Test Library Tutorial and Reference

Bill Rubin
6/3/2021

The Test Library, motivated by the multi-threading code in Bill Collier's BASH program, is a small C++ library which can run multiple algorithms concurrently in separate threads. Unlike in BASH, the Test Library contains no platform-specific code. Instead, it uses the standard C++ thread support library. Therefore, the same code can be used in Windows, MacOS, Linux, or any other platform for which a modern C++ compiler is available.

The Test Library supplies a threading environment for running a collection of tasks, and for making all execution results available in the main thread. The user must supply the algorithms and the code to analyze the results.

Table of Contents

Test Library Tutorial

The four steps of Test execution

The Test Library uses the following steps:

1. Create the Test object.
2. Fill Test object with tasks specifying algorithms to run.
3. Run Test.
4. Analyze Test results.

Simplest possible program

The Test Library's basic functionality using the above four steps can be demonstrated with a simple program which invokes a toy algorithm (not shown) to factor the number 123456789:

```
#include "Test.h"             // The Test Library
#include "FactorAlgorithm.h"  // The factor algorithm
#include "FactorAnalysis.h"   // Function to print the results

int main() {
    Test test;                                // Step 1
    test.fill(FactorAlgorithm());             // Step 2
    test.run();                               // Step 3
    analyze(test.tasks<FactorAlgorithm>());  // Step 4
    return 0;
}
```

Executing the above program on a machine with 8 processors runs the algorithm 8 times in 8 separate threads, and produces the following output:

```
Number to factor: 123456789
Factors:
Task 0:  3 3 3607 3803
Task 1:  3 3 3607 3803
Task 2:  3 3 3607 3803
Task 3:  3 3 3607 3803
Task 4:  3 3 3607 3803
Task 5:  3 3 3607 3803
Task 6:  3 3 3607 3803
Task 7:  3 3 3607 3803
```

Running all tasks in a single thread

Instead of running the algorithm 8 times in 8 separate threads, you can run these 8 tasks on the calling (main) thread by replacing Step 3 with

```
     test.run(false);                     // Step 3 – single threaded
```

The output will be the same as above.

Specifying number of tasks

Instead of defaulting the number of threads to the number of processors, you can specify in Step 1 how many separate tasks to run. For example, to run only 3 tasks, replace Step 1 with:

```
     Test test(3);                        // Step 1 – Specify 3 tasks
```

The output is now:

```
Number to factor: 123456789
Factors:
Task 0:  3 3 3607 3803
Task 1:  3 3 3607 3803
Task 2:  3 3 3607 3803
```

Specifying parameters to the algorithm

Instead of defaulting to factor the number 123456789, you can specify in Step 2 what number to factor. For example, to factor the Mersenne prime $2^{31} - 1 =$ 2147483647, replace Step 2 with:

```
     test.fill(FactorAlgorithm(2147483647));    // Step 2 – parameter
```

The output is

```
Number to factor: 2147483647
Factors:
Task 0:  2147483647
Task 1:  2147483647
Task 2:  2147483647
```

Specifying different parameters for different tasks

Instead of having all tasks run the factor algorithm on the same number, you can specify in Step 2 that different tasks run the factor algorithm on different numbers. For example, the following code runs 3 tasks to factor 123456789, two tasks to factor 2147483647, and the remaining 3 tasks (on a machine with 8 processors) to factor 1073741824:

```
int main() {
    Test test;                                 // Step 1
    test.add(FactorAlgorithm(123456789), 3);   // Step 2 (begin)
    test.add(FactorAlgorithm(2147483647), 2);  // Step 2 (continued)
    test.fill(FactorAlgorithm(1073741824));    // Step 2 (end)
    test.run();                                // Step 3
    analyze(test.tasks<FactorAlgorithm>());    // Step 4
    return 0;
}
```

The output is:

```
Number to factor: 123456789
Factors:
Task 0:  3 3 3607 3803
Task 1:  3 3 3607 3803
Task 2:  3 3 3607 3803

Number to factor: 2147483647
Factors:
Task 3:  2147483647
Task 4:  2147483647

Number to factor: 1073741824
Factors:
Task 5:  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Task 6:  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Task 7:  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Specifying different algorithms for different tasks

Finally, you can run different algorithms on different tasks. For this purpose, I've introduced a subtract algorithm that subtracts one number from another. (My creative spurt clearly waned at this point.) See last page. The following program demonstrates interleaved algorithms:

```
#include "Test.h"
#include "FactorAlgorithm.h"
#include "SubtractAlgorithm.h"
#include "FactorAnalysis.h"

int main() {
    Test test;                                     // Step 1
    test.add(SubtractAlgorithm(61, 31), 2);        // Step 2 (begin)
    test.add(SubtractAlgorithm(29, 61));           // Step 2 (continued)
    test.add(FactorAlgorithm(123456789), 2);       // Step 2 (continued)
    test.fill(SubtractAlgorithm(13, 7));           // Step 2 (end)
    test.run();                                    // Step 3
    analyze(test.tasks<FactorAlgorithm>());        // Step 4 (begin)
    analyze(test.tasks<SubtractAlgorithm>());      // Step 4 (end)
    return 0;
}
```

The resulting output is:

```
Number to factor: 123456789
Factors:
Task 3:  3 3 3607 3803
Task 4:  3 3 3607 3803
Task 0: 61 - 31 = 30
Task 1: 61 - 31 = 30
Task 2: 29 - 61 = -32
Task 5: 13 - 7 = 6
Task 6: 13 - 7 = 6
Task 7: 13 - 7 = 6
```

The above analyze function overloads supplied by the client in Step 4 have been declared as follows:

```
void analyze(const ConstTaskVector<FactorAlgorithm>&);
void analyze(const ConstTaskVector<SubtractAlgorithm>&);
```

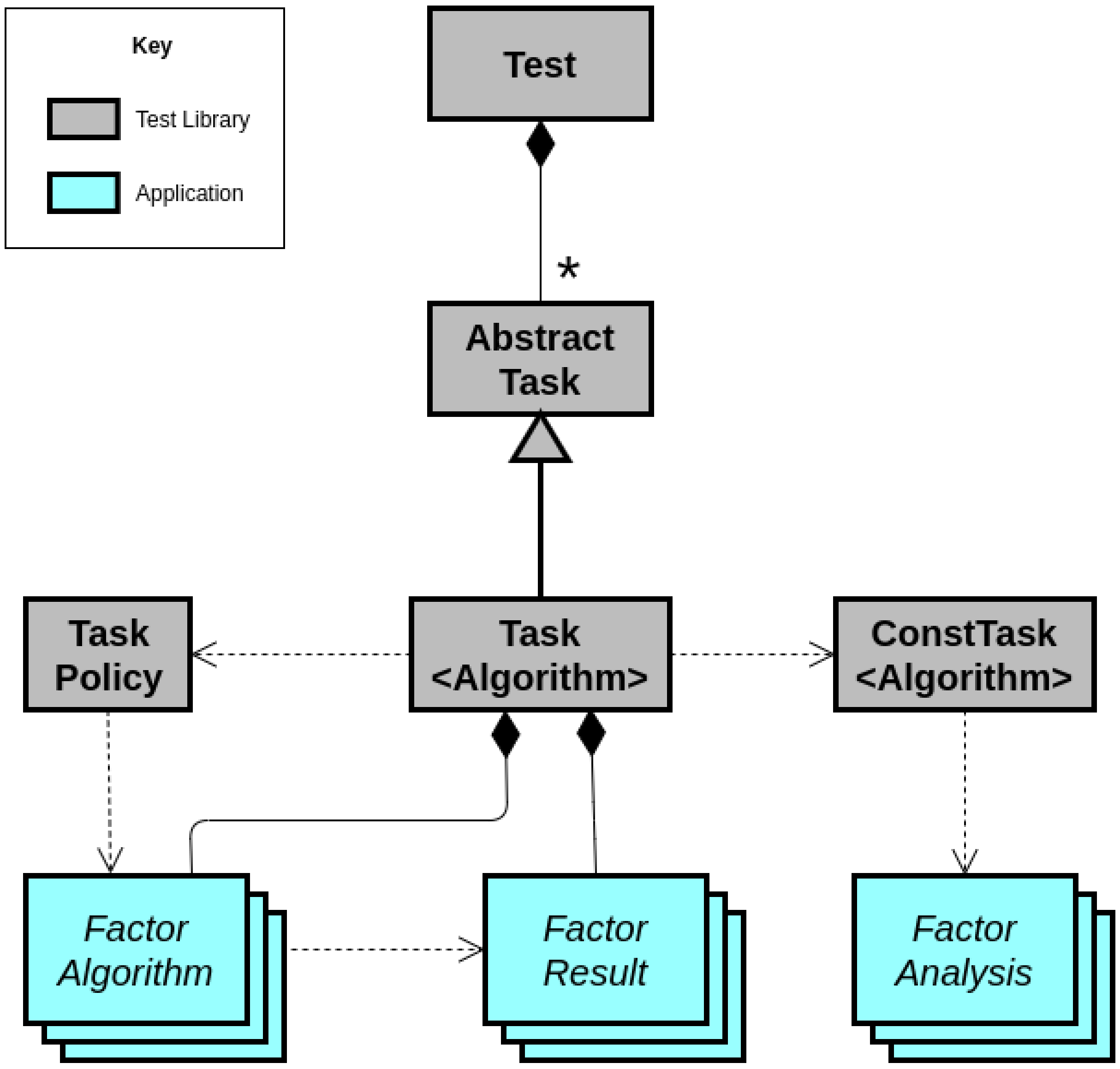
This separation may make sense if the two different algorithm types are to be analyzed separately. However, if the two algorithm types are to be analyzed together, the client could design a single analyze function with signature something like

