www.italiancpp.org

Lambdas Recipes in C++1{1,4}

Gian Lorenzo Meocci

28 Giugno Milano – C++ Community Meetup

About Me!

Software Engineer @ Commprove (Firenze)

C++/C++11, Java7, PHP5, Javascript, HTML5/CSS3

glmeocci@gmail.com
http://www.meocci.it
https://github.com/meox

Preferred OS: Linux especially **Debian Linux**
Preferred PL: **C++11**
Preferred IDE: Sublime Text 3 & vim

Introduction to Lambdas 1/3

- It's a simplified notation for defining an anonymous function object
- It's a shorthand to define a *functor*
- It's something that generates a *closure object*

[<***capture list***>] (<***parameters***>)
mutable noexcept
-> <***return type***>
{ <***body***> }

Introduction to Lambdas 2/3

[<***capture list***>] (<***parameters***>) ***mutable noexcept*** -> <***return type***> { <***body***> }

- Capture list: specify the name of local variables that are used in a lambdas (we can specify if they are taken by value, by ref or by move)
- Parameters: the list of parameters taken by lambdas (optional)
- An optional ***mutable***: a useful way to change internal status of the lambdas
- An optional ***noexcept***
- Return type: the type returned by lambdas (almost optional*)
- Body: the body of the lambdas (could be any kind of statements)

** Optional in C++14. Optional in C++11 iff the lambdas is composed by only one line.*

Introduction to Lambdas 3/3

What we do

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

What compiler roughly does!

```
struct lambda0 {
    int operator() (int a, int b) const {
        return a + b;
    }
}
```

Lambdas in C++14

From C++14 new features are introduced:

- We can use generic lambdas:

```
[] (auto x, auto y ) { return x + y; }
```

- Initialized lambda capture
  - We can use ***move*** in capture list:

```
[ v = move (v) ] { /* do something */ }
```

  - We can define and initialize new variable in capture list:

```
[ s = "hello" ] { cout << s << endl; }
```

Recipe 1: Lambdas in STL (as predicate)

#include <algorithm>

There are a lot of STL functions that accept a callable object and so even lambda:

- std::**for_each**
- std::**count**
- std::**count_if**
- std::**find_if**
- std::**sort**

C++11

```
vector<int> v{1, 2, 3, 4, 5};   int i = 0;
for_each(v.cbegin(), v.cend(), [&i] (int n) {
        cout << i++ << ") val: " << n << endl;
});
```

output

```
0) val: 1
1) val: 2
2) val: 3
3) val: 4
4) val: 5
```

Recipe 1: Lambdas in STL (as predicate)

#include <algorithm>

There are a lot of STL functions that accept a callable object and so even lambda:

- std::**for_each**
- std::**count**
- std::**count_if**
- std::**find_if**
- std::**sort**

C++14

```
vector<int> v{1, 2, 3, 4, 5};
for_each(v.cbegin(), v.cend(), [i = 0] (auto n) mutable {
        cout << i++ << ") val: " << n << endl;
});
```

output

```
0) val: 1
1) val: 2
2) val: 3
3) val: 4
4) val: 5
```

Recipe 1: Lambdas in STL (as predicate)

Count the number of even numbers in a vector

```
int n = 0;
for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
        if (*it % 2 == 0) { n++; }
}
```

C++03

```
const auto n = count_if (v.cbegin(), v.cend(), [] (int e){
        return (e % 2 == 0) ;
});
```

C++11

Recipe 2: Lambdas in STL (as deleter)

Deleter function

```
template <typename T>
struct mydeleter {
   void operator()(T* e) { delete[] e; }
};
boost::shared_ptr<int> ptr (new int[10], mydeleter<int>());
```

C++03

```
std::shared_ptr<int> ptr (new int[10], [](int *e){
   delete[] e;
});
```

C++11

Recipe 2: Lambdas in STL (as deleter)

Deleter function

```
template <typename T>
struct mydeleter {
   void operator()(T* e) { delete[] e; }
};
boost::shared_ptr<int> ptr (new int[10], mydeleter<int>());
```

C++03

```
std::shared_ptr<int> ptr (new int[10], std::default_delete<int[]>());
```

C++11

Recipe 3: Return a const default value

We want initialize a "**const**" variable with a default value in some case and with another value in other case. Consider this code:

```
int const_val = some_default_value;
if (some_condition_is_true)
{
        // ... Do some operations and calculate the value
        // of const_val ...
        const_val = calculate();
}
...
const_val = 1000; // oops, const_val CAN be modified later!
```

