Şevval MEHDER

Lambda Expressions

A lambda expression:

```
[capture list](function arguments){
    function body
};
```

```cpp
void thread_function(){
    std::cout << "thread function\n";
}

int main(){

    std::thread t(&thread_function);    // t starts running

    std::cout << "main thread\n";
    t.join();
    return 0;
}
```

```cpp
int main(){

    std::thread t(
        [](){
            std::cout << "thread function\n";
        }
    );

    std::cout << "main thread\n";
    t.join();
    return 0;
}
```

There are two default capture modes in C++11: by-reference and by-value

by-reference

- [&](){}, which means capture any referenced variable by reference
- [&number](){}, which means capture only variable number by reference

by-value

- [=](){}, which means capture any referenced variable by copy
- [index, &number](){}, which means capture variable number by reference, variable index by value

By-reference

```cpp
int main(){

    // We want to filter some data with different filters
    using FilterContainer = std::vector<std::function<bool(int)>>;
    FilterContainer filters;

    // Add a filter for multiples of 2
    multsOf2();
}
void multsOf2(){

    int divisor = 2;

    filters.emplace_back(

        [&](int value){ return value % divisor == 0; } // danger
        // [&divisor](int value){ return value % divisor == 0; }

    );
}
```

- Reference the divisor will dangle.
- With [&divisor], it's easier to see.

By-value

```
class Filters{
public:
    //...
    void addNewFilter() const;

private:
    int divisor;
};
void Filters::addNewFilter(){
    filters.emplace_back(
        [=](int value){
            return value % divisor == 0;
        }
    );
}
```

- Captures apply only to non-static local variables in the scope where the lambda is created.
- Either by value or reference the **capture won't compile**

```
void Filters::addNewFilter(){
    filters.emplace_back(
        [this](int value){
            return value % divisor == 0;
        }
    );
}
```

```
void Filters::addNewFilter(){

    auto copyDivisor = divisor;

    filters.emplace_back(
        [copyDivisor](int value){
            return value % copyDivisor == 0;
        }
    );
}
```

```
void Filters::addNewFilter(){

    // C++14 solution
    // Default capture variable values
    // Capture copy divisor
    filters.emplace_back(
        [divisor = divisor](int value){
            return value % divisor == 0;
        }
    );
}
```

If you don't capture any variables

```
// [](X& element){ element.operation(); }

class someName{
public:
    void operator()(X& element) const {
        element.operation();
    }
}
```

Capturing by value

```
// int x = 3
// [=](int a){ return a + x; };
class Functor {
public:
    Functor(const int x): m_x(x) {}

    int operator()(int a) {
        return a + m_x;
    }

private:
    int m_x;
};
```

Capturing by reference

```
// int x = 3
// [&](int a){ return a + x++; };
class Functor {
public:
    Functor(int& x): m_x(x) {}

    int operator()(int a) {
        return a + m_x++;
    }

private:
    int& m_x;
};
```

Details: https://web.mst.edu/~nmjxv3/articles/lambdas.html