Item 31) Avoid default capture modes.

들어가기에 앞서 Closure란 무엇인가??

closure란??
함수가 그 함수가 선언된 환경(변수, 상태)을 함께 묶어서 저장하는 기능을 의미한다.
C++에서 람다함수는 외부의 변수나 상태를 "capture"할 수 있는 특성이 있다. 이에 람다함수를 closure라고 정의한다.
즉, *람다 함수의 외부 변수나 상태를 캡쳐할 수 있는 특성* 때문에 "closure"라고 정의된다.

- 람다 함수가 외부 변수를 캡쳐하지 않으면 단순한 함수일 뿐, 클로저는 아님.

C++11에는 by-reference와 by-value의 capture mode가 있다.
default by reference는 dangling의 위험성이 있다.
그렇다면 default by value는 dangling pointer의 문제점이 없이 self-contained할까??

⭐ ⭐ ⭐ 만약 람다에서 만들어진 closure lifetime이 지역변수, 혹은 parameter의 lifetime보다 더 길어진다면 closure안에 있는 reference는 dangling이 된다.

예시 코드

```
using FilterContainer = std::vector<std::function<bool(int)>>;
FilterContainer filters; // 필터링 함수들을 저장할 컨테이너
```

위의 컨테이너에서 필터를 추가하는 방식으로, 5의 배수를 찾는 필터를 람다 함수로 추가한다고 가정하자.

```
filters.emplace_back([](int value) {return value % 5 == 0};);
```

자 여기서 아래의 코드의 문제점을 한 번 살펴보자.

```
void addDivisorFilter() {
    auto calc1 = computeSomeValue1();
    auto calc2 = computeSomeValue2();
    auto divisor = computeDivisor(calc1, calc2);

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

여기서 람다 함수는 외부 변수 `divisor`를 참조 캡쳐 `[&]`하여 사용하고 있다.
문제는 `divisor` 변수의 수명에 있다. `divisor`와 `addDivisorFiler`는 함수 안의 *지역 변수*로 선언되었기 때문에, 함수가 끝나면 메모리에서 해제된다. 하지만 **람다 함수는 이 변수를 참조**하려고 한다. 즉, `addDivisorFilter`함수가 끝난 후에 해당 람다 함수가 호출되면, **이미 소멸된 메모리를 참조하려고 시도**하게 된다. 이로 인해 **Dangling reference**가 발생하게 되며, **undefined Behavior**을 일으킨다.

```
filetrs.emplace_back([&divisor](int value){return value % divisor = 0;});
```

자 그렇다면 default reference capture를 사용하지 않고 divisor만 reference를 하면 문제가 해결되는가??

아쉽게도 같은 문제가 발생한다. 그러나 explicit하게 capture clause를 하면 람다 함수가 `divisor`의 lifetime에 의존한다는 것을 코더가 알 수 있다.

또한 위와 같은 방식으로 참조를 진행하게 된다면, **람다 함수의 수명동안 capture된 객체가 계속 살아있어야 한다는 점**을 상기시키기 때문에, 그 변수를 코더에게 신중하게 관리할 수 있다.

그렇다면 **참조 캡쳐가 안전할 수 있는 경우는 무엇인가??**

람다 함수가 즉시 사용되고 복사되지 않는다면 참조 캡처가 안전할 수 있다. 특히 `STL알고리즘`에 전달되어 즉시 실행된다면, 람다 함수에서 참조한 변수들이 여전히 유효한 상태이기 때문에 Dangling reference문제가 발생하지 않는다.

아래의 코드 예시를 살펴보도록 하자.

```
if (std::all_of(begin(container), end(container),
     [&](const ContElemT& value) { return value % divisor == 0; })) {
    // 모든 요소가 divisor의 배수임
} else {
    // 하나 이상의 요소가 divisor의 배수가 아님
}
```

위의 코드인 경우는 안전할 수 있다. 그러나 특정 람다 함수가 유용하다고 생각되어서 **다른 곳에서 재사용 될 수 있다면**, 즉 람다 함수를 복사해서 `filetrs`와 같은 *긴 수명을 가진 컨테이너에 저장*하는 경우, 참조한 `divisor`는 이미 소멸된 상태가 되어 Dangling reference가 발생된다.

결론적으로

- 참조 캡처는 **람다 함수가 즉시 사용되고, 외부 변수들이 그 함수가 실행될 때까지 살아있는 경우**에는 안전한다.

- **참조한 변수의 수명**이 함수보다 짧아지거나, 람다 함수가 복사되어 다른 곳에서 재사용될 경우 댕글링 참조가 발생할 위험이 있다.
- 장기적으로는, 람다 함수에서 필요한 변수들을 명시적으로 캡쳐하거나(코더에게 주의를 줄 수 있음), 참조 대신 값을 복사하는 방식으로 코드를 작성해야 한다.

📌Tip) C++14부터는 람다 함수의 parameter에 auto를 넣을 수 있다

```
f (std::all_of(begin(container), end(container),
     [&](const auto& value) { return value % divisor == 0; })) {
    // 모든 요소가 divisor의 배수임
} else {
    // 하나 이상의 요소가 divisor의 배수가 아님
}
```

그렇다면 복사를 사용하면 괜찮을까요??

```
void Widget::addFilter() const {
    filters.emplace_back([=](int value) { return value % divisor == 0; });
}
```

```
#include <iostream>
#include <vector>
#include <functional>

void createLambda(std::vector<std::function<void()>>& funcs) {
    int* ptr = new int(42); // 동적으로 메모리 할당

    // 람다에서 ptr을 참조
    funcs.push_back([=]() { std::cout << *ptr << std::endl; });

    delete ptr; // 메모리 해제
}

int main() {
    std::vector<std::function<void()>> funcs;
    createLambda(funcs);

    // 람다 함수 호출 (ptr은 이미 해제됨)
    funcs[0](); // 댕글링 포인터 접근

    return 0;
}
```

NO absolutely Never

default copy capture가 안전할 것처럼 보이지만, **포인터를 값으로 캡쳐하는 경우**에는 문제가 발생할 수 있다. 만약 내부에서 포인터를 값으로 캡쳐한다면, 그 포인터의 복사본이 클로저에 저장되지만, 원본 포인터가 **소멸**되거나 **삭제**될 수 있다.

이 경우, 포인터가 가리키던 메모리가 해제되면, 복사된 포인터는 dangling pointer가 되어, 이후에 접근할 경우 **정의되지 않은 동작**을 유발하게 된다.