C++ 11/14/17(Session 3)

Gong xuan Jun.2021

课程内容

- Lambda
  - Lambda函数
  - 变量捕获
  - Lambda捕获*this
  - 泛型Lambda表达式
  - Lambda表达式用作返回值
  - Lambda表达式用作参数
  - Constexpr Lambda表达式
  - Lambda表达式在STL算法中的使用
- 函数对象
  - 函数对象概述
  - C++中的仿函数类
  - 函数对象适配器
  - 函数对象取反器

Lambda函数（一）

- Lambda函数可以编写内嵌的匿名函数，使代码更容易阅读和理解
- C++引入Lambda的主要目的是让用户能够将类似于函数的表达式用作接受函数指针或函数符的函数参数
- 语法：

  [capture_block](parameters)mutable exceptioin_specification attribute_specifier->return_type{body}

- capture_block: 指定如何捕捉所在所用域中的变量，并供lambda主题部分使用
- parameters: (可选)参数列表。只有在不需要任何参数且没有指定mutable、exception_specification、attribute_specifier和return_type的情况下才能忽略该列表
- mutable: (可选)把lambda表达式标记为mutable
- exception_specification:(可选)用于指定lambda表达式体可以抛出的异常
- attribute_specifier: (可选)用于指定lamba表达式的属性
- return_type: (可选)返回值的类型，如果忽略了return type,编译器会根据函数返回类型推断原则判断返回类型

Lambda函数（二）

- 无参数的 lambda函数

```
auto printStr = []{ std::cout<<“This is  a basic lambda\n”;};
printStr(); // call lambda functioin
```

- 带参数的 lambda函数

```
auto printValue = [](int val){std::cout<<"value="<<val<<"\n";};
printValue(10); // call lambda function
```

- 有返回值的 lambda函数

```
auto getSumOfTwoValue = [](int a,int b)->int{return a+b;};
int sum = getSumOfTwoValue(a,b);
```

Lambda函数

捕捉变量（一）

- 作用域内捕捉变量

```
const int val = 10;
auto captureVarInScope = [val]{cout<<“val=”<<val<<endl;};
int sum = captureVarInScope ();
```

尝试下面的两种情况，有什么不同：

case one:

```
int val = 10;
auto captureVarInScope = [val]{val+=1;cout<<“val=”<<val<<endl;};
int sum = captureVarInScope ();
```

case two:

```
int val = 10;
auto captureVarInScope = [val]() mutable{val+=1;cout<<“val=”<<val<<endl;};
int sum = captureVarInScope ();
```

(lambda表达式中按值捕捉了非const变量， 其表达式也不能修改其副本；如果要修改则需要使用mutable)

Lambda函数

捕捉变量（二）

- 作用域内捕捉变量(按引用捕捉)

```
{
 int val = 10;
 auto captureVarInScope = [&val]{val+=1;std::cout<<“val=”<<val<<endl;};
}
```

（按引用捕捉变量时，必须确保执行lambda表达式时，该引用有效）

采用两种方式捕捉所在作用域中的所有变量：

- [=]:通过值捕捉所有变量
- [&]:通过引用捕捉所有变量

指定捕捉列表：

- [&x]:只通过引用捕捉x
- [x]:只通过值捕捉x
- [=,&x,&y]:默认通过值捕捉，变量x和y是例外，这两个变量通过引用捕捉
- [&,x]:默认通过引用捕捉，变量x是例外，这个变量通过值捕捉
- [&x,&x]:非法，因为标识符不允许重复
- [this] 捕捉周围的对象，即使没有使用this->,也可以在lambda表达式中访问这个对象
- [*this]显式地捕获当前对象的副本

Lambda函数

Lambda捕获*this

- 从C++17开始，Lambda表达式可捕获*this的值，但this及其成员为只读, 即传递到Lambda中的是this对象的拷贝，例如：

```
class A
{
public:
  int getX2() { return [*this]() { return x2; }(); }
private:
    int x1{0};
};
```

请考虑什么情况下会使用到这种情况？

Lambda函数

泛型Lambda 表达式

- 从 C++14开始，可以给Lambda表达式的参数使用自动类型推断，无需显示指定它们的具体类型

```
vector<int> intVec{10,20,30,105,209};
vector<double> doubleVec{11.1,34.2,100.3};
auto isGreaterThan100 = [](auto i){return i>100;};
auto intIter = find_if(begin(intVec),end(intVec),isGreaterThan100);

if(intIter != end(intVec)){
  cout<<"found a value>100:"<<*intIter<<endl;
}
auto doubleIter = find_if(begin(doubleVec),end(doubleVec),isGreaterThan100);
if(doubleIter != end(doubleVec)){
  cout<<"found a value>100:"<<*doubleIter<<endl;
}
```

Lambda函数

捕捉表达式

- C++14支持Lambda捕捉表达式，允许用任何类型的表达式初始化捕捉变量

```
double pi=3.1415;
auto getPi=[capturePi="pi:",pi]{std::cout<<capturePi<<pi;};
```

- 使用按值捕捉使用复制语义，有时对于只能使用移动的对象，通常可以使用移动来捕捉

```
auto ptr = make_unique<double>(3.1415);
auto lmd = [p=move(ptr)]{std::cout<<*p;};
```

