
- 正则表达式
- 智能指针
- ➔ 函数对象相关
- 序列化
- 日期与时间
- 多线程
- 网络

函数对象相关 - boost.bind

- 函数对象相关：
  - boost.bind (In TR1)
  - boost.mem_fn (In TR1)
  - boost.function (In TR1)
  - boost.ref (In TR1)
  - boost.lambda

函数对象相关 - boost.bind

- 关于 bind
  - boost::bind 是标准函数 std::bind1st 和 std::bind2nd 的泛化。它支持任意的函数对象，函数，函数指针，和成员函数指针，它还能将任何参数绑定为一个特定的值，或者将输入的参数发送到任意的位置。 bind 对函数对象没有任何要求，特别是，它不需要 result_type ， first_argument_type 和 second_argument_type 这样的标准 typedefs 。
  - 使用 boost.bind 需包含头文件 <boost/bind.hpp>

函数对象相关 - boost.bind

- bind 接口

```
// no argument
template<class R, class F> unspecified-1 bind(F f);
template<class F> unspecified-1-1 bind(F f);
template<class R> unspecified-2 bind(R (*f) ());

// one argument
template<class R, class F, class A1>
unspecified-3 bind(F f, A1 a1);
template<class F, class A1>
unspecified-3-1 bind(F f, A1 a1);
template<class R, class B1, class A1>
unspecified-4 bind(R (*f) (B1), A1 a1);
template<class R, class T, class A1>
unspecified-5 bind(R (T::*f) (), A1 a1);
template<class R, class T, class A1>
unspecified-6 bind(R (T::*f) () const, A1 a1);
template<class R, class T, class A1>
unspecified-6-1 bind(R T::*f, A1 a1);
```

函数对象相关 - boost.bind

- bind 接口

```
// two arguments
template<class R, class F, class A1, class A2>
unspecified-7 bind(F f, A1 a1, A2 a2);
template<class F, class A1, class A2>
unspecified-7-1 bind(F f, A1 a1, A2 a2);
template<class R, class B1, class B2, class A1, class A2>
unspecified-8 bind(R (*f) (B1, B2), A1 a1, A2 a2);
template<class R, class T, class B1, class A1, class A2>
unspecified-9 bind(R (T::*f) (B1), A1 a1, A2 a2);
template<class R, class T, class B1, class A1, class A2>
unspecified-10 bind(R (T::*f) (B1) const, A1 a1, A2 a2);

// 更多参数,形式同上
// ...
```

函数对象相关 - boost.bind

- bind 用于函数以及函数指针

```
int f(int a, int b) {
    return a + b;
}

int g(int a, int b, int c) {
    return a + b + c;
}

void test() {
    bind(f, 1, 8); // 产生一个无元函数对象，返回f(1,8)
    bind(g, 1, 8, 12); // 同上
    int x = 108;
    bind(f, _1, 5)(x); // 返回f(x,5)
    std::bind2nd(std::ptr_fun(f), 5)(x); // 同上，标准库版本
    bind(f, 5, _1)(x); // 返回f(5, x)
    std::bind1st(std::ptr_fun(f), 5)(x); // 同上，标准库版本

    bind(f, ref(x), _1)(32); // 传参数的引用，而不是参数的副本
    bind(f, cref(42), _1)(28); // 参数的const引用
}
```

函数对象相关 - boost.bind

- bind 用于函数对象

```
struct F {
    int operator()(int a, int b) {
        return a - b;
    }
    bool operator()(long a, long b) {
        return a == b;
    }
};

void func() {
    F f;
    int x = 108;
    // 由于函数对象 F 内部没有定义 return_type 类型,
    // 所以下面需显式写成bind<int>
    bind<int> (f, _1, _1)(x); // f(x, x)

    // 由于函数对象less<>内部定义了return_type类型,
    // 所以下面的bind不需显式写成bind<int>
    bind(std::less<int>(), _1, 9)(x);    // x < 9
}
```

