CS40032: Principles of Programming Languages
Module 05: λ in C++

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

Feb 03 & 08, 2021

Table of Contents

Functors in C++

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++

- 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()

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

Direct Function Pointer

```
#include <stdio.h>

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

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

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

    int result = (*pt2Function)
                     (12, ’a’, ’b’);
    printf("%d", result);

    return 0;
}

int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}

---
DoIt
207
```

Using `typedef`

```
#include <stdio.h>

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);

    return 0;
}

int DoIt (int a, char b, char c) {
    printf ("DoIt\n");

    return a + b + c;
}

---
DoIt
207
```

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

- 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

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, ’/’) ;
    return 0 ;
}
```

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) ;
    return 0 ;
}
```

Function Pointers – Late Binding / Dynamically Loaded Library

- A C Feature in Shared Dynamically Loaded Libraries

Program Part-1

```
#include <dlfcn.h>
int main() {
    void* handle =
    dlopen("hello.so", RTLD_LAZY);
    typedef void (*hello_t)();
    hello_t myHello = 0;
    myHello = (hello_t)
    dlsym(handle, "hello");
    myHello();
    dlclose(handle);
}
```

Program Part-2

```
#include <iostream>
using namespace std;
extern "C" void hello() {
   cout << "hello" << endl;
}
```

Function Pointers – Late Binding / Virtual Function

- A C++ Feature for Polymorphic Member Functions

Code Snippet Part-1

```
class A {
    public:
        void f();
        virtual void g();
};

class B: public A {
    public:
        void f();
        virtual void g();
};
```

Code Snippet Part-2

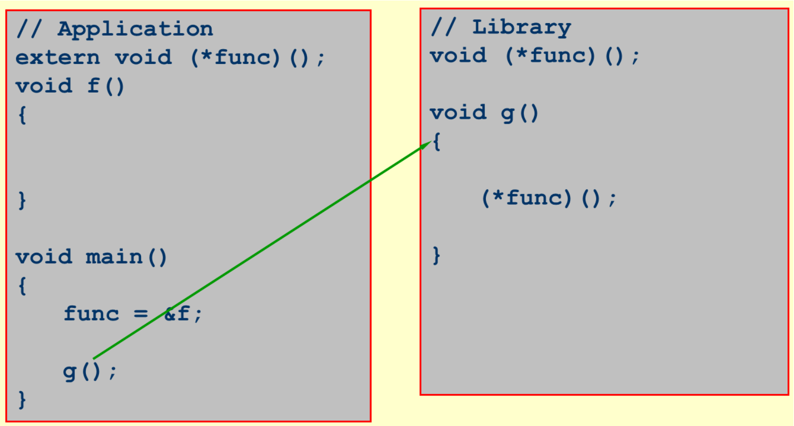
```
void main() {
    A a;
    B b;
    A *p = &b;

    a.f(); // A::f()
    a.g(); // A::g()
    p->f();// A::f()
    p->g();// B::g()
}
```

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)();
}
```

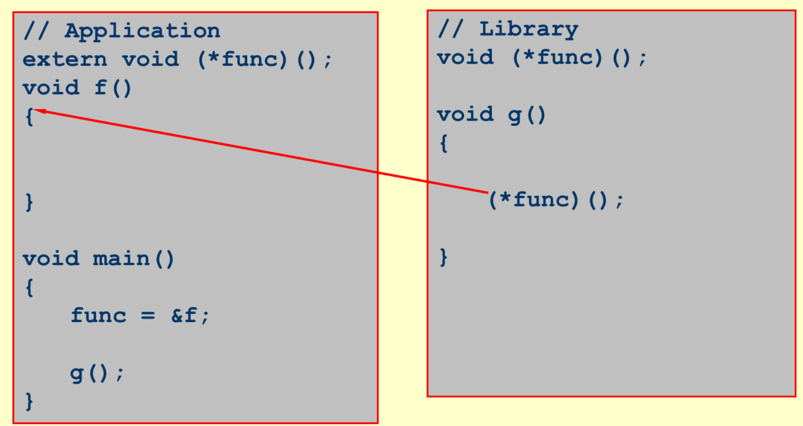
Function Pointers: Callback Illustration (Step-1)



