Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

Principles of Programming Languages

Module M06: λ in C++

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

February 14 & 16, 2022

Table of Contents

1. Functors in C++
   - Callable Entities
   - Function Pointers
     - Replace Switch / IF Statements
     - Late Binding
     - Virtual Function
     - Callback
     - Issues
     - First-Class Objects
   - Basic Functor
     - Elementary Example
     - Functors in STL
2. λ in C++11, C++14, C++17, C++20
   - Closure Object
   - Parameters
   - Capture
   - Examples
   - Curry Function
3. Generic λ in C++14

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Functors in C++

Functors in C++

Source: Scott Meyers on C++

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

Callable Entities in C / C++

- A Callable Entity is an object that
  - Can be called using the function call syntax
  - Supports operator()
- Such objects are often called
  - A Function Object or
  - A Functor

Some authors do distinguish between Callable Entities, Function Objects and Functors.

Several Callable Entities C++

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

- Function-like Macros
- C Functions (Global or in Namespace)
- Member Functions
  - Static
  - Non-Static
- Pointers to Functions
  - C Functions
  - Member Functions (static Non-Static)
- References to functions: Acts like const pointers to functions
- Functors: Objects that define operator()

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers

- Points to the address of a function
  - Ordinary C functions
  - Static C++ member functions
  - Non-static C++ member functions
- Points to a function with a specific signature
  - List of Calling Parameter Types
  - Return-Type
  - Calling Convention

Function Pointers in C

- Define a Function Pointer

```
int (*pt2Function) (int, char, char);
```

- Calling Convention

```
int DoIt (int a, char b, char c);
int DoIt (int a, char b, char c) { printf ("DoIt\n"); return a+b+c; }
```

- Assign Address to a Function Pointer

```
pt2Function = &DoIt; // OR
pt2Function = DoIt;
```

- Compare Function Pointers

```
if (pt2Function == &DoIt) { printf ("pointer points to DoIt\n"); }
```

- Call the Function pointed by the Function Pointer

```
int result = (*pt2Function) (12, 'a', 'b');
```

Function Pointers in C

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

Direct Function Pointer

```
#include <stdio.h>

// Function pointer variable
int (*pt2Function) (int, char, char);

int DoIt(int a, char b, char c);

int main() { pt2Function = DoIt; // &DoIt

    // Paran's needed as operator() has higher
    // precedence than operator*
    int result = (*pt2Function)(12, 'a', 'b');

    printf("%d", result);
}

int DoIt(int a, char b, char c) {
    printf ("DoIt\n"); return a + b + c;
}
---
DoIt
207
```

Using typedef

```
#include <stdio.h>

// Function pointer type alias
typedef int (*pt2Function) (int, char, char);

int DoIt(int a, char b, char c);

int main() { pt2Function f = &DoIt; // DoIt



    int result = f(12, 'a', 'b');

    printf("%d", result);
}

int DoIt(int a, char b, char c) {
    printf ("DoIt\n"); return a + b + c;
}
---
DoIt
207
```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Reference In C++

- Define a Function Pointer

```
int (A::*pt2Member)(float, char, char);
```

- Calling Convention

```
class A {
    int DoIt (float a, char b, char c) {
        cout << "A::DoIt" << endl; return a+b+c;
    }
};
```

- Assign Address to a Function Pointer

```
pt2Member = &A::DoIt;
```

- Compare Function Pointers

```
if (pt2Member == &A::DoIt) { cout <<"pointer points to A::DoIt" << endl; }
```

- Call the Function pointed by the Function Pointer

```
int result = (*this.*pt2Member)(12, 'a', 'b');
```

Function Pointer: Operations

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

- Assign an Address to a Function Pointer
- Compare two Function Pointers
- Call a Function using a Function Pointer
- Pass a Function Pointer as an Argument
- Return a Function Pointer
- Arrays of Function Pointers

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointer: Programming Techniques

- Replacing switch/if-statements
- Realizing user-defined late-binding, or
  - Functions in Dynamically Loaded Libraries
  - Virtual Functions
- Implementing callbacks.

