4-Functional Programming

1. Function Objects

假设我们想要判断一个序列中是否有奇数，借助模板和迭代器，我们可以写一个针对容器通用的版本：

```cpp
// REQUIRES: begin is before or equal to 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 (Iter_type it = begin; it != end; ++it) {
    if (*it % 2 != 0) {
      return true;
    }
  }
  return false;
}
```

然后我们可以将 `any_of_odd()` 用于任何序列类型，只要元素是整数即可：

```cpp
List<int> list;
vector<int> vec;
int array[10];
... cout << any_of_odd(list.begin(), list.end()) << endl;
cout << any_of_odd(vec.begin(), vec.end()) << endl;
cout << any_of_odd(array, array + 10)) << endl;
```

这个函数模板只对迭代器是通用的，对判断条件并不通用，它只能搜索是否有奇数，不能判断容器中是否有其它类型的元素。比如我们想确定一个容器中是否包含偶数，我们可以这样写 `any_of_even()` 模板：

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

这份代码和 `any_of_odd()` 除了判断条件不同，其它地方完全一样。进一步地，假设我们想要判断一个容器是否有正数，我们也只是修改 `if` 中的条件，其它地方不做修改。这又让我们想到了一个超级强大的工具——抽象，这些函数模板，除了判断条件不一样，其它部分都一样，所以我们可以考虑对判断条件进行抽象，在设计代码时，不指定具体的判断条件，而是采用一个像变量一样可以取值为任意判断条件的东西。因为这些判断条件可以看作是一个个函数，而函数在加载到内存后我们可以取它的地址，知道了它的地址就可以调用它了，而对于函数的地址，我们可以使用指针来记录，这样的指针在C/C++里面称为函数指针 Pointers。

有了函数指针，我们就可以写出一个对判断条件也通用的函数模板了：

```
bool is_odd(int x) {
    return x % 2 != 0;
}
bool is_even(int x) {
    return x % 2 == 0;
}
bool is_positive(int x) {
    return x > 0;
}
template <typename iter_type>
bool any_of(iter_type begin, iter_type end, bool (*func)(int)) {
    for (iter_type it = begin; it != end; ++it) {
        if (func(*it)) {
            return true;
        }
    }
    return false;
}
```

To declare an appropriate function pointer, we can use the following steps:

- Start with a function signature:

  `int max_int(int x, int y);`

- Remove the parameter names, which serve only as documentation in a declaration:

`int max_int(int, int);`

- Replace the function name with a variable name and the `*` symbol to indicate it is a pointer, surrounded by parentheses:

  `int (*func3)(int, int);`

The result is that `func3` is a pointer to a function that takes two `int` s and returns an `int` .

1.1 Functors

使用 `any_of` 和 `is_positive` 我们可以判断一个序列中是否有大于零的元素，那我们想要判断一个序列中是否有大于任意一个数的元素呢？比如32或者212，我们可以写一个函数来判断：

```cpp
bool greater(int x, int threshold) {
    return x > 32;
}
```

但这个函数就不能与函数指针适配了，因为函数指针所指的函数只接收一个参数。所以，我们需要的是一种能存储 `threshold` 并且只需一个参数就能调用的东西，显然这里的 `greater` 是不满足需求的。

🏅 More specifically, we want a *first-class entity*, which is an entity that supports the following:

- It can store state.
- It can be created at runtime.
- It can be passed as an argument or returned from a function.

Unfortunately, functions are not first-class entities in C++: they cannot be created at runtime, and they are very limited in how they can store information.

> A function may have a static local variable, but it can only store a single value at a time. We need an arbitrary amount of storage – for instance, we may have any number of threshold values we care about for `greater()` , so we need a way of creating multiple copies of the function that each has their own stored threshold value.

Class types, on the other hand, do define first-class entities. A class-type object can store state in member variables, can be created at runtime, and can be passed between functions. So a class type could satisfy our needs, as long as there were a way to call it like a function. In fact, C++ does allow a class-type object to be called if the class overloads the function-call operator. We refer to such a class as a *functor*.

A functor is a class type that provides the same interface as a function, much like an iterator is a class type that provides the same interface as a pointer.

以下是一个名为 `GreaterN` 的类，它存储了 `threshold` ，并且可以接受一个参数进行调用：

```cpp
class GreaterN {
public:
    // EFFECTS: Creates a GreaterN with the given threshold
    GreaterN(int threshold_in): threshold(threshold_in) {}

    // EFFECTS: Returns whether or not a given value is greater than this
    // GreaterN's threshold
    bool operator(int x) const {
        return x > threshold;
    }
private:
    int threshold;
};
```

🎖 The function-call operator must be overloaded as a member function, so it has an implicit `this` pointer that allows us to access a member variable. We have declared the `this` pointer as a pointer to const, since the function does not modify the `GreaterN` object. We also get to decide what the parameters are, as well as the return type. We have chosen a single parameter of type `int` and a `bool` return type, since we want a `GreaterN` object to act as a predicate on `int` s.

