C++

lambda expression

C++中的 lambda expression 同样用于定义 anonymous function，与 Haskell，Python，Racket 类似
但 C++ 中的 lambda 表达式具有更多的实现细节，并且能向多部分描述其他语言中的情形

Syntax `[captures] (parameters) specifiers exception attr -> ret {body}`
常用形式为：`auto func = [...] (...) mutable? {...};` 调用形式为：`func(...);`

(parameters) 形式参数列表，语法上与普通函数定义一样（传值，传引用，传常量引用） T　T&　const T&
注意：虽然语法规则上可以省略，但在有 mutable 关键字时必须保留 ()，
所以相对而言可以忽略这条省略去规则

{body} {statement*}，函数体，语法上与普通函数定义一样

Expection 声明用于指定函数抛出的异常类型，如 throw(std::runtime_error)　of the closure type

attr provides the attribute specification (属性指定) for the type of the function call operator

->ret 返回值类型，标识函数的返回值类型，如：-> int　(type inference) 类型
当返回值为 void 或函数体中只有一处 return 时，可以省略 -> ret，此时由编译器自动推断返回的值

specifiers 指示符，最常用的为 mutable，使 lambda 函数中可以修改通过传值捕获的变量的拷贝
但是不会影响变量在原环境中的值

[captures] 捕获外部变量列表，[] 同时是 lambda 引出符（类似 Racket Python 的 lambda），编译器据此判断代码为入函数
也称函数对象参数，用于传递给编译器自动生成的函数对象类的构造函数的

注意：只能使用到定义 lambda 时为止，在 lambda 所在作用范围内可见的局部变量，与 Haskell Racket 类似
但特别注意：Python 中允许 lambda 定义时使用仍未定义的变量或不可见的变量
（交互式中还有一些区别）只要在函数调用时相关变量名有定义即可

传值 [=] 以传值方式引用作用范围可见的所有变量（包括 lambda 所在类的 this 指针）
注意 lambda 默认是 const 的，即通过传值捕获的变量不可修改（read-only variable）
这种形式下与 Haskell 最为类似，即通过上层作用域捕获的变量不可修改
特别注意：这种形式实现了闭包，即函数调用时使用的是函数定义时的环境，而非调用时的环境
与 Racket 在 lexer-parser-eval 中的函数式调用类似，
即函数定义时会将定义环境作为函数的一部分装入 closure

传引用 [&] 以传引用方式引用作用范围可见的所有变量（包括 lambda 所在类的 this 指针）
注意：虽然 lambda 默认是 const 的，但不影响对传引用变量的修改（不确定是否为编译器差异）
对变量的修改会影响原环境的变量值，与一般函数传引用（T&）类似
这种形式与 Python 类似，此时 lambda 函数的调用使用的是调用它的环境
而非定义 lambda 函数的环境（注意此时函数具有副作用，是不纯的）

所在类 [this]，lambda 可用通过 this 调用其定义所在类的成员，如 [=]() {this->member();}

混合 [=, &a]，除 a 以外皆传值引用，[&, *b] 除 b 以外皆传引用，[*a, *b] 除 a, b 外皆不捕获
注意不可重复传入，如 [=, *a]，[&, &b]，[a, &a]