深蓝学院
shenlanxueyuan.com

C++ 基础

第 10 章：泛型算法与 Lambda 表达式

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

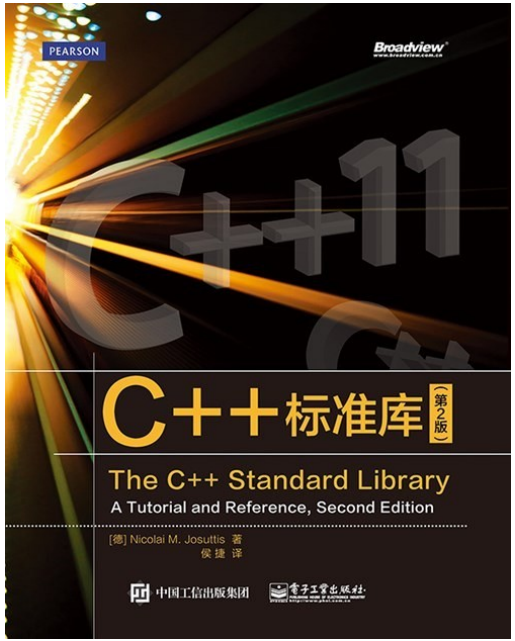
泛型算法

- 泛型算法：可以支持多种类型的算法
  - 这里重点讨论 C++ 标准库中定义的算法 `algorithm, numeric, ranges`
  - 为什么要引入泛型算法而不采用方法的形式
    - 内建数据类型不支持方法
    - 计算逻辑存在相似性，避免重复定义
  - 如何实现支持多种类型：使用迭代器作为算法与数据的桥梁
- 泛型算法通常来说都不复杂，但优化足够好

1.速度快
2.bug比较少

- 一些泛型算法与方法同名，实现功能类似，此时建议调用方法而非算法
  - std::find V.S. std::map::find

因为算法要支持不同类型，就会造成一定的性能损失。因此建议调用方法。



泛型算法（续 1）

- 泛型算法的分类
  - 读算法：给定迭代区间，读取其中的元素并进行计算
    - accumulate / find / count
  - 写算法：向一个迭代区间中写入元素
    - 单纯写操作： fill / fill_n
    - 读 + 写操作： ~~transpose~~ transform / copy
    - 注意：写算法一定要保证目标区间足够大
  - 排序算法：改变输入序列中元素的顺序
    - sort / unique

判断是不是在区间内出现了若干连续相同的元素，如果出现了相同的元素，那么只保留一个元素。最后的元素不进行修改，直接写入=》保证前面的元素是唯一的(需要调用erase把后面的元素删掉)
// 1, 2, 2, 3, 4 -> 1, 2, 3, 4, 4

```
    // a vector containing several duplicate elements
    std::vector<int> v{1,2,1,1,3,3,3,4,5,4};
    auto print = [&] (int id) {
        std::cout << "@" << id << ": ";
        for (int i : v)
            std::cout << i << ' ';
        std::cout << '\n';
    };
    print(1);

    // remove consecutive (adjacent) duplicates
    auto last = std::unique(v.begin(), v.end());
    // v now holds {1 2 1 3 4 5 4 x x x}, where 'x' is indeterminate
    v.erase(last, v.end());
    print(2);

    // sort followed by unique, to remove all duplicates
    std::sort(v.begin(), v.end()); // {1 1 2 3 4 4 5}
    print(3);

    last = std::unique(v.begin(), v.end());
    // v now holds {1 2 3 4 5 x x}, where 'x' is indeterminate
    v.erase(last, v.end());
    print(4);
}
```

Output:

```
@1: 1 2 1 1 3 3 3 4 5 4
@2: 1 2 1 3 4 5 4
@3: 1 1 2 3 4 4 5
@4: 1 2 3 4 5
```

泛型算法（续 2）

- 泛型算法使用迭代器实现元素访问
- 迭代器的分类：迭代器的类别category()
  - 输入迭代器：可读，可递增 —— 典型应用为 find 算法
  - 输出迭代器：可写，可递增 —— 典型应用为 copy 算法
  - 前向迭代器：可读写，可递增 —— 典型应用为 replace 算法
  - 双向迭代器：可读写，可递增递减 —— 典型应用为 reverse 算法
  - 随机访问迭代器：可读写，可增减一个整数 —— 典型应用为 sort 算法
- 一些算法会根据迭代器类型的不同引入相应的优化：如 distance 算法