C++03

Recipe 3: Return a const default value

In C++11 we can use a lambdas to encapsulate all the “logic” and then assign a ***really*** const value to a variable

```
const auto const_val = [&] () {
      if (some_condition_is_true)
      {
            return calculate();
      }
      return some_default_value;
}();   // ← execute lambda here!
```

C++11

Recipe 4: Lambdas as while condition

If you look around your code you can find a lot of this:

```
while (true)
{
    m.lock();               // acquire lock
    if (s.empty())          // if s is empty
    {
        m.unlock();         // unlock the mutex
        break;              // exit from while
    }
    auto e = s.top();       // take the first element
    s.pop();                // remove the element from stack
    m.unlock();             // unlock the mutex
    consume(e);             // consume the element
}
```

C++03

1. A while (true) isn't so expressive
2. Mutex m might be left locked
3. We unlock the mutex in two place

Recipe 4: Lambdas as while condition

Now, using C++11 & lambdas, we can do better!

```
while ([&]{
        unique_lock<mutex> g(m);
        if (s.empty()) { return false; } // exit from scope & release mutex
        else /* body here */
        {
           auto e = s.top();
           s.pop();
           g.unlock();         // release the mutex
           consume(e);      // consume element
           return true;      // exit from lambdas
        }
     }() /* execute the lambdas */ )
  { /* empty body! */ }
```

C++11

1. We use unique_lock to manage mutex
2. We explicit unlock the mutex in only one place

Recipe 4.1: Lambdas as while condition

Problem: Print elements of a vector using a comma as separator

```
template <typename T>
void print_vector(const vector<T>& v)
{
   typename vector<T>::const_iterator it;
   for (it = v.begin(); it != v.end(); ++it)
   {
      cout << *it;
      if (it != v.end() - 1)
         cout << ", ";
   }
   cout << endl;
}
```

C++03

Recipe 4.1: Lambdas as while condition

Problem: Print elements of a vector using a comma as separator

```
template <typename T>
void print_vector(const vector<T>& v)
{
   auto it = begin(v);
   while ([&](){
      if (it == begin(v)) { return true; }
      else if (it == end(v)) {cout << endl; return false; }
      else { cout << ", "; return true; }
   }() /*execute the lambda*/ )
  {
      cout << *it;  ++it;
   }
}
```

C++11

```
vector<int> v{1, 2, 3, 4, 5, 8};
print_vector(v);
```

output

```
1, 2, 3, 4, 5, 8
```

Recipe 5: RAII+Lambdas -> ScopeGuard 1/5

The idea is to take actions when something goes wrong …

```
{
    <TAKE RESOURCE>
    <TAKE DATA>
    <WRITE DATA IN SOME PLACE (file, db, network)>  // BOOMMM!!!!
    <RELEASE RESOURCE>
}
```

Recipe 5: RAII+Lambdas -> ScopeGuard 2/5

The idea is to use RAII (Resource Acquisition Is Initialization) + Lambdas expression

```
template <typename F>
struct ScopeGuard
{
   ScopeGuard(F f) : active_(true), guard(move(f)) {}
   ScopeGuard(ScopeGuard&& rhs) :
         active_(rhs.active_), guard(move(rhs.guard))
  {
      rhs.dismiss();
  }
   void dismiss() { active_ = false; }
   ~ScopeGuard() { if (active_) { guard(); } }
```

Continued ….

Recipe 5: RAII+Lambdas -> ScopeGuard 3/5

The idea is to use RAII (Resource Acquisition is Initialization) + Lambdas expression

```
    ScopeGuard() = delete;
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;

private:
    bool active_;
    F guard;
};
```

Recipe 5: RAII+Lambdas -> ScopeGuard 4/5

We can add a helper function that generate an instance of ***ScopeGuard***:

```
template <typename F>
ScopeGuard<F> scopeGuard(F&& fun)
{
   return ScopeGuard<F>(forward<F>(fun));
}
```

Recipe 5: RAII+Lambdas -> ScopeGuard 5/5

We can use it in this way:

```
{
    db foo("data.dat");
    auto g = scopeGuard([&](){                    /*Here we define our ScopedGuard
      db.rollback();                               and the body of lambdas */
      cerr << "error!" << endl;
   });
...
   recv_data();    //may throw
    /* write data inside foo */
    db.commit();
    g.dismiss(); // ok disable the cleaner lambda
}  /* end of scope */
```

