**CALLBACK FUNCTIONS**

*Functions which take other functions as input (parameter) and use it to implement complex logic*. Callback functions can be implemented using the below mentioned callables (5). In other words, the following entities can be passed as arguments to the callback functions to be called from the callback functions.

1. Function pointers - (since C)

2. Lambda expressions - C++

3. Function objects (Functors) - C++

4. std::function objects from <functional> library - C++

5. Binding expressions - C++

These callables can be called just like any other function - callable()

These can be wrapped by the std::function class present in the <functional> library that is the variable which can store a callable has a data type as specified below. ***std::function<returnType(parameterDatatype)>***

Refer [std::function::function - C++ Reference](http://www.cplusplus.com/reference/functional/function/function/)

Implementation of the callables and the callbacks depends upon the use case (what we need to achieve), for example - looking at the already implemented callback functions in the <algorithm> library of cpp, we have to implement the callables which can work with the implemented callbacks.

**[FUNCTION POINTER] - [RAW FUNCTION POINTER]**

Out of all the callables, Function pointers are available since C while others are newly introduced in C++

Function pointers definition and declaration has a weird syntax.

**[FUNCTION POINTER] - [CONCEPT]**

Address of the instructions in the memory

Name of a function is its function pointer that is - function or &function (Implicit type cast similar to arrays). Function pointer is a way to store the function address in a variable which can be used to be passed as an argument to another function (callback function).

**[FUNCTION SIGNATURE] - [KEY CONCEPT]**

Every function has a function signature which contains the following components:

1. return type

2. function name: Function pointer that is the address of the instructions in the memory

3. parameter list

For example the function below,

void print(int value);

has the function signature as

void(\*AnyNameCanBeSpecifiedHere)(int)

which is also the datatype of the function pointer - print

**[1][DECLARATION OF A NEW FUNCTION POINTER TO A FUNCTION]**

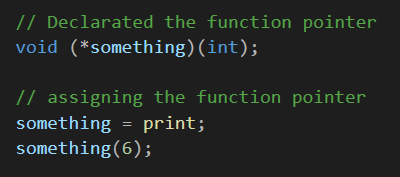
**[1.A][Weird Syntax of function pointers]**

Function signature is tied to

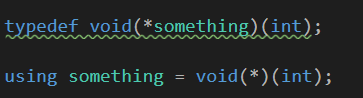
- Declaration of the function pointer

- Datatype of the function pointer : (a) auto (b) typedef (syntax is eased)

Case(a) Function Signature for declaration of function pointer itself.



Case(b) Function Signature for declaration of function pointer data type which later can be used for declaration of function pointers with the same function signature.



**[Important Point]**

*Functions with the same signature have the same function pointer*. Hence, the same function pointer can be used to point to functions with the same signature. The <algorithm> library of cpp contains the callback templates implemented.

[Out of context] Note that we do not need to specify the datatype as template parameter while calling the template functions like std::sort because of 'automatic template parameter deduction' which is present in C++11 for functions but not for the classes however it is introduced for classes as well in C++17

* Implement callback function (ForEach) using function pointers as callables
* Implement callback function (ForEach) using lambda function as a callable

**[LAMBDA EXPRESSIONS] - [ANONYMOUS FUNCTIONS]**

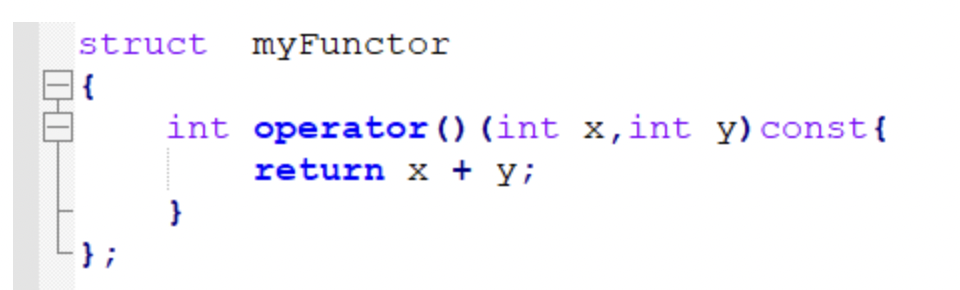
Throw-away function. We get the same functionality as a function but without actually creating a function. Refer [Lambda expressions (since C++11) - cppreference.com](https://en.cppreference.com/w/cpp/language/lambda) for details about syntax.

Use cases -

1. Lambda can be used wherever you can use the raw function pointers.
2. Whenever you want to create a function inside the function itself.
3. A lambda expression can be named by using the *auto* keyword.
4. Important points to be careful about:
   1. If we add anything to the [capture] then, the lambda is no longer a raw function pointer. It has to be wrapped by the *std::function<returnType(paramType)> object.*
   2. If the [capture] is made by value(copy) and we attempt to change the value captured, then we have to use the *mutable* keyword with the lambda. Refer to the documentation for the syntax.
   3. The returnType of the Lambda can also be specified using -> operator. Refer to the documentation.

**[FUNCTION OBJECTS] - [FUNCTORS] - [FUNCTIONAL OPERATOR]**

Function object or Functor or Functional operator is an object of class that has the function call operator() overloaded in its definition. An example is provided below.



Advantages of functors:

1. Function objects can store the state, which means that all the variables need not be passed as an argument to the function but they can be stored as member variables in the object itself. Moreover, it also means that function can behave differently at the run time depending upon the state of the object.
2. Functors are used as a predicate in the algorithms provided in the <algorithm> STL library of C++. These can modify the behavior of the callback functions in the library.

Important point:

1. Do not confuse the constructor call with the functor call.
2. [Object Creation - Functor call]. *Distinguish carefully.*
3. First the object is created and it can be created using a parameterized constructor or a default constructor.
4. Then the functor call is made using the function call operator.

Goodreads:

Functor: [Functors in C++. A functor is a function object which… | by Nishi Tyagi | Medium](https://medium.com/@nishityagi521/functors-in-c-b772e05b69bc)

Functional Programming: [From Objects to Functions. An introduction to functional… | by Severin Perez | Medium](https://medium.com/s/story/from-objects-to-functions-9f945e2a5c16)

**[STD::FUNCTION CLASS OBJECTS IN <functional> C++ LIBRARY]**

It defines commonly used templated function objects which can be passed as arguments to the other callback algorithms.

std::function::function class in <functional> can wrap all the 4 callables (1-4) listed above.

[std::function::function - C++ Reference](https://www.cplusplus.com/reference/functional/function/function/)

1. Function pointers
2. Lambda expressions
3. Function objects
4. std::function objects in <functional> library