Programming in Modern C++

Module M52: C++11 and beyond: General Features: Part 7: Lambda in C++/1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Learnt how Rvalue Reference works as a Universal Reference under template type deduction
- Understood the problem of forwarding of parameters under template type deduction and its solution using Universal Reference and `std::forward`
- Learnt the implementation of `std::forward`
- Understood how Move works as an optimization of Copy

Module Objectives

- To understand λ expressions (unnamed function objects) in C++
  - Closure Objects
  - Parameters
  - Capture

Module Outline

1. λ in C++11, C++14, C++17, C++20
    - Syntax and Semantics
    - Closure Object
        - Lambdas vs. Closures
        - First Class Object
        - Anatomy
    - Parameters
    - Capture
        - By Reference [&]
        - By Value [=]
        - Mutable
        - Restrictions
        - Practice Examples
2. Module Summary

λ in C++11, C++14, C++17, C++20

Source:

- Lambdas, isocpp.org
- Scott Meyers on C++
- Lambda capture, cppreference.com
- Lambdas: From C++11 to C++20, Part 1 and Lambdas: From C++11 to C++20, Part 2, cppstories.com, 2019
- Lambdas: Smart Pointers, Jim Fix, Reed College

λ in C++11, C++14, C++17, C++20

λ in C++: Closure Object

- A *λ expression* is a mechanism for specifying a *function object* or *functor* (Recall **Module 40**)
- The primary use for a λ is to specify a simple action to be performed by some function
- For example, consider a remainder operation `rem` that computes m % n, that is, m modulo n. It has type `int -> int`. To write `rem` in C++, we define a function / functor:

Function	Functor

`int n = 7;`
`int rem(int m) // Function`
`{ return m % n; }`
`// Uses n in context`
	`int n = 7;`
	`struct remainder {  // Functor`
	`int mod;  // State`
	`remainder(int n): mod(n) { } // Ctor (n from context)`
	`int operator()(int m)  // Function call operator`
	`{ return m % mod; }  // Body`
	`};`

`rem(23); // 2`	`struct remainder rem(n);` `rem(23); // 2`

λ: `auto rem = [n](int m) -> int { return m % n; } //` **Captures** `n` from context
`rem(23); // 2`

- Note that `[n]` **Captures** `n` from context to close `rem` and create the **Closure Object** in C++

C++ λ's

- C++11 introduced λ's as syntactically lightweight way to define functions on-the-fly
- λ's can *capture* (or close over) variables from the surrounding scope – by *value* or by *reference*
- First consider callable things that do not capture any variables. C++ offers three alternatives:
  - plain functions (All versions of C & C++)
  - functor classes (C++03 onwards), and
  - lambdas (C++11 onwards)

```
#include <iostream> // cout
using namespace std;

int function (int a) { return a + 3; }
class Functor { public: int operator()(int a) { return a + 3; } };
auto lambda = [] (int a) { return a + 3; };

int main() { Functor functor;
    cout << function(5) << ' ' << functor(5) << ' ' << lambda(5) << endl;
}
8 8 8
```

- For plain functions that capture no variables, lambdas and functors behave the same

C++ λ Syntax and Semantics

- A λ expression consists of the following:

```
[capture list] (parameter list) -> return-type { function body }
```

- The capture list and parameter list can be empty, so the following is a valid λ:

```
[]() { cout << "Hello, world!" << endl; }
```

- *Parameter list* is a sequence of parameter types and variable names as for an ordinary function
- *Function body* is like an ordinary function body
- If the *function body* has only one return statement (which is very common), the *return type* is assumed to be the same as the type of the value being returned
- If there is *no return statement* in the function body, the return type is assumed to be `void`
  - Below λ has return type `void` – can be called without any use of the return value:

    ```
    []() { cout << "Hello from trivial lambda!" << endl; } ();
    ```

  - However, trying to use the return type of the call is an error:

    ```
    cout << []() { cout << "Hello from trivial lambda!" << endl; } () << endl;
    ```

