Back to Basics:
Lambdas

NICOLAI JOSUTTIS

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif

C++

©2021 by josuttis.com

Modern C++

Lambdas

Helper Function as Sorting Criterion

```
class Person {
 public:
  std::string firstname() const;
  std::string lastname() const;
  ...
  friend bool operator< (const Person&, const Person&);   // or op<=> since C++20
};

bool lessPerson(const Person& p1, const Person& p2) {
  // sort ascending to the last name or if equal ascending to the first name:
  return p1.lastname() < p2.lastname() ||
         (p1.lastname() == p2.lastname() && p1.firstname() < p2.firstname());
}

std::vector<Person> coll;
...

// sort elements with operator < :
std::sort(coll.begin(), coll.end());   // pass begin and end of elements to sort
...

// sort elements with a special sorting criterion:
std::sort(coll.begin(), coll.end(),    // elements to sort
          lessPerson);                 // sorting criterion
```

Cannot be defined inside functions
- Need good name
- May be hard to maintain

Lambdas as Sorting Criterion (since C++11)

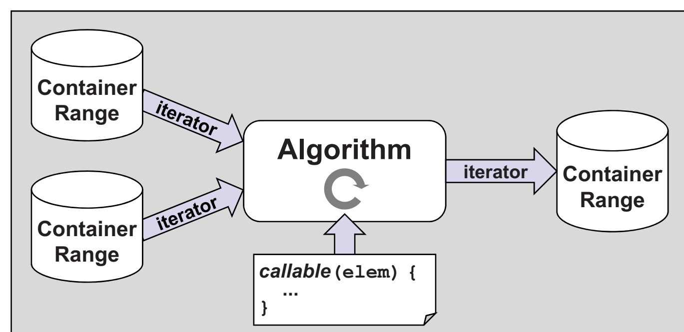
```
class Person {
 public:
  std::string firstname() const;
  std::string lastname() const;
  ...
  std::string getCustNo() const;   // return customer number
  ...
};

std::vector<Person> coll;
...
// sort according to the name:
std::sort(coll.begin(), coll.end(),                    // elements to sort
          [] (const Person& p1, const Person& p2) {    // sorting criterion
            // sort ascending to the last name or if equal ascending to the first name:
            return p1.lastname() < p2.lastname() ||
                   (p1.lastname() == p2.lastname() && p1.firstname() < p2.firstname());
          });

// sort according to the customer number:
std::sort(coll.begin(), coll.end(),                    // elements to sort
          [] (const Person& p1, const Person& p2) {    // sorting criterion
            return p1.getCustNo() < p2.getCustNo();
          });
```


Standard Template Library Architecture

- **Data structures and algorithms**
  - Combine different data structures with different algorithms
  - Some combinations can be a problem (e.g., can't sort read-only elements)
- **Iterators as glue interface**
  - Passed as half-open ranges (including begin, excluding end)
- **Generic rather than OO approach**
  - Everything that *behaves* like a container, *is* a container
  - Everything that *behaves* like an iterator, *is* an iterator
  - Everything you can *call*, *is* a *callable*



Algorithms Using Helper Functions

```
bool isOdd (int value)
{
  return value % 2 != 0;
}
```

Output:

```
odd elems: 6
first odd elem: 15
```

```
std::vector<int> coll{0, 8, 15, 42, 11, 1, 77, -1, 3};

// count number of elements with odd value:
int num = std::count_if(coll.begin(), coll.end(),   // range
                        isOdd);                      // criterion
std::cout << "odd elems: " << num << '\n';

// find position of first element with odd value (returns iterator):
auto pos = std::find_if(coll.begin(), coll.end(),   // range
                        isOdd);                      // criterion
if (pos != coll.end()) {                             // use position if any
  std::cout << "first odd elem: " << *pos << '\n';
}
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Lambdas

- **Lambda: Function defined on the fly**

Output:

```
odd elems: 6
first odd elem: 15
```

```
std::vector<int> coll{0, 8, 15, 42, 11, 1, 77, -1, 3};