```
void analyze(const ConstTaskVector<FactorAlgorithm>&,
             const ConstTaskVector<SubtractAlgorithm>&);
```

and invoke it in a single line.

Test Library Reference

The Test Library comprises interfaces to run tests, to execute algorithm instances, and to analyze results. The overall structure of the Test Library is shown in the following UML class diagram. Dotted arrows are used to show the flow of data. Other symbols are standard UML.



Use by clients: Only the **`Test`** class is to be instantiated by clients. The other Test Library classes are instantiated implicitly by **`Test`** member functions. A const reference to a **`TaskPolicy`** object is passed to the Algorithm's **`operator()`** by the run member function. Const references to **`ConstTask`** objects are returned by the **`tasks`** member function. The **`AbstractTask`** class and the **`Task<Algorithm>`** class template are not used directly by clients.

The `Test` class: The main interface of the Test Library is the **`Test`** class, which is used to run tests. A **`Test`** object is essentially a container for a set of tasks, where a task is an object containing an instance of an algorithm to be run as well as the result of running that algorithm instance. Each task in a test is run in its own thread by default, but it can be specified that all tasks run in the same thread.

Creating a test object: Test objects are created with slots for adding a specified number of tasks. Upon creation, a test is in the **`unfilled`** state.

State transitions: `Test` objects can be put in various states by various non-const member functions, as described in the following narrative and shown in the state diagram on the next page. The state diagram does not show cases where member functions do not complete successfully (see individual member function descriptions). Such member function invocations do not cause state transitions.

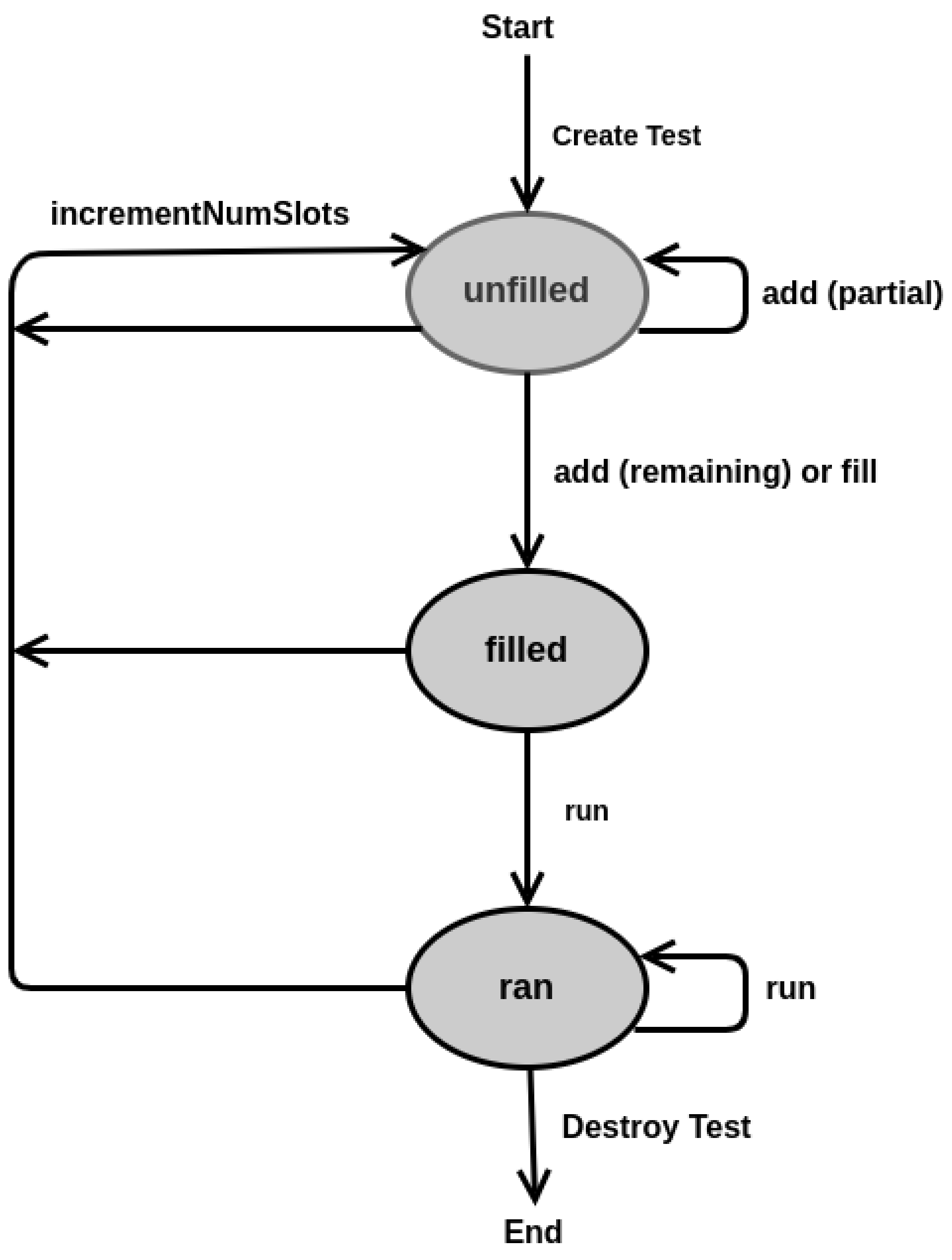
Filling a test object: Tasks can be added to a test using the **`add`** member function or the **`fill`** member function. The **`add`** member function adds one or more tasks to the test, but cannot add more tasks than there are slots. Once all the slots have been filled, the test is in the **`filled`** state. The **`fill`** member function adds enough tasks to fill any empty slots. Both the **`add`** and the **`fill`** member functions leave previously added tasks undisturbed.

Running a test object: A test in the **`filled`** state can be run by invoking the **run** member function. The **`run`** member function runs all the algorithms in all the tasks, waits for them all to complete, and sets the test to the **`ran`** state. A test in the **`ran`** state can also be run.

Analyzing the results: A test in the **`ran`** state can return all the execution results by invoking the **`tasks`** member function, once for each **`Algorithm`** type in the test. These results can be passed to any analysis programs the user may provide.

Increasing the number of slots for tasks: The number of slots in a test can be increased (but not decreased) at any time by invoking the **`incrementNumSlots`** member function. Doing so leaves previously added tasks undisturbed, but puts the test in the **`unfilled`** state.

Test Library State Diagram



Class Test

The **`Test`** class is the interface for assembling and running tests.

Types

Test::Size