C++ λ Syntax and Semantics

- Below λ returns a `bool` value which is true if the first param is half of the second. The compiler knows the return type as `bool` from the return statement:

```
if ([](int i, int j) { return 2 * i == j; } (12, 24))
    cout << "It's true!";
else
    cout << "It's false!";
```

- To specify return type:

```
cout << "This lambda returns " << [](int x, int y) -> int {
        if(x > 5) return x + y;
        else
            if (y < 2) return x - y; else return x * y;
    } (4, 3) << endl;
```

- Below λ, returns an `int`, though the return statement provides a `double`:

```
cout << "This lambda returns " <<
    [](double x, double y) -> int { return x + y; } (3.14, 2.7) << endl;
```

The output is `"This lambda returns 5"`

C++ λ: Syntax and Semantics

- Below λ *captures* `n` *by value* to compute the value of remainder of `m`:

```
int n = 7;
auto rem = [n](int m) -> int { return m % n; };
```

  - `n` is captured by [n] (value – a copy is made from the context) at the time of constructing the closure object. Hence `n` *must be initialized before the construction* of the closure
  - The *value of* `n` *cannot be changed* within the λ (for immutable λ's)
  - The changes to `n` after the construction of the closure object are not reflected
- Below λ *captures* `s` *by reference* to accumulate the value of `m`:

```
int s = 0;
auto acc = [&s](int m){ s += m; };
```

  - `s` is captured by [&s] (reference – a reference is set to the context) at the time of constructing the closure object. Hence it is *optional to initialize* `s` *before the construction* of the closure. However, it must be initialized before the use of the closure
  - The *value of* `s` *can be changed* within the λ
  - The changes to `s` after the construction of the closure object will be reflected

Lambdas vs. Closures

- **Closure**
  - A *closure* (*lexical* / *function closure*), is a technique for implementing *lexically scoped name binding* in a language with first-class functions
  - Operationally, a closure is a *record storing a function* together with an *environment*
  - The *environment* is a mapping associating (*binding*) each *free* variable of the function with the *value* or *reference* to which the name was bound when the closure was created
  - *Unlike a plain function, a closure allows the function to access captured variables through the closure's copies of their values or references, even in its invocations outside their scope*
- **Lambdas vs. Closures** (From Lambdas vs. Closures by Scott Meyers, 2013)
  - A λ expression `auto f = [&](int x, int y){ return fudgeFactor * (x + y); };` *exists only in a program's source code*. A lambda *does not exist at runtime*
  - The runtime effect of a λ expression is the generation of an object, called *closure*
  - Note that *f* is not the closure, it is a *copy of the closure*. The *actual closure object is a temporary* that's typically destroyed at the end of the statement
  - Each λ expression causes a unique class to be generated (during compilation) and also causes an object of that class type – a closure – to be created (at runtime)
  - Hence, closures *are to* lambdas *as* objects *are to* classes

Closure Objects: Implementing λ's

- A **λ-expression** generates a **Closure Object** at *run-time*
- A closure object is *temporary*
- A closure object is *unnamed*
- For a λ-expression, the compiler creates a *functor class* with:
  - *data members*:
    - a value member each for each value capture
    - a reference member each for each reference capture
  - a *constructor* with the captured variables as parameters
    - a value parameter each for each value capture
    - a reference parameter each for each reference capture
  - a *public inline const function call* `operator()` with the parameters of the lambda as parameters, generated from the body of the lambda
  - *copy constructor*, *copy assignment operator*, and *destructor*
- A *closure object* is constructed as an instance of this class and *behaves like a function object*
- A λ-expression without any capture behaves like a function pointer

Source: C++ Lambda Under the Hood, 2019

Closure Objects: Implementing λ's: Example


```cpp
#include <iostream> // lambda & closure object
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;     // for ref. capture init. opt.

    auto check = [val, &ref](int param){
        cout << "val = " << val << ", ";
        cout << "ref = " << ref << ", ";
        cout << "param = " << param << endl;
    };
    // lambda to show captured values
    // constructed with value capture of val
    // and reference capture of ref
    // Also, has a parameter param



    ref = 2;  // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3;  // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4;  // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