```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

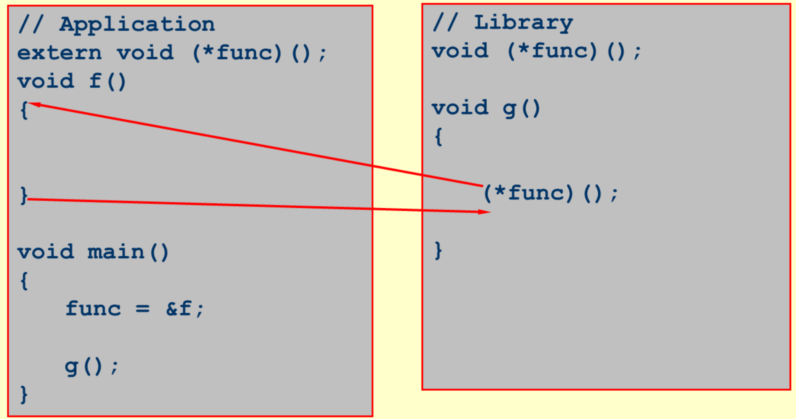
Function Pointers: Callback Illustration (Step-2)



```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

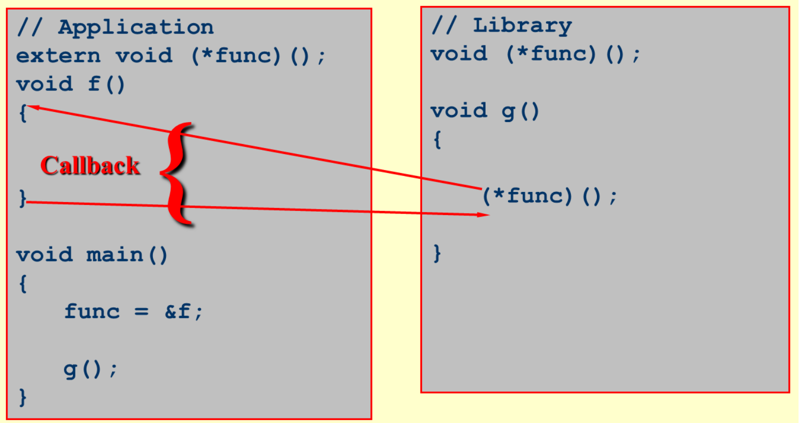
Function Pointers: Callback Illustration (Step-3)



