Functional Programming

In languages that support the functional paradigm, functions are treated as fundamental data types. This means they can be assigned to pointer variables, passed as arguments to other functions, or even returned as results. In modern C++, this capability allows programmers to write cleaner and more expressive code by using features like function pointers, function objects, and lambda expressions. A great way to start experimenting with functional programming is by replacing traditional loops in your day-to-day code with STL algorithms such as `transform`, `copy_if`, and `accumulate`. These algorithms let you pass functions or behaviors directly into the processing logic, helping you think in terms of operations on collections rather than explicit iteration, and guiding you naturally into the functional style of programming.

Function Pointers

Function pointers are variables that store memory addresses of functions, enabling dynamic function invocation. They serve as the foundation for callback mechanisms and flexible program architectures.

Key Characteristics

- Store addresses of executable code
- Enable runtime function selection
- Maintain type safety through signature matching

Declaration Syntax

The formal declaration follows this structure:

```
return_type (*pointer_name)(type_param_1, type_param_2, ...);
```

Important Notes

- Parentheses around `*pointer_name` are mandatory
- The return type and parameters must match any assigned function
- Unlike data pointers, they reference executable code

Example Declarations:

```
int (*math_operation)(int, int);  /* Pointer to a function taking two ints and
   returning int */

int (*ptr)(int arr[], int size); /* ptr is declared as a pointer to a function, which
   takes as parameters an array of integers and an integer and returns an integer. */
```

```
void (*ptr)(double *arr[]); /* ptr is declared as a pointer to a function,  which
   takes as parameters an array of pointers to doubles and returns nothing. */

int test(void (*ptr)(int a)); /* test() returns an integer value and takes as
   parameter a pointer to another function, which takes an integer parameter and
   returns nothing. */
```

Function Pointers as Arguments

When we have variables that can store pointers to functions, we gain the ability to pass functions as arguments to other functions. This powerful feature lets us write code where the processing behavior can change dynamically while the program runs.
Look at the below example code. Here, the `process_number` function doesn't do the actual work itself, instead it relies on whatever function we pass to it to handle the real processing. This means we can completely change what `process_number` does just by giving it different functions when we call it at runtime.

Example

```
#include <iostream>
using namespace std;

int square(int x) { return x * x; }
int cube(int x) { return x * x * x; }
int negate(int x) { return -x; }

int process_number(int (*operation)(int), int x) {
    return operation(x); // Calls whichever function was passed
}

int main() {
    int num = 5;

    cout << "Squared: " << process_number(square, num) << endl;   // 25
    cout << "Cubed: " << process_number(cube, num) << endl;       // 125
    cout << "Negated: " << process_number(negate, num) << endl;   // -5

    return 0;
}
```

Function Objects

A function object, also known as a functor, is any object in C++ that can be called as if it was a function. This is made possible by overloading the function call operator `operator()` inside a class or struct. When

this operator is defined, instances of that class can be "called" like normal functions. Functors are more powerful than function pointers, since functors can carry around state.

What Makes Functors Special?

- **They are objects that act like functions:** You can call them using `()` just like normal functions.
- **They can store information:** Unlike regular functions, they can remember data between calls.
- **Work with STL algorithms:** They're commonly used with algorithms like sort(), for_each(), etc.

Example:

Here's a simple functor that checks if numbers are above a certain limit:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Filter {
    int limit;  // Stores our filter condition
public:
    Filter(int l) : limit(l) {}  // Constructor sets the limit

    // The important part - overloaded () operator
    void operator()(int x) const {
        if (x < limit)
            cout << x << " ";
    }
};
int main() {
    vector<int> numbers = {5, 3, 8, 12, 1};

    cout << "Numbers less than 6: ";
    for_each(numbers.begin(), numbers.end(), Filter(6));
    // Output: 5 3 1

    cout << "\n Numbers less than 4: ";
    for_each(numbers.begin(), numbers.end(), Filter(4));
    // Output: 3 1
    return 0;}
```

Key Parts Explained

- We can create different filters (6, 4, etc.) without writing new functions.

- The filtering logic stays clean and centralized.
- We can easily change the comparison condition.

Lambda Functions

Lambda functions in C++ are **anonymous function objects** that allow you to define small, inline functions without giving them a name. The typical use of lambda functions is as arguments in STL functions that accept a function pointer or functor as a parameter. As we saw, a **functor** is more powerful than a function pointer; however, it requires writing additional code to support its functionality, such as defining a class and overloading the `operator()`.
They are **lightweight, convenient, and powerful**, particularly for functional programming, where functions are passed as arguments or returned from other functions.

General Syntax of a Lambda Function