函数对象相关 - boost.bind

- bind 用于函数对象（用引用避免函数对象的拷贝）

```
struct F2 {
    int s;
    typedef void result_type;
    void operator()(int x) {
        s += x;
    }
};

void func() {
    F2 f2 = { 0 };
    int a[] = { 1, 2, 3 };
    std::for_each(a, a + 3, bind(ref(f2), _1)); // 传f2的引用
    assert( f2.s == 6 ); // 这样f2对象的状态得以延续
}
```

函数对象相关 - boost.bind

- bind 用于成员指针

```
struct X {
    bool f(int a);
};

void func() {
    X x;
    shared_ptr<X> p(new X);

    int i = 5;
    bind(&X::f, ref(x), _1)(i); // x.f(i)
    bind(&X::f, &x, _1)(i); //(&x)->f(i)
    bind(&X::f, x, _1)(i); // (internal copy of x).f(i)
    bind(&X::f, p, _1)(i); // (internal copy of p)->f(i)
}
```

函数对象相关 - boost.bind

- 嵌套 bind
  - bind 可以嵌套，如：
    - `bind(f, bind(g, _1))(x);  // f(g(x))`
  - 注意第一个参数——被绑定函数对象——是不被求值的，即使它是一个由 bind 生成的函数对象或一个占位符参数，所以下面的示例不会如我们预期地工作：

```
typedef void (*pf)(int);
std::vector<pf> v;
std::for_each(v.begin(), v.end(), bind(_1, 5));
```

可以通过将一个辅助函数对象 apply 用作它的第一个参数而获得，作为一个函数对象，它可以支撑它的参数列表。为了方便起见，在 boost/bind/apply.hpp 头文件中提供了一个 apply 的实现。

函数对象相关 - boost.bind

- 嵌套 bind
  - 完整的 apply 示例

```
void f1(int i) {
    cout << "f1() " << i << endl;
}
void f2(int i) {
    cout << "f2() " << i << endl;
}

int main() {
    typedef void (*pf)(int);
    std::vector<pf> v;
    v.push_back(f1);
    v.push_back(f2);

    std::for_each(v.begin(), v.end(),
                  bind(apply<void> (), _1, 5));
}
```

函数对象相关 - boost.bind

- bind 其它示例
  - (DEMO Using Boost Bind Examples)

函数对象相关 - boost.mem_fn

- 函数对象相关：
  - boost.bind (In TR1)
  - boost.mem_fn (In TR1)
  - boost.function (In TR1)
  - boost.ref (In TR1)
  - boost.lambda

函数对象相关 - boost.mem_fn

- 关于 mem_fn
  - boost::mem_fn 是标准函数 std::mem_fun 和 std::mem_fun_ref 的泛化。它支持带有多个参数的成员函数指针，并且返回的函数对象可以持有一个对象实例的指针，引用或者智能指针作为它的第一个参数。 mem_fn 也支持指向数据成员的指针，它把它们看作不持有参数且返回一个成员的（常量）引用的函数。
  - 使用 boost.mem_fn 需包含 <boost/mem_fn.hpp>

函数对象相关 - boost.mem_fn

- mem_fn 接口

```
template<class T> T * get_pointer(T * p);
template<class R, class T> unspecified-1 mem_fn(R (T::*pmf)
());
template<class R, class T> unspecified-2 mem_fn(R (T::*pmf) ()
const);
template<class R, class T> unspecified-2-1 mem_fn(R T::*pm);
template<class R, class T, class A1> unspecified-3 mem_fn(R
(T::*pmf) (A1));
template<class R, class T, class A1> unspecified-4 mem_fn(R
(T::*pmf) (A1) const);
template<class R, class T, class A1, class A2> unspecified-5
mem_fn(R (T::*pmf) (A1, A2));
template<class R, class T, class A1, class A2> unspecified-6
mem_fn(R (T::*pmf) (A1, A2) const);

// 更多参数的重载
```