Function Pointers – Replace Switch/ IF Statements

Solution Using switch

```
#include<iostream>
using namespace std;
// The four arithmetic operations
float Plus (float a, float b) { return a+b ; }
float Minus (float a, float b) { return a-b ; }
float Multiply(float a, float b) { return a*b; }
float Divide (float a, float b) { return a/b ; }
void Switch(float a, float b, char opCode) {
    float result;
    Switch(opCode) { // execute operation
      case '+': result = Plus(a, b); break;
      case '-': result = Minus(a, b); break;
      case '*': result = Multiply(a, b);break;
      case '/': result = Divide(a, b); break;
    }
    cout << "Result of = "<< result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, '+');
    Switch(a, b, '-');
    Switch(a, b, '*');
    Switch(a, b, '/');
}
```

Solution Using Function Pointer

```
#include<iostream>
using namespace std;
// The four arithmetic operations
float Plus (float a, float b)
    { return a+b; }
float Minus (float a, float b)
    { return a-b; }
float Multiply(float a, float b)
    { return a*b; }
float Divide (float a, float b)
    { return a/b; }
// Solution with Function pointer
void Switch(float a, float b,
    float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b);
    cout << "Result := " << result << endl;
}
int main() { float a = 10.5, b = 2.5 ;
    Switch(a, b, &Plus);
    Switch(a, b, &Minus);
    Switch(a, b, &Multiply);
    Switch(a, b, &Divide);
}
```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers: Late Binding / Dynamically Loaded Library

- A C Feature in Shared Dynamically Loaded Libraries

Program Part-1

```
#include <dlfcn.h> // Shared object connection header

int main() {
    // Searching and opening shared object hello.so
    void* handle = dlopen("hello.so", RTLD_LAZY);

    // Type alias and variable for callback
    typedef void (*hello_t)();
    hello_t myHello = 0;

    // Getting the pointer to callback function
    myHello = (hello_t)dlsym(handle, "hello");

    // Invoking the callback function
    myHello();

    // Closing the shared object handle
    dlclose(handle);
}
```

Program Part-2

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

// Callback function
extern "C" void hello() {
   cout << "hello" << endl;
}
```

Function Pointers: Late Binding / Virtual Function

- A C++ Feature for Polymorphic Member Functions

Code Snippet Part-1	Code Snippet Part-2

`class A {`
`    public:`
`        void f();`
`        virtual void g();`
`};`
 	
`class B: public A {`
`    public:`
`        void f();`
`        virtual void g();`
`};`
	`void main() {`
	`    A a;`
	`    B b;`
	`    A *p = &b;`
	
	`    a.f();  // A::f()`
	`    a.g();  // A::g()`
	`    p->f(); // A::f()`
	`    p->g(); // B::g()`
	`}`

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Example: Callback, Function Pointers

- It is a Common C Feature

```
// Application
extern void (*func)();
void f() { }
void main() {
    func = &f;
    g();
}

// Library
void (*func)();
void g() {
    (*func)();
}
```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers: Callback Illustration (Step-1)

Function Pointers: Callback Illustration (Step-2)

Function Pointers: Callback Illustration (Step-3)

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers: Callback Illustration (Step-4)

Function Pointers: Callback Illustration (Step-Final)

Function Pointers: Callback Illustration (whole Process)

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers: Callback: Quick Sort using callback in qsort

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

```
// qsort signature in stdlib.h
void qsort(void *base,
           size_t nitems,
           size_t size,
           int (*compar)(const void *, const void*));

// Type unsafe compare function for int
int CmpFunc(const void* a, const void* b) {
    int ret = (*(const int*)a > *(const int*) b)? 1:
                  (*(const int*)a == *(const int*) b)? 0: -1;
    return ret;
}
void main() {
    int field[10];

    for(int c = 10; c>0; c--)
        field[10-c] = c;

