Concepts of C++ Programming

Lecture 9: Algorithms, Functions, and Lambdas

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Function Objects[119]

- Functions are no objects
  - Cannot be passed as parameters, no state, etc.
- *FunctionObject* named requirement for some type `T`:
  - `T` must be an object
  - For an instance `f` of `T`: `f(args)` must be defined
- Also referred to as *functors*

[119] https://en.cppreference.com/w/cpp/named_req/FunctionObject

Function Pointers[120]

- Functions are not objects, but have addresses
  - Location in memory where the code resides

⇝ Allows declaration of function pointers: `ret-ty (*identifier)(arg-tys)`

- Function pointers satisfy requirements of *FunctionObject*

```
int add(int p1, int p2) { return p1 + p2; }
int callFn(int (*fn)(int, int), int p1, int p2) {
  return fn(p1, p2); // automatic dereference, equivalent to (*fn)(p1, p2)
}

int main() {
  // add gets implicitly converted to function pointer, equiv. to &add
  int res = callFn(add, 1, 1); // 2
}
```

[120] https://en.cppreference.com/w/cpp/language/pointer

Function Pointers

Quiz: What is the output of the program?

```
#include <print>
int fn(int p, int q = 1) { return p * q; }
int callFn(int (*fn)(int), int p) {
  return (*fn)(p);
}
int main() {
  std::println("{}", callFn(fn, 1));
}
```

A. Compile error: `fn` has type `int(int, int)`

B. Compile error: cannot dereference function pointer

C. Undefined behavior: default arguments become undefined

D. Guaranteed output: `1`

Member Function Pointers

- Non-static member functions take an implicit parameter, `this`

$\Rightarrow$ Special pointers to members of a class

```
struct S {
  int a;
  int x(int b, int c) { return a + b + c; }
};

int main() {
  int (S::* memberFnPtr)(int, int) = &S::x;

  S s{32};
  int r1 = (s.*memberFnPtr)(2, 8); // 42
  S* sp = &s;
  int r2 = (sp->*memberFnPtr)(2, 8); // 42
}
```

Stateful Function Objects

- So far: functions are stateless have to pass all state as parameters
- Function objects can be implemented as regular class

$\rightsquigarrow$ Keep arbitrary state as members

```
struct Adder {
  int v;
  int operator()(int param) const {
    return param + v;
  }
};
int main() {
  Adder add42{42};
  int a = add42(10); // 52
  add42.v = 10;
  int b = add42(0); // 10
}
```

Stateful Function Objects

Quiz: What is the output of the program?

```
#include <print>
struct Accumulator {
  int v;
  int operator()(int param) {
    return v += param;
  }
};
int main() {
  Accumulator acc{10};
  int a = acc(10) + acc(20);
  int b = acc(1) + acc.v;
  std::println("{}/{}", a, b);
}
```

A. (compile error) B. (multiple outputs correct) C. 60/81 D. 60/82 E. 70/82

Lambda Expressions (1)[121]

- Function pointers so far have some limitations:
  - Cannot have "local" functions within other functions
  - Cannot capture environment – have to pass all state as parameters
- Lambda expressions: construct closure (function with environment)
- `[captures](params) -> ret-ty { body }`
  - Captures specify parts of environment that should be stored, can be empty
  - Lambda without captures are implicitly converted to function pointers
  - Return type can be omitted if deducible from return statements in body
- Lambda expressions have a unique, unnamed type

⇝ Rely on `auto`/argument deduction when assigning lambda

[121] https://en.cppreference.com/w/cpp/language/lambda

Lambda Expressions

```
int callFn(int (*fn)(int, int), int p1, int p2) {
  return fn(p1, p2); // automatic dereference, equivalent to (*fn)(p1, p2)
}

int main() {
  auto lambda = [](int p1, int p2) {
    return p1 + p2;
  };

  // lambda gets implicitly converted to function pointer
  int res = callFn(add, 1, 1); // 2
  int foo = lambda(2, 4); // 6
}
```

Lambda Expressions

- Lambda types are really unique:

```
// ERROR: Compilation will fail due to ambiguous return type
auto getFunction(bool first) {
  if (first) {
    return []() {
      return 42;
    };
  } else {
    return []() {
      return 42;
    };
  }
}
```

Lambda Captures

- Capture: specify what constitutes the state of a lambda expression
- Refer to automatic variables/`this` in surrounding scopes
- Captured variables can be used in lambda like regular variables

- Capture *by-copy*: `[var]` or `[var = initializer]`
  - Variable *copied* into lambda state on creation
- Capture *by-reference*: `[&var]` or `[&var = initializer]`
  - Variable *reference* stored lambda state on creation
- Each variable may be captured at most once