函数对象相关 - boost.mem_fn

- mem_fn 示例

```
struct X {
    void f();
};

void g(std::vector<X>& v) {
    std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}

void h(std::vector<X*> const& v) {
    std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}

void k(std::vector<boost::shared_ptr<X> > const& v) {
    std::for_each(v.begin(), v.end(), boost::mem_fn(&X::f));
}
```

函数对象相关 - boost.function

- 函数对象相关：
  - boost.bind (In TR1)
  - boost.mem_fn (In TR1)
  - boost.function (In TR1)
  - boost.ref (In TR1)
  - boost.lambda
  - boost.signal2

函数对象相关 - boost.function

- 关于 boost.function
  - Boost.Function 库包含一组作为 function object wrappers（函数对象包装类）的类模板。在概念上类似一个泛化的 callback（回调）。它在两种情况下具有和函数指针相同的特性，一种是定义一个可用于某些可调用实现的调用接口（例如，一个持有两个整型参数并返回一个浮点值的函数），另一种是在整个程序的流程中可能变化的调用。
  - 通常，使用函数指针的任何地方都是用来推迟一个调用或做一个回调， Boost.Function 可以代替函数指针，允许用户在目标的实现上拥有更大的弹性。目标可以是任何“兼容的”函数对象（或函数指针），这意味着传给接口的参数被 Boost.Function 指定为可以转换为目标函数对象的参数。
  - 使用 boost.function 需包含 <boost/function.hpp>

函数对象相关 - boost.function

- 一些简单的示例：

```
struct int_div {
    float operator()(int x, int y) const {
        return ((float) x) / y;
    }
};
float float_div(float x, float y) {
    return x / y;
}
void test(boost::function<float(float x, float y)> const& f) {
    cout << f(12, 45) << endl;
}
int main() {
    boost::function<float(int x, int y)> f1;
    boost::function2<float, int, int> f2;
    f1 = int_div();
    f2 = int_div();
    cout << f1(5, 3) << endl;

    test(&float_div);
}
```

函数对象相关 - boost.function

- 包装类的成员函数的示例：

```
struct X {
    int foo(int);
};

int main() {
    boost::function<int(X*, int)> f;
    f = &X::foo;
    X x;
    f(&x, 5);

    // 或按下面的方式，使用functionN
    boost::function2<int, X*, int> f2;
    f2 = &X::foo;
    X x2;
    f(&x2, 5);

    // 当然最好的是使用bind，而不是function<>！
}
```

函数对象相关 - boost.function

- 使用 function 对象的引用的示例：

```
stateful_type a_function_object;
boost::function<int (int)> f;
f = boost::ref(a_function_object);

boost::function<int (int)> f2(f);

// 或使用下面的方式，使用functionN
stateful_type a_function_object;
boost::function1<int, int> f;
f = boost::ref(a_function_object);

boost::function1<int, int> f2(f);
```

函数对象相关 - boost.function

- functionN<> 接口（部分）：

```
template<typename R, typename T1, typename T2, /*...*/,
typename TN>
class functionN : public function_base {
public:
    // types
    typedef R result_type;
    typedef T1 argument_type; // If N == 1
    typedef T1 first_argument_type; // If N == 2
    typedef T2 second_argument_type; // If N == 2
    typedef T1 arg1_type;
    typedef T2 arg2_type;
    //...
    typedef TN argN_type;
    // static constants
    static const int arity = N;
    // Lambda library support
    template<typename Args>
    struct sig { typedef result_type type; }; // types
```

函数对象相关 - boost.function

- functionN<> 接口（部分）（续 1 ）：

```
    // construct/copy/destruct
    functionN();
    functionN(const functionN&);
    template<typename F> functionN(F);
    template<typename F, typename Allocator> functionN(F,
Allocator);
    functionN& operator=(const functionN&);
    ~functionN();

    // modifiers
    void swap(const functionN&);
    void clear();

    // capacity
    bool empty() const;
    operator safe_bool() const;
    bool operator!() const;
```