```
// Application
extern void (*func)();
void f()
{


}


void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

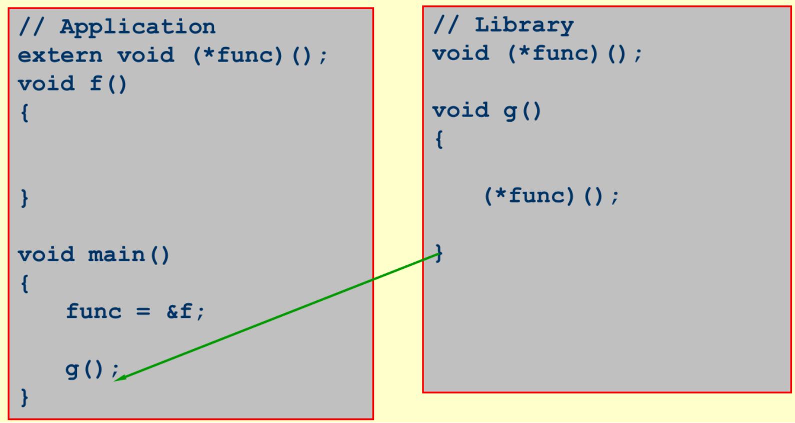
Function Pointers: Callback Illustration (Step-4)



```
// Application
extern void (*func)();
void f()
{

}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

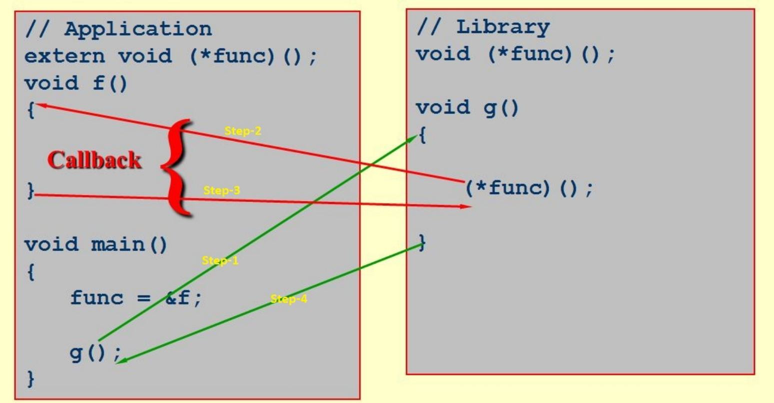
Function Pointers: Callback Illustration (Step-Final)



```
// Application
extern void (*func)();
void f()
{


}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers: Callback Illustration (whole Process)



```
// Application
extern void (*func)();
void f()
{

}

void main()
{
    func = &f;

    g();
}
```

```
// Library
void (*func)();

void g()
{

    (*func)();

}
```

Function Pointers–Callback: Quick Sort Implementation using callback in 'qsort'

```
void qsort(void *base,
           size_t nitems,
           size_t size,
           int (*compar)(const void *, const void*));

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);
}
```

Function Pointers – Issues

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

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*

Basic Functor

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

Functors: Elementary Example

- Look at 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);
}
```

Functors: Examples from 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>
```

  http://www.cplusplus.com/reference/functional/

- Fill a vector with random numbers
  - Function Pointer rand as Function Object

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

- Sort a vector of double by magnitude
  - 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: Examples from STL

- Find the sum of elements in a vector
    - 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;
```

λ in C++11, C++14

Using λ

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

auto twice = [](const function<int(int)>& f, int v) { return f(f(v)); };
auto f = [](int i) { return i + 3; };
auto sqr = [](int i) { return i * i; };
auto comp = [](const function<int(int)>& f,
               const function<int(int)>& g, int v) { return f(g(v)); };
int main() {
    auto a = 7, b = 5, c = 3;

    cout << twice(f, a) << " " << comp(f, f, a) << endl; // 13 13
    cout << twice(sqr, b) << " " << comp(sqr, sqr, b) << endl; // 625 625
    cout << comp(sqr, f, c) << " " << comp(f, sqr, c) << endl; // 36 12

    return 0;
}
```

The lambda's in C++ above correspond to the λ-expressions below:

$$twice \equiv \lambda f.\ \lambda v.\ f\ (f\ v)$$

$$f \equiv \lambda i.\ i + 3$$

$$sqr \equiv \lambda i.\ i * i$$

$$comp \equiv \lambda f.\ \lambda g.\ \lambda v.\ f\ (g\ v)$$

Using Functor

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

struct mod {
    mod() : modulus(8) {}
    int operator()(int v) { return v % modulus; }
    int modulus;
};
int main() {
    vector<int> in, out;

    in.push_back(10); in.push_back(25); in.push_back(40); in.push_back(55);
    out.resize(in.size());

    for (auto it = in.begin(); it != in.end(); ++it) cout << *it << ’ ’;
    cout << endl;

    transform(in.begin(), in.end(), out.begin(), mod());

    for (auto it = out.begin(); it != out.end(); ++it) cout << *it << ’ ’;
    cout << endl;

    return 0;
}

Output:
10 25 40 55
2 1 0 7
```

Using λ

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

int main() {
    vector<int> in, out;

    in.push_back(10); in.push_back(25); in.push_back(40); in.push_back(55);
    out.resize(in.size());

    for (auto it = in.begin(); it != in.end(); ++it) cout << *it << ’ ’;
    cout << endl;

    transform(in.begin(), in.end(), out.begin(), [](int v) { return v % 8; }); // lambda

    for (auto it = out.begin(); it != out.end(); ++it) cout << *it << ’ ’;
    cout << endl;

    return 0;
}
```

```
Output:
10 25 40 55
2 1 0 7
```