类别

因为List的迭代器不是随机访问迭代器，因此就不可以使用sort算法。
我们需要根据迭代器的性质，来决定是否可以使用改算法。
c++标准库里面并不是每一种算法都支持所有的迭代器。

```
// implementation via tag dispatch, available in C++98 with constexpr removed
namespace detail {

template<class It>
constexpr // required since C++17
typename std::iterator_traits<It>::difference_type
    do_distance(It first, It last, std::input_iterator_tag)
{
    typename std::iterator_traits<It>::difference_type result = 0;
    while (first != last) {
        ++first;
        ++result;
    }
    return result;
}

template<class It>
constexpr // required since C++17
typename std::iterator_traits<It>::difference_type
    do_distance(It first, It last, std::random_access_iterator_tag)
{
    return last - first;
}

} // namespace detail
```

泛型算法（续 3）

```
int main()
{
    std::deque<int> q;
    std::back_insert_iterator< std::deque<int> > it(q);

    for (int i=0; i<10; ++i)
        it = i; // calls q.push_back(i)

    for (auto& elem : q) std::cout << elem << ' ';
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

```
std::vector<int> x;
std::fill_n(std::back_insert_iterator<std::vector<int>>(x), 10,3);
//因为上面那句话太长了，因此c++提供了一个back_inserter来简化
//std::fill_n(std::back_inserter(x),10,3);
for(auto i: x) {
  std::cout << i << " "; } // 3 3 3 3 3 3 3 3 3 3
```

- 一些特殊的迭代器
  - 插入迭代器： back_insert_iterator / front_insert_iterator / insert_iterator

    本质上就是用push_back()实现
  - 流迭代器： istream_iterator / ostream_iterator

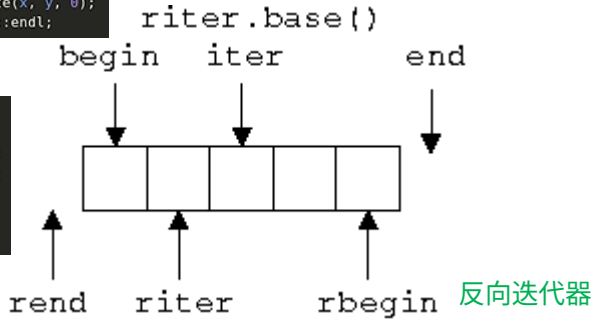
    我们可以把它应用在一些可以使用输入迭代器的算法上。

```
std::istringstream str("1 2 3 4 5");
std::istream_iterator<int> x(str);
std::istream_iterator<int> y{};
int res = std::accumulate(x, y, 0);
std::cout << res << std::endl;
```

  - 反向迭代器（图片选自 www.cs.helsinki.fi ）
  - 移动迭代器： move_iterator

```
std::istringstream str("1 2 3 4 5");
std::istream_iterator<int> x(str);
std::cout << *x << std::endl;   // int tmp; str >> tmp;
std::cout << *x << std::endl;   // int tmp; str >> tmp;

++x;                            // str >> tmp;
std::cout << *x << std::endl;
```

对流进行解析。解析出传入的int参数。



- 迭代器与哨兵（ Sentinel ）
- 并发算法（ C++17 / C++20 ）

SIMD c++ single instruction multiple data 硬件提供的支持

  - std::execution::seq sequence
  - std::execution::par Parallel
  - std::execution::par_unseq

并发非顺序执行，unseq指的就是使用SIMD来进行处理

  - std::execution::unseq

使用SIMD来处理，但是使用单线程。来执行程序

