lambda 表达式

基本概念

- lambda 表达式是 C++ 11 所引入的非常重要的特性，其本质就是一个匿名函数，并且能够捕获一定范围内的变量
- 基本格式
  - [捕获列表](形参列表) -> 返回类型 { /* 函数体 */ };
    - 尾置返回类型
  - ```
    [](const int& a, const int& b) -> bool {
        return  a < b;
    }
    ```
- 特性
  1. 和普通函数不一样的是，lambda 表达式可以省略返回类型，由编译器自行推断，并且不需要使用 decltype
     - ```
       [](const int& a, const int& b) {
           return  a < b;
       };
       ```
     - 由编译器自行推断，返回类型为 bool
  2. 若 lambda 表达式不需要接收参数，那么形参列表也可以省略
     - ```
       [] {
           cout << "lambda" << endl;
       };
       ```
     - 形参列表和返回类型可以都被省略
  3. 捕获列表与函数体不可被省略
- lambda 表达式本身其实就是一个匿名的可调用对象，在实际编程应应尽可能地简短，大部分情况下会在 sort、for_each 等函数中使用

捕获列表

- lambda 表达式比较复杂和容易出错的地方就在于捕获列表，捕获列表可以看作是将外部作用域中的变量"搬移"到 lambda 表达式中的内部
1. [] 表示不捕获任何外部变量
   ```
   int main () {
       int i = 1024;
       auto f = [] {
           cout << i << endl;    // Wrong
       };
   }
   ```
   - lambda 表达式有自己的作用域，虽然 lambda 表达式看起来像个闭包，但是不能直接获取外部作用域的变量
2. [=] 表示按值方式捕获外部作用域中所有的变量
   ```
   int main () {
       int i = 1024;
       auto f = [=] {
           cout << i << endl;    // Right
           i = 2048;             // Wrong
       };
   }
   ```
   - 此时，lambda 表达式外部作用域内的所有变量均通过按值传递的方式传递进 lambda 表达式中，不需要在形参列表中定义
   - **但是，按值传递时相当于"只读"，我们不能修改捕获的变量**
   - 按值捕获将会触发对象的拷贝动作，对于大对象而言，需要慎重使用
3. [&] 表示按引用的方式捕获外部作用域中的所有变量
   ```
   int main () {
       int i = 1024;
       auto f = [&] {
           cout << i << endl;    // Right
           i = 2048;             // Right
       };
   }
   ```
   - 当我们使用按引用的方式捕获外部变量时，它是可读可写的
   - **但是，按引用捕获外部变量在 lambda 表达式延迟调用时必须保证这些变量仍然有效**
4. [变量名] 与 [&变量名]
   - 有时候我们并不需要外部作用域中的全部变量，那么我们就可以使用这种方式进行按需捕获
   - 其中 [变量名] 为按值捕获该变量，[&变量名] 则表示按引用捕获该变量
5. [=, &变量名a, &变量名b, ...]
   - 按值捕获外部所有的变量，但是按引用的方式捕获变量a，变量b 等
6. [&, 变量名a, 变量名b, ...]
   - 按引用捕获外部所有的变量，但是按值的方式捕获变量a，变量b 等
7. [this] 使得 lambda 表达式内部可访问成员变量
   - 用于类中，其作用就是捕获 this 指针，但是在 lambda 表达式中可以直接访问成员变量，无需显式使用 "this->"
   - 另外，如果我们使用了 [=] 或者是 [&]，this 指针也会被捕获

常见陷阱

- **按值捕获时，对象的复制发生在定义 lambda 表达式那一刻**
  ```
  int i = 1024;
  auto f = [=] { return i; };
  i = 2048;

  cout << f() << endl;
  ```
  - 输出结果为 1024，并非 2048
  - **也就是说，凡是按值捕获的外部变量，在 lambda 表达式定义的时刻，所有的外部变量的值就已经被复制到 lambda 表达式中了**
- **按引用捕获时，lambda 延迟调用的时刻捕获对象已失效**
  ```
  vector<function<int(void)>> functions;      // 定义一个 function 容器

  void foo() {
      int i = 1024;
      functions.emplace_back([&]{return i;});    // 按引用捕获变量 i
  }

  int main() {
      foo();

      cout << functions[0]() << endl;              // 调用 lambda 表达式

      return 0;
  }
  ```
  - 运行结果是一个不确定的值
  - 因此，我们要警惕按引用捕获变量，确保 lambda 表达式被调用时这些变量不会失效
  ```