Compare: Functor & Lambda

```
struct mod {
    mod() : modulus(8) {}
    int operator()(int v) { return v % modulus; }
    int modulus;
};

transform(in.begin(), in.end(), out.begin(),
    mod());
```

```
transform(in.begin(), in.end(), out.begin(),
    [](int v) { return v % 8; });
```

Compare: Functor & Lambda

```
struct mod {
    mod(int m) : modulus(m) {}
    int operator()(int v) { return v % modulus; }
    int modulus;
};

int my_mod = 8;

transform(in.begin(), in.end(), out.begin(),
    mod(my_mod));
```

```
int my_mod = 8;

transform(in.begin(), in.end(), out.begin(),
    [my_mod](int v) -> int { return v % my_mod; });
```

- State Variable
- Parameter
- Return Type

Basic λ Syntax

A lambda expression consists of the following:

```
[capture list] (parameter list) {function body}
```

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

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

- The parameter list is just like a sequence of parameter types and variable names, and follows the same rules as for an ordinary function
- The function body is likewise an ordinary function body
- If there is no return statement in the function body, the return type is assumed to be void.
- If the function body consists of only a return statement (which is very common), the return type is assumed to be the same as the type of the value being returned
- For example, with this lambda, the compiler assumes that the return type is void, so calling it without any use of the return value is legal:
  ```
  [](){ cout << "Hello from trivial lambda!" << endl; } ();
  ```
- However, trying to use the return type of the call by outputting it is not legal - there is no return value, so the following won't compile:
  ```
  cout << [](){ cout << "Hello from trivial lambda!" << endl; }()
      << endl;
  ```

Basic λ Syntax

"Hello World" using lambda in C++

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

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

    return 0;
}
```

Basic λ Syntax

- The following lambda takes two integers as parameters and returns a bool value which is true if the first integer is half the value of the second
- The compiler knows a `bool` is returned from the lambda function because that is what the return statement returns:

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

- 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;
```

- In the following lambda, we tell the compiler that an `int` needs to be returned, even 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”.

Anatomy of Lambda Expressions

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

- Introducer: `[my_mod]`
- Capture: `my_mod`
- Parameters: `(int v)`
- Return Type: `-> int`
- Declarator: `(int v) -> int`
- Statement: `{ return v % my_mod; }`
- Lambda Expression:

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

Closure Object

Lambda Expression:

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

$$\Rightarrow$$

Closure Object

- Evaluation of the expression results in a temporary called a **closure object**
- A closure object is unnamed
- A closure object behaves like a function object

Using Closure Objects: Parameters

```
[](){ std::cout << "foo" << std::endl; } ();
```

```
Output: foo
```

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

```
Output: 7*6=42
```

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

```
Output: the correct value is: 42
```

Using Closure Objects: Parameters

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

Output: error: assignment of read-only reference ’v’
```

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

Output: v: 42
```

Using Closure Objects: Parameters

Notice that the lambda's parameters do not affect the namespace

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

```
Output j: 42
```

Lambda expression without a declarator acts as if it were ()

```
[]{ std::cout << "foo" << std::endl; } ();
```

is same as

```
[](){ std::cout << "foo" << std::endl; } ();
```

Using Closure Objects: Capture

- We commonly want to capture state or access values outside our function objects
- With a function object we use the constructor to populate state

```
struct mod {
    mod(int m_) : modulus(m_) {}
    int operator()(int v_) { return v_ % modulus; }
    int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
    mod(my_mod));
```