```cpp
#include <iostream> // Possible functor by compiler
using namespace std;
int main() {
    int val = 0; // for value capture init. must
    int ref;     // for ref. capture init. opt.

    struct check_f { // functor to show captured values
        int val_f;   // value member for value capture
        int& ref_f;  // ref. member for ref. capture
        check_f(int v, int& r):    // Ctor with
            val_f(v), ref_f(r) { } // value & ref params
        void operator()(int param) const { // param
            cout << "val = " << val_f << ", ";
            cout << "ref = " << ref_f << ", ";
            cout << "param = " << param << endl;
        };
    };
    auto check = check_f(val, ref); // Instantiation

    ref = 2;  // init. will be reflected
    check(5); // val = 0, ref = 2, param = 5
    val = 3;  // change will not be reflected
    check(5); // val = 0, ref = 2, param = 5
    ref = 4;  // change will be reflected
    check(5); // val = 0, ref = 4, param = 5
}
```

Closure Objects: First Class Objects (FCOs)

Module M52

Partha Pratim Das


```
struct trace { int i;
    trace() : i(0) { std::cout << "construct\n"; }
    trace(trace const &) { std::cout << "copy construct\n"; }
    ~trace() { std::cout << "destroy\n"; }
    trace& operator=(trace&) { std::cout << "assign\n"; return *this; }
};
```

Code Snippets	Outputs

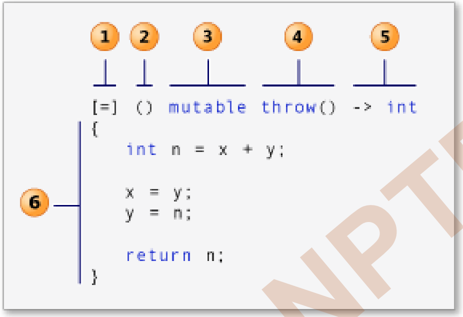
`{ trace t; // t not used so not captured`
`    int i = 8;`
`    auto m1 = [=](){ return i / 2; };`
`}`
	construct
	destroy

`{ trace t; // capture t by value`
`    auto m1 = [=](){ int i = t.i; };`
`    std::cout << "-- make copy --" << std::endl;`
`    auto m2 = m1;`
`}`
	construct
	copy construct
	- make copy -
	copy construct
	destroy
	destroy
	destroy

`{ trace t; // capture t by reference`
`    auto m1 = [&](){ int i = t.i; };`
`    std::cout << "-- make copy --" << std::endl;`
`    auto m2 = m1;`
`}`
	construct
	-- make copy --
	destroy

Closure object has implicitly-declared copy constructor / destructor

Closure Objects: Anatomy



[1] **Capture Clause** (*introducer*)
[2] **Parameter List** (Opt.) (*declarator*)
[3] **Mutable Specs.** (Opt.)
[4] **Exception Specs.** (Opt.)
[5] **(Trailing) Return Type** (Opt.)
[6] λ **body**

λ *Expression*:: $\mathcal{E} \vdash my\_mod : Int,\ \lambda(v : Int).\ v \ \% \ my\_mod : Int$

Closure Object:: `[my_mod](int v) -> int { return v % my_mod; }`

- *Introducer*: `[my_mod]`
- *Capture*: `my_mod`
- *Parameters*: `(int v)`
- *Declarator*: `(int v) -> int`
- *Mutable Spec*: Skipped
- *Exception Spec*: Skipped
- *Return Type*: `-> int`
- λ *Body*: `{ return v % my_mod; }`

Closure Objects: Parameters

Parameter Passing	Remarks
`[](){ std::cout << "foo" << std::endl; }();`	`foo`
`[](int v){ std::cout << v << "*6=" << v*6 << std::endl;}(7);`	`7*6=42`

`int i = 7;`
`[](int & v){ v *= 6; } (i);`
`std::cout << "the correct value is: " << i << std::endl;`
	`the correct value is: 42`

`int j = 7;`
`[](int const & v){ v *= 6; } (j);`
`std::cout << "the correct value is: " << j << std::endl;`
	`// error:`
	`// assignment of read-only reference 'v'`

`int j = 7;` `[](int v){ v *= 6; std::cout << "v: " << v << std::endl;}(j);`	`v: 42`

`int j = 7;`
`[](int & v, int j) { v *= j; } (j, 6);`
`std::cout << "j: " << j << std::endl;`
	`// lambda parameters do not affect`
	`// the namespace`
	`j: 42`

`[] std::cout << "foo" << std::endl; ();` is same as `[]() std::cout << "foo" << std::endl; ();`	`// lambda expression without a` `// declarator acts as if it were ()`

Closure Objects: Capture

- The *captures* is a comma-separated list of zero or more captures, optionally with default
- The capture list defines the outside variables that are accessible from the λ function body
- The only capture defaults are
  - `[&]` (implicitly capture the used automatic variables *by reference*) and
  - `[=]` (implicitly capture the used automatic variables *by copy / value*)
- The *current object* (`*this`) can be *implicitly captured* if either capture default is present
- If implicitly captured, it is always captured by reference, even for `[=]`. Deprecated since C++20

Capture	Meaning	C++
`identifier`	simple by-copy capture	C++11
`identifier ...`	simple by-copy capture that is a pack expansion	C++11
`identifier init`	by-copy capture with an initializer	C++14
`& identifier`	simple by-reference capture	C++11
`& identifier ...`	simple by-reference capture that is a pack expansion	C++11
`& identifier init`	by-reference capture with an initializer	C++14
`this`	simple by-reference capture of the current object	C++11
`*this`	simple by-copy capture of the current object	C++17
`... identifier init`	by-copy capture with an initializer that is a pack expansion	C++20
`& ... identifier init`	by-reference capture with an initializer that is a pack expansion	C++20

Source: Lambda capture, cppreference.com

Closure Objects: Capture

- Optional captures of λ expressions are (C++11):
  - *Default all by reference*

    `[&](){ ... }`
  - *Default all by value*

    `[=](){ ... }`
  - *List of specific identifier(s) by value or reference and/or this*

    `[identifier](){ ... }`
    `[&identifier](){ ... }`
    `[foo,&bar,gorp](){ ... }`
  - *Default and specific identifiers and/or this*

    `[&,identifier](){ ... }`
    `[=,&identifier](){ ... }`

Source: Lambda capture, cppreference.com

Closure Objects: Capture: Simple Examples