```
using Size =  unsigned long;
```

Test::State

```
enum State {unfilled, filled, ran};
```

Member Functions

Returning an error condition: Member functions that can return an error condition are currently declared as returning a **`bool`** type, with **`true`** for error and **`false`** for no error. That way, if later enhancements require member functions to return an error condition as an **`int`** type, previous client code is less likely to be affected. When a member function returns **`true`**, it does not affect the state of the test.

Test::hardwareConcurrency

```
static Size hardwareConcurrency() noexcept;
```

Returns the hardware concurrency (number of processors, maximum number of concurrent tasks) supported by the machine. If the hardware concurrency is unavailable, returns **`1`**.

Test::Test (constructor)

`Test(Size numTasks = hardwareConcurrency());`

If **`numTasks`** > 0, constructs a **`Test`** object with space allocated for **`numTasks`** tasks, and sets **`State`** to **`unfilled`**. If **`numTasks`** == 0, throws a **`std::invalid_argument`** exception.

Test::state

`State state() const noexcept;`

Returns the **`State`** of the **`Test`** object.

Test::numTasks

`Size numTasks() const noexcept;`

Returns the number of tasks added to the test.

Test::numSlots

`Size numSlots() const noexcept;`

Returns the number of slots in the test.

Test::numEmptySlots

`Size numEmptySlots() const noexcept;`

Returns **`numSlots() - numTasks()`**.

Test::incrementNumSlots

`bool incrementNumSlots(Size increment = 1) noexcept;`

If **`increment`** > 0, increments the number of slots allocated for tasks by **`increment`** tasks, destroys all results if **`State`** was **`ran`**, sets **`State`** to **`unfilled`**, and returns **`false`**. Otherwise, returns **`true`**.

Test::add

```
template<class Algorithm>
bool add(const Algorithm& algorithm, Size numTasks = 1);
```

If **`numTasks`** > 0 and **`numTasks <= numEmptySlots()`**, adds **`numTasks`** tasks to the test, each with its own copy of **`algorithm`**, sets the state of the test to **`filled`** if the test is filled, and returns **`false`**. Otherwise, return **`true`**.

Test::fill

`template<class Algorithm> bool fill(const Algorithm&);`

Invokes **`add`** with **`numTasks`** set to the value returned by **`Test::numEmptySlots()`**.

Test::run

`bool run(bool separateThreads = true);`

If **`State`** is **`unfilled`**, returns **`true`**. Otherwise, runs all tasks, waits for them to complete, sets **`State`** to **`ran`**, and returns **`false`**. If **`separateThreads`** is **`true`**, each task is run in its own thread; otherwise, all threads are run in the calling thread.

Test::tasks

```
template<class Algorithm>
const ConstTaskVector<Algorithm> tasks() const noexcept;
```

The return value from **`tasks`** is easier to use than it is to describe. For a simple usage example, see the definition of **`SubtractAlgorithm::analyze`** below, which takes as an argument the return value of **`tasks`** and prints the results.

The **`ConstTaskVector`** class template is defined as:

```
template<class Algorithm>
using ConstTaskVector = std::vector<ConstTask<Algorithm>>;
```

The **`ConstTask<Algorithm>`** elements of this vector correspond to all the **`Algorithm`** tasks added to the test, in the order in which they were added.

Class Template ConstTask

The **`ConstTask`** class template has the following member functions:

```
const Algorithm& algorithm() const noexcept;
```

Returns a const reference to the algorithm of the corresponding task in the test.

```
unsigned long taskNumber() const noexcept;
```

Returns the index of the corresponding task in the test. Note that this index is not necessarily the index of the ConstTask in the ConstTaskVector if the test contains interleaving tasks corresponding to other algorithms.

```
const ResultPtr& result() const noexcept;
```

Returns a const reference to the **`Algorithm::ResultPtr`** of the corresponding task in the test. If **`State`** is **`ran`**, this pointer points to the result of the last invocation of **`run`**. Otherwise,this pointer is effectively **`nullptr`**.

Class TestPolicy

The **`TestPolicy`** class is the interface for executing algorithms. A const reference to an instance of this class is passed as an argument to the **`operator()`** function of each algorithm's function object. The **`TaskPolicy`** parameter allows read access to the number of tasks in the current test, the task number of the current task, and the launch policy of the tasks. It also offers an overload of **`operator<<`** to stream a TestPolicy to a standard output stream.

Types

Test::Size

```
using Size =  unsigned long;
```

Member Functions

```
Size numTasks() const noexcept;
```

Returns the total number of tasks in this test.

```
Size taskNumber() const noexcept;
```

Returns the index of this task in the sequence of slots in this test.

```
std::launch launchPolicy() const noexcept;
```

Returns the standard launch policy for this test, which is determined by the **`separateThreads`** argument to **`Test::run`**. If **`separateThreads`** is **`true`**, **`launchPolicy`** returns **`std::launch::async`**. If **`separateThreads`** is **`false`**, **`launchPolicy`** returns **`std::launch::deferred`**. This interface allows for additional **`std::launch`** values, which would be implementation-defined.

