**boost::bind 详解**

**使用**

boost::bind是标准库函数std::bind1st和std::bind2nd的一种泛化形式。其可以支持函数对象、函数、函数指针、成员函数指针，并且绑定任意参数到某个指定值上或者将输入参数传入任意位置。

**1. 通过functions和function pointers使用bind**

给定如下函数：

[复制代码](javascript:void(0);)

1 int f(int a, int b)

2 {

3 return a + b;

4 }

5

6 int g(int a, int b, int c)

7 {

8 return a + b + c;

9 }

[复制代码](javascript:void(0);)

**可以绑定所有参数**，如：

bind(f, 1, 2)等价于f(1, 2); bind(g, 1, 2, 3)等价于g(1, 2, 3);

**也可以选择性地绑定参数**，如：

bind(f, \_1, 5)(x)等价于f(x, 5)，其中\_1是一个占位符，表示用第一个参数来替换;

bind(f, \_2, \_1)(x, y)等价于f(y, x);

bind(g, \_1, 9, \_1)(x)等价于g(x, 9, x);

bind(g, \_3, \_3, \_3)(x, y, z)等价于g(z, z, z);

**说明：**

传入bind函数的参数一般为变量的copy，如：

int i = 5;

bind(f, i, \_1);

如果想传入变量的引用，可以使用boost::ref和boost::cref，如：

int i = 5;

bind(f, ref(i), \_1);

bind(f, cref(i), \_1);

**2. 通过function objects使用bind**

[复制代码](javascript:void(0);)

1 struct F

2 {

3 int operator()(int a, int b) { return a – b; }

4 bool operator()(long a, long b) { return a == b; }

5 };

6

7 F f;

8 int x = 100;

9 bind<int>(f, \_1, \_1)(x); // f(x, x)

[复制代码](javascript:void(0);)

可能某些编译器不支持上述的bind语法，可以用下列方式代替：

boost::bind(**boost::type<int>()**, f, \_1, \_1)(x);

默认情况下，bind拥有的是函数对象的副本，但是也可以使用boost::ref和boost::cref来传入函数对象的引用，尤其是当该function object是non-copyable或者expensive to copy。

**3. 通过pointers to members使用bind**

bind将传入的成员（数据成员和成员函数）指针作为第一个参数，其行为如同使用boost::mem\_fn将成员指针转换为一个函数对象，即：

bind(&X::f, args);       等价于bind<R>(mem\_fn(&X::f), args)，其中R为X::f的返回类型（成员函数）或类型（数据成员）。

[复制代码](javascript:void(0);)

1 struct X

2 {

3 bool f(int a);

4 };

5

6 X x;

7 shared\_ptr<X> p(new X);

8 int i = 5;

9

10 bind(&X::f, ref(x), \_1)(i); // x.f(i)

11 bind(&X::f, &x, \_1)(i); // (&x)->f(i)

12 bind(&X::f, x, \_1)(i); // x.f(i)

13 bind(&X::f, p, \_1)(i); // p->f(i)

[复制代码](javascript:void(0);)

**4. 使用nested binds**

如bind(f, bind(g, \_1))(x)中：

在外部bind计算之前，内部bind先被计算（如果内部有多个bind，则计算顺序不定）。如上，根据参数x，先计算bind(g, \_1)(x)，生成g(x)，然后计算bind(f, g(x))(x)，最后生成f(g(x))。

**但是要注意：**

bind中的第一个参数不参与计算过程，假设如下程序想要实现对于向量v中的每个函数指针，传入一个参数 5：

typedef void (\*pf)(int);

std::vector<pf> v;

std::for\_each(v.begin(), v.end(), bind(\_1, 5));

上述程序并没有实现我们想要的结果：可以通过使用一个帮助函数对象apply，该对象可以将bind的第一个参数作为一个函数对象，如下：

typedef void (\*pf)(int);

std::vector<pf> v;

std::for\_each(v.begin(), v.end(), bind(apply<void>(), \_1, 5));

其中，apply实现如下：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 template<class R>

2 struct apply

3 {

4 typedef R result\_type;

5 template<class F> result\_type operator()(F & f) const

6 {

7 return f();

8 }

9 ...