Recipe 6: Lambdas as Macro 1/6

C++03

```
UInt32 a_index;  vector<UInt8> v{};  string ip_src, ip_dst, port_src, port_dst;

if ((Status_Ok == blob.get_index_fromstring(".ip_src", a_index)) && blob.get_present_flag(a_index))
   ip_src = render.getAsString(a_index, v);
else
   return;
if ((Status_Ok == blob.get_index_fromstring(".port_src", a_index)) && blob.get_present_flag(a_index))
   port_src = render.getAsString(a_index, v);
else
   return;
if ((Status_Ok == blob.get_index_fromstring(".ip_dst", a_index)) && blob.get_present_flag(a_index))
   ip_dst = render.getAsString(a_index, v);
else
   return;
if ((Status_Ok == blob.get_index_fromstring(".port_dst", a_index)) && blob.get_present_flag(a_index))
   port_dst = render.getAsString(a_index, v);
else
   return;
```

Recipe 6: Lambdas as Macro 2/6

In C++11 we can use lambdas (***get*** in the example) to simplify all the code

C++11

```
auto get = [&](const string& k) -> string {
      UInt32 a_index;
      string r{};
      vector<UInt8> v{};

      if ((Status_Ok == blob.get_index_fromstring(k, a_index))
            && blob.get_present_flag(a_index))
      {
         r = render.getAsString(a_index, v);
      }
      return r;
 };
```

Recipe 6: Lambdas as Macro 3/6

To extract the data it's now possible to write something like that:

```
string ip_src = get(".ip_src");
string port_src = get(".port_src");

string ip_dst = get(".ip_dst");
string port_dst = get(".port_dst");
```

C++11

1. Don't repeat yourself
2. Code more readable

Great! But we've just forgotten to exit from function if a field is missing.
So we have to wrapper with an ***if*** (…) ***return*** our code or …
We can define an helper function (using variadic template feature of C++11)
that run the "next" lambdas *iff* the previous one has returned true!

Recipe 6: Lambdas as Macro 4/6

```
template <typename F>
void run_if(F f)
{
    f();
}

template <typename F, typename ...Funs>
void run_if(F f, Funs ...funs)
{
    if ( f() ) // if f return true … carry on
    {
        run_if(forward<Funs>(funs)...); // call run_if with others Funs
    }
}
```

C++11

Recipe 6: Lambdas as Macro 5/6

We have to redefine our get function to return a bool:

C++11

```
auto get = [&](const string& k, string& val) -> bool {
      UInt32 a_index;
      vector<UInt8> v{};
      if ((Status_Ok == blob.get_index_fromstring(k, a_index))
            && blob.get_present_flag(a_index))
      {
         val = render.getAsString(a_index, v);
         return true;
      }
      else
         return false;
};
```

Recipe 6: Lambdas as Macro 6/6

```
string ip_src, ip_dst, port_src, port_dst;

run_if (
    [&] { return get(".ip_src", ip_src); }
    [&] { return get(".port_src", port_src); }
    [&] { return get(".ip_dst", ip_dst); }
    [&] { return get(".port_dst", port_dst); },
    [&] { /*success!: use here ip_{src,dst} & port_{src,dst} */ }
);
```

C++11

Recipe 7: Lambdas as message passing system

In GUI application (classical or web) is very common to bind a message to a function and then send a message to the main-thread in order to execute this particular function in that thread.

General speaking lambdas permit us to implement this kind of mechanism in a lot of places and in a very simple way. In the next slides we'll see a Log example taken from Herb Sutter talk on Concurrency.

The goal is very straightforward: PERFORMANCE!

std::thread([]{ cout << "I'm leaving in a separate thread!" << endl; });
Is a new way to define a thread.

Recipe 7: Lambdas as message passing system

Imagine a simple Logger class with a very special method: ***send_log***

```
class Logger
{
public:
   Logger() : wth{} {}
   void info(string msg)  { wth.send_log([=]{ cout << "[INFO]"  <<  msg << endl; }); }
   void debug(string msg) { wth.send_log([=]{ cout << "[DEBUG ]"  <<  msg << endl; }); }

   ~Logger() { wth.stop(); }
private:
   Worker wth;
};
```

Take a Lambdas!

- No explicit mutex -> no blocking!
- High customizable function

Recipe 7: Lambdas as message passing system

```
class Worker {
public:
   Worker() : stopped{false}  {
      wth = thread([&] {
         while (!stopped)
         {
            unique_lock g(m_);
            if (!funs.empty())
            {
               auto f = move(funs.front());
               funs.pop_front();
               g.unlock();
               f(); //execute
            }
            else
            {
               g.unlock();
               this_thread::sleep_for (chrono::milliseconds(50));
            }
         }
         cout << "Exit from worker!" << endl;
      });
   }
```

```
   void stop() { send_log([this]{ stopped = true; }); }

   template <typename F>
   void send_log(F&& f)
   {
     lock_guard<std::mutex> g(m);
     funs.push_back(forward<F>(f));
   }

   ~Worker() { stop(); wth.join(); }

   Worker(const Worker& w) =delete;
   Worker& operator=(const Worker& w) =delete;

private:
   deque<function<void(void)>> funs;
   mutable mutex m_;
   bool stopped;
   thread wth;
};
```