Using Closure Objects: Capture

- Lambda expressions provide an optional capture

```
[my_mod](int v_) ->int { return v_ % my_mod; }
```

- We can capture by:
  - Default all by reference
  - Default all by value
  - List of specific identifier(s) by value or reference and/or this
  - Default and specific identifiers and/or this

Using Closure Objects: Capture

- 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](){ ... }
  ```

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

Capture default all by reference:

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

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

```
template< typename T >
void fill(std::vector<int> & v, T done) {
    int i = 0;
    while (!done()) {
        v.push_back(i++);
    }
}
std::vector<int> stuff;

fill(stuff, [&] { return stuff.size() >= 8; });
for(auto it = stuff.begin(); it != stuff.end(); ++it)
    std::cout << *it << ’ ’;
std::cout << std::endl;

Output: 0 1 2 3 4 5 6 7
```

- Capture by value:

```
[=] { return stuff.size() >= 8; };
```

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

```
template< typename T >
void fill(std::vector<int> & v, T done) {
    int i = 0;
    while (!done()) {
        v.push_back(i++);
    }
}
std::vector<int> myvec;

// Fill the vector with 0, 1, 2, ... till the sum of
// elements exceeds 10
fill(myvec, [&] {
                    int sum = 0;
                    std::for_each(myvec.begin(), myvec.end(),
                            [&](int i){ sum += i; }); // [=] is error
                    return sum >= 10;
                }
    );
for(auto it = myvec.begin(); it != myvec.end(); ++it)
    std::cout << *it << ’ ’;
std::cout << std::endl;

Output: 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;
```

```
Output: 0 1 2 0 1 2 0 1 2 0
```

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

Where is the value captured?

```
int x = 42;
auto fL = [=] () { std::cout << x << std::endl; };
std::cout << "Lambda Eval: ";
x = 8;
fL();
```

At the time of evaluation: `Output:  Lambda Eval:  42`

```
struct functor {
    functor(int x_) : x(x_) {};
    void operator()() { std::cout << x << std::endl; };
    int x;
};

int x = 42;
auto fF = functor(x);
std::cout << "Functor Eval: ";
x = 8;
fF();
```

```
Output: Functor Eval: 42
```

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

```
    int x = 42; int y = 37;
    auto fLi = [&x, y]() { std::cout << "Value?" << std::endl;
        std::cin >> x; std::cout << x << " " << y << std::endl; };
    std::cout << "Lambda Eval: ";
    x = 8; y = 20; fLi();

Input: 17
Output:
Lambda Eval: Value?
17 37
```

```
    struct ftor { int& x, y;
        ftor(int& x_, int y_) : x(x_), y(y_) {};
        void operator()() { std::cout << "Value?" << std::endl;
            std::cin >> x; std::cout << x << " " << y << std::endl;
        };
    };

    int x = 42; int y = 37;
    auto fFi = ftor(x, y);
    std::cout << "Functor Eval: ";
    x = 8; y = 20; fFi();

Input: 17
Output:
Functor Eval: Value?
17 37
```

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

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

Compile error:

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

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

Lambda 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;
```

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

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

```
    int h = 10;
    auto f1 = [=] () mutable { h *= 2; return h; };
    std::cout << "2h:" << f1() << std::endl;
    std::cout << " h:" << h << std::endl;
```

Output:

```
    int h = 10;
    auto g1 = [&] () { h *= 2; return h; };
    std::cout << "2h:" << g1() << std::endl;
    std::cout << " h:" << h << std::endl;
```

Output:

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

```
    int h = 10;
    auto f1 = [=] () mutable { h *= 2; return h; };
    std::cout << "2h:" << f1() << std::endl;
    std::cout << " h:" << h << std::endl;
```

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

```
    int h = 10;
    auto g1 = [&] () { h *= 2; return h; };
    std::cout << "2h:" << g1() << std::endl;
    std::cout << " h:" << h << std::endl;
```