    qsort((void*) field, 10, sizeof(field[0]), CmpFunc);
}
```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Function Pointers: Issues

- No value semantics
- Weak type checking
- Two function pointers having identical signature are necessarily indistinguishable
- No encapsulation for parameters

First-Class Object

- A **First-Class Object**[1] **(FCO)**[2] (also **citizen**, **type**, **entity**, or **value**) in a programming language is an entity which supports all the operations generally available to other entities. These operations typically include[3,4]
  - All items can be the *actual parameters* of functions
  - All items can be *returned as results* of functions
  - All items can be the *subject of assignment* statements (*copied*)
  - All items can be *tested for equality*

Source:

- First-class citizen, Wikipedia
- About first-,second- and third-class value, StackOverflow

[1]Object does not mean object in the object-oriented programming sense - it is just a thing
[2]In '60s, Christopher Strachey used FCO & SCO to contrast real numbers and procedures in ALGOL
[3]Defined by Robin Popplestone
[4]In '90s, Raphael Finkel defined of SCO & TCO, but these definitions have not been widely adopted

First-Class Objects: Examples

Module M06

Partha Pratim Das

Functors
- Callable Entities
- Function Pointers
  - Replace Switch / IF Statements
  - Late Binding
  - Virtual Function
  - Callback
  - Issues
  - FCO
- Basic Functor
  - Elementary Example
  - Functors in STL

λ in C++
- Closure Object
- Parameters
- Capture
- Examples
- Curry Function

Generic λ

- **First-Class Objects**
  - *Scalar data types*, like integer and floating-point numbers, are usually FCO
  - *Line Numbers* are FCO in **Fortran**
  - *Objects* of UDTs or classes are FCO in **C++** and **Java**
  - `vector`s and `string`s (in STL) are FCO in **C++**. So are several data structures (containers) like `list`, `deque`, `stack`, queue, and `map`
  - *Iterator* (STL) are FCO in **C++**
  - *Classes* in **Smalltalk**, **Objective-C**, **Ruby**, **Python**, etc. are FCO, are instances of metaclasses
  - *Functions* (*Closures* / *Anonymous Functions*) are FCO in **Smalltalk**, **Scheme**, **ML**, **Haskell**, **Python**, **Javascript**, etc.
  - *λ Functions* (*Closures*) are FCO in **C++11** and **Java**
  - *Functors* are FCO in **C++**. For example algorithms in `algorithm` (STL)
  - *Smart Pointers* in **C++** / **C++11** (STL)

Source:
- First-class citizen, Wikipedia
- First-class function, Wikipedia
- What does it mean for a function to be a first class object?, Quora

Not First-Class Objects: Examples

- **Not First-Class Objects**
  - *C-Strings* are not FCO in **C** / **C++**
  - *Arrays* are not FCO in **Fortran IV** and **C**. They cannot be assigned as objects. When they are passed as parameters, only the position of their first element is actually passed, their size is lost
  - *Labels* are not FCO in **C** / **C++**
  - *Pointers*, including *Function Pointers*, are not FCO in **C** / **C++**
  - *Classes* are not FCO in **C++**
  - *Functions*[5] are not FCO in **C** / **C++**

Source:

- First-class citizen, Wikipedia
- First-class function, Wikipedia
- What does it mean for a function to be a first class object?, Quora

[5] Funarg problem deals with issues surrounding functions as first class objects

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Functors or Function Objects

- Smart Functions
  - Functors are functions with a state
  - Functors encapsulate C / C++ function pointers
    - Uses templates and
    - Engages polymorphism
- Has its own Type
  - A class with zero or more private members to store the state and an overloaded `operator()` to execute the function
- Usually faster than ordinary Functions
- Can be used to implement callbacks
- Provides the basis for *Command Design Pattern*
- Functor is a first-class object

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Basic Functor

- Any class that overloads the function call operator:
  - `void operator()();`
  - `int operator()(int, int);`
  - `double operator()(int, double);`
  - ...

Functors: Elementary Example

- Consider the code below

```
int AdderFunction(int a, int b) {
    return a + b;
}

class AdderFunctor {
public:
    int operator()(int a, int b) {
        return a + b;
    }
};

void main() {
    int x = 5;
    int y = 7;
    int z = AdderFunction(x, y);

    AdderFunctor aF;
    int w = aF(x, y); // aF.operator()(x, y)
}
```

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Functors in STL

- Function objects are objects specifically designed to be used with a syntax similar to that of functions. In C++, this is achieved by defining member function operator() in their class, like for example:

```
struct myclass {
    int operator()(int a) { return a; }
} myobject;
int x = myobject (0);   // function-like syntax with object myobject
```

They are typically used as arguments to functions, such as predicates or comparison functions passed to standard algorithms. Several such algorithms are available in STL component:

```
#include <functional>
```

Source: `<functional>`, cplusplus.com

Functors in STL: Examples 1

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

- Fill a vector with random numbers
  - Include headers

```
#include <vector>
#include <algorithm> // generate
```

  - Function Pointer `rand` as Function Object

```
vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Functors in STL: Example 2

- Sort a vector of double by magnitude
  - Include headers

```
#include <vector>
#include <algorithm>  // sort
#include <functional> // binary_function
```

  - User-defined Functor `less_mag`

```
struct less_mag: public
binary_function<double, double, bool> {
    bool operator()(double x, double y)
        { return fabs(x) < fabs(y); }
};
vector<double> V; //...
sort(V.begin(), V.end(), less_mag());
```

Functors in STL: Example 3

- Find the sum of elements in a vector
  - Include headers

```
#include <vector>
#include <algorithm>  // for_each
#include <functional> // unary_function
```

  - User-defined Functor adder with local state

```
struct adder: public
unary_function<double, void> {
    adder() : sum(0)
    double sum;
    void operator()(double x) { sum += x; }
};

vector<double> V;
...
adder result = for_each(V.begin(), V.end(), adder());
cout << "The sum is " << result.sum << endl;
```

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

λ in C++11, C++14, C++17, C++20

Source:

- Scott Meyers on C++
- Lambda capture, cppreference.com
- Lambdas: From C++11 to C++20, Part 1
- Lambdas: From C++11 to C++20, Part 2

λ in C++: Closure Object

- Consider the λ-expression: *define* rem $\equiv \mathcal{E} \vdash n: Int, \lambda(m: Int).\ m \% n: Int$, where
  - λ variable m (type *Int*) is the *parameter* and is *bound* in rem
  - n (type *Int*) is *free* in rem and set from the *environment (context)* $\mathcal{E}$
  - rem computes $m \% n$, that is, m modulo n. It has type *Int* (return type)
- To write rem in the *Imperative & OO paradigm of C++*, we define a function / functor:

```
int rem(int m) // Function
    { return m % n; }
// Uses n in context
```

```
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
};
```

λ **Environment**: $\mathcal{E} \vdash n$ $\Longrightarrow$ **Imperative Context**: `int n; n = 7;`

```
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++

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

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

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

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;
    ```

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

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

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

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: FCOs


```
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

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Include_06/lambdaexpsyntax.png

[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

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

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: 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

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

- *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

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ


```
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

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

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ


```
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

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`

`[=,&identifer]()->rt{...}`: Capture

- 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

function<R(Args...)>

- Polymorphic wrapper for function objects applies to anything that can be called:
  - Function pointers
  - Functors (including closure objects)
  - Member function pointers
- Function declarator syntax

```
std::function< R ( A1, A2, A3...)  > f;
```

- Summary

Type	Old School Define	std::function
Free	`int(*callback)(int,int)`	`function< int(int,int) >`
Functor	`object_t callback`	`function< int(int,int) >`
Member	`int (object_t::*callback)(int,int)`	`function< int(int,int) >`

function<R(Args...)>

- **Function pointers**

```
int my_free_function(std::string s) { return s.size(); }
std::function< int(std::string) > f = my_free_function;
int size = f("ppd");
```

- **Functors**

```
struct my_functor { std::string s_; my_functor(std::string const & s) : s_(s) { }
    int operator()() const { return s_.size(); }
};
my_functor mine("ppd");
std::function< int() > f = std::ref(mine);
int size = f();
```

- **Closure Objects**

```
std::function< int(std::string const &) > f = [](std::string const & s){ return s.size(); };
int size = f("ppd");
```

- **Member function pointers**

```
struct my_struct { std::string s_; my_struct(std::string const & s) : s_(s) { }
    int size() const { return s_.size(); }
};
my_struct mine("ppd");
std::function< int() > f = std::bind(&my_struct::size, std::ref(mine));
int size = f();
```