```
int x = 2, y = 3; // Global Context
const auto l0 = []()  { return 1; };                  // No capture
typedef int (*l1) (int);                              // Function pointer
const l1 f = [](int i){ return i; };                  // Converts to a func. ptr. w/o capture
const auto l2 = [=]() { return x; };                  // All by value (copy)
const auto l3 = [&]() { return y; };                  // All by ref
const auto l4 = [x]() { return x; };                  // Only x by value (copy)
const auto lx = [=x](){ return x; };                  // wrong syntax, no need for
                                                      // = to copy x explicitly
const auto l5 = [&y]()    { return y; };              // Only y by ref
const auto l6 = [x, &y](){ return x * y; };           // x by value and y by ref
const auto l7 = [=, &x](){ return x + y; };           // All by value except x
                                                      // which is by ref
const auto l8 = [&, y]() { return x - y; };           // All by ref except y which
                                                      // is by value
const auto l9 = [this]() { }                          // capture this pointer
const auto la = [*this](){ }                          // capture a copy of *this
                                                      // since C++17
```

[&]()->rt{...}: Capture

- *Capture default all by reference*

```
int total_elements = 1;
for_each(cardinal.begin(), cardinal.end(),
    [&](int i) { total_elements *= i; } ); // total_elements
                                           // can be changed
```

- **Errors**

