C++ LAMBDA EXPRESSION

C++ 11 introduced lambda expressions to allow inline functions which can be used for short snippets of code that are not going to be reused. Therefore, they do not require a name. They are mostly used in STL algorithms as callback functions. The syntax of a lambda is shown below:

```
[capture-clause] (parameters) -> return-type {
   // definition
}
```

- **Return Type**
  - Generally, the return-type in lambda expressions is evaluated by the compiler itself and we don't need to specify it explicitly. However, in some complex cases e.g. conditional statements, the compiler can't determine the return type and explicit specification is required.
- **Parameters**
  - These parameters are similar to the function parameters in every way.
- **Capture Clause**
  - A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope. We can capture external variables from the enclosing scope in three ways using capture clause
    - [&]: capture all external variables by reference.
    - [=]: capture all external variables by value.
    - [a, &b]: capture 'a' by value and 'b' by reference.
  - A lambda with an empty capture clause [] can only access variables which are local to it.

Implement and run the following in a new project, within a file titled **Lambda.cpp**

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

int main() {

    // Defining a lambda
    auto res = [](int x) {
        return x + x;
    };
    cout << res(5);

    return 0;
}
```

The lambda expression in the above program takes an integer x as input and returns the sum of x with itself. For instance, The result of **res(5)** prints 10, as the lambda doubles the value of 5.

Implement and run the following in a new project, within a file titled **CaptureClause.cpp**.

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

void print(vector<int> v) {
    for (auto x : v) cout << x << " ";
    cout << endl;
}

int main() {
    vector<int> v1, v2;

    // Capture v1 and v2 by reference
    auto byRef = [&] (int m) {
        v1.push_back(m);
        v2.push_back(m);
    };

    // Capture v1 and v2 by reference instead of value to modify outer vectors
    auto byVal = [&] (int m) {
        v1.push_back(m);
        v2.push_back(m);
    };

    // Capture v1 by reference and v2 by reference
    auto mixed = [&v1, &v2] (int m) {
        v1.push_back(m);
        v2.push_back(m);
    };

    // Push 20 in both v1 and v2
    byRef(20);

    // Push 234 in both v1 and v2
    byVal(234);

    // Push 10 in both v1 and v2
    mixed(10);

    print(v1);
    print(v2);

    return 0;
}
```

In the **CaptureClause.cpp** program, we have the following:

- **byRef** captures all by reference. So pushing 20 will push it into original v1 and v2.
- **byVal** captures all by value. So pushing 234 will not do anything to original vectors.
- mixed captures v1 by reference and v2 by value. So pushing 10 will only push it into v1.
- The **mutable** keyword here is used in capture by value lambdas only because, by default, value captured objects are **const**.

Lamda expressions are extensively used in STL in place of callback i.e. functions passed as arguments.

Implement and run the following in a new project, within a file titled **VectorSort.cpp**. In addition find out about **#include <algorithm>** (STL algorithms).

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

int main() {
        vector<int> v = {5, 1, 8, 3, 9, 2};

        // Sort in descending order
        sort(v.begin(), v.end(), [] (const int& a, const int&b) {
                return a > b;
        });

        for (int x : v)
                cout << x << " ";
        return 0;
}
```

The program in **VectorSort.cpp** sorts the contents contained within a vector, in descending order.

Implement and run the following in a new project, within a file titled **VectorFind.cpp**.

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

int main() {
        vector<int> v = {5, 1, 8, 3, 9, 2};

        // Sort in descending order
        auto it = find_if(v.begin(), v.end(), [] (const int& a) {
                return a % 3 == 0;
        });

    if (it != v.end()) cout << *it;
        else cout << "No such element";
        return 0;
}
```

The program in **VectorFind.cpp** finds the first within a vector, which is divisible by 3, if that number exists.

Lambda expressions' main purpose was to replace the functions in callbacks by providing inline definitions. Following are the common applications of lambda expressions in C++

- Inline, Anonymous Functions: Write small functions directly where needed without naming them.
- STL Algorithms: Pass custom comparison or transformation logic to algorithms like sort, for_each, etc.
- Callbacks and Event Handling: Use lambdas as callbacks for asynchronous operations or event handlers.
- Threading and Concurrency: Pass lambdas to threads for quick, inline tasks without defining separate functions.
- Custom Comparators in Containers: Use lambdas as comparators for containers like priority_queue, set, etc.

References

- https://www.geeksforgeeks.org/cpp/lambda-expression-in-c/