23 Functors and Imposter Syndrome

23.1 `template <typename Iter_type>`

上一讲讲了 Iterator: Iterator 是一种 generalized pointer.

但是回想起来，我们写的 Iterator 是内嵌于 `List<T>` 的 class。因而：

1. 我们每次写 Iterator 作为 参数并且的函数都要

   ```
   template <typename T>
   int func(typename List<T>::Iterator, ...)
   ```

2. 不仅如此，我们的 Iterator 只能用于 List。但是很显然，其他任何 Container 的 Iterator 的运行逻辑也是类似的，所以会写一些确定 typename 的代码，这些是不需要的。

因而我们可以考虑写一个 Iterator 的 template function:

```
template <typename Iter_type>
Iter_type max_element(Iter_type begin, Iter_type end) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (*it > *maxIt) {
      maxIt = it;
    }
  }
  return maxIt;
}
```

这里，我们的 template class 并不是某个 container，而是某种 Iterator.

Compiler 会自动 infer Iterator 是哪个 Iterator.

比如:

```
int main() {
  List<int> list;
  cout << *max_element(list.begin(), list.end()) << endl;
  // 这里 compiler infers Iter_type 为: List<int>::Iterator

  List<Card> cards;
  cout << *max_element(cards.begin(), cards.end()) << endl;
  // 这里 compiler infers Iter_type 为: List<Card>::Iterator

  vector<int> vec;
  cout << *max_element(vec.begin(), vec.end()) << endl;
  // 这里 compiler infers Iter_type 为: vector<int>::Iterator
}
```

再比如: 真正的 pointer 也可以作为之类的 `Iter_type`.

```
int main() {
  int const SIZE = 10;
  double arr[SIZE]; //array
  cout << *max_element(arr, arr + SIZE) << endl;
  // 这里 compiler infers Iter_type 为: double*
  //
}
```

23.1.1 Exercise

```
// REQUIRES: begin is before end (or begin == end)
// EFFECTS: Returns true if any element in the
// sequence [begin, end) is odd.
template <typename Iter_type>
bool any_of_odd(Iter_type begin, Iter_type end) {
  for ( ; begin != end; ++begin) {
    if (*begin & 2 != 0) {return true;}
  }
  return false;
}
```

23.2 function pointer

我们目前为止使用的 pointer 都是指向 data 的 pointer，但是 c++ 还支持指向 function 的 pointer。

格式为:

```
int (*fn)(int, int) = max;
```

这里的意思是：`fn` 是一个指向一个 参数为 `(int, int)` 并且返回值为 `int` 的 function 的 pointer；并且我们给它指向的对象 assign 了 `max(int ..., int ...)` 这个函数。

23.2.1 functor 的 declaration 和 assignment syntax

梳理一下：

1. `int (*fn)` 表示：我们这里 declare 一个 function pointer 叫做 `fn`，这个 funtor 是一个指向返回值为 `int` 的 function 的 pointer.
2. `(int, int)` 表示的是：`fn` 指向的 function 的参数的类型必须是 `int` 和 `int`.
3. `=max` 表示我们把 `max` 这个函数作为 `fn` 当前指向的函数。

另，注意 1：c++ declaration 是 inside out 的。**`[]`, `()` 这种 postfix operators 比起 prefix operators 如 `&`, `*` 等有更高的优先级。**

注意 2: 我们给普通的 ptr 指向的对象赋值需要先解引用，或者把需要指向的 data object 的 address 赋给 ptr：

```
int *p;
*p = 1; // dereference 之后 assign value
int a = 2;
p = &a; // 把 a 的 address 赋给 p
```

但是我们给 function pointer 赋值是**没有先 dereference 再赋值的这种 syntax 的，只有把需要指向的 function object 的 address 赋给 ptr 这种方法！**；