```
[=](int i) { total_elements *= i; } );

error C3491: 'total_elements': a by-value capture cannot be modified
in a non-mutable lambda

[](int i) { total_elements *= i; } );

error C3493: 'total_elements' cannot be implicitly captured because
no default capture mode has been specified
```

[&] ()->rt{...}: Capture: Scope & Lifetime

Wrong Capture by Reference

- Closures may outlive their creating function

```
std::function<bool(int)>
    returnClosure(int a) { // returns bool
    int b, c;
    ...
    // won't compile, but assume it would
    return [](int x)
        { return a*x*x + b*x + c == 0; };
}

// f is essentially a copy of
// lambda's closure
auto f = returnClosure(10);
...
if (f(22))  // invoke the closure
```

- What are the values of a, b, c in the call?
  - returnClosure no longer active!
- Non-static locals referenceable only if **captured**

Correct Capture by Reference

- This version has no such problem

```
int a; // now at global or namespace scope
std::function<bool(int)>
    returnClosure() {
    static int b, c; // now static ...

    // now compiles
    return [](int x)
        { return a*x*x + b*x + c == 0; };
}

// as before

auto f = returnClosure();
...
if (f(22))   // as before
```

- a, b, c outlive returnClosure's invocation
- Variables of static storage duration always referenceable

[&]()->rt{...}: Capture


```
// #include <iostream>, <algorithm>, <vector>
template< typename T >
void fill(std::vector<int>& v, T done) { int i = 0; while (!done()) { v.push_back(i++); } }
int main() {
    std::vector<int> stuff; // Fill the vector with 0, 1, 2, ... 7
    fill(stuff, [&]{ return stuff.size() >= 8; }); // [=] compiles but is infinite loop
    for(auto it = stuff.begin(); it != stuff.end(); ++it) std::cout << *it << '  ';
    std::cout << std::endl;

    std::vector<int> myvec; // Fill the vector with 0, 1, 2, ... till the sum exceeds 10
    fill(myvec, [&]{ int sum = 0; // [=] compiles but is infinite loop
        std::for_each(myvec.begin(), myvec.end(), [&](int i){ sum += i; });
                                  // [=] is error: assignment of read-only variable 'sum'
        return sum >= 10;
    }
    );
    for(auto it = myvec.begin(); it != myvec.end(); ++it) std::cout << *it << '  ';
    std::cout << std::endl;
}
0 1 2 3 4 5 6 7
0 1 2 3 4
```

[=]()->rt{...}: Capture

- *Capture default all by value*

```
std::vector<int> in, out(10);
for (int i = 0; i < 10; ++i)
    in.push_back(i);

int my_mod = 3;
std::transform(in.begin(), in.end(), out.begin(),
                            [=](int v) { return v % my_mod; });

for (auto it = out.begin(); it != out.end(); ++it)
    std::cout << *it << ' ';
std::cout << std::endl;
```

```
0 1 2 0 1 2 0 1 2 0
```

[=]()->rt{...}: Capture: Mutable

- Consider

```
int h = 10;
auto two_h = [=] () { h *= 2; return h; };
std::cout << "2h:" << two_h() << " h:" << h << std::endl;

error C3491: 'h': a by-value capture cannot be modified in a non-mutable lambda
```

- λ closure objects have a *public inline function call operator* that:
  - Matches the parameters of the lambda expression
  - Matches the return type of the lambda expression
  - Is declared const
- *Make mutable*

```
int h = 10;
auto two_h = [=] () mutable { h *= 2; return h; };
std::cout << "2h:" << two_h() << " h:" << h << std::endl;

2h:20 h:10
```

[=] ()->rt{...}: Capture: Mutable

```
int h = 10;
auto f = [=] () mutable { h *= 2; return h; }; // h changes locally
std::cout << "2h:" << f() << std::endl;
std::cout << " h:" << h << std::endl;
```

