Lambda:

<https://www.programiz.com/cpp-programming/lambda-expression>

Lambda expressions are inline anonymous functions i.e. they do not have a name. Lambda has various parts such as capture clause, parameters, [keywords](https://www.scaler.com/topics/cpp/keywords-in-cpp/), body, exceptions, etc.

[Capture clause] (parameters) mutable exception ->return\_type

{

// Method definition;

}

* **capture clause** - it is a list of variables that are to be copied inside the lambda function in C++. We can also use it to initialize variables.
* **parameters** - zero, one or more than one argument to be passed to the lambda at execution time.
* **mutable** - Mutable is an optional keyword. It lets us modify the value of the variables that are captured by the call-by-value when written in the lambda expression.
* **return type** - It is optional as the compiler evaluates it. Though, in some complex cases compiler can't make out the return type and thus we need to specify it.
* **body of the method** - It is the same as the usual method definition. All the operations to be performed when the lambda expression is called are written here.

The lambda mutable keyword allows lambda to modify the variable **captured by value** inside the lambda body without affecting its original value in the enclosing function.

However the variables **captured by reference** are not affected. For example,

#include <iostream>

using namespace std;

int main() {

int a = 1;

int b = 1;

cout << "In main():" << endl;

cout << "a = " << a << ", ";

cout << "b = " << b << endl;

cout << endl;

auto add\_one = [a, &b] () mutable {

// modify both a & b

a++;

b++;

cout << "In add\_one():" << endl;

cout << "a = " << a << ", ";

cout << "b = " << b << endl;

};

add\_one();

cout << endl;

cout << "In main():" << endl;

cout << "a = " << a << ", ";

cout << "b = " << b << endl;

return 0;

}

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

// initialize vector of integers

vector<int> nums = {1, 2, 3, 4, 5, 8, 10, 12};

int even\_count = count\_if(nums.begin(), nums.end(), [](int num) {

return num % 2 == 0;

});

cout << "There are " << even\_count << " even numbers.";

return 0;

}

<https://www.programiz.com/cpp-programming/functors>

Lambda functions are often used in situations where you need a simple function for short-lived tasks, such as sorting or filtering elements in a container, defining custom comparison functions, or providing callbacks for standard library algorithms like **std::for\_each** and **std::sort**. They offer a more concise and convenient way to define such functions compared to traditional function objects or functors.

: Lambda functions allow you to write small, inline functions without having to declare a separate named function. This can make your code more concise and easier to read, especially when the function is simple and used only in one place

Lambda functions can improve the readability of your code by keeping the logic of a function close to where it's used. This can make your code easier to understand, as you don't need to jump to a different part of your codebase to see the function's implementation.

Lambda functions capture variables from their surrounding scope. This can be very useful when you need to pass variables to a function without making them global or passing them explicitly as function arguments. The lambda captures variables by reference or by value, giving you control over how variables are accessed and modified.

-----------------------------------------------------------------------------------------------------------------------------------

FUNCTION POINTER:

<https://www.geeksforgeeks.org/function-pointer-in-cpp/>

Functors:

It works as function and achieved by overloading “() operator” in some class.

It overloads the **function-call operator** () and allows us to use an object like a function.

In C++, functors (function objects) are objects that can be treated as if they are functions. They are instances of classes that define the `operator()` function, allowing them to be called like functions. Functors provide a way to encapsulate behavior and are commonly used in various situations.

**Functors are flexible and can have internal state, making them more versatile than regular functions or function pointers. They are often used in scenarios where you want to pass a callable object to a function, such as when working with standard library algorithms that accept callable objects as arguments.**

Here are some of the common uses of functors in C++:

1. Sorting: Functors can be used as predicates in sorting algorithms like `std::sort` to define custom sorting criteria. By overloading the `operator()` function, you can define how two elements should be compared.

2. Comparison: Functors can be used for custom comparisons in containers like `std::map` and `std::set`. By providing a functor that defines the less-than (`<`) operator, you can control the ordering of elements in the container.

3. Filtering: Functors can be used for filtering elements based on specific criteria. They can be used with algorithms like `std::remove\_if` or `std::copy\_if` to selectively operate on elements of a container.

5. Custom Algorithms: Functors can be used to define custom algorithms or operations on a set of elements. By implementing the required behavior in the `operator()`, you can use the functor in algorithms like `std::for\_each` or `std::transform`.

**Stateful Behavior:**

* Functors can have internal state, allowing them to maintain information between calls.

**Flexibility**: Functors provide more flexibility than simple functions because they can have member variables and complex behavior.

These are just a few examples of how functors can be used in C++. The flexibility and versatility of functors make them a powerful tool for expressing complex behavior and customizing the behavior of algorithms and containers.

#include <iostream>

using namespace std;

class Multiply

{

private:

int val;

public:

Multiply (int a) : val(a) {}

int operator() (int num)

{

return val\*num;

}

};

int main()

{

Multiply m1(12); // saves the state

cout<<m1(2)<<endl;

cout<<m1(3);

}

---------------------------------------------------------------------------------------------------------------------------------

#include <iostream>

using namespace std;

class Add {

public:

int operator() (int a, int b) {

return a + b;

}

};

int main() {

Add add;

// call the add object

int sum = add(100, 78);

cout << "100 + 78 = " << sum;

return 0;

}

#include <iostream>

#include <vector>

#include <algorithm>

// Define a functor for custom string length comparison

class LengthComparator {

public:

bool operator()(const std::string& a, const std::string& b) const

{

return a.length() > b.length();

}

};

int main() {

std::vector<std::string> words = {"apple", "banana", "cherry", "date"};

// Sort the vector using the LengthComparator functor

std::sort(words.begin(), words.end(), LengthComparator());

for (const std::string& word : words) {

std::cout << word << " ";

}

return 0;

}

**When to Use Functors (Function Objects)**:

* Use functors when you need to create callable objects with state. Functors can have member variables that maintain state across multiple calls, making them suitable for situations where you need to remember information between function calls.
* Functors are also useful when you want to provide custom behavior for operators like **()**, allowing you to define complex or specialized actions when the functor is called.

**When to Use Lambda Functions**:

* Use lambda functions for short, simple, and one-off operations where creating a separate named function or functor is unnecessary and would clutter the code.
* Lambda functions are particularly useful when you want to capture and use variables from the enclosing scope without explicitly passing them as arguments.
* They are handy for tasks like sorting, filtering, mapping, or applying simple transformations to data.

In summary, the choice between functors and lambda functions depends on your specific needs. If you require state or custom operator behavior, functors are appropriate. For concise, one-time operations with easy access to surrounding variables, lambda functions are a more practical choice.



Top of Form