Lambda Expressions and callbacks

1. Lambda expressions, usage
2. Capture Syntax
3. Generic Lambdas
4. std::function
5. std::bind

1.What is Lambda in c++?

-> Lambda is basically small line snippets of code that can be used inside functions or even function call statements. They are not named or reused.

-> Lambdas can be declared as “auto” and use them anywhere in program.

->Lambdas for effective callback mechanism, without the need for defining named functions in advance

2. Syntax of Lambda ?

-> General syntax of defining lambdas is as follows:

```
(Capture clause) (parameter_list) mutable exception ->return_type
{
Method definition;
}
```

-> Capture closure: Lambda introducer as per c++ specification.

-> Parameter list: Also called as lambda declarations. Is optional and is similar to the paramter list of a method.

-> Mutable: Optional. Enables variables captured by value to be modified.

-> exception: Exception specification. Optional. Use “noexcept” to indicare that lambda does not throw exception.

-> Return_type: Optional. The compiler deduces the return type of the expression of its own. But as lambdas get more complex, it is better to include return type as the compiler may not be able to deduce the return type.

-> Method definition: Lambda body.

-> Capture clause: Used to specify which variables are captured (stored in object) and whether they are captured and whether they are captured by reference or by value.

Empty capture closure [], indicates that no variables are used by lambda which means it can only access variables that are local to it.

“Capture-default” mode indicates how to capture outside the variables referenced in lambda.

* Capture closure [&] means the variables are captured by reference.
* Capture closure [=] indicates that the variables are captured by value.
* [ref] capture all variables by reference
* [val] capture single variable by value
* [&val] capture single variable by reference
* [=, &x] capture all by value, specific variable by reference
* [x, y] capture both x, y by value
* [&x, &y] capture both x, y by reference
* [&x, y] capture x by reference, y by value
* [this] capture current object(*this) by reference
* [std::move(expr)] Move capture
* [] no capture

3. Some examples of lambda expressions?

3.1The below example explains the usage of lambda expression.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
auto sum = [](auto a, auto b) {
return a + b;
};
cout <<"Sum of two integers:"<< sum(5, 6) << endl;

return 0;
   }
```

```
Output: Sum(5,6) = 11
             Sum(2.0,5.6)= 7.6
               Sum((string("SoftwareTesting"), string("help.com")) = SoftwareTestinghelp.com
```

-> In above example, we can see instead of using seperate function of sum(), which we were doing traditionally, here we have used lambda expression.

-> The above example shows generic lambda, because we can concatenate strings, integer or any other data type.

3.2 Example to demonstrate capture syntax

Example1: Simple example

```
#include <iostream>

using namespace std;

int main()
{
   int i = 0;
auto foo = [i](){ cout << i << endl; }; //capture by value
auto bar = [&i](){ cout << i << endl; }; //capture by reference
i=10;

foo();
bar();

    return 0;
}
```

```
Output: 0
            10
```

Example 2: Detailed Example

```
// 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(), [N](int i)
{
return i > N;
});

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

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

cout << "The number of elements greater than or equal to 5 is : "
<< count_N << endl;
}
```

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

-> We can also do move capture by doing following:

```
    [ rr= std::move(s1)] (int x) {
//captured move instance of s1
```

4. **std::function**

-> Instances of std::function can store, copy, and invoke any Callable target -- functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.

-> Parameters in std::function are: **R**-result_type
argument_type- T if sizeof..(Args)==1 and T is first and only type in Args

-> Syntax: std:function<R(argument_type)> name.
Eg: std::function<void(int)> f_display

-> Alternative way of declaring **callbacks**, instead of conventional **function pointers** / named functions.

-> Function Pointer: It is basically passing pointers to functions.
Example:

```
 #include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
printf("Value of a is %d\n", a);
}

int main()
{
// fun_ptr is a pointer to function fun()
void (*fun_ptr)(int) = &fun;

/* The above line is equivalent of following two
void (*fun_ptr)(int);
fun_ptr = &fun;
*/

// Invoking fun() using fun_ptr
(*fun_ptr)(10);

return 0;
}
```

Output:

```
Value of a is 10
```

-> *Callback:* If a reference of a function is passed as an argument to another function as an argument, then it is called as callback function.

Example:

```
// A simple C program to demonstrate callback
#include<stdio.h>

void A()
{
printf("I am function A\n");
}

// callback function
void B(void (*ptr)())
{
(*ptr) (); // callback to A
}

int main()
{
    void (*ptr)() = &A;
    // calling function B and passing
    // address of the function A as argument
    B(ptr);

return 0;
}

Output: I am function A
```

-> Here we are passing address of function A as parameter to function B and we are calling function A via function B.

-> Now let us come back to std::function.
  Let us understand it with an example:

```
#include <functional>
#include <iostream>

struct Foo {
   Foo(int num) : num_(num) {}
   void print_add(int i) const { std::cout << num_+i << '\n'; }
   int num_;
};

void print_num(int i) {
   std::cout << i << '\n';
}

struct PrintNum {
   void operator()(int i) const {
      std::cout << i << '\n';
   }
};

int main() {
   std::function<void(int)> f_display = print_num;
   f_display(-9);

   std::function<void()> f_display_42 = []() { print_num(42); };
   f_display_42();

   std::function<void()> f_display_31337 = std::bind(print_num, 31337);
   f_display_31337();

   std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
   const Foo foo(314159);
   f_add_display(foo, 1);

   std::function<int(Foo const&)> f_num = &Foo::num_;
   std::cout << "num_: " << f_num(foo) << '\n';

   using std::placeholders::_1;
   std::function<void(int)> f_add_display2= std::bind( &Foo::print_add, foo, _1
);
   f_add_display2(2);

   std::function<void(int)> f_add_display3= std::bind( &Foo::print_add, &foo, _1
);
   f_add_display3(3);

   std::function<void(int)> f_display_obj = PrintNum();
   f_display_obj(18);
}
```

Output:

```
-9
42
31337
314160
num_: 314159
314161
314162
18
```

-> In the above example, we have demonstrated the example of std::function. But one more observation, we have also used std::bind in the above example. Details about std::bind will be explained in the next section.

5. <u>**std::bind**</u>

-> Sometimes we need to manipulate the operation of a function according to the need. (i.e changing some arguments to default etc.)

-> Predefining function to have default arguments restricts the versatility of function and forces us to use default arguments that too with similar value each time.

-> Bind function with help of placeholders helps to manipulate the position and number of values to be used by the function and modifies the function according to the desired output.

-> Placeholders are namespace which direct the position of a value in a function. They are represented by _1,_2,_3 .

-> Let us see below example to understand std::bind function.

```
// C++ code to demonstrate bind() and
// placeholders
#include <iostream>
#include <functional> // for bind()
using namespace std;

// for placeholders
using namespace std::placeholders;

// Driver function to demonstrate bind()
void func(int a, int b, int c)
{
cout << (a - b - c) << endl;
}

int main()
{
// for placeholders
   using namespace std::placeholders;

    // Use of bind() to bind the function
    // _1 is for first parameter and assigned
    // to 'a' in above declaration.
    // 2 is assigned to b
    // 3 is assigned to c
    auto fn1 = bind(func, _1, 2, 3);

    // 2 is assigned to a.
    // _1 is for first parameter and assigned
    // to 'b' in above declaration.
    // 3 is assigned to c.
    auto fn2 = bind(func, 2, _1, 3);

    // calling of modified functions
    fn1(10);
fn2(10);

  return 0;
}

Output:
5
-11
```

-> From the above example, we can observe that bind modified the call of a function to take 1 argument and return the desired output.

-> It gives flexibility because we can fix default parameters of function at any point of time and then pass the value we want at desired position in function.

5.1 Binding objects using std::bind

-> Let us understand it with a example:

```
 class Box {
int l;
Converting member functions as
int b;
global functions, this is useful
int h;
when some API can accept global
public:
functions only
Box(int x, int q, int r):l(x),b(y),h(r) { }
int volume() { return l * b * h; }
void zoom(double scale) { }
Void update(int p,int q, int r);
};
Box b1(10,12,5);
auto fvolume = std::bind(&Box::volume, b1);
auto fzoom = std::bind(&Box::zoom, b1, ::1);
Auto fupdate = std::bind(&Box::update, b1 , ::1, ::2, ::3);
//fvolume(15) ==> b1.volume();
//fzoom(1.5) ==> b1.zoom(1.5);
```

6. Differences between lambda and std::bind

-> Lambdas are useful when we need to add little bit logic on top of an existing function.

-> In bind, you are foeced to create new function, even if logic is needed in one place.

-> With lambda, you can add new logic inside lambda.

-> For using in STL algorithms, lambdas are better choice.

-> Expressions cant be captured, only identifiers can. For bind you can write:

```
        auto f1 = std::bind(f, 42, _1, a + b);
```