函数对象相关 - boost.function

- functionN<> 接口（部分）（续 2 ）：

```
    // target access
    template<typename Functor> Functor* target();
    template<typename Functor> const Functor* target() const;
    template<typename Functor> bool contains(const Functor&)
const;
    const std::type_info& target_type() const;
    // invocation
    result_type operator()(arg1_type, arg2_type, ...,
argN_type) const;
};
```

- 注意：
  - functionN<> 并不是该类型的名称，而是一个代称，泛指 function0<>~function10<>

函数对象相关 - boost.function

- function<> 接口：
  - 类模板 function<> 继承于 functionN<>，接口类似，
  - 注意 function<> 和 functionN<> 的声明方式有区别：

```
// 接受 float f(int x, int y) 形式的函数和函数对象
boost::function<float(int x, int y)> f1;
boost::function2<float, int, int> f2;
```

函数对象相关 - boost.ref

- 函数对象相关：
  - boost.bind (In TR1)
  - boost.mem_fn (In TR1)
  - boost.function (In TR1)
  - boost.ref (In TR1)
  - boost.lambda

函数对象相关 - boost.ref

- 关于 boost.ref
  - Ref 库是一个小型库，对于把引用传递给函数模板（算法）非常有用，这些函数模板（算法）通常都接收它们的参数的拷贝。它定义了类模板 boost::reference_wrapper<T>，两个返回 boost::reference_wrapper<T> 实例的函数 boost::ref 和 boost::cref，以及两个 traits classes（特征类）boost::is_reference_wrapper<T> 和 boost::unwrap_reference<T>。
  - boost::reference_wrapper<T> 的目的是容纳一个引向类型为 T 的对象的引用。它主要用于把引用“喂”给那些以传值方式持有它们的参数的函数模板（算法）。为了支持这个用法，boost::reference_wrapper<T> 提供一个到 T& 的隐式转换。这通常允许函数模板可以不加改变地工作在引用上。

函数对象相关 - boost.ref

- 关于 boost.ref（续）
  - boost::reference_wrapper<T> 是 CopyConstructible（可拷贝构造）的，也是 Assignable（可赋值）的（普通引用不是 Assignable（可赋值）的）。
  - 表达式 boost::ref(x) 返回一个 boost::reference_wrapper<X>(x)， X 是 x 的类型。类似地，boost::cref(x) 返回一个 boost::reference_wrapper<X const>(x)。
  - 如果 T 是一个 reference_wrapper，则表达式 boost::is_reference_wrapper<T>::value 的值为 true，否则为 false。
  - 如果 T 是一个 reference_wrapper，则类型表达式 boost::unwrap_reference<T>::type 的值为 T::type，否则为 T。

函数对象相关 - boost.ref

- boost.ref 相关接口

```
template<typename T>
class reference_wrapper {
public:
    // types
    typedef T type;
    // construct/copy/destruct
    explicit reference_wrapper(T&);

    // access
    operator T&() const;
    T& get() const;
    T* get_pointer() const;
};

// constructors
reference_wrapper<T> ref(T&);
reference_wrapper<T const> cref(T const&);
```

函数对象相关 - boost.lambda

- 函数对象相关：
  - boost.bind (In TR1)
  - boost.mem_fn (In TR1)
  - boost.function (In TR1)
  - boost.ref (In TR1)
  - boost.lambda

函数对象相关 - boost.lambda

- 关于 boost.lambda
  - Boost Lambda Library（简称为 BLL）是一个 C++ 模板库，为 C++ 实现了 lambda abstractions 的形式。这个术语起源于函数式编程和 lambda 演算，一个 lambda abstraction 定义一个无名函数。BLL 的主要动机是为定义供 STL 算法使用的无名函数对象提供灵活性和便利性
  - 一个 lambda 表达式定义一个函数。在求值的时候，一个 C++ lambda 表达式实际上构造了一个函数对象，一个 functor（仿函数）。我们用名字 lambda functor（lambda 仿函数）来指涉这样一个函数对象。因此，在此采纳的术语中，一个 lambda 表达式的求值结果是一个 lambda 仿函数。