10 };

[复制代码](javascript:void(0);)

**示例程序**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 // bind\_test.cc

2 #include <boost/config.hpp>

3

4 #if defined(BOOST\_MSVC)

5 #pragma warning(disable: 4786) // identifier truncated in debug info

6 #pragma warning(disable: 4710) // function not inlined

7 #pragma warning(disable: 4711) // function selected for automatic inline expansion

8 #pragma warning(disable: 4514) // unreferenced inline removed

9 #endif

10

11 #include <boost/bind.hpp>

12 #include <boost/ref.hpp>

13

14 #if defined(BOOST\_MSVC) && (BOOST\_MSVC < 1300)

15 #pragma warning(push, 3)

16 #endif

17

18 #include <iostream>

19

20 #if defined(BOOST\_MSVC) && (BOOST\_MSVC < 1300)

21 #pragma warning(pop)

22 #endif

23

24 #include <boost/detail/lightweight\_test.hpp>

25

26 //

27 long f\_0() { return 17041L; }

28 long f\_1(long a) { return a; }

29 long f\_2(long a, long b) { return a + 10 \* b; }

30

31

32 long global\_result;

33

34 void fv\_0() { global\_result = 17041L; }

35 void fv\_1(long a) { global\_result = a; }

36 void fv\_2(long a, long b) { global\_result = a + 10 \* b; }

37

38 void function\_test()

39 {

40 using namespace boost;

41

42 int const i = 1;

43

44 BOOST\_TEST( bind(f\_0)(i) == 17041L );

45 BOOST\_TEST( bind(f\_1, \_1)(i) == 1L );

46 BOOST\_TEST( bind(f\_2, \_1, 2)(i) == 21L );

47

48 BOOST\_TEST( (bind(fv\_0)(i), (global\_result == 17041L)) );

49 BOOST\_TEST( (bind(fv\_1, \_1)(i), (global\_result == 1L)) );

50 BOOST\_TEST( (bind(fv\_2, \_1, 2)(i), (global\_result == 21L)) );

51 }

52

53

54 //

55 struct Y

56 {

57 short operator()(short & r) const { return ++r; }

58 int operator()(int a, int b) const { return a + 10 \* b; }

59 };

60

61 void function\_object\_test()

62 {

63 using namespace boost;

64 short i(6);

65

66 BOOST\_TEST( bind<short>(Y(), ref(i))() == 7 );

67 BOOST\_TEST( bind(type<short>(), Y(), ref(i))() == 8 );

68 }

69

70

71 //

72 struct X

73 {

74 mutable unsigned int hash;

75

76 X(): hash(0) {}

77

78 int f0() { f1(17); return 0; }

79 int g0() const { g1(17); return 0; }

80

81 int f1(int a1) { hash = (hash \* 17041 + a1) % 32768; return 0; }

82 int g1(int a1) const { hash = (hash \* 17041 + a1 \* 2) % 32768; return 0; }

83

84 int f2(int a1, int a2) { f1(a1); f1(a2); return 0; }

85 int g2(int a1, int a2) const { g1(a1); g1(a2); return 0; }

86 };

87

88 void member\_function\_test()

89 {

90 using namespace boost;

91

92 X x;

93

94 // 0

95 bind(&X::f0, &x)();

96 bind(&X::f0, ref(x))();

97 bind(&X::g0, &x)();

98 bind(&X::g0, x)();

99 bind(&X::g0, ref(x))();

100

101 // 1

102 bind(&X::f1, &x, 1)();

103 bind(&X::f1, ref(x), 1)();

104 bind(&X::g1, &x, 1)();

105 bind(&X::g1, x, 1)();

106 bind(&X::g1, ref(x), 1)();

107

108 // 2

109

110 bind(&X::f2, &x, 1, 2)();

111 bind(&X::f2, ref(x), 1, 2)();

112 bind(&X::g2, &x, 1, 2)();

113 bind(&X::g2, x, 1, 2)();

114 bind(&X::g2, ref(x), 1, 2)();

115 }

116

117 void nested\_bind\_test()

118 {

119 using namespace boost;

120

121 int const x = 1;

122 int const y = 2;

123

124 BOOST\_TEST( bind(f\_1, bind(f\_1, \_1))(x) == 1L );

125 BOOST\_TEST( bind(f\_1, bind(f\_2, \_1, \_2))(x, y) == 21L );

126 }

127

128 int main()

129 {

130 function\_test();

131 function\_object\_test();

132 member\_function\_test();

133 nested\_bind\_test();

134

135 return boost::report\_errors();

136 }

137

138 //output

139 No errors detected.

[复制代码](javascript:void(0);)

**易错点**

**1. 参数个数不正确**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 int f(int, int);

2

3 int main()

4 {

5 boost::bind(f, 1); // error, f takes two arguments

6 boost::bind(f, 1, 2); // OK

7 }

8 一个此类错误的变形为：忘记成员函数有一个隐式参数this:

9 struct X

10 {

11 int f(int);

12 }

13

14 int main()