// count number of elements with odd value:
int num = std::count_if(coll.begin(), coll.end(),   // range
                        [] (int elem) {              // criterion (defined on the fly)
                          return elem % 2 != 0;
                        });
std::cout << "odd elems: " << num << '\n';

// find position of first element with odd value:
auto pos = std::find_if(coll.begin(), coll.end(),   // range
                        [] (int elem) {              // criterion (defined on the fly)
                          return elem % 2 != 0;
                        });
if (pos != coll.end()) {                             // use position if any
  std::cout << "first odd elem: " << *pos << '\n';
}
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Lambdas

- **Lambda: Function object defined on the fly**

Output:

```
odd elems: 6
first odd elem: 15
```

```
std::vector<int> coll{0, 8, 15, 42, 11, 1, 77, -1, 3};

auto isOdd = [] (int elem) {
               return elem % 2 != 0;
             };

// count number of elements with odd value:
int num = std::count_if(coll.begin(), coll.end(),   // range
                        isOdd);                     // criterion
std::cout << "odd elems: " << num << '\n';

// find position of first element with odd value:
auto pos = std::find_if(coll.begin(), coll.end(),   // range
                        isOdd);                     // criterion
if (pos != coll.end()) {                            // use position if any
  std::cout << "first odd elem: " << *pos << '\n';
}
```

idOdd is an object that can be used like a function

C++
©2021 by josuttis.com

josuttis | eckstein IT communication

Combining Lambdas with Standard Algorithms

Output:

```
Berlin Cologne Here LA London are cities some
are Berlin cities Cologne Here LA London some
```

```
int main()
{
  std::vector<std::string> coll{"Here", "are", "some", "cities", "Berlin", "LA",
                                "London", "Cologne"};

  std::sort(coll.begin(), coll.end());
  print(coll);

  std::sort(coll.begin(), coll.end(),                            // range to sort
            [] (const std::string& s1, const std::string& s2) {  // sort criterion
              return std::lexicographical_compare(s1.begin(), s1.end(),   // string as 1st range
                                                  s2.begin(), s2.end(),   // string as 2nd range
                                                  [] (char c1, char c2) { // compare criterion
                                                    return std::toupper(c1)
                                                            < std::toupper(c2);
                                                  });
            });
  print(coll);
}
```

Compare elements of two containers

C++
©2021 by josuttis.com

josuttis | eckstein IT communication

Combining Lambdas with Standard Range Algorithms (C++20)

Output:

```
Berlin Cologne Here LA London are cities some
are Berlin cities Cologne Here LA London some
```

```
int main()
{
  std::vector<std::string> coll{"Here", "are", "some", "cities", "Berlin", "LA",
                                "London", "Cologne"};

  std::ranges::sort(coll);
  print(coll);

  std::ranges::sort(coll,                                      // range to sort
          [] (const std::string& s1, const std::string& s2) {  // sort criterion
            auto toUpper = [](char c){return std::toupper(c);};
            return std::ranges::lexicographical_compare(s1,          // string as 1st range
                                                        s2,          // string as 2nd range
                                                        std::less{}, // compare criterion
                                                        toUpper,     // projection for s1 elem
                                                        toUpper)     // projection for s2 elem
          });
  print(coll);
}
```


C++14: Named Generic Lambdas

```
// define a generic lambda object:
auto twice = [] (const auto& x) {
                 return x + x;
             };

// and call/use it:
auto i = twice(3);                    // i is int => 6
auto d = twice(1.7);                  // d is double => 3.4
auto s = twice(std::string{"hi"});    // s is std::string => "hihi"
auto t = twice("hi");                 // Error: const char[3] + const char[3]

// print all elements of any kind:
for (const auto& elem : coll) {
  std::cout << "- " << twice(elem) << '\n';
}

// replace all elements of coll by the sum of adding them to themselves:
std::transform(coll.begin(), coll.end(),  // source range
               coll.begin(),              // destination range
               twice);                    // transformation
```