```
[capture_list](parameter_list) -> return_type {
    // function body
};
```

Parts of a Lambda Function:

- **Capture List** (`[]`): Captures variables from the surrounding scope.
- **Parameter List** (`()`): Like a normal function's parameter list.
- **Return Type** (`->`): Optional if the return type can be inferred.
- **Function Body**: The statements to execute.

Example:

```
#include <iostream>
using namespace std;
int main()
{
    int val = 11;
    cout << [&](){ return val*val; }() << endl;

return 0;
}
```

Lambda Capture Types in C++

Syntax	Type of Capture	Description
`[x]`	Capture by value	Captures variable `x` by value (a copy is made). Changes inside lambda won't affect the original variable.
`[&x]`	Capture by reference	Captures variable `x` by reference. Changes inside lambda will reflect outside.
`[=]`	Capture all by value	Captures all outer variables used inside the lambda by value.
`[&]`	Capture all by reference	Captures all outer variables used inside the lambda by reference.
`[=, &x]`	Mixed capture	Captures all by value, but `x` by reference.
`[&, x]`	Mixed capture	Captures all by reference, but `x` by value.

Example Usage:

```
int a = 10, b = 20;

auto val = [a]() { return a + 5; };     // [x]        Capture a by value
auto ref = [&b]() { b += 5; };          // [&x]       Capture b by reference
auto allVal = [=]() { return a + b; }; // [=]        Capture all by value
auto allRef = [&]() { a += b; };        // [&]        Capture all by reference
auto mix1 = [=, &b]() { b += a; };      // [=, &x]    a by value, b by reference
auto mix2 = [&, a]() { b += a; };       // [&, x]     all by reference, a by value
```

Using Lambda Functions with Common STL Algorithms in C++

Lambdas are often used as an argument in STL algorithms when the default behavior is not desirable. Most commonly used STL algorithms are mentioned below:

1. `std::transform`

Applies a function to a range and stores the result in another range.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4};
    std::vector<int> result(v.size());

    std::transform(v.begin(), v.end(), result.begin(), [](int x) {
```

```
        return x * x;
    });

    for (int i : result) std::cout << i << " "; // Output: 1 4 9 16
}
```

2. `std::copy_if`

Copies elements from a range to another container if they satisfy a condition.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::vector<int> evens;

    std::copy_if(v.begin(), v.end(), std::back_inserter(evens), [](int x) {
        return x % 2 == 0;
    });

    for (int i : evens) std::cout << i << " "; // Output: 2 4
}
```

3. `std::remove_if`

Removes elements from a container that match a condition (requires `.erase()` to actually remove).

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};

    v.erase(std::remove_if(v.begin(), v.end(), [](int x) {
        return x % 2 == 0; // remove even numbers
    }), v.end());

    for (int i : v) std::cout << i << " "; // Output: 1 3 5
}
```

4. `std::accumulate`

Computes the sum (or any accumulation) of elements in a range.

Header required: `#include <numeric>`

```cpp
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::vector<int> v = {1, 2, 3, 4};

    int sum = std::accumulate(v.begin(), v.end(), 0);
    std::cout << "Sum: " << sum << "\n"; // Output: Sum: 10

    int product = std::accumulate(v.begin(), v.end(), 1, [](int a, int b) {
        return a * b;
    });

    std::cout << "Product: " << product << "\n"; // Output: Product: 24
}
```

5. `std::sort`

`std::sort` is used to sort the elements in a range (typically a container like `vector`) in ascending order by default. You can customize the sorting criterion using a comparator, including lambda functions.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {4, 2, 9, 1, 5};