15 {

16 boost::bind(&X::f, 1); // error, X::f takes two arguments

17 boost::bind(&X::f, \_1, 1); // OK

18 }

[复制代码](javascript:void(0);)

**2. 函数对象不能被指定参数调用**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 int f(int);

2

3 int main()

4 {

5 boost::bind(f, "incompatible"); // OK so far, no call

6 boost::bind(f, "incompatible")(); // error, "incompatible" is not an int

7 boost::bind(f, \_1); // OK

8 boost::bind(f, \_1)("incompatible"); // error, "incompatible" is not an int

9 }

[复制代码](javascript:void(0);)

**3. 访问不存在的参数**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 占位符\_N需要在调用时从指定的参数表中选择第N个参数：

2 int f(int);

3

4 int main()

5 {

6 boost::bind(f, \_1); // OK

7 boost::bind(f, \_1)(); // error, there is no argument number 1

8 }

[复制代码](javascript:void(0);)

**4. bind(f, ...)形式和bind<R>(f, ...)形式的不当用法**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 bind(f, a1, a2, ..., aN)会对f自动进行类型识别，f必须是一个函数或者成员函数指针。当f是函数对象时，大多数编译器将不能工作。

2 bind<R>(f, a1, a2, ..., aN)支持任意类型的函数对象。虽然在有些编译器上，传入函数或者成员函数指针也能工作，但是不推荐这么做。

**5. 绑定一个非标准函数**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 bind(f, a1, a2, ..., aN)形式识别<普通的>C++函数和成员函数指针。如果一个函数使用了不同的调用约定或者可变参数列表（如std::printf），那么bind(f, a1, a2, ..., aN)将不能工作；如果确实需要使用此类非标准函数，那么bind<R>(f, a1, a2, ..., aN)将能满足这种要求。

**6. 绑定一个重载函数**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 通常情况下，bind一个重载函数会导致错误，因为无法确定到底bind重载函数的哪个形式：

2 struct X

3 {

4 int& get();

5 int const& get() const;

6 };

7

8 int main()

9 {

10 boost::bind(&X::get, \_1);

11 }

12 这种二义性可以通过将成员函数指针转换到特定类型来解决：

13 int main()

14 {

15 boost::bind(static\_cast< int const& (X::\*) () const >(&X::get), \_1);

16 }

17 此外，一个更可具可读性的解决办法为引入一个临时变量：

18

19 int main()

20 {

21 int const& (X::\*get) () const = &X::get;

22 boost::bind(get, \_1);

23 }

[复制代码](javascript:void(0);)

**7. boost的特定编译器实现问题**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 // 7.1 MSVC 6.0编译器

2 在函数签名中不支持const：（移除const就可以了）

3 int f(int const);

4

5 int main()

6 {

7 boost::bind(f, 1); // error

8 }

9 // 7.2 MSVC 7.0以下编译器

10 (1) 如果通过using声明引入boost::bind，如：using boost::bind，那么bind<R>(f, ...)语法将不能工作。

11 解决办法为直接使用限定名boost::bind或者使用using指令：using namespace boost;

12 (2) 一个嵌套的命名为bind的类模板将隐藏函数模板boost::bind，使得bind<R>(f, ...)语法不能工作。

13 (3) MSVC将可变参数中的省略号看作一种类型，因此，其可以接受如下形式：

14 bind(printf, "%s\n", \_1);

15 但是拒绝正确的形式如：

16 bind<int>(printf, "%s\n", \_1);

[复制代码](javascript:void(0);)

**调用约定**

根据调用约定的不同，不同的平台可能支持几种类型的（成员）函数。例如：

Windows API函数和COM接口成员函数使用\_\_stdcall;

Borland VCL使用\_\_fastcall;

Mac toolbox函数使用pascal。

与\_\_stdcall函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_BIND\_ENABLE\_STDCALL;

与\_\_stdcall成员函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_MEM\_FN\_ENABLE\_STDCALL;

与\_\_fastcall函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_BIND\_ENABLE\_ FASTCALL;

与\_\_fastcall成员函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_MEM\_FN\_ENABLE\_ FASTCALL;

与pascal函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_BIND\_ENABLE\_ PASCAL;

与\_\_cdecl成员函数一起使用bind时，在包含<boost/bind.hpp>之前#define the macro BOOST\_MEM\_FN\_ENABLE\_CDECL;

**一个比较好的建议是：**如果需要使用bind，要么提前在工程选项中定义这些宏，要么通过命令行选项-D定义，要么直接在使用bind的.cc文件中定义。否则如果包含bind.hpp的文件中，发生了在定义这些宏之前including bind.hpp，那么可能导致难以发现的错误。