并且，我们甚至也不用这么写。**我们可以直接写 function pointer = 某个 function，compiler 会自动推测意思，补全为把该 function 的 address 赋给这个 function pointer！**

```
int (*fn)(int, int) = nullptr;
*fn = std::max; // error, 不支持这种写法

int (*fn2)(int, int) = nullptr;
fn2 = &std::max;  // ok, 把 std::max 的 address 赋给 p

int (*fn3)(int, int) = nullptr;
fn3 = std::max; // ok, 这里 compiler 自动理解意思, 把 std::max 的 address 赋给 p
```

23.2.2 使用 function pointers

Function pointer 不仅 declaration, assignment 和普通的 pointer 有区别，具体的使用也和普通的 pointer 有区别：我们使用 function pointer 不需要 dereference.

```
int (*fn2)(int, int) = max;
cout << fn2(9, 8) << endl;
```

`fn2(9, 8)` 的值就是 `max(9, 8)` 的值。

（其实我感觉 function pointer 更像是对 function 的引用而不像是 pointer，或者说是 reference 和 pointer 的结合：pointer 的 declaration 方式和 reference 的使用方式）

23.2.3 Advanced application: 把 function pointer 作为函数参数从而减少 code duplication

Def: 一个 predicate 是一个返回 bool 值的函数.

假设我们需要一个这样的函数：

我们有一个 predicate，以及一个随意的 `int` container。如果这个 container 中有任何满足这个 predicate 的 data，那么就返回 true.

我们发现在这种应用情境下，function pointer 是很有用的。我们可以试想：我们对这个 `any_of` 判断函数传入一个（输入 `int` 输出 `bool` 的 function 的 function pointer），并且 iterate 这个 container；如果有元素，在输入这个 function pointer 指向的 function 之后输出结果为 true，那么我们就返回 true.

如果我们不是用 function pointer 的话，那么我们每次换一个 predicate 就要写一个新的 `any_of` 函数。

```
template <typename Iter_type>
bool any_of(Iter_type begin, Iter_type end, bool (*fn)(int)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) {return true;}
  }
  return false;;
}
```

23.2.4 使用多个 template type

我们甚至可以使用多个 template type，这样可以使用不止对 int container 判断一个 predicate，还可以对任何 container ADT 判断这个 predicate.

这里我们 template：`Iter_type` 为任何 Iterator type，`T` 为任何 data type，这样我们就可以写入一个指向（输入为 `T&` (&用来防止复制太复杂的数据)，输入为 `bool` 的 function）的 funtion pointer `fn`，将其指向的 function 作为 `any_of` 的 predicate.

```
template <typename Iter_type, typename T>
bool any_of(Iter_type begin, Iter_type end, bool (*fn)(const T&)) {
  for (Iter_type it = begin; it != end; ++it) {
    if (fn(*it)) {return true;}
  }
  return false;;
}
```

我们这里应用我们刚才写的 `any_of` function:

```
bool is_odd(int x) {return x % 2 != 0;}
bool greater0(double x) {return x > 0;}

int main() {
  List<int> list;
  List<double list2;
  //...fill
  cout << any_of(list.begin(), list.end(), is_odd);
  cout << any_of(list2.begin(), list.end(), greater0);
}
```

23.3 Functors: 模拟 function 的行为

我们之前学了 Iterator 的理念：iterator 就是用一个 class 模拟 pointer 的行为。

现在我们引入一个新的理念 functor：用一个 class 来模拟 function 的行为。（命名借鉴于 category theory 中的 functor，但是我感觉在定义上没什么联系。）

我们为什么要引入这个东西呢：还是为了缩短代码，减少 code duplication.

比如我们看到之前的 `greater0` 函数，我们会想：如果我们先要写很多 `greatern` 函数呢?

```
bool greater0(double x) {return x > 0;}
bool greater1(double x) {return x > 1;}
bool greater2(double x) {return x > 2;}
//...
bool greater255(double x) {return x > 255;}
```

这是不符合我们写代码尽量减少 code duplication 的原则的。

于是 C++ 引入 functor 的理念：通过写一个 class，用这个 class 的 date member 来模拟相似函数的不同条件；constructor 来模拟函数的创建；通过 **overload the function call operator** `()` 来模拟函数的输入。

```
class GreaterN {
private:
  int threshold;

public:
  GreaterN(int threshold_in)
    : threshold(threshold_in) {}

  bool operator()(int n) const {  // overload () 这个 operator, 参数为 int n.
    return n > threshold;
  }
}
```

