Performer 1.0 User Manual

This user manual aims to provide an introduction to the framework and the basic uses that will help the user understand how it works. It will help give you the skills necessary to build on and eventually becoming more proficient and advance with the framework. This manual is by no means a complete guide to everything about the framework but will enable to use full framework features with ease, after reading this manual it is suggested to read through the frameworks API (header files).

The Performer 1.0 is a dynamic performance analysis framework designed that gives users the ability to do performance analysis on their applications by utilizing the framework directly in user code. For the purpose of this manual, the same example code will be used to demonstrate the frameworks capability and instructions for use.

All the example source code shown here can be found in the frameworks /example folder.

Table of Contents:

1. How to Install

The framework already provides pre-built libraries for Linux in the /libs folder, you only need to put the root directory (performer-1.0) in your include environment variables and the /libs folder in your linker environment variables. You must also either have the boost 1.3.9+ and Xerces-C 3.0+ installed or if you do not have them then the required libraries are provided in /extlibs folder.

If you wish to build from source then there is a .mak file provided that can be ran from console in Linux using "make -C *path to .mak* -f *nameOfMakFile*" or in Windows using nmake tool.

The framework was built using Code::Blocks IDE and the necessary project file is provided if you wish to open it in Code::Blocks and build from there. Code::Blocks is available in Windows and Linux (Highly Recommended).

2. Basic Use and Introduction to the Framework

Throughout this manual, we will be using our framework to do performance analysis on the following example code:

```
 9  int main()
10  {
11      //create a vector, list and set of ints
12      vector<int> intVec;
13      list<int> intList;
14      set<int> intSet;0
15
16      //insert operatins to fill all three containers
17      for(int i=0; i<10; i++){
18          intVec.push_back(i);
19      }
20      for(int i=0; i<10; i++){
21          intList.push_back(i);
22      }
23      for(int i=0; i<10; i++){
24          intSet.insert(i);
25      }
26
27      //search a value in the container
28      find(intVec.begin(), intVec.end(), 5);
29      find(intList.begin(), intList.end(), 5);
30      find(intSet.begin(), intSet.end(), 5);
31
32      //erase
33      intVec.erase(intVec.begin(), intVec.end());
34      intList.erase(intList.begin(), intList.end());
35      intSet.erase(intSet.begin(), intSet.end());
36
37      //insert operatins to refill containers
38      for(int i=0; i<=100; i++){
39          intVec.push_back(i);
40      }
41      for(int i=0; i<100; i++){
42          intList.push_back(i);
43      }
44      for(int i=0; i<100; i++){
45          intSet.insert(i);
46      }
47
48      //re-fill values in container
49      int y = 1;
50      fill(intVec.begin(), intVec.end(), (++y)*2);
51      y = 1;
52      fill(intVec.begin(), intVec.end(), (++y)*2);
53      y = 1;
54      fill(intVec.begin(), intVec.end(), (++y)*2);
55
56      //clear all containers
57      intVec.clear();
58      intList.clear();
59      intSet.clear();
60
```

This example code does not do anything magical, its a simple code that creates a vector, list and a set and does some standard operations of add, delete, edits and search. All containers get treated the same way, with same elements being added, deleted and etc. They only differ with the operations they each use to complete these tasks e.g. push_back vs insert.

Now suppose we want to time how fast it takes to add all the elements, we can time it before and after the for-loop to get a total time for all the elements.

To do this first we must include the necessary header file:

<performer/performer.h> //this header file will include the whole framework

```
#include <vector>
#include <list>
#include <set>
#include <iostream>
#include <algorithm>
#include <performer/performer.h>
```

To do our measurements we will use a **Data Collector** object that will take care of using the necessary platform specific components to do the measuring and also manage the results for us including saving it. The Data Collector needs however a **configuration object** to tell it important information like where to save and file name to use. A configuration object is helpful as it allows you to centralize all the settings in one place for a single **source** that is being measured. All these objects will be created using **factory** functions and a **factory class**. This is the **recommended** way and will help keep the code portable between Linux and Windows.

You maybe wondering what a **source** is, a source is what you are interested in measuring, i.e. vector, list and set are all sources your interested in measuring so we need a configuration object for each on to manage theme separately. With a source we will have **sections** that we are interested in, for example its vague to say we want to measure a vector. What do we want to measure about it? What are we interested in? Well in this case we want to measure performance for its add operations, search operations and delete operations. These are all **sections** that we want to measure. Its good to familiarize with these terminologies as they will be used in the examples and in the framework.

When including header files make sure to always use the <performer/....> style, append the world **performer/** for all header files. When trying to include specific classes from one of the three modules make sure to append the module name after the “performer/”. You may wish to examine the “performer” folder in your installation directory to view the directory structure.