函数对象相关 - boost.lambda

- 以往的方式（一个简单的示例）

```
void add5(int& src) {
    src += 5;
}

struct PrintV {
    void operator()(int v) {
        cout << v << ' ';
    }
};

int main() {
    int a[] = { 12, 3, 6, 98, 23 };
    for_each(a, a + 5, add5); // 使用函数指针
    for_each(a, a + 5, PrintV()); // 使用函数对象
}
```

函数对象相关 - boost.lambda

- 使用 boost.lambda 的方式（一个简单的示例）

```
#include <algorithm>
#include <iostream>
#include <boost/lambda/lambda.hpp>
using namespace std;

int main() {
    using namespace boost::lambda;

    int a[] = { 12, 3, 6, 98, 23 };
    for_each(a, a + 5, cout << _1 << ' ');
    cout << endl;
    for_each(a, a + 5, _1 += 5);
    for_each(a, a + 5, cout << _1 << ' ');
    cout << endl;

    // 还可以多个操作
    for_each(a, a + 5, (--_1, cout << _1 << ' '));
    cout << endl;
}
```

函数对象相关 - boost.lambda

- boost.lambda 相关的文件

boost/lambda/lambda.hpp	为不同的 C++ 操作符定义了 lambda 表达式
boost/lambda/bind.hpp	为最多 9 个参数定义了 bind 表达式
boost/lambda/if.hpp	定义了相当于 if 语句和条件操作符的 lambda 函数
boost/lambda/loops.hpp	定义了相当于循环结构的 lambda 函数
boost/lambda/switch.hpp	定义了相当于 switch 语句的 lambda 函数
boost/lambda/construct.hpp	提供了用于写带有构造函数，析构函数，new 和 delete 调用的 lambda 表达式的工具
boost/lambda/cast.hpp	提供了各种强制转型以及 sizeof 和 typeid 的 lambda 版本
boost/lambda/exceptions.hpp	给出了在 lambda 函数中抛出和捕获异常的工具
boost/lambda/algorithm.hpp boost/lambda/numeric.hpp	（与标准 algorithm 和 numeric 头文件相对）允许嵌入 STL 算法调用

函数对象相关 - boost.lambda

- 关于 lambda 表达式
  - lambda 表达式在函数式编程语言中很普通。在不同的语言之间（以及不同的 lambda 演算形式之间）语法有所区别，但是一个 lambda 表达式的基本形式如下：
    - `lambda x1 ... xn.e`
      - `x1~xn` 是参数，`e` 是表达式，如：
      - `lambda x y.x+y` 使用该表达式：
        `(lambda x y.x+y) 2 3 = 2 + 3 = 5`
        对应的 `boost.lambda` 形式：`_1 + _2`
      - `lambda x y.foo(x,y)` 对应的 `boost.lambda` 形式：`bind(foo,_1,_2)`，而不是 `foo(_1,_2)`

函数对象相关 - boost.lambda

- 占位符 (Placeholders)
  - BLL 定义了三个占位符类型：placeholder1_type，placeholder2_type 和 placeholder3_type。BLL 为每一个占位符类型提供了一个预定义的占位符变量：_1，_2 和 _3。
  - 占位符在 lambda 表达式中的使用决定了结果函数是无元的，一元的，二元的还是三元的。这由最高的占位符索引决定。例如：

```
_1 + 5                  // unary
_1 * _1 + _1            // unary
_1 + _2                 // binary
bind(f, _1, _2, _3) // 3-ary
_3 + 10                 // 3-ary
```

函数对象相关 - boost.lambda

- 占位符 (Placeholders)（续）
  - 为一个占位符提供真正的参数的时候，参数传递的方式总是传引用。这就意味着任何影响占位符的副作用都会反映到实际参数上。例如：

```
int i = 1;
(_1 += 2)(i);           // i is now 3
(++_1, cout << _1)(i) // i is now 4, outputs 4
```

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)
  - 基本规则是任何 c++ 操作符调用，只要它的参数中至少有一个是 lambda 表达式，则这个调用本身也是 lambda 表达式。几乎所有的能重载操作符都已被支持。例如，下面就是一个合法的 lambda 表达式：

```
cout << _1, _2[_3] = _1 && false
```

