Object-Oriented Programming

Lambda Functions

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Function Pointer Usage

```
int compare(const void* e1, const void* e2) {
    auto pn1 = static_cast<const int*>(e1);
    auto pn2 = static_cast<const int*>(e2);

    if (*pn1 < *pn2)
         return -1;
    else if (*pn1 > *pn2)
         return 1;
    else
         return 0;
}
```

```
std::array<int, 100000000> v;
// ...
qsort(v.data(), v.size(), sizeof(v[0]), compare);
```

```
int compare(const void* e1, const void* e2) {
    auto pn1 = static_cast<const int*>(e1);
    auto pn2 = static_cast<const int*>(e2);

    if (*pn1 < *pn2)
         return -1;
    else if (*pn1 > *pn2)
         return 1;
    else
         return 0;
}
```

Problems

- Have to create the function and name it
- Naming things is good when we have a good name, but a nuisance when we don't
- Functions often cannot be defined close to where they are used, so we have to search for them

Solution: Lambda Function

- Also known as an *anonymous function*

- Originated from Church's lambda calculus (*λ-calculus*)

- C++11 feature

- The simple form of a lambda function can be passed to a function pointer parameter

- Lambda Function Syntax

Trailing Return Type

```
int add(int n1, int n2) {
    return n1 + n2;
}
```

```
auto add(int n1, int n2) ->int {
    return n1 + n2;
}
```

Trailing Return Type

```
auto add(int n1, int n2) ->int {
    return n1 + n2;
}

auto add(int n1, double n2) -> decltype(n1 + n2) {
    return n1 + n2;
}

template<typename T, typename U>
auto add(T n1, U n2) -> decltype(n1 + n2) {
    return n1 + n2;
}
```

- The *trailing return type* syntax was introduced in C++11
- The return type is after the parameter list, marked with a –>
- The `auto` is a placeholder for the return type
- Required when the return type depends on the types of the parameters
- Most valid use cases are for *templates*
- Only use when necessary, stick to the traditional syntax

Lambda Parts

Code	Term	Description
`[]()->void {}`	Lambda	
`[]`	Capture	Access to *state* from where the lambda is defined. We will leave it empty for now, as this is a whole topic and only an empty capture can be used with function-pointer parameters.
`()`	Parameters	Pass *state* from the code that *calls the lambda function*
`->void`	Return type	Can be any function return type. Optional if `void`. In many cases, the compiler can derive the return type.
`{}`	Body	*Definition* of the lambda function. Where the statements go. Anything you can do in a *free function* you can do in a *lambda function*

Empty Lambda

```
[]() {}

[]() {
    std::cerr << "TRACE\n";
}
```

- Only isolated usage because there is no *state* for the function to work on

- Can be used for tracing or debugging

Lambda Parameter

```
[](int n) {}
```

- Useful because the current state of the caller is passed to the callback
- I.e., can compute based on the parameter

Lambda Parameters

lambda	Description

`[](char c) {`
`  std::cout << c;`
`}`
	IN parameter

`[](char& c) {`
`  c = std::toupper(c);`
`}`
	IN/OUT parameter

Lambda: `bool` comparision

```
[](int n1, int n2)->bool {
    return n1 < n2;
}
```

Lambda: `int` comparision

```
[](const void* e1, const void* e2)->int {
    int n1 = *(int*) e1;
    int n2 = *(int*) e2;

    if (n1 < n2)
        return -1;
    else if (n1 > n2)
        return 1;
    else
        return 0;
}
```

Call qsort() with lambda function

```
std::array<int, 100000000> v;
// ...
qsort(v.data(), v.size(), sizeof(v[0]), [](const void* v1, const void* v2) {
    auto n1 = static_cast<const int*>(v1);
    auto n2 = static_cast<const int*>(v2);

    if (*n1 < *n2)
         return -1;
    else if (*n1 > *n2)
         return 1;
    else
         return 0;
});
```

Call std::sort() with lambda function

```
std::vector<int> v;
// ...
std::sort(v.begin(), v.end(), [](int n1, int n2)->bool {

    return n1 < n2;
});
```

Comparison

Aspect	Function Pointer	Lambda Function
Definition	The function must be defined before it can be passed to a pointer	Defined inline where it's used, with its body enclosed in `{}` and parameters in `()`
Scope	Global or namespace scope for the function being pointed to	Scope is limited to where the lambda is defined
Capturing External Variables	Not directly possible. External variables must be passed as parameters	Can capture external variables from the surrounding scope either by value or by reference `[&]`
Portability	Highly portable and compatible with C interfaces	C++11
Performance	Slightly slower than a direct call	Provides more potential for compiler optimization

Issues

```
std::vector<int> v;

// ...

const int pivot = 5;
int comparisons = 0;
std::sort(v.begin(), v.end(), [pivot, &comparisons](int n1, int n2)->bool {
    ++comparisons;;
    return std::abs(n1 - pivot) < std::abs(n2 - pivot);
});
```

- The *capture* ([ ]) and *captured variables* are very important
- Inline lambda functions create several formatting issues