C++程序设计精要教程

华中科技大学

Lambda表达式

- lambda表达式本质上是一个匿名、自动内联的函数
- 与任何函数类似，一个lambda表达式具有返回类型、参数列表、函数体{}
- 和普通函数不一样的是：lambda表达式有捕获列表，可以定义在一个函数内部(C++是不允许在一个函数里定义另外一个函数的)
- 一个lambda表达式具有如下形式

  [捕获列表](形参列表) mutable 异常说明 -> 返回类型 {函数体}

其中：

捕获列表是必须有的（即使列表为空也必须有[]）； 函数体{}是必须有的

mutable和异常说明是可选的

当不需要任何参数时，(形参列表)可以没有

当函数返回类型可以推断时，->返回类型 可以没有

注意：

形参列表指传递给lambda表达式参数（和函数形参列表一样），函数参数称为绑定变量，它在函数上下文有明确的定义

捕获列表指lambda表达式要使用的、定义在lambda表达式外面的变量，称为自由变量。

lambda表达式可以赋值给一个变量，变量的类型用auto

Lambda表达式

- C++不允许在一个函数里定义另外一个函数

```
void f1(){
    void f2(){}
}
```

- 但是可以在一个函数里定义lambda表达式(而且这是最常见的lambda应用场景)

Lambda表达式

```
//定义了一个lambda表达式，捕获列表为空,不需要任何参数，返回类型省略（可以从函数 体推断出//返回类型为int）
  auto f1 = [] {return 42;}; //等价于   auto f1 = []()->int {return 42;};
//将一个lambda表达式赋值给变量f1后，f1就可以看成函数名，可以通过f1进行调用
  int rtn_f1 = f1();
  cout << "rtn_f1 = " << rtn_f1 << endl;
  //这里定义了一个lambda表达式，捕获列表为空，形参int x(默认值为1)，返回类型为int
  auto f2 = [ ](int x=1)->int { return x; };
  int rtn_f2 = f2(10);
  cout << "rtn_f2 = " << rtn_f2 << endl;
  auto f3 = [](int i) { return i < 0?-i:i;}; //功能：返回整数i的绝对值
  int rtn_f3 = f3(-20);
  cout << "rtn_f3 = " << rtn_f3 << endl;
  //如果有兴趣，可以用typeid函数看看f3被自动推断出什么类型
  //如果用gcc
  cout << "type of f1:" << abi::__cxa_demangle(typeid(f1).name(), nullptr, nullptr, nullptr);
  //如果在VisualStudio  cout << "type of f1:" << typeid(f1).name() << endl;
```

上面代码的输出为

```
rtn_f1 = 42
rtn_f2 = 10
rtn_f3 = 20
type of f1 : test_lambda_demo_1() : : {lambda()#1}
```

注意最后一行f1的类型就明白为什么接受lambda表达式的变量类型必须是auto(自动推导)

Lambda表达式

lambda捕获变量

lambda可以使用它所在函数的局部变量（包括它所在函数的形参），这个过程叫捕获。**捕获只限于局部非static变量**。

lambda可以直接使用它所在函数的局部static变量，以及在它所在函数之外声明的名字（例如全局的名字、所在文件的静态的名字），但这种情况不是捕获。

Lambda表达式

```
void test_lambda_demo_2(){
   //函数test_lambda_demo_2定义了局部变量 x,y
   int x = 10, y = 10;
   //现在在test_lambda_demo_2里定义lambda表达式
  //问题：lambda表达式除了自己的形参外，是否可以使用lambda表达式所位于的test_lambda_demo_2函数的
  //局部变量x、y呢？可以！lambda表达式使用所在函数中的局部变量叫做捕获变量，可以在[捕获列表]里指明要
  //捕获那些局部变量
 //lambda表达式要使用(捕获)所在函数test_lambda_demo_2的局部变量x，是值捕获
  auto f1 = [x] (int i) ->int {return x * i;}; //调用lambda表达式
   cout << "f1() = " << f1(2) << endl; //输出20
   ++x;  //函数test_lambda_demo_2定义了局部变量 x变了
   cout << "f1() = " << f1(2) << endl; //还是输出20，而不是22
}
```

为什么会这样？

lambda表达式捕获发生在lambda表达式被创建的时候（即定义lambda表达式的代码执行时），这个例子里是语句

```
auto f1 = [x] (int i) ->int {return x * i;};
```

这个时刻捕获x的值是10，而且是值捕获，因此x（值为10）被复制到lambda表达式的上下文里（更准确的说应该是闭包）。

因此++x没有改变已经创建好的lambda表达式闭包里的x，所以调用f1(2)时，函数体{ return x * i; }里的x还是10

顺便说一下，当lambda表达式被创建好以后，可以多次执行，例如f1(2),f1(3)，...