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 1 ）
  - 不能重载的操作符
    - 有些操作符根本不能重载（ `::`, `.`, `.*` ）。对于有些操作符，对返回类型的要求阻碍了它们为创建 `lambda` 函数而重载。这些操作符有 `->.`, `->`, `new`, `new[]`, `delete`, `delete[]` 和 `?:`（条件操作符）。

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 2 ）
  - 赋值和下标操作符
    - 这些操作符必须被实现为类成员。因此，左操作数必须是一个 lambda 表达式。例如：

```
int i;
_1 = i; // ok
i = _1; // not ok. i is not a lambda expression
```

    - 关于这一限制有一个简单的解决方案，简而言之，就是通过用一个特殊的 var 函数进行包装，左侧参数可以被显式转变为 lambda 仿函数：

```
var(i) = _1; // ok
```

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 3 ）
  - 逻辑操作符
    - 逻辑操作符服从短路求值规则。例如，在下面的代码中， i 没有被增加：

```
bool flag = true;
int i = 0;
(_1 || ++_2)(flag, i);
```

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 4 ）
  - 逗号操作符
    - 逗号操作符在 lambda 表达式中是“语句分隔符”。因为逗号也是函数调用中的参数分隔符，所以有时需要额外的括号：

```
for_each(a.begin(), a.end(), (++_1, cout << _1));
```

如果没有包围 ++_1, cout << _1 的额外括号，这行代码的意图会被解释为用四个参数调用 for_each。

    - 使用逗号操作符建立的 lambda 函数对象遵守 C++ 中左操作数的求值总是先于右操作数的规则。在上面的示例中， a 中的每一个元素首先被增 1，然后才写入流中。

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 5 ）
  - 函数调用操作符
    - 函数调用操作符有求 `lambda` 仿函数的值的作用。用过多的参数调用会导致一个编译时错误。

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 6 ）
  - 成员指针操作符
    - 成员指针操作符 operator->* 可以随意重载，因此，对于用户定义类型，成员指针操作符没有特定的情况。然而，它的内建的意义，却稍微有些复杂。内建成员指针操作符被用于以下情况：左参数是一个指向某个类 A 的对象的指针，而右手参数是一个指向 A 的一个成员的指针，或者是一个指向从 A 派生的类的一个成员的指针。我们必须区分以下两种情况：
      - 右手参数是一个数据成员的指针。在这种情况下， lambda 仿函数简单地执行参数替换并调用内建成员指针操作符，它返回一个引向它所指向的成员的引用。例如：

```
struct A { int d; };
A* a = new A();
// ...
(a->*&A::d); // returns a reference to a->d
(_1->*&A::d)(a); // likewise
```

函数对象相关 - boost.lambda

- 操作符表达式 (Operator expressions)（续 7 ）
  - 成员指针操作符（续）
    - 右侧参数是一个指向成员函数的指针。对于一个像这样的内建调用，结果有点儿像一个被延迟的成员函数调用。这样一个表达式必须在后面跟一个函数参数列表，以使得这个被延迟的成员函数调用可以被执行。例如：

```
struct B { int foo(int); };
B* b = new B();
//...
(b->*&B::foo) //returns a delayed call to b->foo
// a function argument list must follow
(b->*&B::foo)(1) // ok, calls b->foo(1)
(_1->*&B::foo)(b); // returns a delayed call to b->foo,
(_1->*&B::foo)(b)(1); // calls b->foo(1)
```