Now that you are familiar with the basic usage guideline and with some of the terminologies, the rest of the manual will demonstrate a variety of tasks that can be accomplished with the framework.

Note 1: For all the examples in the rest of the document, it is assumed you always include the <performer/performer.h> header file, you will not always be reminded of this!

Note 2: The framework does not use any namespaces so there is no need to use any “using” statements.

Note 3: All objects retrieved using factory methods including the factory class are all dynamically allocated so make sure to delete t hem!

3. Measuring Speed and Memory

Speed is broken down in different sub categories of “wall-clock time” and “CPU time”. CPU time is also further broken down in “system time” and “user time”.

Memory similar is broken down into various categories, up to 27 different measures.

As a reminder, first make sure you have read the above section and have included the framework header file(s).

First we will use the main factory function to get our platform specific implementation of PerformerFactory object:

```
12      //create factory
13      PerformerFactory *pf = getPerfFactory();
```

We get a pointer to a dynamically allocated PerformerFactory object. Now using the factory object we need to create a configuration object to manage each of our sources (vector, list and set).

```
15      //config objects
16      ConfigFile *vecConf = pf->createConfigObject("vector", "vecResult", "results");
17      ConfigFile *listConf = pf->createConfigObject("list", "listResult.xml", "results/");
18      ConfigFile *setConf = pf->createConfigObject("set", "setResult.xml", "results/");
```

Here we are creating configuration objects for measuring and saving new data, therefore we create the configuration objects by giving a name of the source, the name of the file we want to save it as and the location to save it. Notice that for file name we have used ".xml" in one and left out ".xml" in another, for the file location similar we used "/" at the end and left "/" out. Either way it does not matter, the framework is smart enough to figure out what to do. However for save location if you specify a directory you **MUST** make sure it **exists**! The framework will not create it for you.

If you would like to specify the current directory as the save location just use "" or " " or leave it blank.

Just as we created configuration objects for each source we want to manage, we need to create data collector objects for each source we want to measure and pass the configuration objects for each source to the respective data collectors.

```
20      //create data collectors
21      DataCollector *vecDc = pf->createDataCollector(vecConf);
22      DataCollector *listDc = pf->createDataCollector(listConf);
23      DataCollector *setDc = pf->createDataCollector(setConf);
```

Now we can measure a section of code, we shall measure performance for total of all the add operations in a vector by placing the necessary code before and after the for loop.

```
30      //insert operatins to fill all three containers
31
32      vecDc->startSection("push_back"); //measure from here
33
34      for(int i=0; i<10; i++){
35          intVec.push_back(i);
36      }
37
38      vecDc->stopSection(); //stop measuring here
```

Notice for the startSection() call we gave a parameter, this specifies the name of the section we are measuring. You can use the same name multiple times and the framework will group all the same section names together, analysis will also be based on results of each section. Note: For memory this measures the change in memory readings between the start and stop section. In some applications you may get many 0's. To take a memory snapshot of the process use memPoll() function instead. Like this: vecDc->memPoll("section name", true) where true means to save this result.

The above method can be repeated for all the operations for each source, see the example folder for the file “simple_measure.cpp” for this examples source code, . Finally if we build our file and run it, a .xml file is created with our results.

4. Using the Data Collector to View Results

Continuing on from our previous example, we managed to gather our performance data and even save it. However we can also use the Data Collector object to view our result during runtime through the standard output

```
136     speed_map sm;
137     mem_map mm;
138
139     vecDc->getSpeedResult(&sm);
140     vecDc->getMemResult(&mm);
141
142     cout << sm << endl;
```

Create speed_map and mem_map structures and pass their pointers to be filled in. Do not pass pointers unless they were created from the new operator!

You can simply print the structures using standard stream operations, when run you should get a similar output.

```
****************** Speed Results ******************

##### section-name = clear #####

wallTime: 1.65432e+07
cpuTime: 16001
usrTime: 12000
sysTime: 4001

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

##### section-name = erase #####

wallTime: 1.72599e+07
cpuTime: 16001
usrTime: 16001
sysTime: 0
```

Note for printing you can use printResult() function in the data collector object, it is a short cut for printing all the results gathered so far. You can also measure individual results like this:

```
38     ResultInfo ri = vecDc->stopSection(); //stop measuring here
39
40     cout << ri << endl; //print the result that was just measured
```

5. Perform Statistical Analysis

In the previous examples we have managed to gather results, save them and view them during runtime. This examples continues on and moves on to the analyzer module which will generate statistical information from the result. This is achieved by using PerformerFunctors (part of the framework) that generate these statistics, the framework provides 4 built-in (min, max, avg, and std-deviation). You can also create your own however that is an advanced topic not covered in this manual and guides on this may be released in the near future.

