Lambda expression

MAJOR ASSIGNEMET CPP OOP

Sir Muhammad Adnan

City University of Science and Information Technology

Taimoor Ul Islam (14031)

# Table of content

* Introduction
* Background
* Methods Detail
* Real Life Examples / Common Use Cases
* Consideration and Limitations
* Conclusion

# Introduction

Lambda expressions are a powerful feature found in modern programming languages that support functional programming paradigms. They provide a concise and expressive way to define anonymous functions, allowing developers to write more readable and maintainable code

# background

Lambda expressions were introduced in the C++ programming language with the release of C++11. They provide a concise way to define anonymous functions and are a significant addition to the language, enabling developers to write more expressive and flexible code.

Prior to the introduction of lambda expressions, C++ programmers typically used function objects (functors) or function pointers to achieve similar functionality. However, these approaches often involved writing more code and required the explicit definition of separate classes or functions.

Lambda expressions in C++ follow a syntax that closely resembles lambda expressions in other programming languages like Python or JavaScript. They offer a compact and readable way to define inline functions without the need for explicit declarations.

# Method detail

## Syntax

The syntax of a lambda expression in C++ follows the pattern:

**[capture-list] (arguments) -> return-type { body }.**

The capture-list is an optional part that allows lambda functions to capture variables from their surrounding scope.

Arguments represent the input parameters of the function.

The return-type specifies the return type of the lambda function.

The body contains the computation or operation performed by the lambda function.

## Example

**CPP Code**

auto addition = [ ](int x, int y) -> int

{ return x + y; };

main(){

int result = addition(3, 5);

cout << result << endl;

}

*// Output: 8*

## Capture List

The capture list, denoted by [ ], allows lambda functions to capture variables from their surrounding scope.

The capture list can be used to access and use variables in the lambda function's body.

There are three types of capture in C++:

* Value Capture: [x] captures the variable x by value. It creates a copy of the variable within the lambda function.
* Reference Capture: [&x] captures the variable x by reference. It allows the lambda function to access and modify the original variable.
* Generalized Capture: [=] captures all variables from the surrounding scope by value, and [&] captures all variables by reference.

## Example with Capture List

**CPP Code**

int value = 10;

auto increment = [value]() mutable { value++; };

main(){

increment();

cout << value << endl;

*// Output: 10 (value in the //outer scope remains //unchanged)*

}

In this example, the lambda expression captures the variable value by value.

The mutable keyword allows modification of the captured variable within the lambda function.

However, the modification does not affect the value of value in the outer scope.

# REal life examples/ Common Use Cases

Following are some common use cases of lambda expressions

## Higher-order functions

Higher-order functions are functions that take other functions as arguments or return functions as results.

Lambda expressions are commonly used with higher-order functions to provide custom operations on elements of collections.

**Example:**

Let's consider the transform algorithm in C++ that applies a given operation to each element of a range and stores the results in another range:

In this example, the lambda expression [](int n) { return n \* n; } defines a function that squares a number.

The lambda function is passed as the operation argument to transform, which applies the lambda function to each element of the numbers vector and stores the results in the squares vector.

In this example, the lambda expression [](int n) { return n \* n; } defines a function that squares a number.

The lambda function is passed as the operation argument to transform, which applies the lambda function to each element of the numbers vector and stores the results in the squares vector.

**CPP Code**

main(){

vector<int> numbers = {1, 2, 3, 4, 5};

vector<int> squares;

transform( numbers.begin(), numbers.end(),back\_inserter(squares), [ ](int n) { return n \* n; });

*// squares will contain {1, 4, 9, 16, 25}*

}

## Asynchronous operations

Lambda expressions are frequently used with asynchronous operations to serve as inline callbacks to handle the results of asynchronous tasks.

This allows for more concise and readable code, as the callback logic is defined directly where it is needed.

**Example:**

Let's consider the usage of async to execute a function asynchronously and retrieve its result:

In this example, the lambda expression []() { return performTimeConsumingTask(); } defines the task that will be executed asynchronously.

The lambda function is passed to async, which launches the task in a separate thread.

The future object represents the result of the asynchronous task and can be used to retrieve the result later using future.get().

**CPP Code**

auto future = async([]() { return performTimeConsumingTask(); });

*// Do other work concurrently*

auto result = future.get(); *// Retrieve the result when needed*

## Custom predicates

Lambda expressions are useful for defining custom predicates in algorithms that require condition checking based on specific criteria.

Predicates are often used in algorithms like find\_if, remove\_if, and sort to determine element matches or sorting order.

**Example**

Let's consider the usage of find\_if to find the first element in a vector that satisfies a specific condition:

In this example, the lambda expression [](int n) { return n % 2 == 0; } defines a function that checks if a number is even.

The lambda function is passed as the predicate argument to find\_if, which searches for the first element in the numbers vector that satisfies the condition.

**CPP Code**

vector<int> numbers = {1, 2, 3, 4, 5};

auto it = find\_if(numbers.begin(), numbers.end(), [](int n) { return n % 2 == 0; });

if (it != numbers.end()) {

cout << "First even number found: " << \*it << endl;

}

# Considerations and Limitations

Following are some considerations and limitations of lambda expressions:

## Dangling references

A dangling reference refers to a situation where a reference is used to access a memory location that has been deallocated or is no longer valid. In the context of lambda functions, a dangling reference can occur when a lambda captures a reference to a variable that goes out of scope before the lambda is executed.

**Example**

In this example, we dynamically allocate an int using new and assign its address to the pointer variable ptr. The lambda function captures ptr by value using [=]. However, after the memory is deallocated using delete ptr, the lambda function is still called, attempting to access the captured reference ptr. This results in a dangling reference because the memory previously pointed to by ptr has been deallocated.

**CPP Code**

main() {

int\* ptr = new int(10);

auto lambda = [=]() {

std::cout << "Value pointed by ptr: " << \*ptr << std::endl;

};

delete ptr; // Deallocate memory pointed by ptr

lambda(); // Accessing a captured dangling reference

}

## Unnecessary Copying

Be mindful of capturing large objects by value, as it may lead to unnecessary copying. Consider capturing them by reference instead.

Capturing large objects by value in lambda expressions can result in unnecessary copying of those objects, which may lead to performance overhead.

Copying large objects can be time-consuming and memory-intensive, impacting the efficiency of the program.

Instead, capturing large objects by reference avoids the overhead of copying and allows the lambda function to directly operate on the original object.

## Exception handling

Lambda expressions cannot have their own try-catch blocks. Exception handling should be done outside the lambda if needed.

Lambda expressions cannot directly contain their own exception handling blocks (try-catch).

If an exception occurs within a lambda function, it will propagate out of the lambda and can be caught and handled outside the lambda expression.

It's important to consider the appropriate exception handling strategy for lambda functions and ensure that exceptions are properly caught and handled outside the lambda when necessary.

## Return type

The return type of a lambda function can be omitted in C++14 and later if it can be deduced from the return statement. Otherwise, it should be specified explicitly.

In C++14 and later versions, the return type of a lambda function can be deduced by the compiler if it can be determined from the return statement.

This allows for more concise lambda expressions by omitting the explicit return type specification.

However, if the return type cannot be deduced or if explicit return type specification is desired for clarity, it should be specified explicitly using the -> syntax.

# Conclusion

Lambda expressions provide a concise and flexible way to define anonymous functions in C++.

They offer benefits such as code conciseness, flexibility in capturing variables, and customization of operations.

Lambda expressions are commonly used with higher-order functions, asynchronous operations, and custom predicates.

Understanding lambda expressions in C++ can enhance your coding skills and enable more expressive and efficient code.