Understanding Lambda Capture in C++

Contents

Case 1: No capture, no outside variables used

```
auto sayHi = []() {
    cout << "Hi!" << endl;
};
sayHi();  // Output: Hi!
```

Case 2: Capture nothing, but tries to use an outside variable (Error)

```
int x = 5;
auto printX = []() {
    // cout << x;  //  Error: 'x' is not captured
};
```

Case 3: Capture by value [=] (copy)

```
int x = 5;
auto printX = [=]() {
    cout << "x = " << x << endl;  // x is copied
};
printX();  // Output: x = 5
```

Case 4: Capture by reference [] (live link to outside)

```
int x = 5;
auto modifyX = [&]() {
    x += 10;  // modifies the original x
};
modifyX();
cout << x << endl;  // Output: 15
```

Optional: Capture specific variables

```
int x = 5, y = 10;

auto custom = [x, &y]() {
    cout << "x (copied): " << x << ", y (referenced): " << y << endl;
    // y += 1; // allowed
    // x += 1; //  x is read-only (captured by value)
};
```

Lambda with Parameters and Return Types

Example 1: Lambda with parameter list

```
auto square = [](int x) {
```

```
    return x * x;
};
cout << square(4);  // Output: 16
```

Example 2: Lambda with parameters and explicit return type

```
auto divide = [](double a, double b) -> double {
    return b != 0 ? a / b : 0;
};
cout << divide(10, 2);  // Output: 5
```

Example 3: Mixed capture and parameter list

```
int base = 10;
auto addToBase = [base](int x) {
    return base + x;
};
cout << addToBase(5);  // Output: 15
```

Lambda Capture Quiz

1. What will the following code print?

```
int x = 3;
auto f = [=]() {
    return x + 2;
};
cout << f();
```

a) 2 **b)** 3 **c)** 5 **d)** Error

2. Which capture mode allows modifying the original variable?
a) `[]` **b)** `[=]` **c)** `[]` **d)** `[x]`

3. What will happen in the following case?

```
int x = 10;
auto f = []() {
    cout << x;
};
```

a) Prints 10 **b)** Error **c)** Prints 0 **d)** Undefined behavior

Answers:
1. c) 5
2. c) []
3. b) Error