Lambda表达式

```
void test_lambda_demo_2(){
    int x = 10, y = 10;
/* 如果是引用捕获呢?
    lambda表达式捕获发生在lambda表达式被创建的时候（即定义lambda表达式的代码执行时），下面例子
        auto f2 = [&y] (int i) ->int {return y * i;};
     这个捕获是引用捕获，即y的引用复制到lambda表达式的上下文里（更准确的说应该是闭包）。因此++y以后再调用f2(2)时，函数体{ return y * i; }里的y变成了11
    */
    auto f2 = [&y] (int i) ->int {return y * i;}; //调用lambda表达式
    cout << "f2() = " << f2(2) << endl; //输出20
    ++y;
    cout << "f2() = " << f2(2) << endl; //输出22
    /* 可以捕获多个局部变量，这时在[捕获列表]里面多个名字用逗号分割例如 */
    auto f3 = [x, y] (int i)->int { return i * (x + y);};  //同时值捕获x和y
    auto f4 = [&x, &y] (int i)->int { return i * (x + y);}; //同时引用捕获x和y
    auto f5 = [x, &y] (int i)->int { return i * (x + y);}; //值捕获x，引用捕获y
}
```

Lambda表达式

```
/**
   隐式捕获
   除了在[捕获列表]显式列出lambda需要捕获的变量外，也可以让编译器根据lambda表达式函数体代码来推断要捕获哪些变量 .但为了指示编译器如何推断捕获列表，应该在[]里写＝或&，来告诉编译器是值捕获还是引用捕获
 */
void test_lambda_demo_3(){
    int x = 10, y = 10, z = 10;
    auto f1 = [=]()-> int{ return x + y + z;}; //隐式值捕获，根据代码可以推断出：值捕获了x , y, z三个变量
    cout << "f1() = " << f1() << endl; //输出30

    auto f2 = [&]()-> int{ return x + y + z;}; //隐式引用捕获，根据代码可以推断出：引用捕获了x ,y ,z三个变量
    cout << "f2() = " << f2() << endl; //输出30

    ++x;++y;++z;
    cout << "f1() = " << f1() << endl; //还是输出30
    cout << "f2() = " << f2() << endl; //输出33
}
```

Lambda表达式

```
/**
   隐式捕获
   除了在[捕获列表]显式列出lambda需要捕获的变量外，也可以让编译器根据lambda表达式函数体代码来推断要捕获哪些变量 .但为了指示编译器如何推断捕获列表，应该在[]里写＝或&，来告诉编译器是值捕获还是引用捕获
 */
void test_lambda_demo_3(){
   int x = 10, y = 10, z = 10;
/*
      可以混合使用隐式捕获和显式捕获
      当混合使用隐式捕获和显示捕获时，捕获列表的第一个元素必须是一个=或者&，指定隐式捕获方式；后面是显式捕获变量名列表，而且显式捕获的方式必须和隐式捕获不一样
      即：如果隐式捕获是引用捕获，则显式捕获必须用值捕获；如果隐式捕获是值捕获，则显式捕获必须用引用捕获；
    */
   auto f3 = [=,&z]()-> int{ return x + y + z;}; //隐式值捕获x , y，显式引用捕获变量z
   auto f4 = [&,x]()-> int{ return x + y + z;}; //隐式引用捕获y , z，显式值捕获变量x}
```

Lambda表达式

既然lambda表达式可以赋值给一个变量，那么lambda表达式可以作为函数的返回值

我们希望有这样一个函数：natural_number_generator(自然数产生器)，每次调用natural_number_generator会自动产生下一个自然数，

我们还希望有一个函数能帮我们产生这样的自然数产生器，这个函数比如叫natural_number_generator_factory（自然数发生器的制造工厂）

即：我们希望函数natural_number_generator_factory能返回另外一个函数natural_number_generator(自然数产生器)。我们可以也只能用lambda表达式