```
std::istringstream str("1 2 3 4 5");
std::istream_iterator<int> x(str);
std::istream_iterator<int> y{};
for (; x != y; ++x)
{
    std::cout << *x << std::endl;
}
```

使用缺省的方式构造流迭代器的话，它可以用来表示迭代器的结尾位置

bind 与 lambda 表达式

- 很多算法允许通过可调用对象自定义计算逻辑的细节
  - transform / copy_if / sort…
- 如何定义可调用对象
  - 函数指针：概念直观，但定义位置受限 只能再函数外部定义。对于某些只用一次的函数，就比较浪费
  - 类：功能强大，但书写麻烦
  - bind ：基于已有的逻辑灵活适配，但描述复杂逻辑时语法可能会比较复杂难懂
  - lambda 表达式：小巧灵活，功能强大

bind

bind：通过绑定的方式修改可调用对象的调用方式

- 早期的 bind 雏形： std::bind1st / std::bind2nd 和第一个参数进行绑定 和第二个参数进行绑定
  - 具有了 bind 的基本思想，但功能有限 只有部分可调用对象可以使用，
- std::bind （C++11 引入）：用于修改可调用对象的调用方式 使用bind需要引入namespace：using namespace std::placeholders

```
std::copy_if(x.begin(), x.end(), std::back_inserter(y), std::bind(MyPredict2, 1, 3));
```

  - 调用 std::bind 时，传入的参数会被复制，这可能会产生一些调用风险 31分，可以看一下例子
  - 可以使用 std::ref 或 std::cref 避免复制的行为
- std::bind_front （C++20 引入）： std::bind 的简化形式

```
bool MyPredict(int val) {
  return val > 3; }
int main() {
  std::vector<int> x{1,2,3,4,5,6,7,8,9};
  std::vector<int> y;
  std::copy_if(x.begin(), x.end(), std::
back_inserter(y), std::bind2nd(
std::greater<int>(), 3));
  for ( auto p: y) {
    std::cout << p << ' ';}
}
```

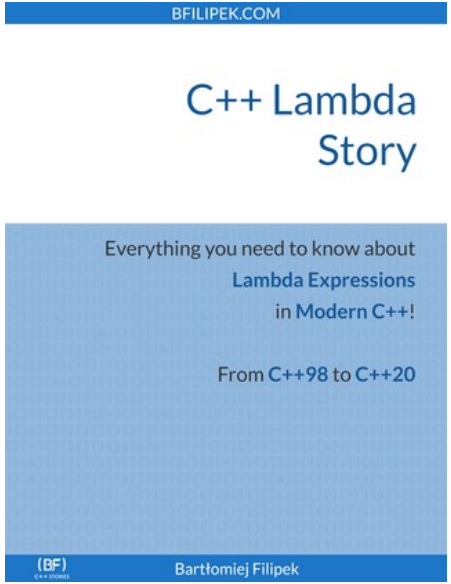
```
auto x = std::bind(MyPredict2, _1, 3);  //可调用对象
//  1表示我接下来调用x的时候所传入的第几个参数。
即把50和3传入参数，不是指MyPredict2的第几个参数
_1是定义在std::placeholders下的
x(50); // x会调用MyPredict2的第一个参数，


auto x1 = std::bind(MyPredict2, _1, 3);
auto x2 = std::bind(MyPredict2, 10, _1);
auto x3 = std::bind(MyAnd, x1, x2);
// 先调用x3，然后尝试用5来对x1，x2来进行计算。
因为x1，x2中有_1,那么他会用5来替换_1.x1和x2都
计算出来之后，再去计算x3
```

```
void Proc(int& x) {
  ++x; }
int main() {
  int x = 0;
  auto b = std::bind(Proc, x);
  b();
  std::cout << x << std::endl;
  // 此时x为0，因为bind传入的参数会被
复制，因此，b其实没有调用x，而是调用x
的复制值。
  // 改进方法：
  auto b = std::bind(Proc, std::ref
(x));
}
```

17:01

lambda 表达式

- lambda 表达式（https://leanpub.com/cpplambda）：
  - 为了更灵活地实现可调用对象而引入
  - C++11 ~ C++20 持续更新
    - C++11 引入 lambda 表达式
    - C++14 支持初始化捕获、泛型 lambda
    - C++17 引入 constexpr lambda， *this 捕获
    - C++20 引入 concepts，模板 lambda
- lambda 表达式会被编译器翻译成类进行处理



lambda 表达式（续 1）

