Lambda expression in C++

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 and therefore do not require a name. In their simplest form a lambda expression can be defined as follows:

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

Generally, the return-type in lambda expressions is evaluated by the compiler itself and we don’t need to specify it explicitly. Also the -> return-type part can be ignored. However, in some complex cases e.g. conditional statements, the compiler can’t determine the return type and explicit specification is required.

Various uses of lambda expression with standard functions are given below :

```
// C++ program to demonstrate lambda expression in C++
#include <bits/stdc++.h>
using namespace std;

// Function to print vector
void printVector(vector<int> v)
{
    // lambda expression to print vector
    for_each(v.begin(), v.end(), [](int i)
    {
        std::cout << i << " ";
    });
    cout << endl;
}

int main()
{
    vector<int> v {4, 1, 3, 5, 2, 3, 1, 7};
    printVector(v);

    // below snippet find first number greater than 4
    // find_if searches for an element for which
    // function(third argument) returns true
    vector<int>:: iterator p = find_if(v.begin(), v.end(), []
    {
        return i > 4;
    });
    cout << "First number greater than 4 is : " << *p << endl


    // function to sort vector, lambda expression is for sort
    // non-increasing order Compiler can make out return type
    // bool, but shown here just for explanation
    sort(v.begin(), v.end(), [](const int& a, const int& b) -
    {
        return a > b;
    });

    printVector(v);

    // function to count numbers greater than or equal to 5
    int count_5 = count_if(v.begin(), v.end(), [](int a)
    {
        return (a >= 5);
    });
    cout << "The number of elements greater than or equal to
        << count_5 << endl;

    // function for removing duplicate element (after sorting
    // duplicate comes together)
    p = unique(v.begin(), v.end(), [](int a, int b)
    {
        return a == b;
    });

    // resizing vector to make size equal to total different
    v.resize(distance(v.begin(), p));
    printVector(v);

    // accumulate function accumulate the container on the ba
    // function provided as third argument
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int f = accumulate(arr, arr + 10, 1, [](int i, int j)
    {
        return i * j;
    });

    cout << "Factorial of 10 is : " << f << endl;

    //   We can also access function by storing this into var
    auto square = [](int i)
    {
        return i * i;
    };

    cout << "Square of 5 is : " << square(5) << endl;
}
```

Output

```
 4 1 3 5 2 3 1 7
First number greater than 4 is : 5
7 5 4 3 3 2 1 1
The number of elements greater than or equal to 5 is : 2
7 5 4 3 2 1
Factorial of 10 is : 3628800
Square of 5 is : 25
```

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 :

Capture by reference

Capture by value

Capture by both (mixed capture)

Syntax used for capturing variables :

[&] : 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.

Different methods of capturing are demonstrated below :

```
// C++ program to demonstrate lambda expression in C++
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> v1 = {3, 1, 7, 9};
    vector<int> v2 = {10, 2, 7, 16, 9};

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

    // it pushes 20 in both v1 and v2
    pushinto(20);

    // access v1 by copy
    [v1]()
    {
        for (auto p = v1.begin(); p != v1.end(); p++)
        {
            cout << *p << " ";
        }
    };

    int N = 5;

    // below snippet find first number greater than N
    // [N] denotes, can access only N by value
    vector<int>:: iterator p = find_if(v1.begin(), v1.end(),
    {
        return i > N;
    });

    cout << "First number greater than 5 is : " << *p << endl

    // function to count numbers greater than or equal to N
    // [=] denotes, can access all variable
    int count_N = count_if(v1.begin(), v1.end(), [=](int a)
    {
        return (a >= N);
    });

    cout << "The number of elements greater than or equal to
        << count_N << endl;
}
```

Output

```
First number greater than 5 is : 7
The number of elements greater than or equal to 5 is : 3
```