Lambda is compiled for different parameter types

Lambdas Without Captures

- **Lambdas with no captures**
  - Can be used as ordinary function pointers
  - Can be used as sorting criterion / hash function type (since C++20)

```
#include <cstdlib>          // for atexit()
#include <unordered_set>
...

int main()
{
  std::atexit([](){                       // lambda to be called on regular exit
                std::cout << "good bye\n";
              });

  auto hashFunc = [](const auto& obj) {   // lambda to be used as hash function
                    return ...;
                  };
  ...
  std::unordered_set<MyType, decltype(hashFunc)> coll;
  ...
}
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Modern C++

Lambdas
as Better Functions

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Lambdas are More Than Functions

- **Local functions**
- **Convenient way to define functions at runtime**
  - Functions with state

```
bool less7(int v)
{
  return v < 7;
}

bool less8(int v)
{
  return v < 8;
}

count_if(c.begin(), c.end(),
         less7);

count_if(c.begin(), c.end(),
         less8);

void foo(int max) {
  count_if(c.begin(), c.end(),
           lessMax);    // ???
}
```

```
count_if(c.begin(), c.end(),
         [] (int v) {
           return v < 7;
         });

count_if(c.begin(), c.end(),
         [] (int v) {
           return v < 8;
         });

void foo(int max) {
  count_if(c.begin(), c.end(),
           [max] (int v) {
             return v < max;
           });
}
```

function (object) created on the fly

Lambdas Captures

- **Lambdas**
  - *Capture* **behavior parameters**
    - Functionality can depend on run-time parameters
  - to deal with **call parameters**

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> coll{0, 8, 15, 42, 11, 1, 77, -1, 3};
  int max = 30;
  …
  // count number of elements less or equal max
  int num = std::count_if (coll.begin(), coll.end(),   // range
                           [max] (int elem) {          // criterion (defined on the fly)
                             return elem < max;
                           });
  std::cout << "elems <" << max << ": " << num << '\n';
}
```

Output:

```
elems <30: 7
```

Generic Lambdas (since C++14)

- **Generic Lambdas**
  - Call arguments may have a generic type
    - `auto`, `const auto&`, ...
  - Not possible in normal functions before C++20

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> coll{0, 8, 15, 42, 11, 1, 77, -1, 3};
  int max = 30;
  …
  // count number of elements less or equal max
  int num = std::count_if (coll.begin(), coll.end(),   // range
                           [max] (auto elem) {         // criterion (defined on the fly)
                             return elem < max;
                           });
  std::cout << "elems <" << max << ": " << num << '\n';
}
```

Output:

```
elems <30: 7
```


Using Lambdas

- **Lambdas can be used as objects**
  - Local functions (behavior defined at runtime)
  - Type has to be `auto` (type name is platform dependent)

```
void processInvoice(const Invoice& iv)
{
  double tax = getTax(iv);
  auto plusTax = [tax] (double d) {
                   return d * (1 + tax);
                 };

  // add tax at various places of the invoice:
  std::cout << plusTax(iv.getSum()) << '\n';   // prints result of sum * (1 + tax)

  for (const auto& item : iv) {
    std::cout << plusTax(item) << '\n';        // prints items plus tax
  }

  std::transform(iv.begin(), iv.end(),         // transform elements of the invoice
                 iv.begin(),                   // writing back the result
                 plusTax);                     // to add tax
}
```

Local function object that adds tax for invoice `iv`

Using Lambdas

- **Lambdas can be used as objects**
  - Functions passed around (behavior defined at runtime)
  - Multiple "functions" can co-exist

```
auto makeAddTaxFunc(const Invoice& iv)
{
  double tax = getTax(iv);
  return [tax] (double d) {
           return d * (1 + tax);
         };
}