    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a < b;  // Ascending order
    });

    for (int n : numbers)
        std::cout << n << " ";
    return 0;
}
```

Lab exercises

Exercise 1

Employee Sorting and Filtering: You are tasked with managing a list of employees. Each employee has the following details:

- Name (string)
- Age (integer)
- Salary (float)
- Years of Experience (integer)

Your goal is to implement a program that allows sorting and filtering of employees based on different criteria using function pointers and STL containers. The program should offer two main functionalities:

1. Filtering Employees based on salary or experience range.
2. Sorting Employees based on age, salary, or years of experience.

You will be using `std::vector` to store the employee details and `std::sort()` to sort the employees based on the chosen criteria.

Tasks:

1. **Filter Employees:** The program should allow filtering employees by salary range or years of experience range.

   Create two functions for filtering:

   - `filterBySalaryRange()`: Filters employees based on a given salary range (`minSalary`, `maxSalary`).
   - `filterByExperienceRange()`: Filters employees based on a given experience range (`minExperience`, `maxExperience`).

   Use a function pointer to call the appropriate filtering function based on user input.

2. **Sort Employees:** The program should allow sorting employees based on:

   - Age
   - Salary
   - Years of Experience

   Use `std::sort()` and pass the appropriate comparison function as a function pointer to sort employees.

Requirements:

- Define an `Employee` struct with the fields: `name`, `age`, `salary`, and `yearsOfExperience`.
- Use `std::vector` to store the list of employees.
- Implement functions to compare employees by age, salary, and experience.
- Implement functions to filter employees by salary and experience range.

- Implement a menu-driven program allowing the user to:
  - Choose between filtering or sorting employees.
  - Choose the filtering or sorting criterion (salary, experience, age, etc.).
  - Display the filtered or sorted employee list.

Exercise 2

Managing and Sorting Product Inventory using Functors and STL: You are tasked with managing a list of products for an e-commerce platform. Each product has the following details:

- Name (string)
- Category (string)
- Price (float)
- Quantity in stock (integer)

Your goal is to implement a program that offers two functionalities:

1. Sort the products based on either price or quantity.
2. Apply a price increase to all products by a specified amount.

The program will use **functors** for comparison and for applying the price increase, and the products will be stored in a `std::vector`. The program should allow the user to choose between sorting or applying the price increase, and then execute the corresponding operation.

Tasks:

1. **Sort Products:** The program should allow sorting products based on:
   - Price
   - Quantity in stock

   Use the following functors:
   - `CompareByPrice`: A functor to compare products by price.
   - `CompareByQuantity`: A functor to compare products by quantity in stock.

   The program should prompt the user to choose between sorting by price or quantity and sort the products accordingly.
2. **Apply Price Increase:** The program should allow the user to apply a price increase to all products. The amount of increase should be specified by the user.

   Use the functor:
   - `ApplyPriceIncrease`: A functor that applies the price increase to each product.

   The program should update the prices of all products based on the user's input.

Requirements:

- Define a `Product` class with the fields: `name`, `category`, `price`, and `quantityInStock`.
- Use `std::vector` to store the list of products.
- Implement the functors `CompareByPrice`, `CompareByQuantity`, and `ApplyPriceIncrease`.
- Implement a menu-driven program allowing the user to:
  - Choose between sorting products or applying a price increase.
  - Choose the sorting criterion (price or quantity).
  - Display the sorted or updated product list.

Exercise 3

Print 100 random numbers. Each random number must be obtained by calling a lambda function that generates a random number between 1 and 100.

Exercise 4

Use `std::sort` to sort (in ascending order) a vector of integers containing the elements {1, 3, 8, 6, 4, 5, 7, 2, 0, 9}. After sorting is completed, print the number of comparisons that were needed during sorting.

Exercise 5

Given a vector of integers, use `std::transform` and a lambda function to half each number in the vector.

Exercise 6

From a vector of integers, create a new vector that contains only the odd numbers using `std::copy_if` and a lambda function.

Exercise 7

Given a vector of integers, remove all elements that are greater than 50 using `std::remove_if` and a lambda function.

Exercise 8

Given a vector of integers, use `std::accumulate` and a lambda function to calculate the sum of all the elements in the vector.

Exercise 9

Sort a vector of double in descending order using `std::sort` and a lambda function.