Here is the analysis code:

```
132     //get the speed and memory results
133     speed_map sm;
134     mem_map mm;
135
136     vecDc->getSpeedResult(&sm);
137     vecDc->getMemResult(&mm);
138
139     //analyze and print analyzed result
140     Analyzer *anz = pf->createAnalyzer(vecConf, sm, mm);
141     cout << anz->getResult() << endl;
```

When the analyzer is constructed with both speed and/or memory results, the results can be printed immediately after construction. When run you will get a console output similar to this:

```
******************** Result of Analysis ******************

result-type = speed:

        section-name = clear:

                category = average:
                        cpuTime: 8001
                        sysTime: 0
                        usrTime: 8001
                        wallTime: 8.39131e+06

                category = max:
                        cpuTime: 8.39131e+06
                        sysTime: 8.39131e+06
                        usrTime: 8.39131e+06
                        wallTime: 8.39131e+06

                category = min:
                        cpuTime: 8.39131e+06
                        sysTime: 8.39131e+06
                        usrTime: 8.39131e+06
                        wallTime: 8.39131e+06
```

6. Loading Existing Results with Data Collector

So far we have created our own results and saved them, what if we wanted to load existing results and carry on using it to add more results or analyze them again? We can do this using a Data Collector object.

```
12      //create factory
13      PerformerFactory *pf = getPerfFactory();
14
15      //use config object to set what file to load
16      ConfigFile *vecConf = pf->createConfigObject("result/vecResult.xml");
17
18      //pass config to data collector and it will load the result
19      DataCollector *vecDc = pf->createDataCollector(vecConf);
```

Here the framework has loaded an xml file and constructed and xml object in memory and linked it to the file it was constructed from, the data collector object will also take care of updating the configuration object with any data such as the source this file was measuring. You can use this in the same way as you have done in the previous examples with any differences.

7. Create Timers and Memory Profilers without Data Collector

Using a Data Collector object is the recommended way to measure performance, it assists in code portability across platforms and also takes care of management duties. In the windows version only the factory route is allowed, however with Linux libraries you can bypass the factory methods and use them directly. Using timers and memory profilers directly gives you direct access but you must also manage it yourself. With this example you can include only the headers you need as so:

```
1   #include "performer/data_collection/resultxml.h"
2   #include "performer/data_collection/linux/lintimer.h"
3   #include "performer/data_collection/linux/linmemprof.h"
4   #include "performer/analysis/analyzer.h"
5   #include <iostream>
```

Here only Linux components are included as well as other header files that you may wish to use like the analyzer and resultxml for analyzing and saving. Below is the code extract that creates a timer and memory profiler for the Linux platform and uses it to measure performance. The API should be self explanatory if you have read all previous parts of the manual:

```
10      LinTimer lt;
11      LinMemProfiler lmp;
12      lt.startTimer();
13
14      lt.stopTimer();
15      lt.startTimer();
16
17      lmp.poll();
18      lt.stopTimer();
19
20      struct MemInfo mi = lmp.getMemInfo();
21
22      cout << mi << endl;
```

8. Using the Result XML objects Directly

In the previous example it was demonstrated how to use platform specific timers and memory profilers directly, however when using that method you also need to manage that data and store it somewhere. Here it is demonstrated how to use a ResultXML object in combination with the previous example to store you results:

```
11          LinTimer lt;
12          LinMemProfiler lmp;
13          lt.startTimer();
14
15          ResultXML xp("wtest.xml", "atsource");
16
17          lt.stopTimer();
18          lt.startTimer();
19
20          xp.addSpeedResult(lt.getResult(), "xmlTest");
21          xp.addMemResult(lmp.getMemInfo(), "xmlTest2");
22
23          lmp.poll();
24          lt.stopTimer();
25
26          xp.addSpeedResult(lt.getResult(), "xmlTest");
27
28          struct MemInfo mi = lmp.getMemInfo();
29
30          cout << mi << endl;
31
32          xp.addMemResult(mi, "xmlTest3");
33
34          xp.save();
```

You can see that a ResultXML object was created by giving a file name to save it as and name of the source to measure. These parameters required, if you leave out the source then the framework will assume you want to load the file in first parameter. The first parameter does not have to be just a name it can be directory location to a file as in it can be “/dir/subdir/file.xml”.

To add results to the XML document you may call addSpeedResult() and addMemResult() by providing the respective structures as first parameter and the section that was measure as second parameters. This can be seen in the example above.

When an ResultXML object is destroyed, it will automatically save your document to file but you can request to save the document before that by calling save().