## Introduction to std::packaged\_task

This lesson gives an introduction to std::packaged\_task which is used in C++ for multithreading.

we'll cover the following ^
• Explanation:

std::packaged\_task pack is a wrapper for a callable, which allows it to be
invoked asynchronously. By calling pack.get\_future(), we get the associated
future. Invoking the call operator on pack (pack()) executes the
std::packaged\_task and, therefore, executes the callable.

Dealing with std::packaged\_task usually consists of four steps:

I. Wrap our work:

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a + b; });
```

II. Create a future:

```
std::future<int> sumResult= sumTask.get_future();
```

III. Perform the calculation:

```
sumTask(2000, 11);
```

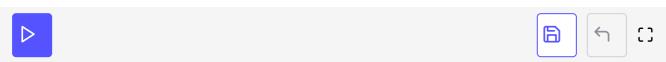
IV. Query the result:

```
sumResult.get();
```

Here is an example showing the four steps.

```
// packagedTask.cpp
#include <utility>
#include <future>
#include <iostream>
#include <thread>
#include <deque>
class SumUp{
 public:
    int operator()(int beg, int end){
      long long int sum{0};
     for (int i = beg; i < end; ++i ) sum += i;
     return sum;
};
int main(){
  std::cout << std::endl;</pre>
 SumUp sumUp1;
  SumUp sumUp2;
 SumUp sumUp3;
 SumUp sumUp4;
 // wrap the tasks
  std::packaged task<int(int, int)> sumTask1(sumUp1);
  std::packaged_task<int(int, int)> sumTask2(sumUp2);
  std::packaged_task<int(int, int)> sumTask3(sumUp3);
  std::packaged_task<int(int, int)> sumTask4(sumUp4);
  // create the futures
  std::future<int> sumResult1 = sumTask1.get_future();
  //std::future<int> sumResult2 = sumTask2.get_future();
  auto sumResult2 = sumTask2.get_future();
  std::future<int> sumResult3 = sumTask3.get future();
  //std::future<int> sumResult4 = sumTask4.get future();
  auto sumResult4 = sumTask4.get_future();
  // push the tasks on the container
  std::deque<std::packaged_task<int(int,int)>> allTasks;
  allTasks.push_back(std::move(sumTask1));
  allTasks.push back(std::move(sumTask2));
  allTasks.push_back(std::move(sumTask3));
  allTasks.push_back(std::move(sumTask4));
  int begin{1};
  int increment{2500};
  int end = begin + increment;
  // perform each calculation in a separate thread
 while (not allTasks.empty()){
    std::packaged_task<int(int, int)> myTask = std::move(allTasks.front());
    allTasks.pop_front();
    std::thread sumThread(std::move(myTask), begin, end);
   begin = end;
    end += increment;
    sumThread.detach();
```

0



The purpose of the program is to calculate the sum of all numbers from 0 to 10000 with the help of four std::packaged\_task s, each running in a separate thread. The associated futures are used to sum up the final result; of course, we can also use the Gaußschen Summenformel.

## Explanation: #

- I. Wrap the tasks: We packed the work packages in std::packaged\_task (lines 28 31) objects. Work packages are instances of the class SumUp (lines 9 16). The work is done in the call operator (lines 11 15) which sums up all numbers from beg to end -1 and returns the sum as the result.

  std::packaged\_task (lines 28 31) can deal with callables that need two int s and return an int: int(int, int).
- II. **Create the futures**: We have to create the future objects with the help of std::packaged\_task objects (lines 34 to 3). The packaged\_task is the promise in the communication channel. The type of the future is defined explicitly as std::future<int> sumResult1= sumTask1.get\_future(), but the compiler can do that job for us with auto sumResult1= sumTask1.get\_future().
- III. **Perform the calculations**: Now the calculation takes place. First, the packaged\_task is moved onto the std::deque (lines 43 46), then each packaged\_task (lines 54 59) is executed in the while loop. To do that, we moved the head of the std::deque in an std::packaged\_task (line 54), moved the packaged\_task in a new thread (line 56) and let it run in the background (line 59). We used the move semantic in lines 54 and 56 because std::packaged\_task objects are not copyable. This restriction holds for all promises, but also for futures and threads. However, there is one exception to

this rule: std::shared\_future.

IV. **Pick up the results**: In the final step we asked all futures for their values and summed them up (line 63).

Let's solve the exercise in the next lesson.