```
std::ostream& operator<<(std::ostream&, const TaskPolicy&);
```

This **`operator<<`** overload is a convenience function for debugging. Typical output:

```
  Number of tasks: 8
  Task number: 3
  Launch policy: async
```

Algorithm Interface

Algorithms used to execute tasks within the Test Library must be constructed as function objects whose **`operator()`** performs the desired algorithm to run in the task. The following code sketches a typical implementation, where code in ***`italics`*** indicates application-specific code:

```
#include "Test.h"
#include <memory>

class TaskPolicy;

class SomeAlgorithm {
    public:
        SomeAlgorithm(ctor_parms ...);    // Optional initialization
        using ResultPtr = std::unique_ptr<result_type>;  // Required

        ResultPtr operator()(const TaskPolicy&) {        // Required
            auto result = std::make_unique
               <ResultPtr::element_type>         // Initialize result
               (optional_result_type_ctor_args ...);
            ...                                    // Compute result
            return result;                         // Return result
            }
};
```

Constructor: The algorithm's constructor is optional, but is often useful to specify different behaviors for different instances. Algorithm instances must be copyable because the Test Library will copy them into each task.

Result of algorithm: The algorithm class must specify a ***`result_type`*** with a public declaration of the form:

```
using ResultPtr = std::unique_ptr<result_type>;
```

The ***`result_type`*** is intended to hold the results of computation for an instance of the algorithm class. The Test Library will make an encapsulated const pointer to the ***`result_type`*** instance available to analysis functions. The ***`result_type`*** can be any class (or even a built-in type) suitable to the purposes of the algorithm.

Operator(): The function object's **`operator()`** must have the signature **`ResultPtr operator()(const TaskPolicy&)`**. The **`operator()`** must instantiate the ***`result_type`*** and return a smart pointer to it. The instantiation is most easily accomplished with the statement:

```
auto result = std::make_unique<ResultPtr::element_type>
     (optional_result_type_ctor_args ...);
```

See the Reference section below for optional use of the **`TaskPolicy`** parameter.

Analysis Interface

The interface to analysis functions is the **`Test::tasks()`** member function template which takes the type of the algorithm as a template parameter. After a run completes, **`tasks<`*`algorithm`*`>()`** can be called for each ***`algorithm`*** in the run. It returns a result of type **`const ConstTaskVector<`*`algorithm`*`>`** where **`ConstTaskVector`** is of the form **`std::vector<ConstTask<`*`algorithm`*`>>`** and **`ConstTask`** is a class template giving access to a const reference to the algorithm function object for the task, the task number for the task, and a const reference to the **`ResultPtr`** of the task. In this way, analysis functions can readily access all the inputs and outputs of all the tasks in the run.

The following page shows how the subtract algorithm, and its analysis function, interface with the Test Library.

Example: Subtract Algorithm and Analysis Function

Following is the entire source code for the subtract algorithm and its analysis function, comprising a file named **`SubtractAlgorithm.h`**. Text that is required for the Test Library is in color:

```
#include "Test.h"
#include <memory>
#include <iostream>

class TaskPolicy;

class SubtractAlgorithm {
    public:
        using long_number_type = long long;
        using ResultPtr = std::unique_ptr<long_number_type>;

        SubtractAlgorithm(long_number_type minuend,
                          long_number_type subtrahend)
            : minuend_(minuend), subtrahend_(subtrahend) {}

        ResultPtr operator()(const TaskPolicy&)
            {return std::make_unique<ResultPtr::element_type>(
                                             minuend_ - subtrahend_);}

        friend std::ostream& operator<<(std::ostream& os,
                                        const SubtractAlgorithm& sa)
            {os<<sa.minuend_<<" - "<<sa.subtrahend_<<" = ";
             return os;}

    private:
        const long_number_type minuend_;
        const long_number_type subtrahend_;
};


void analyze(const ConstTaskVector<SubtractAlgorithm>& tasks) {
    for(auto& task : tasks)
        std::cout<<"Task "<<task.taskNumber()<<": "
                 <<task.algorithm()<<*task.result()<<'\n';
}
```