现在我们使用这个 functor: functor 就是我们刚刚创建用来模拟 function 的 class 的 instances.

```
int main() {
  GreaterN g0(0);
  GreaterN g32(32);
  GreaterN g212(212);

  cout << g0(-3);  // false
  cout << g0(3);  // true
  cout << g32(9);  // false
  cout << g212(189);  // false
}
```

23.3.1 使用 template class 作为 Predicate 参数以适用任何 Functor

我们刚才用的是 `GreaterN` 这个 functor 作为 predicate. 这个 functor class 的 `operator()` 的 overload 方式是: `return n > threshold`. 因而这个 functor class 只有比较某个 Container 中的元素和另一个元素的大小这一个作用。

那如果我们想要这个 `any_of` functor 可以嵌入任何 predicate 呢?

我们这个时候可以选择把 `GreaterN` 这个 class 名参数改成一个 template class 的参数，这样就可以让这个函数可以起到真正的 "any of" 的作用：如果 Container object 中有任何一个元素满足我们现在指定的 predicate，那么就返回 true.

```
template <typename Iter_type, typename Predicate>
bool any_of(Iter_type begin, Iter_type end, Predicate pred) {
  for (Iter_type it = bein; it != end; ++it) {
    if (pred(*it)) {return true;}
  }
  return false;
}
```

23.3.2 example 2

再举一个例子：

```
class Duck {
public:
  Duck()
    : name(""), duckling_count(0) { }
  Duck(string name_in, int duckling_count_in)
    : name(name_in), duckling_count(duckling_count_in) { }
  const string & getName() const { return name; }
  const int getDucklingCount() const { return duckling_count; }
private:
  string name;
  int duckling_count;
};
```

`Duck` 这个 class 有两个 private members: 一个代表名字，一个代表生产数量。

我们现在写一个 template function，来获取任意一个装 Ducks 的 Container 中某个属性最大的鸭子是哪个。

首先我们写两个 functor，一个代表对鸭子名字的字母顺序的比较的 predicate，另一个代表对鸭子生产数量的比较的 predicate.

```
// functor 1
class DuckNameLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.getName() < d2.getName();
  }
};

// functor 2
class DuckDucklingsLess {
public:
  bool operator()(const Duck &d1, const Duck &d2) const {
    return d1.getDucklingCount() < d2.getDucklingCount();
  }
};
```

然后我们 implement 这个函数：

```
template <typename Iter_type, typename Comparator>
Iter_type max_element(Iter_type begin, Iter_type end, Comparator less) {
  Iter_type maxIt = begin;
  for (Iter_type it = begin; it != end; ++it) {
    if (less(*maxIt,*it)) {maxIt = it;} // 检验 predicate (compare)
  }
  return maxIt;
}
```

我们现在可以使用这个函数来比较随机 Container 中鸭子属性的情况：

```
int main() {
  List<Duck> pond; // fill…
  cout << *max_element(pond.begin(), pond.end(), DuckNameLess()) << endl;
  cout << *max_element(pond.begin(), pond.end(), DuckDucklingsLess()) << endl;
}
```

23.3.3 example 3

我们之前写的 `any_of` 函数表示：只要 Container 中有任意一个 element 满足 predicate，那么就返回 true.

那么如果我们要在 traversal 每一个元素后做一点事情呢?

```
template <typename Iter_t, typename Func_t>
Func_t for_each(Iter_t begin, Iter_t end, Func_t func) {
  for (Iter_t it = begin; it != end; ++it) {
    func(*it);
  }
  return func;  //We return the functor in case it contains some information about the
result.
}
```

使用：

```
template <typename T>
class Printer { //一个 functor
public:
  void operator()(const T &n) const {
    cout << n;
  }
};

int main() {
  List<int> list; // Fill with numbers
  for_each(list.begin(), list.end(), Printer<int>());
}
```