std::packaged\_task enables you to write a simple wrapper for a callable, which you can invoke later.

std::packaged\_task

To deal with std::packaged\_task, you have to perform four steps usually:

1. Wrap the task
2. Create the future
3. Perform the calculation
4. Pick up the result

These steps are easy to get with an example.

// 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;

SumUp sumUp1;

SumUp sumUp2;

SumUp sumUp3;

SumUp sumUp4;

// define 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);

// get the futures

std::future<int> sumResult1= sumTask1.get\_future();

std::future<int> sumResult2= sumTask2.get\_future();

std::future<int> sumResult3= sumTask3.get\_future();

std::future<int> 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;

// execute each task 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();

}

// get the results

auto sum= sumResult1.get() + sumResult2.get() + sumResult3.get() + sumResult4.get();

std::cout << "sum of 0 .. 10000 = " << sum << std::endl;

std::cout << std::endl;

}

The job of the program is not so heavy. Calculate the sum from 0 to 10000 with the help of four threads and sum up the results with the associated futures. Of course, you can use the [Gaußschen Summenformel](https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Summenformel). (Strange, there is no English page, describing this famous algorithm. But math is international.).

In the**first step**, I pack the work packages in std::packaged\_task (line 28 - 31) objects. Work packages are instances of the class SumUp (line 9 - 16). The current work is done in the call operator (line 11-15). The call operator sums up all numbers from beg to end and returns the sum as result. std::packaged\_task in line 28 - 31 can deal with callables, that need two ints and return an int.

Now, I have to create in the **second step** the future objects with the help of std::packaged\_task objects. Exactly that is done in the lines 34 to 37. The packaged\_task is the promise in the [communication channel](http://modernescpp.com/index.php/tasks). I define in these lines explicitly the type of the future: std::future<int> sumResult1= sumTask1.get\_future(), but of course, I can do it also implicitly: auto sumResult1= sumTask1.get\_future().

In the **third step**, the work takes place. The packaged\_task are moved onto the std::deque (line 40 -44). In the while loop, each packaged\_task (line 51-58) is executed. For doing that, I move the head of the std::deque in a std::packaged\_task (line 52), move the packaged\_task in a new thread (line 54) and let it run in the background (line 56). std::packaged\_task object are not copyable. That's the reason, why I used move semantic in the lines 52 and 54. These restriction holds for all promises, but also futures and threads. There is one exception to this rule: std:.shared\_future.

In the **fourth and last step**, the four futures ask with the get call for the results and sum them up (line 61).

Honestly, std::packaged\_task was not made for simple use case like std::async. But the result is simple.

### Optimization potential

C++11 has a function std::thread\_hardware\_concurrency. It provides a hint for the numbers of cores on your system. In case the C++ runtime has no glue, it's conforming to the standard to return 0. So you should verify that value in your program (line 31). With the current GCC, clang or Microsoft compiler, I get the right answer 4. I use this number of cores In the variation of the program packagedTask.cpp, to adjust the software to my hardware. So, my hardware is fully utilized.

// packagedTaskHardwareConcurrency.cpp

#include <algorithm>

#include <future>

#include <iostream>

#include <thread>

#include <deque>

#include <vector>

class SumUp{

public:

SumUp(int b, int e): beg(b),end(e){}

int operator()(){

long long int sum{0};

for (int i= beg; i < end; ++i ) sum += i;

return sum;

}

private:

int beg;

int end;

};

static const unsigned int hwGuess= 4;

static const unsigned int numbers= 10001;

int main(){

std::cout << std::endl;

unsigned int hw= std::thread::hardware\_concurrency();

unsigned int hwConcurr= (hw != 0)? hw : hwGuess;

// define the functors

std::vector<SumUp> sumUp;

for ( unsigned int i= 0; i < hwConcurr; ++i){

int begin= (i\*numbers)/hwConcurr;

int end= (i+1)\*numbers/hwConcurr;

sumUp.push\_back(SumUp(begin ,end));

}

// define the tasks

std::deque<std::packaged\_task<int()>> sumTask;

for ( unsigned int i= 0; i < hwConcurr; ++i){

std::packaged\_task<int()> SumTask(sumUp[i]);

sumTask.push\_back(std::move(SumTask));

}

// get the futures

std::vector< std::future<int>> sumResult;

for ( unsigned int i= 0; i < hwConcurr; ++i){

sumResult.push\_back(sumTask[i].get\_future());

}

// execute each task in a separate thread

while ( not sumTask.empty() ){

std::packaged\_task<int()> myTask= std::move(sumTask.front());

sumTask.pop\_front();

std::thread sumThread(std::move(myTask));

sumThread.detach();

}

// get the results

int sum= 0;

for ( unsigned int i= 0; i < hwConcurr; ++i){

sum += sumResult[i].get();

}

std::cout << "sum of 0 .. 100000 = " << sum << std::endl;

std::cout << std::endl;

}