Example

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

```
#include <iostream>
#include <functional>

int main() {
    std::function<int(int)> f2 =
        [&](int i) {
            std::cout << i << " ";
            if (i > 5) { return f1(i - 2); } else { return 0; }
        };

    f1 = [&](int i) { std::cout << i << " "; return f2(++i); };

    f1(10);

    return 0;
}
```

10 11 9 10 8 9 7 8 6 7 5 6 4 5

Example: Factorial

```
#include <iostream>
#include <functional>

int main() {
    std::function<int(int)> fact;
    fact =
        [&fact](int n) -> int
        { return (n == 0) ? 1 : (n * fact(n - 1)); };

    std::cout << "factorial(4) : " << fact(4) << std::endl;

    return 0;
}
```

Example: Fibonacci

```
#include <iostream>
#include <functional>
using namespace std;

int main() {
    std::function<int(int)> fibo;
    fibo =
        [&fibo](int n)->int
        { return (n == 0) ? 0 :
                 (n == 1) ? 1 :
                 (fibo(n - 1) + fibo(n - 2)); };

    cout << "fibo(8) : " << fibo(8) << endl;

    return 0;
}
```

Example: Pipeline

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
struct machine {
    template< typename T >
    void add(T f) { to_do.push_back(f); }
    int run(int v) {
        std::for_each(to_do.begin(), to_do.end(),
            [&v](std::function<int(int)> f) { v = f(v); });
        return v;
    }
    std::vector< std::function<int(int)> > to_do;
};
int foo(int i) { return i + 4; }
int main() { machine m;
    m.add([](int i){ return i * 3; });
    m.add(foo);
    m.add([](int i){ return i / 5; });
    std::cout << "run(7) : " << m.run(7) << std::endl;
    return 1;
}
```

```
run(7) : 5
```

Currying with C++ λ

```
#include <iostream>     // std::cout
#include <functional>

int main() {
    auto add = [](int x, int y) { return x + y; };
    auto add5 = [=](int y) { return add(5, y); };  // Curry

    std::cout << "W/o curry:\n" << add(5, 3);
    std::cout << "W/  curry:\n" << add5(3);
}

Output:
W/o curry:8
W/  curry:8
```

- On the 'Curry' line, we can capture also by [&], [&add], or [add]. However, it does not work without default or explicit capture as the symbol add is used in the body. So [] fails
- This is a hard-coded solution. There is built-in solution. Generic operator for Curry can be built separately using variadic templates, variadic functions and lambda functions. This is outside of our current scope.

Source: C++11: lambda, currying, StackOverflow

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Generic λ in C++14

Generic λ in C++14

Source:

- Generic lambdas, ISO
- Lambda expressions (since C++11), cppreference.com
- Scott Meyers on C++
- Lambda expressions in C++, Microsoft
- Generalized Lambda Expressions in C++14
- Generic code with generic lambda expression

Generic / Generalized λ in C++

- C++11 introduced λ expressions as short inline anonymous functions that can be nested inside other functions and function calls
- C++ 14 buffed up λ expressions by introducing Generic or Generalized λ
- Following λ function returns the sum of two integers

```
[](int a, int b) -> int { return a + b; } // Return type is optional
```

- Whereas we need a different λ to obtain the sum of two floating point values:

```
[](double a, double b) -> double { return a + b; } // Return type is optional
```

- In C++11 we could unify these two λ functions using template parameters:

```
template<typename T>
[](T a, T b) -> T { return a + b } // Return type is optional - compiler may infer
```

- C++ 14 circumvent this by the keyword auto in the input parameters of the λ expression. Thus the compilers can now deduce the type of parameters during compile time:

```
[](auto a, auto b) { return a + b; } // Compiler must infer return type
```

Generic / Generalized λ in C++

```
#include <iostream>, #include <string>, using namespace std;