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

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

```
    int h = 10;
    auto f2 = [=] () mutable { h *= 2; return h; };
    std::cout << "2h:" << f2() << " h:" << h << std::endl;
    std::cout << "2h:" << f2() << " h:" << h << std::endl;
```

Output:

```
    int h = 10;
    auto g2 = [&] () { h *= 2; return h; };
    std::cout << "2h:" << g2() << " h:" << h << std::endl;
    std::cout << "2h:" << g2() << " h:" << h << std::endl;
```

Output:

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

```
    int h = 10;
    auto f2 = [=] () mutable { h *= 2; return h; };
    std::cout << "2h:" << f2() << " h:" << h << std::endl;
    std::cout << "2h:" << f2() << " h:" << h << std::endl;
```

```
Output:
2h:20 h:10
2h:40 h:10
```

```
    int h = 10;
    auto g2 = [&] () { h *= 2; return h; };
    std::cout << "2h:" << g2() << " h:" << h << std::endl;
    std::cout << "2h:" << g2() << " h:" << h << std::endl;
```

```
Output:
2h:20 h:10
2h:40 h:20
```

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

```
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();

Output: extras: 6
```

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

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

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

```
struct foo {
    foo() : i(0) {}
    void amazing(){ [=]{ i=8; }(); }
    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

- Capture restrictions:
  - Identifiers must only be listed once

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

  - Default by value, explicit identifiers by reference

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

  - Default by reference, explicit identifiers by value

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

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

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

```
    int i = 8;
    {
        int j = 2;
        auto f = [=]{ std::cout << i / j; };
        f();
    }
Output: 4
```

```
    int i = 8;
    auto f =
        [=]()
    {
        int j = 2;
        auto m = [=]{ std::cout << i / j; };
        m();
    };
    f();
Output: 4
```

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

```
    int i = 8;
    auto f =
        [i]()
    {
        int j = 2;
        auto m = [=]{ std::cout << i / j; };
        m();
    };
    f();
Output: 4
```

```
    int i = 8;
    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
```

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

```
    int i = 8;
    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
```

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

```
Output: inner: 4 outer: 8
```

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

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

```
Output: ?
```

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

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

```
Output: 426 : 126
```

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

- Closure object has implicitly-declared copy constructor / destructor

```
struct trace
{
    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; }
    int i;
};
```

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

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

```
Output:
construct
destroy
```

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

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

```
Output:
construct
copy construct
- make copy -
copy construct
destroy
destroy
destroy
```

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

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

```
Output:
construct
-- make copy --
destroy
```

Storing / Passing Lambda Objects

Seen two ways so far:

```
template<typename T> void foo(T f)
```

```
auto f = []{};
```

Function pointer

If the lambda expression has no capture it can be converted to a function pointer with the same signature

```
typedef int (*f_type) (int);
f_type f = [](int i) { return i+20; };
std::cout << f(8);
```

```
Output:
28
```

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

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

function<R(Args...)>

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;
f = my_free_function;

int size = f("ppd");
```

function<R(Args...)>

Functors

```
struct my_functor
{
    my_functor( std::string const & s) : s_(s) {}
    int operator()() const { return s_.size(); }
    std::string s_;
};

my_functor mine("ppd");
std::function< int() > f;

f = std::ref(mine);
int size = f();
```

function<R(Args...)>

Member function pointers

```
struct my_struct
{
    my_struct( std::string const & s) : s_(s) {}
    int size() const { return s_.size(); }
    std::string s_;
};

my_struct mine("ppd");
std::function< int() > f;

f = std::bind( &my_struct::size, std::ref(mine) );
int size = f();
```

function<R(Args...)>

Closure Objects

```
std::function< int(std::string const &) > f;

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

Example

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

int main() {
    std::function<int(int)> f1;
    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;
}

Output: 10 11 9 10 8 9 7 8 6 7 5 6 4 5
```

