Lambda Expression

IN MODERN C++

Formal definition

A lambda is an ad-hoc, locally-scoped functor. Basically, lambdas are syntactic sugar, designed to reduce a lot of the work required in creating ad-hoc functor classes

Callback Concept

The callback is basically an additional logic, usually in the form of a function, defined by the user of a service, to personalize and/or expand the service itself

Example: personalizing a «find» method providing the criteria for finding a specific item

Lambda expression are useful especially (but not only) when a callback should be passed

```cpp
class myClass
{
public:
    string name;
    double a, b;
};

bool bGreaterA(const myClass& element)
{
    return  element.b >  element.a;
}


int main(unsigned int argc, char** argv)
{
    std::vector<myClass> myvec =
    {
        {"foo", 10, 20},
        {"bar", 100, 1}
    };
    std::find_if(myvec.begin(), myvec.end(),bGreaterA); // returns foo);
    return 0;
}
```

Pre-Lambda ways to pass a callback

- Defining a function, often only for a single callback usage, and pass its address
- Defining a functor, creating and valorizing it and pass it as a callback. A functor is a class that provides an implementation of operator(). Its advantage is that being an object it can carry data along with him and his operator() can interact with his data

```
class MyMultiplicator
{
public:
    double y;
    double operator()(double x){return x*y;}
}
```

```
int main(int argc, char** argv)
{
    MyMultiplicator multi;
    multi.y = 15;
    auto res = multi(10); //will return 150
    return 0;
}
```

Major flaws of the pre-lamda era

Creating bespoke functors or functions can be a lot of effort; especially if the functor/function is only used in one specific place.

These bespoke functors also unnecessarily ‘clutter up’ the code.

Lambda expression sintax

[ ] → here we can specify some object in the same scope of the lambda to be visible inside the body of the function. We can «Capture» them. We can also specify to capture them by value or by reference with the use of «&»

{ }→this will contain the body of our function

[<capture>](<arguments>){<body>}

( )→this will contain the arguments of the function as usual

Capture list

- Nothing: []
- All by reference: [&]
- All by value: [=]
- this by reference: [this]
  - implicit when using [=] or [&]
- [C++17] *this by value : [*this]
  - never implicit
- Specific variables:

```
int x = 2;
auto multiply_val = [x] (int a) { return x*a; };
auto multiply_ref = [&x] (int a) { return x*a; };
x = 3;
multiply_val(3); // returns 6
multiply_ref(3); // returns 9
```

Return type

- [C++14] Multiple return statements allowed for auto return type deduction
  - Same as return type deduction for normal functions
  - Types of multiple return statements need to match
- [C++11] If body consists of single return statement, return type deduced automatically
- If automatic deduction not possible: specify with "-> Type"

```
auto clamp = [](float angle) -> float {
    if (angle < 0)
        return 0; // int
    else
        return angle; // float
};
```

Lambda expression sintax

Here is the example of before but using the lamba

```
class myClass
{
public:
    string name;
    double a, b;
};



int main(unsigned int argc, char** argv)
{
    std::vector<myClass> myvec =
    {
        {"foo", 10, 20},
        {"bar", 100, 1}
    };
    std::find_if(myvec.begin(), myvec.end(), [](const auto& elem){ return elem.a > elem.b}); // returns foo
    return 0;
}
```

C++98

```
class Unnamedclass
{
    int* pX;
public:
    Unnamedclass(cont int& pXin) : pX(&pXin){ }

    double operator()(const double& y){ return y*(*pX); }
};
```

```
int x = 10;
double y = 15;
Unnamedclass myfunctor(x);
auto res = myfunctor(y); // returns 150
```

C++11

```
int x = 10;
double y = 15;
auto mylambda = [ &x ] (const double& y) { return y*x; }
auto res = mylambda(y);
```

Other pourposes

- Lambda expression can be used for defining function/subroutine into other funtion, limiting the code pollution
- Can effectively substitute std::bind or boost::bind, wrapping other function