// C++11 lambda's - separate lambda for every type. Return type is optional
auto add_i = [](int a, int b) { return a + b; };                // Compiler may infer return type
auto add_d = [](double a, double b) { return a + b; };          // Compiler may infer return type
auto add_s = [](string a, string b) { return a + b; };          // Compiler may infer return type
// C++11 templatized lambda - one lambda for multiple types. Return type is optional
template<typename T> auto add_t = [](T a, T b) { return a + b; }; // Compiler may infer return type
// C++14 generic lambda. Return type cannot be specified
auto add = [](auto a, auto b) { return a + b; };                 // Compiler must infer return type
int main () {
    // Different name of each lambda for each type: No inference
    cout << add_i(3, 5);            // add_i for int type
    cout << add_d(2.6, 1.3);        // add_d for double type
    cout << add_s("Good ", "Day"); // add_s for string type converts from const char*

    // Same name of the lambda for all types, type must be specified: No inference
    cout << add_t<int>(3, 5);                 // add_t<int> for int type
    cout << add_t<double>(2.6, 1.3);          // add_t<double> for double type
    cout << add_t<string>("Good ", "Day"); // add_t<string> for string type converts from const char*

    // Same name of the lambda for all types and no type need to be specified: It is inferred
    cout << add(3, 5);                               // add for int type
    cout << add(2.6, 1.3);                           // add for double type
    cout << add(string("Good "), string("Day")); // add for string type - cannot convert from const char*
}
```

Return Type of Generic λ in C++

- The return type may be inferred by the compiler from the return expression:

```
auto add = [](auto a, auto b) { return a + b; };
cout << add(2, 3);       // 5   // rt = int
cout << add(4, 3.3);     // 7.3 // rt = double
cout << add(2.6, 2);     // 4.6 // rt = double
cout << add(3.8, 4.5);   // 8.3 // rt = double
```

- The return type may be specified from the parameters:

```
auto add = [](auto a, auto b) -> decltype(a) { return a + b; };
cout << add(2, 3);       // 5   // rt = int
cout << add(4, 3.3);     // 7   // rt = int
cout << add(2.6, 2);     // 4.6 // rt = double
cout << add(3.8, 4.5);   // 8.3 // rt = double

auto add = [](auto a, auto b) -> decltype(b) { return a + b; };
cout << add(2, 3);       // 5   // rt = int
cout << add(4, 3.3);     // 7.3 // rt = double
cout << add(2.6, 2);     // 4   // rt = int
cout << add(3.8, 4.5);   // 8.3 // rt = double
```

- The return type may be explicitly specified:

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

Use of Generic λ in C++

```
// Sorting integers, floats, strings using a generalized lambda and sort function
// #include <iostream>, #include <string>, #include <vector>, #include <algorithm>, using namespace std;

// Utility Function to print the elements of a collection
void printElements(auto& C) {
    for (auto e : C) cout << e << " ";
    cout << endl;
}
int main() {
    // Declare a generalized lambda and store it in greater. Works for int, double, string, ...
    auto greater = [](auto a, auto b) -> bool { return a > b; };

    vector<int> vi = { 1, 4, 2, 1, 6, 62, 636 };              // Initialize a vector of integers
    vector<double> vd = { 4.62, 161.3, 62.26, 13.4, 235.5 }; // Initialize a vector of doubles
    vector<string> vs = { "Tom", "Harry", "Ram", "Shyam" };  // Initialize a vector of strings

    sort(vi.begin(), vi.end(), greater); // Sort integers
    sort(vd.begin(), vd.end(), greater); // Sort doubles
    sort(vs.begin(), vs.end(), greater); // Sort strings

    printElements(vi); // 636 62 6 4 2 1 1
    printElements(vd); // 235.5 161.3 62.26 13.4 4.62
    printElements(vs); // Tom Shyam Ram Harry
}
```

Generic Recursive λ

- A λ does not have a named specific type. So a recursive λ expression needs `std::function` wrapper
- The generic λ expression allows recursive λ functions without using `std::function`

```
#include <iostream>  // C++11 solution using std::function. Works for int base
#include <functional>
int main() {
    std::function<int(int,int)> pow; // pow recursive function. std::function used to define type of pow
    pow = [&pow](int base, int exp) { return exp==0 ? 1 : base*pow(base, exp-1); };
    std::cout << pow(2, 10);       // 2^10 = 1024
    std::cout << pow(2.71828, 10); // e^10 = 1024 // 2.71828 is cast to int giving 2 and wrong result
}