Lambda函数

lambda表达式用作返回值

- 使用std::function, 可以给lambda表达式指定名称，并从函数中返回

```
std::function<int(void)> multiplyBy2Lambda(int x)
{
  return [x]{return 2*x;};
 }
std::function<int(void)> fn = multiplyBy2Lambda(5);
std::cout<<"fn="<<fn()<<endl;
```

- C++14支持函数返回类型推断，上面的函数和其调用可改写为：

```
 auto multiplyBy2Lambda(int x)
{
 return [x]{return 2*x;};
 }
auto fn = multiplyBy2Lambda(5);
cout<<"fn="<<fn()<<endl;
```

(std::function定义在<functional>,是一个多态的函数对象包装，类似函数指针)

Lambda函数

Lambda表达式用作参数

- Lambda表达式可以作为函数的参数

```
void printValueOfMeetCondtion(const vector<int>&vec, const function<bool(int)>& fun)
{
  for(const auto& i:vec)
{
    if(! fun(i)))
   {
      break;
    }
    cout<<i<<" ";
  }
  cout<<endl;
}
vector<int> vec{1,2,3,4,5,6,7,8,9,10};
printValueOfMeetCondtion(vec, [](int i){return i<4;});
```

Lambda函数

constexpr Lambda表达式

- 从c++17起，Lambda在满足编译时期constexpr的要求的情况下，会隐式转换为constexpr表达式。

```
auto squared = [](auto val) { // implicitly constexpr since C++17
   return val * val;
};
 int main()
{
   std::array<int, squared(5)> arr;
    auto size = arr.size();
    return 0;
}
```

Lambda函数

Lambda表达式用作参数(constexpr)

- 任何Lambda都可以在编译时上下文中使用，前提是它使用的特性对编译时上下文中有效。即满足如下要求:
- Lambda表达式内没有静态变量；
- Lambda表达式内没有虚函数；
- Lambda表达式内没有 try/catch语句；
- Lambda表达式内没有new/delete；

```
auto squared2 = [](auto val) {
static int calls = 0; // OK, but disables lambda for constexpr contexts
...
return val*val;
};
```

Lambda函数

lambda表达式在STL算法中的使用

- count_if

```
vector<int> vec{1,2,3,4,5,6,7,8,9};
int val = 3;
int cnt = count_if(cbegin(vec),cend(vec),[val](int i){return i>val});
std::cout<<“found”<<cnt<<“values>”<<val<<std::endl;
```

- 参见示例：

```
bool Pools::atLeastOneL2HiPoolConfigured() const
{
    return std::any_of(pools.cbegin(), pools.cend(),
                       [](const auto& pool) { return pool.configured && pool.type ==
    MessageStructs::OAM::PoolType::L2_NRT; });
}
```

函数对象

函数对象概述

- 重载函数调用符，使类的对象可以取代函数指针，这些对象称为函数对象或仿函数（functor）
- C++提供了一些预定义的仿函数类（定义在头文件 <functional>中），执行最常用的回调操作

lambda表达式可解决创建函数或仿函数类时的名称冲突问题，所以尽可能使用lambda 表达式，而不是函数对象

函数对象

C++中的仿函数类

几类预定义的仿函数类

- 算术函数对象：C++提供了5类二元算术运算符的仿函数类模板：plus,minus,multiplies,divides and modulus
- 比较函数对象:equal_to,not_equal,less,greater,less_equal and greater_equal
- 逻辑函数对象: logical_not(operator!),logical_and(operator&&) and logical_or(operator||)
- 按位函数对象: bit_and(operator&),bit_or(operator|) and bit_xor(operator), bit_not(operator~)(C++14)

透明运算符仿函数（C++14）,例如multiplies<>()可以取代multiplies<int>()

（应总是使用透明访函数）

函数对象适配器（一）

- 使用基本的函数对象时，通常会出现参数不匹配的问题,下面的例子是查找大于等于2的所有元素

```
vector<int> vec{1,2,3,4};
find_if(vec.begin(), vec.end(), greater_equal<int>()); //what's problem?
```

- 对于上面的例子需要解决的问题是：
  - 如何将接受两个形参的函数对象转换为接受一个形参的函数对象？
  - 如何指定被比较的值？

函数对象适配器（二）

- 函数对象适配器将会解决这个问题：

auto it = find_if(vec.begin(), vec.end(),
bind(greater_equal<>(), placeholders::_1, 2));

注：1.绑定器(binder)用于将函数的参数绑定至特定的值

```
template< class F, class... Args >
/*unspecified*/ bind( F&& f, Args&&... args );
```

2. _1,_2,_3等表示没有绑定至指定的值的参数，这些定义在std:: placeholders

3.c++11已经废除了bind2nd() 和 bind1st(),请改用lambda和bind（）

函数对象

函数对象适配器（取反器）

- 取反器类似绑定器，但取反器计算谓词结果的反结果：

```
bool GreatAndEqual(int num) {return (num >= 100); }
function<bool(int)> f = GreatAndEqual;
auto it = find_if(vec.begin(), vec.end(), not1(f));
```

注：1.not1中的”1”表示这个操作数必须是一元函数，即接收一个参数的的函数；
如果接收两个参数则用not2()

- 尽量使用lambda表达式

```
auto it = find_if(vec.begin(), vec.end(), [](int i){ return i < 100; });
```

Q&A

/