```
auto x = [](int val) { return val > 3; };
// 小括号内是函数的参数部分。{}内是函数体
```

```
//初始化捕获
std::string a = "hello";
auto lam = [y=std::move(a)]() {
  std::cout << y;};
std::cout << a << std::endl;
// 此时a已经空了，因为在构造lambda表达式的时候，a就已经被move了。
优点：1.可以通过这种方式引入一些更复杂的捕获逻辑。2.可以一定程度上提升系统性能 auto lam = [z=x+y](int val)
```

- lambda 表达式的基本组成部分 本质上是生成了一个类
  - 参数与函数体 返回类型可以自动推导 也可以指定
  - 返回类型 显式指定返回类型
    ```
    auto x = [](int val) -> float
    ```
  - 捕获：针对函数体中使用的局部自动对象进行捕获

    如果是静态对象，则编译器会报错 static int y = 10; // 报错 静态对象是不需要捕获的，可以直接在lambda中使用

    ```
    int y = 10;
    auto x = [y](int val)
      {
        return val > y;
      };
    //捕获，把y值复制到lambda表达式内部
    实现：在class里面定义了一个私有变量y
    ```

    值捕获：捕获的值不会变 [y]
    引用捕获：[&y]：捕获值会改变
    混合捕获：[&y, z]
    [=]：编译器自动分析，需要捕获哪些局部自动对象。值捕获
    [&]:编译器自动分析，需要捕获哪些局部自动对象，引用捕获
    [&,z]:表示通常采用引用捕获，而z采用值捕获
    - 值捕获、引用捕获与混合捕获
    - this 捕获

      this捕获
      ```
      struct Str {
        auto fun() {
          int val = 3;
          auto lam = [val, this] () {
            return val > x; };
        }
        int x;
      };
      // this 是个对象，指向str
      ```
    - 初始化捕获（C++14）
    - *this 捕获（C++17） 44分
  - 说明符
    - mutable / constexpr (C++17) / consteval (C++20)…… 可以在编译器进行调用
  - 模板形参（C++20）
    ```
    auto lam = []<typename T>(T val) {
      return val + 1;
    };
    ```

lambda 表达式（续 2）

- lambda 表达式的深入应用
  - 即调用函数表达式（Immediately-Invoked Function Expression, IIFE）
  - 捕获时计算（C++14）
  - 使用 auto 避免复制（C++14）
  - Lifting （C++14）
  - 递归调用（C++14）

```
//捕获时计算
int x = 3;
int y = 5;
auto lam = [x,y]() {
  return x+y;};
// 优化版
auto lam = [z=x+y]() {
  return z; };
```

```
//即调用函数表达式
const auto val = []() {return 1;}();
//先构造了一个lam表达式，然后
马上执行这个lambda表达式
//优点：在初始化常量表达式const auto val的时候，
我们需要在定义的时候就赋值。如果使用一个函数来
初始化，那么既不利于阅读，也不利于维护。
如果改成lambda表达式，就比较简单、易读。
```

```
//使用auto 避免复制,可以使用
const auto&来避免编程不小心而
引入的复制
auto lam = [](const auto&p) {
  return p.first + p.second; };
```

```
// lifting
auto fun(int val) {
  return val + 1;}
auto fun(double val) {
  return val + 1;
}
int main() {
  auto b = std::bind(fun, 3);
//编译器没有办法区分应该绑定哪个fun函数，
所以会报错。
  std::cout << b() << std::endl;
  //解决办法
  auto lam = [](auto x) {
    return fun(x);};
  std::cout << lam(3) << std::endl;
  std::cout << lam(3.5) << std::endl;
}
```