Invoice iv1, iv2;
…
auto plusTax1 = makeAddTaxFunc(iv1);
auto plusTax2 = makeAddTaxFunc(iv2);
…
std::cout << plusTax1(iv1.getSum()) << '\n';
std::cout << plusTax2(iv2.getSum()) << '\n';
…
```

Return function object
that adds tax for invoice `iv`

Modern C++

Lambdas
are Function Objects

Lambdas are Function Objects

- **Lambdas are function objects defined "on the fly"**
  - Simple way to define objects that can be used like functions
    - Using `operator()`
- **have a "**unique, unnamed non-union class type**"**
  - "*closure type*"

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

Usage:
```
int i = 4;
...
i = add(17, i);
```

calls:
```
add.operator()(17, i);
```

has the effect of:

```
class lambda??? {
  public:
    lambda???();   // only callable by compiler before C++20
    int operator() (int x, int y) const {
      return x + y;
    }
};
auto add = lambda???{};
```

For each lambda in the source code, the compiler
- **defines a class** ("closure type")
- **and creates an object of this class**

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

Lambdas are Function Objects

```
while (...) {
  int min, max;
  ...                  // set min and max
  p = std::find_if(coll.begin(), coll.end(),   // find element between min and max
                   [min, max](int i) {
                     return min <= i && i <= max;
                   });
}
```

has the effect of:

```
class lambda??? {
  private
    int _min, _max;
  public:
    lambda???(int min, int max)   // only calla
     : _min{min}, _max{max} {
    }
    bool operator() (int i) const {
      return _min <= i && i <= _max;
    }
};
```

```
template<typename Iter, typename Callback>
Iter find_if (Iter beg, Iter end,   // range
              Callback op)          // predicate
{
  for (Iter pos = beg; pos != end; ++pos) {
    if (op(*pos)) {   // if op() yields true for element
      return pos;     // return iterator to element
    }
  }
  return end;
}
```

calls:
```
op.operator()(*pos);
```

```
while (...) {
  int min, max;
  ...
  p = std::find_if(coll.begin(), coll.end(),
                   lambda???{min, max});
}
```

For each lambda in the source code, the compiler
- **defines a class** ("closure type")
- **and creates an object of this class**

C++

©2021 by josuttis.com

josuttis | eckstein IT communication

C++14: Generic Lambdas

- **Lambdas for generic parameter types**
  - Using `auto`
  - Defines function template member
    - Usual template deduction rules apply

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

calls:
`plus.operator()<double,int>(7.7, i);`

Usage:

```
int i = 42;
...
double d = plus(7.7, i);

std::string s{"hi"};
...
std::cout << plus("s: ", s);
```

has the effect of:

```
class lambda??? {
  public:
    lambda???();   // only callable by compiler before C++20
    template<typename T1, typename T2>
    auto operator() (T1 x, T2 y) const {
      return x + y;
    }
};

auto plus = lambda???{};
```

For each lambda in the source code, the compiler
- defines a class ("closure type")
- and creates an object of this class

Generic Functions vs. Generic Lambdas

- **Generic lambdas since C++14:**

```
auto printLmbd = [] (const auto& coll) {
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 };
```

Function object with generic `operator()`

- **Generic functions:**

```
template<typename T>
void printFunc(const T& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Function template (generic before the call)

```
std::vector<int> v;
...
printFunc(v);
printLmbd(v);
printFunc<std::string>("hello");   // OK
printLmbd<std::string>("hello");   // ERROR

call(printFunc, v);                // ERROR
call(printFunc<decltype(v)>, v);   // OK
call(printLmbd, v);                // OK
```

Generic Functions vs. Generic Lambdas (since C++20)

- **Generic lambdas since C++14:**

```
auto printLmbd = [] (const auto& coll) {
                   for (const auto& elem : coll) {
                     std::cout << elem << '\n';
                   }
                 };
```

Function object with generic `operator()`

- **Generic functions since C++20:**

```
void printFunc(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

Function template (generic before the call)

```
std::vector<int> v;
…
printFunc(v);
printLmbd(v);
printFunc<std::string>("hello");    // OK
printLmbd<std::string>("hello");    // ERROR

call(printFunc, v);                 // ERROR
call(printFunc<decltype(v)>, v);    // OK
call(printLmbd, v);                 // OK
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Modern C++