Lambda Captures: Example

```
int main() {
  int i = 42;

  // Lambda stores a copy of i
  auto lambda1 = [i]() { return i + 42; };
  // Lambda stores a reference to i
  auto lambda2 = [&i]() { return i + 42; };

  i = 0;

  int a = lambda1(); // a = 84
  int b = lambda2(); // b = 42
}
```

Lambda Capture: Default Captures

- First capture can be a capture-default: = (copy)/& (reference)
- Allows use of any variable in surrounding scope
- Diverging capture types can be specified for individual variables

```
int main() {
  int i = 0;
  int j = 42;

  auto lambda0 = [=](){}; // j and i by-copy
  auto lambda1 = [&](){}; // j and i by-reference
  auto lambda2 = [&, i](){}; // j by-reference, i by-copy
  auto lambda3 = [=, &i](){}; // j by-copy, i by-reference
  auto lambda4 = [&, &i](){}; // ERROR: non-diverging capture types
  auto lambda5 = [=, i](){}; // ERROR: non-diverging capture types
}
```

Lambda Captures

Quiz: What is problematic about this code?

```
auto createAdder(int n) {
  return [&](int x) {
    return x + n;
  };
}
int main() {
  auto add42 = createAdder(42);
  return add42(10);
}
```

A. Compile error: return type must be named ⇒ cannot return lambda.

B. Compile error: cannot copy lambda.

C. Undefined behavior: function call uses a dangling reference.

D. No problem: the program exits with status 52.

Lambda Captures

Quiz: What is the output of the program?

```
struct Too {
  int x;
  auto makeBuzzer() {
    return [*this](int y) { return x + y; };
  }
};
int main() {
  auto buzzer = Too{32}.makeBuzzer();
  std::println("{}", buzzer(10));
}
```

A. Compile error: cannot capture `*this`.

B. Compile error: `x` is not captured.

C. Undefined behavior: `x` used after `Too` got destroyed.

D. Guaranteed output: 42

Lambda Capture: this

- *this (copy)/this (reference)

```
struct Foo {
  int i = 0;

  void bar() {
    auto lambda1 = [*this]() { return i + 42; };
    auto lambda2 = [this](){ return i + 42; };

    i = 42;

    int a = lambda1(); // a = 42
    int b = lambda2(); // b = 84
  }
};
```

std::function[122]

- `std::function`: general-purpose wrapper for all callable targets
- Allows storing, copying, and calling the wrapped target
- Often adds considerable overhead $\rightsquigarrow$ **avoid where possible**

```
#include <functional>

std::function<int()> getFunction(bool first) {
  int a = 14;
  if (first)
    return [=]() { return a; };
  else
    return [=]() { return 2 * a; };
}

int main() {
  return getFunction(false)() + getFunction(true)(); // 42
}
```

[122] https://en.cppreference.com/w/cpp/utility/functional/function

Passing Function Objects as Parameters

```
#include <functional>

int bad(int (*fn)()) { return fn(); }
int slow(const std::function<int()>& fn) { return fn(); }
template <typename Fn> int good(Fn&& fn) { return fn(); }

struct Functor {
    int operator()() { return 42; }
};
int main() {
    Functor ftor;
    bad([]() { return 42; }); // OK
    bad(ftor); // ERROR!
    slow([]() { return 42; }); // OK
    slow(ftor); // OK
    good([]() { return 42; }); // OK
    good(ftor); // OK
}
```

Quiz: Have you filled out the lecture evaluation?

A. Yes!

B. Of course!

C. Not yet, but I really, *really* promise to do it later today.

Algorithms Library[123]

- C++ standard library provides several functions for sorting, searching, etc.
- Operations on ranges of elements `[begin, end)`
  - Operate on appropriate iterator types (incl. pointers)
- Mostly in `<algorithm>`, but also `<numeric>`, `<memory>`, `<cstdlib>`

[123] https://en.cppreference.com/w/cpp/algorithm

std::sort[124]

- Sort elements in range [begin, end) on *RandomAccessIterators*
- Elements must be swappable, move-assignable and move-constructible
- $\mathcal{O}(n \log n)$ comparisons, not stable

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

int main() {
  std::vector<unsigned> v = {3, 4, 1, 2};
  std::sort(v.begin(), v.end()); // 1, 2, 3, 4
}
```

[124] https://en.cppreference.com/w/cpp/algorithm/sort

Custom Comparison[125]

- Comparator supplied as functor: `bool cmp(const T&, const T&)`
- Must establish strict weak ordering: `true` iff `a < b`
  - `!(a < a)`, `a < b` $\Rightarrow$ `!(b < a)`, `a < b && b < c` $\Rightarrow$ `a < c`, `!(a < b) && !(b < a)` $\Rightarrow$ `a` $\approx$ `b`

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

int main() {
  std::vector<unsigned> v = {3, 4, 1, 2};
  std::sort(v.begin(), v.end(), [](unsigned lhs, unsigned rhs) {
    return lhs > rhs;
  }); // 4, 3, 2, 1
}
```