```
//递归调用
int factorial(int n) {
  return n > 1 ? n * factorial(n-1):1; }
// 把上面的函数转化为lambda表达式
int main() {
  auto factorial = [](int n) {
    return n>1? n* factorial(n-1):1; };
  //直接这么写会报错。因为当我们解析到
auto factorial这句时，这个对象的类型是不
确定的。因为是个迭代的表达，因此不能确定
是什么类型
//改进
  auto factorial = [](int n) {
    auto f_impl = [](int n, const auto&
impl) -> int {
      return n > 1 ? n*impl(n-1,impl):1;
    };
    return f_impl(n, f_impl);
  };
//在第一层的factorial里面没有出现
factorial,即没有出现递归，那么就可以解
析。在第二层中，f_impl也没有出现f_impl，
所以也是可以解析的。
如果删除->int,那么程序就会报错

}
```

泛型算法的改进—— ranges

```
std::vector<int> x{1,2,3,4,5,6};
// 使用迭代器
auto it = std::ranges::find(x.begin(), x.end(), 3);
// 使用容器
auto it = std::ranges::find(x, 3);
```

- 可以使用容器而非迭代器作为输入
  - 通过 std::ranges::dangling 避免返回无效的迭代器
- 从类型上区分迭代器与哨兵
- 引入映射概念，简化代码编写
- 引入 view ，灵活组织程序逻辑

```
auto fun() {
  return std::vector<int>{1,2,3,4,5};
}
int main() {
  auto it = std::ranges::find(fun(), 3);
  std::cout << *it ;
// error, fun返回了一个局部对象，在调用完auto那条语句之后，这个局部对象就会被删除掉。换而言之，这个fun返回的是右值。那么it会指向右值的一个位置。那么it就会指向一个非法的地方。
```

```
//引入映射概念，简化代码编写
std::map<int, int> m{{2,3}};
//如果你想找value为3的值，那么之前的时候你需要写一个lambda表达式来找
现在可以直接使用ranges中的proj的功能
auto it = std::ranges::find(m, 3,
&std::pair<const int, int>::second);
std::cout << it->first;
// &std::pair<const int, int>::second本质上是一个指针，指向了pair中second元素。
```

```
//引入view
优点1：
view不是对输入立即计算，是需要的时候计算。这种推迟可以某种程度上提高计算的性能。2.模糊了算法和容器的概念。3.灵活组织程序逻辑
```

深蓝学院
shenlanxueyuan.com

并发计算

```
for (int i = 0; i < 5; ++i)
{
    using namespace std::chrono;
    std::vector<double> sorted(vals);
    const auto startTime = high_resolution_clock::now();
    std::sort(std::execution::par, sorted.begin(), sorted.end());
    const auto endTime = high_resolution_clock::now();
    std::cout << "Latency: "
              << duration_cast<duration<double, std::milli>>(endTime - startTime).count()
              << std::endl;
}
```

使用par速度减少了。par的本质会建立多个线程=》提高系统性能

```
std::random_device rd;

std::vector<double> vals(10000000);
for (auto& d : vals) {
  d = static_cast<double>(rd());
}

for (int i = 0; i < 5; ++i)
{
    using namespace std::chrono;
    std::vector<double> sorted(vals);
    const auto startTime = high_resolution_clock::now();
    std::sort(sorted.begin(), sorted.end());
    const auto endTime = high_resolution_clock::now();
    std::cout << "Latency: "
              << duration_cast<duration<double, std::milli>>(endTime - startTime).count()
              << std::endl;
}
```

感谢聆听 !

```
/usr/bin/clang++  -c  "/home/cpp_course/demo/demo/main.cpp" -O3 -Wall --std=c++2a -DNDEBUG  -o ./Release/main.cpp.o -I. -I.
/usr/bin/clang++ -o ./Release/demo @"demo.txt" -L.    -ltbb
```

```
std::random_device rd;

std::vector<double> vals(10000000);
for (auto& d : vals) {
  d = static_cast<double>(rd());
}

for (int i = 0; i < 5; ++i)
{
    using namespace std::chrono;
    std::vector<double> sorted(vals);
    const auto startTime = high_resolution_clock::now();
    std::sort(std::execution::unseq, sorted.begin(), sorted.end());
    const auto endTime = high_resolution_clock::now();
    std::cout << "Latency: "
              << duration_cast<duration<double, std::milli>>(endTime - startTime).count()
              << std::endl;
}
```

时间并没有什么变化，对于老师的机器来说，使用unseq没有提高速度