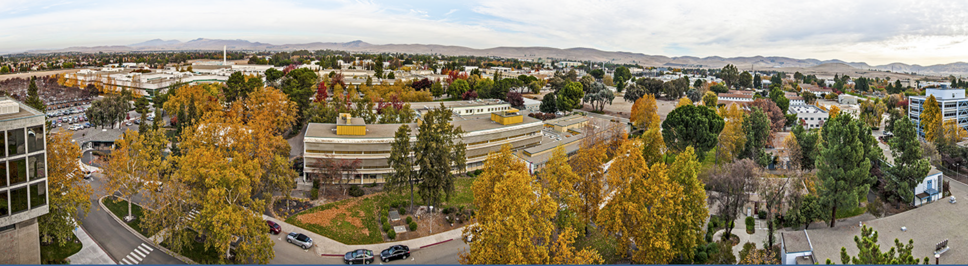
An Introduction to C++ Lambdas

August 2024

Seth Bromberger



LLNL-PRES-867878

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Disclaimer

A gentle introduction to lambdas

- A lambda expression is an anonymous function that can inherit state from its surroundings.
- With a few (language-specific) exceptions, anything you can do in a normal function can be done with a lambda expression.
- A C++ lambda expression is an object that can be assigned to a variable or passed as an argument.
- Lambda expressions are important because they provide a concise way to control another function's behavior.
- If you've done functional programming, you've undoubtedly used lambda expressions.

Higher-order functions

Consider a function that adds two to its input:

```
int add_two(int x) { return x + 2; }
int five = add_two(3);  // 5
```

This function has a name, `add_two`. You can pass this function as an argument to another function:

```
// executor takes an int and any unary function \\
// that has an int → int signature.
template <typename Function>
int executor(int n, Function fn) {
    return fn(n);
}
int result = executor(3, add_two);  // 5
```

`executor` is a *higher-order* function.

Lambdas: Anonymous functions

```
int add_two(int x) { return x + 2; }
        template <typename Function>
int executor(int n, Function fn) {
    return fn(n);
}
int result = executor(3, add_two)  // 5
```

If you only need `add_two` as an input to `executor`, then there is no need to assign a name to it. You can pass the function in as a *lambda*:

```
auto add_two { [](int x) -> int { return x + 2; }};
int result2 = executor(3, add_two);  // 5
```

or, inline:

```
int result2 = executor(3, [](int x) -> int {
    return x + 2;
});  // 5
```

Lambda syntax in C++

Capture (with optional default value)

Keywords (optional)

Parameters

Return type (optional)

```
[int n = 3](char c) mutable -> std::string {
    return std::string(n++, c);
}
```

Body

Captures

by copy

Because lambdas can be defined inline, they can *capture* the surrounding environment, bringing existing variables into the lambda body's scope:

```
int f(int x) {
    int y = 1;
    return executor(2, [y](int x) -> int {
        return 2 + x + y;
    }  // 5
}
```

Captures

by reference

In the previous example, the lambda body has access to a *copy* of `y`. It is also possible to pass *y* by *reference*:

```
int f(int x) {
    int y = 1;
    return executor(2, [&y](int x) -> int {
        return 2 + x + y++;
    }  // 5, but now y == 2
}
```

Mutable Captures

```
int main() {
    int health = 3;
    auto hit{[health]() mutable -> void {
        health--;
        std::cout << "Ouch! Health now " << health << "!\n";
    }};
    std::cout << "Health is " << health << ".\n";
    hit();
    hit();
    std::cout << "Health is " << health << ".\n";
}
```

```
Health is 3.
Ouch! Health now 2!
Ouch! Health now 1!
Health is 3.
```

More Captures

There are two special capture designations that can help if you need to capture lots of variables:

- $=$ captures everything in scope *by value*
- & captures everything in scope *by reference*
- These can be combined with specific captures.

Example:

```
int y = 5;
int z = 10;
            // this captures y by reference and everything else by value.
int r = executor(2, [&y, =](int x) -> int { return x + y + z; y += 2; }  // r=17, y=7
```

Note: it is better to be explicit about captured variables instead of relying on these “wildcard” values:

```
int r = executor(2, [&y, z](int x) -> int { return x + y + z; y += 2; }  // r=17, y=7
```

Capture Persistence

Values of captured variables that are copied persist:

```
int main() {
    int health = 3;
    auto hit{[health]() mutable -> void {
        health--;
        std::cout << "Ouch! Health now " << health << "!\n";
    }};
    std::cout << "Health is " << health << ".\n";
    hit();
    hit();
    std::cout << "Health is " << health << ".\n";
    hit();
}
```

```
Health is 3.
Ouch! Health now 2!
Ouch! Health now 1!
Health is 3.
Ouch! Health now 0!
```

Captures in YGM

In YGM, the ability to use captures depends on the function:

- In non-async functions, there are no restrictions.
- In async functions, captures are limited to *copies* of data structures that are *standard layout*[1] and *trivially copyable*[2]; *e.g.*, `bool`, `char`, `int`, `float`, `double`, and structs of these types.
- In many scenarios (*e.g.*, all async functions and `for_all()`), the lambda's return value is ignored and should not be specified.

[1] `std::is_standard_layout`
[2] `std::is_trivially_copyable`

Examples

Pass by copy, no rebinding

```
int main() {
    char c = '.';
    int n = 3;
    auto repeat{[c, n]() -> void { // c and n are copied.
        std::cout << std::string(n, c) << "\n"; }
    };

    repeat();                       // output: ...
    c = '+';                        // this doesn't affect repeat
    repeat();                       // output: ...
    [c, n]() -> void {
        std::cout << std::string(n, c) << "\n";
    }();                            // output: +++
}
```

Examples

Pass by reference

```
int main() {
    char c = '.';
    int n = 3;
    auto repeat{[&c, &n]() -> void {
        std::cout << std::string(n, c) << "\n"; }
    };

    repeat();                        // output: ...
    c = '+';                         // wait for it!
    repeat();                        // output: +++
    [&c, &n]() -> void {
        std::cout << std::string(n, c) << "\n";
    }();                             // output: +++
}
```

Examples

Another pass by reference

```cpp
int main() {
    char c = '.';
    int n = 3;
    auto repeat{[&c, &n]() -> void {
        std::cout << std::string(n, c) << "\n"; }
        n += 3;
    };

    repeat();                       // output: ...
    c = '+';
    repeat();                       // output: ++++++
    [&c, &n]() -> void {
        std::cout << std::string(n, c) << "\n";
    }();                            // output: +++++++++
}
```

History

Functors

In the beginning, there was the *functor*:

```
struct add_x {
    add_x(int n): x(n) {};                         // constructor
    int operator()(int y) { return x + y; }  // override
private:
    int x;
};

auto add_2 = add_x(2);                              // this is a struct
int z = add_2(10);                                  // 12
```

Functors capture state via the constructor, and are callable by overriding the `operator()` method.

Functors vs Lambdas

	C++ Lambda	C++ Functor
Can recurse	-	✓
Requires explicit captures	-	✓
More explicit behavior	-	✓
One-liners	✓	-
Optional return types	✓	-

Lawrence Livermore National Laboratory