现在，我们就可以创建 `GreaterN` 对象并使用了。

```cpp
int main() {
    GreaterN greater0(0);
    GreaterN greater32(32);
    GreaterN greater212(212);

    cout << greater0(-5) << endl; // 0
    cout << greater0(3) << endl;  // 1
    cout << greater0(-5) << endl; // 0 (false)
    cout << greater0(3) << endl;  // 1 (true)

    cout << greater32(9) << endl;  // 0 (false)
    cout << greater32(45) << endl; // 1 (true)

    cout << greater212(42) << endl;  // 0 (false)
    cout << greater212(451) << endl; // 1 (true)
    return 0;
}
```

接着，修改 `any_of` 就可以让它们协作了：

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

修改后，我们的 `any_of` 就支持任何类型的接受一个参数就可以调用的 `functor` 了。

```
int main() {
    list<int> list;
    vector<int> vec;
    int array[10];
    for (int i = 0; i < 10; ++i) {
        list.push_back(i);
        vec.push_back(i);
        array[i] = i;
    }
    cout << any_of(list.begin(), list.end(), GreaterN(0)) << endl; // 1
    cout << any_of(vec.begin(), vec.end(), GreaterN(1)) << endl;   // 1
    cout << any_of(array, array + 10, GreaterN(10)) << endl;       // 0
    return 0;
}
```

2. Recursion

🏅 If one thing uses itself as part of its definition, it is recursive.

比如，很经典的求阶乘

```
// REQUIRES: n >= 0
// EFFECTS: Computes and returns n!.
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

这函数是递归的，因为它调用它自己。

🏅 Operationally, the recursive definition works because each invocation of `factorial()` gets its own activation record, and the body of the function is executed within the context of that activation record. More conceptually, it works by using `factorial()` as an abstraction, even while we are still in the midst of implementing that abstraction! We call this “the recursive leap of faith” – we trust that the abstraction works, even if we haven’t finished writing it yet.

In general, a recursive abstraction requires the following:

- *base cases*, which are cases we can implement directly without recursion
- *recursive cases*, where we break down the problem into *subproblems* that are *similar* to the problem but “smaller,” meaning closer to a base case.

2.1 Tail Recursion

考虑如下函数：

```
// EFFECTS: Reverses the array starting at 'left' and ending at
//          (and including) 'right'.
void reverse(int *left, int *right) {
    if (left < right) {
        int temp = *left;
        *left = *right;
        *right = temp;
        reverse(left + 1, right - 1);
    }
}
```

🏅 Here, the recursive call is a *tail call*, meaning that the call is the last thing that happens in the function, and no work is done after the tail call returns. So no work is done after the tail call, there is no need to retain the activation record of the caller, and it can be discarded.

Many compilers recognize tail calls and perform *tail-call optimization (TCO)*, where the activation record for the tail call reuses the space of the caller’ s activation record. In the `g++` compiler, TCO is enabled at optimization level 2 (`-O2`). TCO is not restricted to recursive functions; as long as a function call is the last thing that happens in its caller, it is a tail call and the optimization can apply. However, tail-call optimization is most important for recursive functions, since it can reduce the space usage of the computation by a large factor.

A function is said to be *tail recursive* if it is recursive, and all recursive calls are tail calls. Rather than using a linear amount of space, a tail-recursive computation requires only constant space when tail-call optimization is applied. For instance, the tail-recursive version of `reverse()` uses space for only a single activation record under TCO.

In order for a computation to be tail recursive, it must do all its work before making a recursive call. A computation that does its work after a recursive call completes, so that the recursive call is not a tail call.

比如之前的 `factorial` 就不是一个尾递归，因为在递归调用返回后 `caller` 还有做一次乘法，为了将其改成尾递归，我们必须在递归调用之前就完成乘法的计算。就像这样

```cpp
int factorial(int n, int resultSoFar) {
    if (n == 0 || n == 1) {
        return resultSoFar;
    } else {
        return factorial(n - 1, n * resultSoFar);
    }
}
```

但这个接口不太一样，所以我们借助辅助函数对其进行抽象：

```cpp
int factorial(int n, int resultSoFar) {
    if (n == 0 || n == 1) {
        return resultSoFar;
    } else {
        return factorial(n - 1, n * resultSoFar);
    }
}
int factorial(int n) {
    factorial(int n, 1);
}
```

2.2 Kinds of Recursion

🏅

- A function is ***linear recursive*** if it is recursive, but each invocation of the function makes at most one recursive call.
- A function is ***tail recursive*** if it is linear recursive, and every recursive call is the last thing that happens in the invocation that makes the recursive call.
- A function is ***tree recursive*** if a single invocation of the function can make more than one recursive call.

2.3 Iteration vs. Recursion

🏅 Both iteration and recursion have the same computational power; an iterative algorithm can be converted to a tail-recursive implementation and vice versa. A non-tail-recursive computation can also be rewritten iteratively; however, the iterative version may require explicit storage. The non-tail-recursive algorithm can store data in multiple activation records, while the iterative one only has a single activation record to work with, so it may need an explicit data structure such as a vector to store its data.