Example: Factorial

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

int main_fact() {
    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>
using namespace std;

struct machine {
    template< typename T >
    void add(T f) { to_do.push_back(f); }
    int run(int v) {
        for_each(to_do.begin(), to_do.end(),
            [&v](std::function<int(int)> f) { v = f(v); });
        return v;
    }
    vector< 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; });
    cout << "run(7) : " << m.run(7) << endl;

    return 1;
}
```

Output:

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;
}
```

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

Currying with C++ Lambda

```
#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);

    return 0;
}
Output:
W/o curry:8
W/  curry:8
```

Note: 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.

http://stackoverflow.com/questions/39468955/c11-lambda-currying

More on λ in C++

Source: Scott Meyer on C++

Functor Example

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find_if
#include <vector>       // std::vector

bool IsOdd(int i) { return ((i % 2) == 1); }

int main() { std::vector<int> v;
    v.push_back(10); v.push_back(25);
    v.push_back(40); v.push_back(55);

    std::vector<int>::iterator it =
        std::find_if(v.begin(), v.end(), IsOdd);

    std::cout << "The first odd value is " << *it << ’\n’;

    return 0;
}
```

Using Lambda

```
#include <iostream>     // std::cout
#include <algorithm>    // std::find_if
#include <vector>       // std::vector

int main() { std::vector<int> v;
    v.push_back(10); v.push_back(25);
    v.push_back(40); v.push_back(55);

    auto it = std::find_if(v.begin(), v.end(),
                 [](int i) { return ((i % 2) == 1); });

    std::cout << "The first odd value is " << *it << ’\n’;

    return 0;
}
```

Generates:

```
class MagicType1 {
public:
    bool operator() (int i) const { return ((i % 2) == 1); }
};
```

Lambda Expressions

Another example:

```
typedef std::shared_ptr<Widget> SPWidget;
std::deque<SPWidget> d;
...
std::sort(d.begin(), d.end(),
    [](const SPWidget& sp1, const SPWidget& sp2)
    { return *sp1 < *sp2; });
```

Essentially generates:

```
class MagicType2 {
public:
    bool operator()(const SPWidget& p1, const SPWidget& p2) const
    { return *p1 < *p2; }
};
...
std::sort(d.begin(), d.end(), MagicType2());
```

Function objects created through lambda expressions are **closures**

Variable References in Lambdas

Closures may outlive their creating function:

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

auto f = returnClosure(10);                   // f is essentially a
                                              // copy of lambda's
                                              // closure
```

In this call,

```
if (f(22)) ...                                // invoke the closure
```

what are the values of a, b, c?

`returnClosure` no longer active!

Variable References in Lambdas

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 ...

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

auto f = returnClosure();                      // as before

...

if (f(22)) ...                                 // as before
```

a, b, c outlive `returnClosure`'s invocation

Variable References in Lambdas

Rules for variables lambda's may refer to:

- Non-static locals referenceable only if **captured**

```
std::function<bool(int)> returnClosure(int a)
{
    int b, c; ...

    return [](int x)
           { return a*x*x + b*x + c == 0; }; // to compile, must
}                                            // capture a, b, c;
                                             // this example
                                             // won't compile
```

- Variables of static storage duration always referenceable

```
int a;

std::function<bool(int)> returnClosure()
{
    static int b, c; ...

    return [](int x)
           { return a*x*x + b*x + c == 0; };  // no need to
}                                             // capture a, b, c
```

Capturing Local Variables

Capturing locals puts copies in closures:

```
{
    int minVal; double maxVal;
    ...
    auto it = std::find_if(v.cbegin(), v.cend(),
        [minVal, maxVal](int i)
            { return i > minVal && i < maxVal; }
        );
}
```

Essentially corresponds to:

```
class MagicType {
public:
    MagicType(int v1, double v2): _minVal(v1), _maxVal(v2) {}
    bool operator()(int i) const
        { return i > _minVal && i < _maxVal; }
private:
    int _minVal; double _maxVal;
};
auto it =
    std::find_if(v.cbegin(), v.cend(), MagicType(minVal, maxVal));
```

Capturing Local Variables

Captures may also be by reference:

```
{
    int minVal; double maxVal;
    ...
    auto it = std::find_if(v.cbegin(), v.cend(),
        [&minVal, &maxVal](int i)
            { return i > minVal && i < maxVal; }
        );
}
```

Essentially corresponds to:

```
class MagicType {
public:
    MagicType(int&v1, double& v2): _minVal(v1), _maxVal(v2) {}
    bool operator()(int i) const
        { return i > _minVal && i < _maxVal; }
private:
    int& _minVal; double& _maxVal;
};
auto it = std::find_if(v.cbegin(), v.cend(), // same as
    MagicType(minVal, maxVal));              // before
```

Capturing Local Variables

Different (non-static) locals may be captured differently:

```
{
    int minVal; double maxVal;
    ...
    auto it = std::find_if(v.cbegin(), v.cend(),
        [minVal, &maxVal](int i)
            { return i > minVal && i < maxVal; }
        );
}
```

Essentially corresponds to:

```
class MagicType {
public:
    MagicType(int v1, double& v2): _minVal(v1), _maxVal(v2) {}
    bool operator()(int i) const
        { return i > _minVal && i < _maxVal; }
private:
    int _minVal; double& _maxVal;
};

auto it = std::find_if(v.cbegin(), v.cend(), // same as
    MagicType(minVal, maxVal));               // before
```

Capturing Local Variables

Capture mode defaults may be specified:

```
auto it = std::find_if(v.cbegin(), v.cend(), // default is
                   [=](int i)                 // by value
                       { return i > minVal && i < maxVal; }
               );

auto it = std::find_if(v.cbegin(), v.cend(), // default is
                   [&](int i)                 // by ref
                       { return i > minVal && i < maxVal; });
```

With a default capture mode, captured variables need not be listed (As in examples above)

Capturing Local Variables

Default overridable on a per-variable basis:

```
auto it = std::find_if(v.cbegin(), v.cend(), // default capture is
        [=, &maxVal](int i)                  // by value, but maxVal
        { return i > minVal &&               // is by reference
                 i < maxVal; }
    );
```

Essentially corresponds to:

```
class MagicType {
public:
    MagicType(int v1, double& v2): _minVal(v1), _maxVal(v2) {}
    bool operator()(int i) const
        { return i > _minVal && i < _maxVal; }
private:
    int _minVal; double& _maxVal;
};
auto it =
    std::find_if(v.cbegin(), v.cend(), MagicType(minVal, maxVal));
```

Capturing Class Members

To access class members within a member function, capture `this`:

```
class Widget {
public:
    void doSomething();
private:
    std::list<int> li;
    int minVal;
};
void Widget::doSomething() {
    auto it = std::find_if(li.cbegin(), li.cend(), // error! attempt
        [minVal](int i) { return i > minVal; }     // to capture
        );                                         // "this->minVal"
    ...
}

void Widget::doSomething() {
    auto it = std::find_if(li.cbegin(), li.cend(),
        [this](int i)
        { return i > minVal; } // fine
   );                          // ("minVal"  ...
                               // "this->minVal")
}
```

Capturing Class Members

A default capture mode also makes `this` available:

```
class Widget {
public:
    void doSomething();
private:
    std::list<int> li; int minVal;
};
void Widget::doSomething() {
    auto it = std::find_if(li.cbegin(), li.cend(),
        [=](int i) { return i > minVal; }      // fine, copies
    );                                         // "this" into closure
...
}

void Widget::doSomething() {
    auto it = std::find_if(li.cbegin(), li.cend(),
        [&](int i) { return i > minVal; }   // also fine, holds
    );                                      // ref to "this" in
...                                         // closure
}
```