[125] https://en.cppreference.com/w/cpp/named_req/Compare

Other Sorting Operations

- `std::sort` is unstable (order of equal-ranked elements not maintained)
- `std::stable_sort` is stable

- `std::partial_sort` to find smallest n elements
  - More efficient if only top-k elements are interesting

- `std::is_sorted` to check whether a range is sorted
- `std::is_sorted_until` to find first unsorted element

- `std::partition` to reorder elements based on result of predicate
- Some others, see reference

Searching – Unsorted[126]

- Find first element, returns iterator
  - `std::find`, `std::find_if`, `std::find_if_not`
- Count matching elements: `std::count`/`std::count_if`
- Search for a range of elements: `std::search`
- Check if condition holds: `any_of`, `all_of`, `none_of`

- Many more operations, see reference

[126] https://en.cppreference.com/w/cpp/algorithm#Non-modifying_sequence_operations

Searching – Unsorted

```
#include <algorithm>
#include <vector>
int main() {
  std::vector<int> v = {2, 6, 1, 7, 3, 7};
  auto res1 = std::find(vec.begin(), vec.end(), 7);
  int a = std::distance(vec.begin(), res1); // 3
  auto res2 = std::find(vec.begin(), vec.end(), 9);
  assert(res2 == vec.end());

  auto res1 = std::find_if(vec.begin(), vec.end(),
      [](int val) { return (val % 2) == 1; });
  int a = std::distance(vec.begin(), res1); // 2
  auto res2 = std::find_if_not(vec.begin(), vec.end(),
      [](int val) { return val <= 7; });
  assert(res2 == vec.end());
}
```

Searching – Sorted

- On sorted ranges, binary search is more efficient
- $\mathcal{O}(\log n)$ for *RandomAccessIterator*
  - $\mathcal{O}(n)$ when called with *ForwardIterator*!

- `std::binary_search` – check whether element is contained
- `std::lower_bound` – iterator to first element $\geq$ search value
- `std::upper_bound` – iterator to first element $>$ search value
- `std::equal_range` – pair of `lower_bound` and `upper_bound`

Searching – Sorted

```
#include <algorithm>
#include <vector>
#include <cassert>
int main() {
  std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};
  assert(true == std::binary_search(v.begin(), v.end(), 3));
  assert(false == std::binary_search(v.begin(), v.end(), 0));

  assert(v.begin()+3 == std::lower_bound(v.begin(), v.end(), 3));
  assert(v.begin() == std::lower_bound(v.begin(), v.end(), 0));
  assert(v.begin()+6 == std::upper_bound(v.begin(), v.end(), 3));
  assert(v.end() == std::upper_bound(v.begin(), v.end(), 4));
}
```

Searching and Sorting

- Sort + binary search useful if:
  - Separated insert and lookup phases
  - Many searches are required
- Sorting might not be a good idea if:
  - Order cannot be changed and would need to make copy
  - There are frequent updated or insertions

Permutations[127]

- Functions for iterating over permutations in lexicographical order
- `std::next_permutation`
  - `false` if permutation was the last permutation (sorted in descending order)
- `std::prev_permutation`
  - `false` if permutation was the last permutation (sorted in ascending order)

```cpp
#include <algorithm>
#include <vector>
#include <cassert>
int main() {
  std::vector<int> v = {1, 2, 3};
  std::next_permutation(v.begin(), v.end()); // true, v == {1, 3, 2}
  std::next_permutation(v.begin(), v.end()); // true, v == {2, 1, 3}
  std::prev_permutation(v.begin(), v.end()); // true, v == {1, 3, 2}
  std::prev_permutation(v.begin(), v.end()); // true, v == {1, 2, 3}
  std::prev_permutation(v.begin(), v.end()); // false, v == {3, 2, 1}
}
```

[127] https://en.cppreference.com/w/cpp/algorithm/next_permutation

Additional Functionality

- `std::min_element`/`std::max_element` – operate over range of elements
- `std::merge`/`std::inplace_merge`
- `std::copy` – copy elements
- Many set operations, sampling, heap operations, ...
- `std::iota` – initialize with increasing values

```
#include <numeric>
#include <memory>

int main() {
  auto heapArray = std::make_unique<int[]>(5);
  std::iota(heapArray.get(), heapArray.get() + 5, 2);
  // heapArray is now {2, 3, 4, 5, 6}
}
```