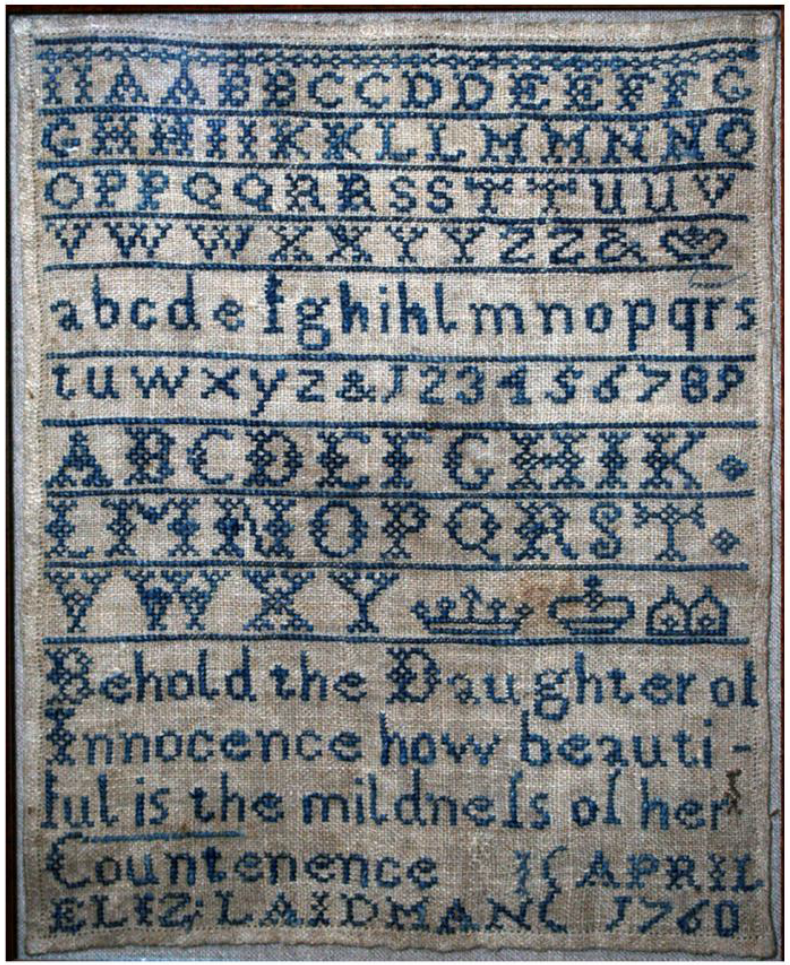
- We can use a Lambdas as body of a thread
- We can store lambdas in a container
- We can pass lambdas between threads

Recipe 7: Lambdas as message passing system

Finally example: two threads that “send” a lambdas message on a logger

```
Logger log;
log.info("Hello World!");

auto th1 = thread([&log]{
   for (auto i = 0; i < 150; ++i) {log.info("Th A");}
});

auto th2 = thread([&log]{
   for (auto i = 0; i < 150; ++i) {log.info("Th B");}
});

log.debug(“after threads starter!");

th1.join();
th2.join();
```

Effective Modern C++ (Scott Meyers)



Chapter 5 Lambda Expression

- Item 25: Avoid default capture modes.
- Item 26: Keep closures small.
- Item 27: Prefer lambdas to std::bind.
- Item xx: dont' use uniform initialitazion inside lambdas capture list

Some references

- The C++ Programming Language 4th edition (Bjarne Stroustrup)
- C++11 Rocks (Alex Korban)

- Lambdas, Lambdas Everywhere

http://herbsutter.com/2010/10/07/c-and-beyond-session-lambdas-lambdas-everywhere/

- Scott Meyers blogpost

http://scottmeyers.blogspot.it/

- Fun with Lambdas: C++14 Style (part 1, 2)

http://cpptruths.blogspot.it/2014/03/fun-with-lambdas-c14-style-part-1.html
http://cpptruths.blogspot.it/2014/03/fun-with-lambdas-c14-style-part-2.html

- Mix RAII & lambdas for deferred execution (Marco Arena)

http://marcoarena.wordpress.com/2012/08/27/mix-raii-and-lambdas-for-deferred-execution/

Italian C++ Community

www.italiancpp.org

Thanks!