Introduction to Boost.Function

Yuhao Zhou
miskcoo@gmail.com

Department of Computer Science and Technology

April 11, 2017

What’s Boost?

Here is a good material.

Function Object

A *function object* is any object for which the function call operator is defined.

- ```
  auto is_negative_lambda = [](int x) { return x < 0; }
  ```
- ```
  class LessThanFunc
  {
      int x_;
  public:
      LessThanFunc(int x) : x_(x) {}
      bool operator()(int a) const { return a < x_; }
  };

  LessThanFunc is_negative_object(0);
  ```

They have similar behavior with typical function.

```
is_negative_object(1);
is_negative_lambda(1);
```

Function Object

Suppose we have a function counting the number of elements with some properties (e.g. std::count).

We need to pass a function (or function objet) to std::count.

```
int A[] = { -1, 0, 1, 2 };
std::count(A, A + 4, is_negative_lambda);
std::count(A, A + 4, is_negative_object);
```

Here, is_negative can be lambda function, function object, or function.

Function Object

Suppose we have a function counting the number of elements with some properties (e.g. std::count).

We need to pass a function (or function objet) to std::count.

```
int A[] = { -1, 0, 1, 2 };
std::count(A, A + 4, is_negative_lambda);
std::count(A, A + 4, is_negative_object);
```

Here, is_negative can be lambda function, function object, or function.

How to represent the type of the callable object?

Template

You may want to use the template!

```
template<typename Iter, typename Property>
int count(Iter beg, Iter end, Property property)
{
    int num = 0;
    for(Iter p = beg; p != end; ++p)
        if(property(*p)) ++num;
    return num;
}
```

Template

You may want to use the template!

```
template<typename Iter, typename Property>
int count(Iter beg, Iter end, Property property)
{
    int num = 0;
    for(Iter p = beg; p != end; ++p)
        if(property(*p)) ++num;
    return num;
}
```

However, what if we need a series of callable objects and store them into an array?

How to represent the type of the array? They have similar calling grammar, but the type of them can be different.

boost::function

boost::function is a function wrapper in which we can store any callable object with specific calling grammar.

```
boost::function<bool(int)> is_negative;

// store a lambda function
is_negative = [](int x) {
    return x < 0;
};

// store a function object
is_negative = LessThanFunc(0);
```

Implementation

First, consider how to store a single callable object.

Implementation

First, consider how to store a single callable object.

Calling grammar is the same $\rightarrow$ interface!

Callable objects are different $\rightarrow$ Implementation!

Implementation

First, consider how to store a single callable object.

Calling grammar is the same → interface!

Callable objects are different → Implementation!

We can use the polymorphism!

```
// abstract interface
template<typename Ret, typename Param>
class function_impl
{
public:
    virtual Ret invoke(Param p) = 0;
    virtual ~function_impl() = default;
};

// particular implementation
template<typename Func, typename Ret, typename Param>
class function_impl_derive : public function_impl<Ret, Param>
{
    Func f_;
public:
    function_impl_derive(Func f) : f_(f) {}
    Ret invoke(Param p) { return f_(p); }
};
```

Implementation

Next, we are going to write a wrapper.

```
template<typename Ret, typename Param>
class function
{
    function_impl<Ret, Param> *f_;
public:
    // function template can help us deduce the type of callable object
    template<typename Func>
    function(Func f) : f_(new function_impl_derive<Func, Ret, Param>(f)) {}

    ~function() { if(f_) delete f_; }

    template<typename Func>
    function& operator = (Func f)
    {
        if(f_) delete f_;
        f_ = new function_impl_derive<Func, Ret, Param>(f);
        return *this;
    }

public:
    Ret operator() (Param p) { return f_->invoke(p); }
};
```

Implementation

How to support the class member function like this?

```
struct AA
{
    void f() { std::cout << "F" << std::endl; }
};

int main()
{
    AA obj;
    function<void, AA> f1 = &AA::f;
    function<void, AA*> f2 = &AA::f;
    f1(obj);
    f2(&obj);
    return 0;
}
```

Implementation

We can use template specialization.

```
template<typename Ret, typename ClassParam>
class function_impl_derive<Ret(ClassParam::*)(), Ret, ClassParam>
    : public function_impl<Ret, ClassParam>
{
    typedef Ret(ClassParam::*function_type)();
    function_type f_;
public:
    function_impl_derive(function_type f) : f_(f) {}
    Ret invoke(ClassParam p) { return (p.*f_)(); }
};

template<typename Ret, typename ClassParam>
class function_impl_derive<Ret(ClassParam::*)(), Ret, ClassParam*>
    : public function_impl<Ret, ClassParam*>
{
    typedef Ret(ClassParam::*function_type)();
    function_type f_;
public:
    function_impl_derive(function_type f) : f_(f) {}
    Ret invoke(ClassParam* p) { return (p->*f_)(); }
};
```

Multi-parameter Function

How to support multi-parameter function?

Multi-parameter Function

How to support multi-parameter function?

Copy the code?

Multi-parameter Function

How to support multi-parameter function?

Copy the code?

This is not a good idea...

Multi-parameter Function

How to support multi-parameter function?

Copy the code?

This is not a good idea...

Macro, or variable template in C++11, would be better.

Q&A

Thanks for listening!