Ranges[128]

- Ranges provide an abstraction of iterator pairs seen so far
- Views of ranges can be manipulated through adaptors

```
#include <ranges>
#include <print>
#include <map>
int main() {
  std::map<int, int> map{{10, 22}, {1, 2}, {3, 4}};
  for (auto key : map | std::views::keys)
    std::println("{}", key);
  // Prints: 1 3 10
}
```

[128] https://en.cppreference.com/w/cpp/ranges

Range Factories

- Most containers can be used directly as ranges
  - Details specified in range concepts, `ranges::range` and `ranges::viewable_range` (for ranges convertible into view for further transformation)
- Range factories: create commonly used views without dedicated container
  - `views::empty`, `views::single`, `views::iota`

```
#include <ranges>
#include <print>
int main() {
  for (auto i : std::views::iota(1, 5))
    std::println("{}", i);
  // Prints: 1 2 3 4
}
```

Range Adaptors

- (Lazily) Transform elements of range, return a view
- Might take additional arguments for transformation
- Can be chained, either by $C_2(C_1(R))$ or `R | C1 | C2`

```
#include <ranges>
#include <print>
#include <map>

int main() {
  std::map<int, int> map{{1, 2}, {3, 4}};
  for (auto key : (map | std::views::keys | std::views::reverse))
    std::println("{}", key);

  auto square = [](auto x) { return x * x; };
  for (auto sq : (map | std::views::keys | std::views::transform(square)))
    std::println("{}", sq);
}
```

Range Adaptors

Quiz: What is the output of the program?

```
#include <print>
#include <ranges>
#include <vector>
int fn() {
  auto print = [](int x) { std::print("{}␣", x); return x; };
  std::vector<int> vec{1, 2, 3, 4, 5, 6};
  auto v = vec | std::views::reverse | std::views::transform(print)
         | std::views::drop(2);
  return *v.begin();
}
int main() {
  std::println("{}", fn());
}
```

A. (compile error) B. 6 5 4 3 2 1 4 C. 6 5 4 4 D. 4 4

Random Number Generators[129]

- C++ standard library defines pseudo-random number generators/distributions
- PRNGs can (and should) be seeded; not thread-safe
- Example: `std::mt19937` (32-bit)/`std::mt19937_64`

```
#include <cstdint>
#include <random>
int main() {
  std::mt19937 engine(42); // seed = 42
  unsigned a = engine(); // a == 1608637542
  unsigned b = engine(); // b == 3421126067
}
```

[129] https://en.cppreference.com/w/cpp/header/random

Other Random Number Generators

- `std::mt19937` provides typically good enough pseudo-randomness
- `std::random_device` provides *true* randomness
  - Typically rather slow, might degrade to pseudo-randomness if no entropy is available
  - Not good for testing, where determinism is wanted
  - Typical use: get seed `std::mt19937 engine(std::random_device()());`
- `std::default_random_engine` – implementation-defined
  - Non-portable

- `rand()` from `<cstdlib>`
  - Quality of random numbers often rather bad
  - Avoid, strongly prefer C++ random functionality

Distributions

- Random number generators have fixed output range, approximately uniform
- Distributions transform output of RNG
  - Uniform, normal, Bernoulli, Possion, ...

```
#include <random>
int main() {
  std::mt19937 engine(42);
  std::uniform_int_distribution<int> dist(-2, 2); // range [-2, 2]
  int d1 = dist(engine); // d1 == -1
  int d2 = dist(engine); // d2 == -2
}
```

Remainder/Modulo for Uniform Distributions

Quiz: What is problematic about this function?

```
unsigned genRand8() { /* perfect RNG for 3 bits of randomness */ }
unsigned rollDice() {
  return genRand8() % 6 + 1;
}
```

A. Integer remainder is a very slow operation.

B. Some values are more likely than others.

C. There is no problem.

Algorithms, Functions, and Lambdas – Summary

- Function pointers can refer to functions, but have no state
- Member function pointers can refer non-static member functions
- Functors are a concept representing callable objects
- Lambdas are unnamed structures that can capture values
- Several algorithms are provided in the standard library
- Ranges provide an abstraction for iterator pairs
- Ranges can be transformed, creating views
- (Pseudo) random number generators and distributions are provided

Algorithms, Functions, and Lambdas – Questions

- What are requirements for a type to be a function object?
- What is the type of a lambda expression?
- Where are lambda captures stored?
- When can by-reference captures be problematic?
- How to write functions that take functors as argument?
- How to find the insertion point for some value into a sorted array?
- Why is modulo for random numbers generally not a good idea?