Lambda表达式

```
//函数natural_number_generator_factory是一个自然数发生器的工厂
//它要返回lambda表达式（自然数发生器）
//因此函数natural_number_generator_factory的返回类型为auto
auto natural_number_generator_factory(){
   int i = 0;

   //定义一个lambda表达式，其功能为自然数发生器，产生下一个自然数
   //特别说明：如果需要在lambda表达式函数体内修改通过值捕获方式捕获到的变量的值（如++i），则必须在（形参列表）后面加上mutable
   auto natural_number_generator = [i] () mutable  ->int { return ++i;};

   return natural_number_generator; //返回自然数发生器
}
```

Lambda表达式

```
void test_lambda_demo_4(){
    //调用natural_number_generator_factory()得到一个自然数产生器generator1
    auto generator1 = natural_number_generator_factory();
    cout << "generator1 generates ..." << endl;
    int n1 = generator1();    //调用generator1，得到下一个自然数，n1 = 1;
    cout << "n1 = " << n1 << endl;
    int n2 = generator1();    //调用generator1，得到下一个自然数，n2 = 2;
    cout << "n2 = " << n2 << endl;
    //可以一直这样产生下一个自然数

    //调用natural_number_generator_factory()得到另一个一个自然数产生器generator2
    auto generator2 = natural_number_generator_factory();
    cout << "generator2 generates ..." << endl;
    int m1 = generator2();    //调用generator2，得到下一个自然数，m1 = ?;
    cout << "m1 = " << m1 << endl;
    int m2 = generator2();    //调用generator2，得到下一个自然数，m2 = ?;
    cout << "m2 = " << m2 << endl;
    //可以一直这样产生下一个自然数
}
```

上面代码输出结果为

```
generator1 generates ...
n1 = 1
n2 = 2
generator2 generates ...
m1 = 1
m2 = 2
```

Lambda表达式

为什么第二个自然数产生器generator2的输出也是从1开始而不是从3开始

因为generator1捕获的i和generator2捕获的i是不同的i，而且捕获时i的值都是0，前面讲过：

lambda表达式捕获发生在lambda表达式被创建的时候（即定义lambda表达式的代码执行时）

```
auto natural_number_generator = [i] () mutable  ->int { return ++i;};
```

Lambda表达式+捕获的自由变量i叫闭包。闭包(Closure)并不是一个新鲜的概念，很多函数式语言中都使用了闭包。例如在JavaScript中，当你在内嵌函数中使用外部函数作用域内的变量时，就是使用了闭包。用一个常用的类比来解释闭包和类（Class）的关系：类是带函数的数据，闭包是带数据的函数。

闭包的本质：代码块+上下文

闭包中使用的自由变量有一个特性，就是它们不在父函数返回时释放，而是随着闭包生命周期的结束而结束。generator1和generator2就是二个不同的闭包，它们分别使用了相互独立的变量i（在generator2的i自增1的时候，generator2的i并不受影响，反之亦然），只要generator2或generator2这两个变量没有被回收，他们各自的自由变量i就不会被释放。

Lambda表达式

- Lambda表达式是C++引入的一种匿名函数。存储Lambda表达式的变量被编译为临时类的对象。
- 该临时类变量的对象被构造时，此时Lambda表达式被计算。
- 若未定义存储该临时类对象的变量，则称该Lambda表达式没被计算。
- Lambda表达式的声明格式为
  - [捕获列表](形参列表) mutable 异常说明 -> 返回类型 {函数体}
  - 例如，auto f = [ ](int x=1)->int { return x; };
- 捕获列表的参数用于捕获Lambda表达式的外部变量。
- 临时类重载operator( ) (…) ；当调用f(3)时，等价于f.operator( )(3)
  - Lambda表达式被编译成函数对象，函数对象名为f

重载函数调用操作符（ ）

为多目运算符

```
int sum(int x, int y) { return x + y;}
int s = sum (1,2); //三个操作数sum, 1, 2
//第一个操作数为函数名，这里把函数名看做函数对象
```

()只能通过类的普通成员函数重载，这意味着()第一个操作数必须是this指针指向的类对象，这样的对象称为函数对象。

```
class AbsInt{
public:
   //语法：returnType operator( ) (参数列表)
   //调用：objectofAbsInt(实参)，类AbsInt的对象称为函数对象
   int operator()( int val) { return val > 0? val:-val; }
} absInt;
int i = absInt(-1); //函数对象可以作为函数的参数，这意味着我们可以将函数当做对象传递. 在模板编程里广泛使用。在C+11里，还可以用lamda表达式
```

函数指针也可以作为函数的参数，但函数指针主要的缺点是：
被函数指针指向的函数是无法内联的。而函数对象则没这个问题。

```
void  main() {
   int m = 1, p = 3;
   const int n = 2, q = 4;
   auto  g = [m, n, &p, &q](int x) ->int { p++; /*错： m++; n++; q++;*/; return m+n+x; };
   int z = g(0);                  //m=1,p=4,z=3, 等价于g.operator()(0)
   z = g(0);          //m=1,p=5,z=3
}
//auto g = lambda表达式这条语句，被编译为匿名类对象g，生成的匿名类及其函数如下
class 匿名类{
    const int  m, n;          //Lambda表达式无mutable时，非&捕获的可写外部变量m为const成员
    int &p;                   //&捕获的外部变量保持其原有类型int+&
    const int &q;             //&捕获的外部变量保持其原有类型const int+&
public:
    构造函数(int m, const int n, int &p, const int &q) : m(m), n(n), p(p), q(q){ }
    int operator()(int x) { p++; return m + n + x; }
}g(m, n, p, q);    //调用构造函数初始化匿名类对象g
```

m, n为值捕获;p，q为引用捕获

当捕获列表以值的方式传递后，lambda表达式不能修改这个变量的值，只能使用

引用捕获的非const值可以修改

```
void  main() {
   int m = 1, p = 3;
   const int n = 2, q = 4;
   auto  f = [m, n, &p, &q](int x)mutable->int { p++; /*错:n++;q++;*/ return m++ + x; };
   int z = f(0);                  //m=1,p=4, z=1, 等价于f.operator()(0);
   z = f(0);          //m=1,p=5, z=2 （匿名类里的m变成2）
}
//为匿名类对象f生成的匿名类及其函数如下
class 匿名类{
    mutable int  m;          //Lambda表达式有mutable时， 非&捕获的可写外部变量m为mutable成员
    const int n;             //保持n的原有类型为const int
    int &p;                  //保持p的原有类型为int+&
    const int &q;            //保持q的原有类型为const int+&
public:
    构造函数(int m, const int n, int &p, const int&q):m(m), n(n), p(p),q(q){ }
     int operator()(int x) {p++; /*错:n++;q++;*/ return m++ + x; }
}f(m, n, p, q);              //调用构造函数初始化匿名类对象f
```

m, n为值捕获;p，q为引用捕获

lambda表达式有mutable时，值捕获的非const值变成mutable，即可以修改

Lambda表达式

◆捕获列表的参数

- Lambda表达式的外部变量不能是全局变量或static定义的变量。
- Lambda表达式的外部变量不能是类的成员。
- Lambda表达式的外部变量可以是函数参数或函数定义的局部自动变量。
- 出现“&变量名”表示引用捕获外部变量，[&]表示引用捕获所有函数参数或函数定义的局部自动变量。其可写特性同被捕获的外部变量一致。
- 出现“=变量名”表示值捕获外部变量的值（值参传递），[=]表示捕获所有函数参数或函数定义的局部自动变量的值。
- 参数表后有mutable表示在Lambda表达式中可以修改“值参传递的可写变量的值”，但调用后不影响Lambda表达式捕获的对应该可写外部变量的值（值捕获）。

```
//例12.21
int main( ) {                                                    准函数：函数对象不包含其他成员，除了operator()(...)
   int a = 0;
   auto f = [](int x = 1)->int { return x; };                    //捕获列表为空，对象f当准函数用
   auto g = [](int x)throw(int) ->int { return x; };             //g同上：匿名函数可能抛出异常， throw(int) 是异常声明
   int(*h)(int) = [](int x)->int { return x * x; };              //捕获列表为空，可以用函数指针h指向“准函数”
   h = f;                                                        //正确：f的Lambda表达式捕获列表为空，f当准函数使用
   auto m = [a, f](int x)->int { return x * f(x); };             //m是函数对象：捕获a初始化函数对象实例成员
   //int(*k)(int)=[a](int x)->int{return x;};      //函数指针k不能指向函数对象（捕获列表非空,函数对象包含其他成员）
   //h = m;                                        //错误：m的Lambda表达式捕获列表非空，m是函数对象
   //printf(typeid([ ](int x)->int{return x;}).name( ));     //错：临时Lambda表达式未计算，无类型
   printf("%s\n", typeid(f).name());                           //输出class <lambda_...>
   printf("%s\n", typeid(f(3)).name());                        //输出int，使用实参值调用x=3
   printf(“%s\n”, typeid(f.operator( )()).name()); //输出int，使用默认值调用x=1,f是函数对象，显式调用operator()()
   printf("%s\n", typeid(f.operator( )).name());               //输出int __cdecl(int)
   printf("%s\n", typeid(g.operator( )).name());               //输出int __cdecl(int)
   printf("%s\n", typeid(h).name());                           //输出int (__cdecl*)(int)
   printf("%s\n", typeid(m).name());                           //输出class <lambda_...>
   return f(3) + g(3) + (*h)(3);                               //用对象f、g调用Lambda表达式
}
```

第12章 类型解析、转换与推导

◆捕获列表的参数

- 捕获列表的参数可以出现this，但实例函数成员中的Lambda表达式默认捕获this，而静态函数成员中的Lambda表达式不能捕获this。
- 由于this不是变量名或参数名，故不能使用“&this”或者“=this”。
- Lambda表达式的捕获列表非空，倾向于用作“准对象”，否则倾向于用作“准函数”。
- 早期编译器实例函数成员中的Lambda表达式默认捕获this，故它是一个准对象。但vs2019必须明确捕获this才能访问实例数据成员。
- 只有作为“准函数”才能获得其函数入口地址。

第12章 类型解析、转换与推导

```
int m = 7;                          //全局变量m不用被捕获即可被Lambda表达式使用
static int n = 8;                   //模块变量n不用被捕获即可被Lambda表达式使用
class A {
   int x;                           //由于this默认被捕获，故可访问实例数据成员A::x
   static int y;                    //静态数据成员A::y不用捕获即可被Lambda表达式使用
public:
   A(int m): x(m) { }
   void f(int &a) {                 //实例函数成员f()有隐含参数this
      int b = 0;
      static int c=0;               //静态变量c不用被捕获即可被Lambda表达式使用
      auto h = [&, a, b](int u)mutable->int{    //this默认被捕获，创建对象h，a,b值捕获，其它引用捕获
         a++;                       //f()的参数a被值捕获并传给h的实例成员a：a++不改变f()的参数a的值
         b++;                       //f()的局部变量b被捕获并传给h实例成员b：b++不改局部变量b的值
         c++;                       //f()的静态变量c可直接使用，c++改变f()的静态变量c的值
         y=x+m+n+u+c;               //this默认被捕获：可访问实例数据成员x
         return a;
      };
```

第12章 类型解析、转换与推导

```
    h(a + 2);                    //实参a+2值参传递给形参u，调用h.operator( )(a+2)
}
static void g(int &a){           //静态函数成员g()没有this
    int b = 0;
    static int c = 0;            //静态变量c不用被捕获即可被Lambda表达式使用
    auto h = [&a, b](int u)mutable->int{      //没有this被捕获，创建函数对象h
        a++;                     //g()的参数a被捕获并传给h的引用实例成员a：a++改变参数a的值，a是引用捕获
        b++;                     //g()的局部变量b被捕获传给h实例成员b：b++不改局部变量b的值，b是值捕获
        c++;                     //g()的静态变量c可直接使用，c++改变g()的静态变量c的值
        y=m+n+u+c;               //没有捕获this，不可访问实例数据成员A::x
        return a;
    };
    auto k = [ ](int u) ->int { return u; }; //静态成员函数g没有this，没有this被捕获，创建对象k
    auto p = k;                              //p的类型为class<lambda…>
    int (*q)(int) = k;                       //问题：若将红色部分放入void f(int &a)中，如何？ int (*q)(int) = k;就不成
立
```

第12章 类型解析、转换与推导

```
        h(a + 2);                    //实参a+2值参传递给h形参u
    }
}a(10);
int A::y = 0;                        //静态数据成员必须初始化
void  main( ) {
    int p = 2;
    a.f(p);                          //p=2，a.x=10，A::y=30   x（10）+m（7）+n（8）+u（4）+c（1）
    a.f(p);                          //p=2，a.x=10，A::y=31
    A::g(p);                         //p=3，a.x=10，A::y=20
    A::g(p);                         //p=4，a.x=10，A::y=22
}
```