```
2h:20
 h:10
```

```
int h = 10;
auto g = [&] () { h *= 2; return h; }; // h changes globally
std::cout << "2h:" << g() << std::endl;
std::cout << " h:" << h << std::endl;
```

```
2h:20
 h:20
```

[=]()->rt{...}: Capture: Mutable

```
int i = 1, j = 2, k = 3; // Global i, j, k
auto f = [i, &j, &k]() mutable
{
    auto m = [&i, j, &k]() mutable
    {
        i = 4; // Local i of f
        j = 5; // Local j of m
        k = 6; // Global k
    };
    m();
    std::cout << i << j << k; // Local i of f, Global j, Global k
};
f();
std::cout << " : " << i << j << k; // Global i, j, k
```

426 : 126

[=] ()->rt{...}: Capture: Mutable

- Will this compile? If so, what is the result?

```
struct foo {
    foo() : i(0) { }
    void amazing(){ [=]{ i = 8; }(); } // i is captured by value
                                        // Can it be changed without mutable?
    int i;
};
foo f;
f.amazing();
std::cout << "f.i : " << f.i;

Output: f.i : 8
```

- **this** implicitly captured
- `i` actually is `this->i` which can be written from a member function as a data member. So no **mutable** is required

Capture: Restrictions

- Capture restrictions
  - Identifiers must only be listed once

```
[i,j,&z](){...} // Okay
[&a,b](){...}   // Okay
[z,&i,z](){...} // Bad, z listed twice
```

  - Default by value, explicit identifiers by reference

```
[=,&j,&z](){...} // Okay
[=,this](){...}  // Bad, no this with default =
[=,&i,z](){...}  // Bad, z by value
```

  - Default by reference, explicit identifiers by value

```
[&,j,z](){...}  // Okay
[&,this](){...} // Okay
[&,i,&z](){...} // Bad, z by reference
```

- Scope of Capture
  - Captured entity must be defined or captured in the immediate enclosing lambda expression or function

Closure Object: Capture: Mixed Examples

```
class A { std::vector<int> values; int m_;
public: A(int mod) : m_(mod) { }
    A& put(int v) { values.push_back(v); return *this; }
    int extras() { int count = 0;
        std::for_each(values.begin(), values.end(),
            [=, &count](int v){ count += v % m_; });
        return count;
    }
};
A g(4);
g.put(3).put(7).put(8);
std::cout << "extras: " << g.extras();
```

```
extras: 6
```

- Capture default by value
- Capture `count` by reference, accumulate, return
- How do we get `m_`?
- Implicit capture of '`this`' by value

Closure Objects: Capture: Mixed Examples

Capture	Remarks
`int i = 8; // Global context for all lambda's`	

`{ int j = 2; auto f = [=]{ std::cout << i / j; };`
`    f();`
`}`
	4

`auto f = [=](){ int j = 2; auto m = [=]{ std::cout << i / j; };`
`    m(); };`
`f();`
	4

`auto f = [i](){ int j = 2; auto m = [=]{ std::cout << i / j; };`
`    m(); };`
`f();`
	4

`auto f = [](){ int j = 2; auto m = [=]{ std::cout << i / j; };`
`    m(); };`
`f();`
	`// Error C3493: 'i' cannot be implicitly`
	`// captured because no default capture`
	`// mode has been specified`

`auto f = [=](){ int j = 2; auto m = [&]{ i /= j; }; m();`
`    std::cout << "inner: " << i; };`
`f(); std::cout << " outer: " << i;`
	`// Error C3491: 'i': a by-value capture`
	`// cannot be modified in a non-mutable`
	`// lambda`

`auto f = [i]() mutable { int j = 2;`
`    auto m = [&i, j]() mutable { i /= j; }; m();`
`    std::cout << "inner: " << i; };`
`f(); std::cout << " outer: " << i;`
	`inner: 4`
	`outer: 8`

Module Summary

- Understood λ expressions (unnamed function objects) in C++ with
  - Closure Objects
  - Parameters
  - Capture