Capturing in Detail

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Lambdas: Capture by Value

```
#include <iostream>
#include <deque>
#include <algorithm>

int main()
{
  std::deque<int> coll{1, 3, 19, 5, 13,
                       7, 11, 2, 17};

  int min = 5;
  int max = 12;
  ...
  // find position of first element greater than x and less than y:
  auto pos = std::find_if(coll.begin(), coll.end(),
                          [min, max](int i) {     // capture x and y
                            return min <= i && i <= max;
                          });
  if (pos != coll.end()) {
    std::cout << "first found element: " << *pos << '\n';
  }
}
```

Objects captured by value are by default **read-only copies**
- to make the lambda **stateless** (can't change its behavior)

C++
©2021 by josuttis.com

josuttis | eckstein IT communication

Capturing in Detail

- **Capturing is performed when the lambda is created**
  - Local read-only copy (unless passed by reference)

```
std::string prefix = "elem: ";

auto printval = [prefix] (int i) {     // captures local copy
                  std::cout << prefix << i << '\n';
                };

prefix = "value: ";   // no impact on printval

printval(17);         // prints:  elem: 17
```

C++
©2021 by josuttis.com

josuttis | eckstein IT communication

Lambdas: Capture by Reference

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
   std::vector<double> coll;
   ...

   // process average value of all elements
   double sum = 0.0;
   std::for_each (coll.begin(), coll.end(),
                  [&sum](double d) {          // [&] would catch all by reference
                      sum += d;
                  });
   std::cout << "average: " << sum / coll.size() << '\n';
}
```

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

C++14: Lambda Capture Initializers

- **Since C++14, you can pass initializers to lambda captures**
  - Enables adding local state
  - Enables capturing by const reference with `std::as_const()`

```
// compute price function on the fly:
auto price = [disc = getDiscount(cust)] (auto item) {  // compute discount once for cust
               return getPrice(item) * disc;           // and use it for all function calls
             };
...
for (const auto& item : order) {
  std::cout << "your price: " << price(item) << '\n';  // use computed discount
}
```

Initialization happens on lambda definition

```
// capture by const&:
auto coll = ...;
...
auto dealWithColl = [&coll = std::as_const(coll)] {    // ensure coll can't be modified
                       ...
                    };

dealWithColl();   // coll can be used inside the lambda but is not mutable
...
```

Available since C++17

C++

©2021 by josuttis.com

josuttis | eckstein
IT communication

Capture by Value

- **Lambdas are stateless by default**
  - Not allowed to modified values captured by-value

```
auto changed = [prev = 0] (auto val) {
                  bool changed = prev != val;
                  prev = val;       // ERROR: prev is read-only copy
                  return changed;
                };
```

C++
©2021 by josuttis.com

josuttis | eckstein
IT communication

Mutable Lambdas

- **Lambdas are stateless by default**
  - Not allowed to modified values captured by-value
- **`mutable` makes them stateful** (modifications allowed)

```
auto changed = [prev = 0] (auto val) mutable {
                  bool changed = prev != val;
                  prev = val;       // OK due to mutable
                  return changed;
                };

std::vector<int> coll{7, 42, 42, 0, 3, 3, 7};

std::copy_if(coll.begin(), coll.end(),
             std::ostream_iterator<int>{std::cout, " "},
             changed);

std::copy_if(coll.begin(), coll.end(),
             std::ostream_iterator<int>{std::cout, " "},
             changed);

changed(7);
std::copy_if(coll.begin(), coll.end(),
             std::ostream_iterator<int>{std::cout, " "},
             changed);
```

Output:

```
7 42 42 0 3 3 7
7 42 0 3 7
7 42 0 3 7
42 0 3 7
```

Standard algorithms take callables by value
- Operate on a **copy** of `changed`

C++
©2021 by josuttis.com

josuttis | eckstein
IT communication

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis