**How to Implement Coroutines in C++?**

## What are Coroutines in C++?

The C++ coroutines are a control structure in which the control flow is passed along with the different routines without returning. The C++ coroutine feature is added in the C++20. The C++ coroutine is a function that can prevent the execution that needs to be resumed later. Coroutines are utilized in numerous stages of development just like the execution of code non-concurrently with a set of legitimate consecutive codes in place. The C++ coroutines are of stackless behavior which is why this function returns the result to the caller and continues the specified coroutine that's again stored within the isolated area or stack that's as of now defined.

## Need of C++ Coroutines

To read a record and define it whereas reading into a few essential and meaningful pieces of information, you can either read step by step at each line, which is fine. It is also possible that we can load all the large content in the memory, but that would not be suggested for large text file cases such as MS Word, text editors, etc.

For solving this problem, Donald Knuth introduced a solution for this problem in computer programming. Donald Knuth said that we can remove the stack concept completely. We do not need to let any process as the caller and caller. We can just consider them as cooperating equals.

## Implementation of Coroutines in C++

For implementing C++ coroutines, there is the requirement of the following two things:

1. Resume control to its last state
2. Make data persist through calls

int read(void)

{

int i;

for (i = 0; i < 10; i++)

return i; */\* here the function itself*

*destroyed on the first run\*/*

}

The above problem can be solved by using static variables. But how to keep in mind the state additionally returns to the same execution state as some time recently i.e., the lines of code after the return or the loop? We can use here the GOTO statements. Let us see the solution below.

*// here is the C++code to demonstrate how the coroutines*

*// can be implemented in C++.*

#include <iostream>

using namespace std;

int range(int a, int b)

{

static long long int i;

static int state = 0;

switch (state) {

case 0: */\* this is the start of the function \*/*

state = 1;

for (i = a; i < b; i++) {

return i;

*/\* Returns control \*/*

case 1:

cout << "control at range"

<< endl; */\* here, resume control straight*

*after the return \*/*

}

}

state = 0;

return 0;

}

*// here is the main function code*

int main()

{

int i; *// For really large numbers*

for (; i = range(1, 5);)

cout << "control at main :" << i << endl;

return 0;

}

We now basically have numerous return explanations defined inside the for loop and each time we return to a diverse state to execute the program on the off chance that it is programmed to do so. This approach represents a function in python named the **range** function of python. That function of python is also based on the C++ coroutine concept of C++.

## How Do C++ Coroutines Work?

The C++ coroutines work as described below.

* **C++ coroutines** are functions that can be resumed at a later point even when the primary execution gets completed. The C++ coroutines mainly call the main function and then the data is saved in another location. Whenever we use coroutines as a function then we must associate its data type to make it work.
* The C++ coroutines are always attached with variating arguments and return statements which makes them perfect coroutines. But there still exist some restrictions. The constructors, destructors, constant expression function, and the main function can not form the perfect coroutines.

Some important concepts related to C++ coroutines are:

**Coroutine execution paradigm:**

A C++coroutine is associated with two things. The first one is the promise object and the second one is the coroutine object. A promise object is an object which can store a value that can be retrieved by any future a=object (generally another thread) and it offers a synchronization point. And the coroutine state is used with a heap where the heap is available in an optimized manner. The state object has the following properties:

1. available Parameters.
2. Until the current suspension point, the local variables and the temporary variables have a little time stamp and scope.
3. Promise objects for manipulation within.
4. To understand where to resume and continue the execution, we represent the proper state and value of the local variables.

**coroutine handles:**

Whenever we need to compile and execute the outside scope of the coroutine frame we use the coroutine handles. We use the non-use handling handlers of the coroutine hold to resume the execution of the coroutine and simultaneously delete the C++ coroutines from the frame. A C++coroutine handle is somewhat similar to a c pointer which can be easily copied but we do not have an associated destructor to destroy the occupied memory. So, we must use the thecoroutine\_handle:: destroy method to destroy the coroutine and avoid memory leakage. When the coroutine is destroyed, the coroutine handler which was pointing to the deleted coroutine. The coroutine handler will now point to a garbage value and hence it will not work when invoked. Even if the control flows in and out of the coroutines, a coroutine handle is still valid for executing the entire coroutine.

**coroutine return object:**

The return type of the C++coroutine is an object with a nested type ::promise\_type. The **r::promise\_type** should include a method namely r get\_return\_object which returns an outer object of type r. The get\_return\_object function returns a coroutine function.

**The promise object:**

The coroutine state has an instance of promise type. We can add a field namely value to transmit the values from the coroutine to the main function. We do not convert the coroutine handle to a std::coroutine\_handle<>, we keep it as std::coroutine\_handle<ReturnObject3::promise\_type>.

**The co\_yield operator:**

The co\_yield operator yields a value from the expression and returns the value to the caller. It suspends the current coroutine and it is one of the most common building blocks of the resumable generator functions.

**The co\_return operator:**

The co\_return operator signals the end of a coroutine. There are three ways of doing the signaling.

1. Co\_return e can be used to return the value e.
2. To denote the end of a coroutine without a final value, the co\_routine can use the co\_return operator.
3. If we want to fall off the end of the function, we can use the co\_return operator just like the previous point.

## Important Points to Remember When Using Coroutines

Following are some important points that we should always keep in mind while using the C++ coroutines.

* The C++ coroutines can only use the co\_return operator but it can not use return.
* We can not use varargs and it can not be a constexpr.
* It can not be a constructor or a destructor.
* The coroutines can not be in the main function as well.
* When we are using the C++ coroutines, we should use parameters by value to be on the safer side and avoid the dangling reference.

## Coroutines and Reference Parameters

The C++ coroutines are implemented in such a way that the entire call stack is not saved when the coroutine is suspended. We only store the local variable. This kind of implementation of the coroutine is known as a stack full coroutine. On the other hand, we have a stack-less coroutine in which the entire call stack is saved and restored. A coroutine reference parameter similar to any other asynchronous parameter takes a reference parameter that the caller object must provide. This ensures that the referenced object continues to exist throughout the existence of the object.

## Examples of C++ Coroutines

Given below are some examples that will help us to understand how the C++ coroutines work.

* **Example - 1:**  
  This example shows how the coroutine greedy generator works when there is a program in which the function returns all the integers from the start to the end of the given program.
* #include <iostream>
* #include <vector>
* std::vector<int> get\_the\_greedy\_Num(int begin, int end, int inc = 1) {
* std::vector<int> greedy\_nums;
* for (int j\_0 = begin; j\_0 < end; j\_0 += inc) {
* greedy\_nums.push\_back(j\_0);
* }
* return greedy\_nums;
* }
* int main() {
* std::cout << std::endl;
* const auto greedy\_nums= get\_the\_greedy\_Num(-4, 15);
* for (auto n: greedy\_nums) std::cout << n << " ";
* std::cout << "\n\n";
* for (auto n: get\_the\_greedy\_Num(1, 123, 5)) std::cout << n << " ";
* std::cout << "\n\n";
* }

**Output:**

-4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

1 6 11 16 21 26 31 36 41 46 51 56 61 66 71 76 81 86 91 96 101 106 111 116 121

* **Example -2:**  
  In this example, we used the C++ coroutines for storing the values that are being returned from the output. And that output is called by the returning value of the future as given below.
* #include <coroutine>
* #include <iostream>
* #include <memory>
* template<typename T\_0>
* struct date\_future {
* std::shared\_ptr<T\_0> value;
* date\_future(std::shared\_ptr<T\_0> p\_1): value(p\_1) {}
* ~date\_future() { }
* T\_0 get() {
* return \*value;
* }
* struct promise\_type {
* std::shared\_ptr<T\_0> ptr = std::make\_shared<T\_0>();
* ~promise\_type() { }
* date\_future<T\_0> get\_return\_object() {
* return ptr;
* }
* void return\_value(T\_0 k) {
* \*ptr = k;
* }
* std::suspend\_never initial\_suspend() {
* return {};
* }
* std::suspend\_never final\_suspend() noexcept {
* return {};
* }
* void unhandled\_exception() {
* std::exit(1);
* }
* };
* };
* date\_future<int> createFuture\_0() {
* co\_return 2022;
* }
* int main() {
* std::cout << '\n';
* auto fut\_1 = createFuture\_0();
* std::cout << "fut\_1.get(): " << fut\_1.get() << '\n';
* std::cout << '\n';
* }

**Output:**

// Possible output-

fut\_1.get(): 2022

## Conclusion

* The C++ coroutines are a control structure in which the control flow is passed along with the different routines without returning.
* The C++ coroutine was introduced afterward from the C++11.
* The C++coroutine is a stack-based operation.
* The C++ coroutines are functions that can be resumed at a later point even when the primary execution gets completed.
* Two things are associated with the C++ coroutines:
  + the promise object and
  + the coroutine object.
* The **coroutine handle** is used to pause or stop the currently executing coroutine.