#include <iostream>  // C++14 solution without using std::function. Works for any numeric base
int main() {
    auto power = [](auto self, auto base, int exp) -> decltype(base) { // any numeric 'base' type
        return exp==0 ? 1 : base*self(self, base, exp-1);
    };
    // Wrapper of power to avoid passing power as first parameter to the call
    auto pow = [power](auto base, int exp) -> decltype(base) { return power(power, base, exp); };

    std::cout << power(power, 2, 10);       // 2^10 = 1024 // Needs to pass itself as first parameter
    std::cout << power(power, 2.71828, 10); // e^10 = 22026.3

    std::cout << pow(2, 10);       // 2^10 = 1024 // Wrapper provides a clean solution
    std::cout << pow(2.71828, 10); // e^10 = 22026.3
}
```

Generic Recursive λ: Factorial

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL

λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function

Generic λ

```
#include <iostream>  // Factorial lambda in C++11 solution using std::function
#include <functional>
int main () {
    std::function < int (int) > fact;
    fact = [&fact](int n) -> int
        { return (n == 0) ? 1 : (n * fact(n - 1));
    };
    std::cout << "factorial(4) : " << fact(4) << std::endl;
}
```

```
#include <iostream>  // Factorial lambda in C++14 without using std::function
int main () {
    auto factorial = [](auto self, int n) -> int
        { return (n == 0) ? 1 : (n * self(self, n - 1));
    };
    auto fact = [factorial](int n) -> int // Wrapper of factorial to skip passing factorial
        { return factorial(factorial, n);
    };
    std::cout << "factorial(4) : " << fact(4) << std::endl;
}
```

Module M06

Partha Pratim Das

Functors
Callable Entities
Function Pointers
Replace Switch / IF Statements
Late Binding
Virtual Function
Callback
Issues
FCO
Basic Functor
Elementary Example
Functors in STL
λ in C++
Closure Object
Parameters
Capture
Examples
Curry Function
Generic λ

Generic Recursive λ: Fibonacci

```
#include <iostream>  // Fibonacci lambda in C++11 solution using std::function
#include <iostream>
#include <functional>
int main () {
    std::function < int (int) > fibo;
    fibo = [&fibo](int n) -> int {
        return (n == 0) ? 0 : (n == 1) ? 1 : (fibo(n - 1) + fibo(n - 2));
    };
    std::cout << "fibo(8) : " << fibo(8) << std::endl;
}
```

```
#include <iostream>  // Fibonacci lambda in C++14 without using std::function
int main () {
    auto fibonacci = [](auto self, int n) -> int {
        return (n == 0) ? 0 : (n == 1) ? 1 : (self(self, n - 1) + self(self, n - 2));
    };
    auto fibo = [fibonacci](int n) -> int { // Wrapper of fibonacci to skip passing fibonacci
        return fibonacci(fibonacci, n);
    };
    std::cout << "fibo(8) : " << fibo(8) << std::endl;
}
```

Capture-less Generic λ to Function Pointers

Module M06

Partha Pratim Das

Functors
- Callable Entities
- Function Pointers
  - Replace Switch / IF Statements
  - Late Binding
  - Virtual Function
  - Callback
  - Issues
  - FCO
- Basic Functor
  - Elementary Example
  - Functors in STL

λ in C++
- Closure Object
- Parameters
- Capture
- Examples
- Curry Function

Generic λ

- A non-generic λ with an empty capture-list can be converted to a function pointer

```
typedef int (*fp) (int);             // Function pointer
const fp f = [](int i){ return i; }; // Converts to a func. ptr. w/o capture
```

- A capture-less generic λ behaves in the same way
- Further, it can be converted to more than one compatible function pointers

```
#include <iostream>

void f(void(*fp)(int))    { fp(1); /*...*/ }
void g(void(*fp)(double)) { fp(2.2); /*...*/ }

int main () {
    auto op = [](auto x) { // generic code for x
        std::cout << "x = " << x << std::endl;
    };

    // use 'op' as a generic callback function pointer
    f(op); // x = 1
    g(op); // x = 2.2
}
```