LAMBDAS

Things we stole from functional programming

Florian Warg, Max Staff

June 15, 2017

FUNCTORS

- in C++ we can have callable objects
- to do this we have to overload the function call operator
- any instantiated objects of those classes are called functors

```cpp
class AddOne {
public:
    int operator()(int i) { return i + 1; }
};
/* ... */
AddOne myFunctor;
std::cout << myFunctor(41);
```

CLOSURES

- closures are a concept in functional programming
- a closure is a function that can capture variables of its outer scope
- closures have implicit identity, state and behavior

CREATING CLOSURES WITH FUNCTORS

- we can simulate closures with functors
- captured variables are stored in private members

```
class Closure {
private:
    int val;
    int& ref;
public:
    Closure(int val, int& ref)
        : val(val), ref(ref) {}
    int operator()() { return val + ref; }
};
/* ... */
int x = 42;
int y = 31;
Closure f(x, y);
std::cout << f();
```

- anonymous functions originate from the lambda calculus
- in lambda calculus every function is anonymous
- e.g. $\lambda x.\lambda y.\lambda z.xyz$ is an anonymous function with 3 arguments
- anonymous functions are mostly used as arguments for higher order functions
- they can also be used to return a function as a result
- in C++ anonymous functions that represent closures are called **lambdas**

```
[captureList](argumentList) {functionBody}
```

```
auto plus1 = [](int x) { return x + 1; };
```

- lambda terms create functors of an anonymous type
- only `auto` can deduce the type
- the capture list can be used to capture variables of the enclosing scope by value or reference

```
int x = 42;
int y = 1;
// capture implicitly by value
auto f = [=]() { return x + y; };
// capture implicitly by reference
auto g = [&]() { return x + y; };
// capture explicitly int x and int& y
auto h = [x, &y] { return x + y; }
```

HIGHER-ORDER FUNCTIONS

- higher-order functions are functions that can take functions as arguments or return functions
- in standard C this is done via function pointers
- C++ uses templates or type erasure instead

```
struct Entry {
    int id;
    char name[32];
};
/* ... */
vector<Entry> v = {{3, "Charlie"},
                   {1, "Alice"},
                   {2, "Bob"}};
sort(begin(v), end(v), [](Entry& x, Entry& y) {
    return x.id < y.id; });
// v == {{1, "Alice"}, {2, "Bob"}, {3, "Charlie"}};
```

PARTIAL FUNCTION APPLICATION

- we can bind parameters of a function and get a new function of smaller arity as a result
- lambda expressions are naturally used for that

```
int f(int x, int y, int z) {
    return x * x + 2 * y + z;
}
/* ... */
auto partial = [](int z) {
    return f(1, 9, z);
};
```

- functions can return lambdas because they are objects
- lambdas can be used as function parameters by using templates

```
auto makeDice(int sides) {
    return [=]() {
        random_device r;
        default_random_engine e(r());
        uniform_int_distribution<int> d(1, sides);
        return d(e);
    };
}
/* ... */
auto cube = makeDice(6);
auto octahedron = makeDice(8);
cout << cube() << " " << octahedron() << "\n";
```

- template argument deduction can find the type of any lambda

```
template<class List, class Functor>
void forEach(List l, Functor f) {
    for (auto i : l) {
        f(i);
    }
}
/* ... */
vector<int> v = {1, 2, 3, 4};
auto print = [](int i) { cout << i << ' '; };
forEach(v, print);
```

- we can the keyword `auto` as a parameter type to create generic lambdas
- generic lambdas accept any type of arguments (as long as they are well-formed)

```
auto f = [](auto container) {
    for (auto i : container) {
        cout << " ";
    }
}
/* ... */
int a[] = {1, 2, 3, 4};
vector<string> v = {"Lambdas", "are", "great"};
f(a);
f(v);
```