函数对象相关 - boost.lambda

- bind 表达式
  - bind 表达式可以有两种形式：
    - bind(target-function, bind-argument-list)
    - bind(target-member-function, object-argument, bind-argument-list)
  - 一个 bind 表达式延迟了一个函数的调用。如果这个 target function 是 n 元的，那么 bind-argument-list 也必须同样包含 n 个参数。在 BLL 的当前版本中，必须保证 0 <= n <= 9。对于成员函数来说，参数的数目最高为 8，因为对象参数要占有一个参数位置。总的来说，除了任何一个参数都能被一个占位符（更一般地说，是 lambda 表达式）取代之外，还要求 bind-argument-list 对于目标函数来说必须是一个合法的参数列表。注意，目标函数也可以是一个 lambda 表达式。根据在 bind-argument-list 中占位符的使用，一个 bind 表达式的结果可以是无元的，一元的，二元的或三元的函数对象

函数对象相关 - boost.lambda

- bind 表达式（续 1 ）
  - 以函数指针或引用为目标
    - 目标函数可以是指向一个函数的指针或引向一个函数的引用，而且它可以使已被绑定的或未被绑定的。例如：

```
X foo(A, B, C); A a; B b; C c;
bind(foo, _1, _2, c)(a, b);
bind(&foo, _1, _2, c)(a, b);
bind(_1, a, b, c)(foo);
```

    - 重载函数不能直接用于一个 bind 表达式：

```
void foo(int);
void foo(float);
int i;
bind(&foo, _1)(i); // error
void (*pf1)(int) = &foo;
bind(pf1, _1)(i); // ok
bind(static_cast<void(*)(int)> (&foo), _1)(i); // ok
```

函数对象相关 - boost.lambda

- bind 表达式（续 2 ）
  - 以成员函数为目标
    - 在 bind 表达式内使用指向成员函数的指针的语法为：

      ```
      bind(target-member-function, object-argument,
      bind-argument-list)
      ```

    - 对象参数可以是一个引向对象的引用或指向对象的指针， BLL 以同样的接口支持这两种情况：

```
bool A::foo(int) const;
A a;
vector<int> ints;
//  ...
find_if(ints.begin(), ints.end(), bind(&A::foo, a, _1));
find_if(ints.begin(), ints.end(), bind(&A::foo, &a, _1));
```

函数对象相关 - boost.lambda

- bind 表达式（续 3 ）
  - 以数据成员为目标
    - 一个指向成员变量的指针不是一个真正的函数，但是 bind 函数的第一个参数依然可以是一个指向成员变量的指针。调用这样一个 bind 表达式会返回一个引向这个数据成员的引用。例如：

```
struct A { int data; };
A a;
bind(&A::data, _1)(a) = 1;  // a.data == 1
```

    - 成员被访问的那个对象的 cv 修饰符（c 为 const，v 为 volatile —— 译者注）也需要被考虑。例如，下面的例子就是试图写到一个 const 区域中：

```
const A ca = a;
bind(&A::data, _1)(ca) = 1;      // error
```

函数对象相关 - boost.lambda

- bind 表达式（续 4 ）
  - 以函数对象为目标
    - BLL 支持标准库的在一个函数对象类中用一个名为 result_type 的成员 typedef 声明一个函数对象的返回类型的惯例。这是一个简单的示例：

```
struct A {
    typedef B result_type;
    B operator()(X, Y, Z);
};
```

    - 另一种可以让 BLL 感知一个函数对象的返回类型的机制是定义一个成员模板结构 sig<Args>，其中包含一个指定返回类型的 typedef type。这是一个简单的示例：

```
struct A {
    template <class Args> struct sig {
        typedef B type;
    }
    B operator()(X, Y, Z);
};
```

函数对象相关 - boost.lambda

- 覆盖推导出的返回类型
  - 为了明确的确定返回值，可以使用 ret<T>

```
A a;
B b;
C operator+(A, B);
D operator-(C);
//   ...
ret<D> (-(_1 + _2))(a, b); // error
ret<D> (-ret<C> (_1 + _2))(a, b); // ok
```

函数对象相关 - boost.lambda

- 延迟常量和变量
  - 一元函数 constant， constant_ref 和 var 将它们的参数变成一个实现恒等映射的 lambda 仿函数，前两个用于常量，后面的用于变量。为了明确 lambda 表达式的语法，这些延迟常量和变量的使用有时是有必要的。

函数对象相关 - boost.lambda

- 延迟常量和变量（续1）

```
int a[] = { 12, 6, 92 };
for_each(a, a + 3, cout << _1 << ' '); // 12 6 92
cout << endl;
// 下面的 ' ' 立即求值
for_each(a, a + 3, cout << ' ' << _1); // 12692
cout << endl;

for_each(a, a + 3, cout << constant(' ') << _1); // 12 6 92
cout << endl;

int index = 0;
// 输出: 1:12
//       6
//       92
for_each(a, a + 3, cout << ++index << ':' << _1 << '\n');
// 下面输出正常:
for_each(a, a + 3, cout << ++var(index) << ':' << _1 << '\n');
```

函数对象相关 - boost.lambda

- 延迟常量和变量（续2）
  - 延迟常量和变量都可以是命名对象：

```
int i = 0; int j;
for_each(a, a + 3, (var(j) = _1, _1 = var(i), var(i) =
var(j)));

// 等价于:
int i = 0; int j;
var_type<int>::type vi(var(i)), vj(var(j));
for_each(a, a + 3, (vj = _1, _1 = vi, vi = vj));

// 下面是命名延迟常量的示例:
constant_type<char>::type space(constant(' '));
for_each(a, a + 3, cout << space << _1);
```

函数对象相关 - boost.lambda

- 控制结构的 lambda 表达式
  - BLL 支持以下用于控制结构的函数模板：

```
if_then(condition, then_part)
if_then_else(condition, then_part, else_part)
if_then_else_return(condition, then_part, else_part)
while_loop(condition, body)
while_loop(condition) // no body case
do_while_loop(condition, body)
do_while_loop(condition) // no body case
for_loop(init, condition, increment, body)
for_loop(init, condition, increment) // no body case
switch_statement(...)
```

函数对象相关 - boost.lambda

- 控制结构的 lambda 表达式（续 1 ）
  - 示例：

```
using namespace boost::lambda;
// 下面代码需要包含<boost/lambda/if.hpp>
int a[] = { 98, 21, 32 };
for_each(a, a + 3, if_then(_1 % 2 == 0, cout << _1 << ' '));
for_each(a, a + 3, if_(_1 % 2 == 0)
                       [cout << _1 << ' ']
                   .else_
                       [cout << ++_1 << ' ']); // same above

// 下面代码需要包含<boost/lambda/loops.hpp>
int am[5][10];
int i;
for_each(am, am + 5, for_loop(var(i) = 0, var(i) < 10, ++var(i),
        _1[var(i)] += 1));
for_each(am, am + 5, for_(var(i) = 0, var(i) < 10, ++var(i))
        [_1[var(i)] += 1]); // same above
```

函数对象相关 - boost.lambda

- 控制结构的 lambda 表达式（续 2 ）
  - switch 语句
    - switch 控制结构的 lambda 表达式更加复杂，因为 cases 的数量是可以变化的。一个 switch lambda 表达式的常规形式是：

```
switch_statement(condition,
    case_statement<label>(lambda expression),
    case_statement<label>(lambda expression),
    // ...
    default_statement(lambda expression)
);
```

函数对象相关 - boost.lambda

- 控制结构的 lambda 表达式（续 3 ）
  - switch 语句示例

```
// 下面代码需要包含<boost/lambda/switch.hpp>
int a[] = { 12, 31, 92 };

std::for_each(a, a + 3,
        (switch_statement(_1,
        case_statement<0> (std::cout << constant( "zero")),
        case_statement<1> (std::cout << constant("one")),
        default_statement(cout << constant("other: ") << _1)),
        cout << constant("\n")));
```

函数对象相关 - boost.lambda

- 关于 boost.lambda 的其它话题
  - 诸如异常、构造 & 析构等方面请参考 boost.lambda 的文档

函数对象相关 - More stuffs...

- boost 函数对象相关的组件很丰富，除了本单元列出的几个组件外，还有：
  - boost.signal2： 可管理的回调机制的实现
  - boost.functional 中一些增强的函数对象适配器
  - ...