Algorithms

C++ Merge Sort Project Example

0. Objectives

This small example shows you what a C++ project looks like using the merge sort that we teach in class. You will learn coding style, directory structure, compilation flow, naming convention, etc. Although you do not have to submit anything, you are strongly recommended to run this example on EDA union lab machines before you start working on your PA.

1. Readme

It is a good habit to write a README file to tell people the basic information about this software project. The README file should contain the copyright, directory structure, compilation information, revision history, etc.

```
1.  This is a mergesort example for NTUEE Algorithms students.
2.  Author: created by CM Li, NTUEE
3.
4.  =====
5.  DIRECTORY
6.  bin/    executable binary
7.  doc/    documents
8.  lib/    library files about time and memory usage
9.  src/    source C++ codes
10. README   This file
11. ======
12. GETTING STARTED
13. Please read the word file in doc/mergesort.doc.
14. To start the demo (debug version), simply follow the following steps
15.         cd lib
16.         make lib
17.         cd ../src
18.         make demo_dbg
19.         cd ../bin
20.         ./demo_dbg
21.
22. To use the optimized version,
23.         make demo_opt
24.         cd ../bin
25.         ./demo_opt
26. ======
27. CONTACT
28. If any question, please email: cmli@cc.ee.ntu.edu.tw or TA.
```

README

2. Time and Memory Usage

A *software package* is a collection of related classes. Many useful classes are provided in *library* so that programmers can reuse them easily. Library can be *dynamic* or *static*. In this example, we show how to generate a useful static library for time and memory usage. A *static library* contains an archive of many object files (*lib**.*a*) and a user interface file (*.*h*).

2.1 tm_usage.h

First, enter the *lib* directory and read the *tm_usage.h* header file, which provides an interface of the *TmUsage* library, please type:

```
cd lib
```

```
1.  // ************************************************************************
2.  // File          [ tm_usage.h ]
3.  // Author        [ littleshamoo ]
4.  // Synopsis      [ Get CPU time and Memory usage ]
5.  // Date          [ Ver 2.0 started 2010/03/23 ]
6.  // History       [ created TmStat structure to store usage ]
7.  // ************************************************************************
8.
9.  #ifndef _COMMON_TM_USAGE_H_
10. #define _COMMON_TM_USAGE_H_
11.
12. namespace CommonNs {
13.
14.         // define a data structure to store time and memory usage
15.         struct TmStat {
16.                 long vmSize; // in kilobytes
17.                 long vmPeak; // in kilobytes
18.                 long vmDiff; // in kilobytes
19.                 long rTime;    // real time in micro seconds
20.                 long uTime;    // user time in micro seconds
21.                 long sTime;    // system time in micro seconds
22.         };
23.
24.         class TmUsage {
25.                 public:
26.                         TmUsage();
27.                         ~TmUsage();
28.
29.                 bool totalStart();          // start the total timer.    this is called only once.
30.                 bool periodStart();         // start the period timer. this can be called many times
31.                 bool getTotalUsage(TmStat &st) const;      // get the total tm usage
32.                 bool getPeriodUsage(TmStat &st) const;    // get the priod tm usage
33.                 bool checkUsage(TmStat &st) const;          //
34.
35.                 private:
36.                 TmStat tStart_;
37.                 TmStat pStart_;
38.         };
39. }; // namespace
40.
41. #endif
42.
43. // ************************************************************************
44. // HOW TO USE TMUSAGE?
45. //   ….
```

tm_usage.h

Lines 1-7: title of this *.h* file. Keep information about author, synopsis, and revision date. This is a good habit to write this information for reusability and portability.

Lines 9-10: these two lines are *compiler preprocessor* that prevent a redundant inclusion of *tm_usage.h*. Note that *_TM_USAGE_H_* start and end with underscores '_'. This naming tells us that this word is not a regular variable. The reason for defining this *_TM_USAGE_H_* is to prevent the compiler from including the same header file again (in case some other programmer also includes this header file in his code.)

Line 12: Define the *name space*, CommonNs. This is to avoid other programmers use same names in other codes. This is useful when many programmers work together in a large project.

Lines 15-22: Declare a data structure to store time and memory usage. Please note that we have three different time:

real time: the real world time

user time: the time to run your code in your program

system time: that time to run system jobs (such as memory allocation, file I/O) called by your program.

Normally, we use user time + system time as the total run time of your program.

Line 24: Declare class *TmUsage*. Please note that the class name starts with a capital letter. Each subsequent word also starts with a capital letter.

Lines 25-33: Define the public functions. Please add comments so that users know how to call these functions. We add *const* to these functions so that they can not change values of any variables.

Lines 35-37: Define the private variable that are used only in the *TmUsage* class. You can add *underscores* behind the variable names to remind the programmers that these variables are private.

Lines 43-: Show how to use this TmUsage (not fully shown).

2.3 tm_usage.cpp

The implementation of *TmUsage* is in the *tm_usage.cpp.*

```
1.  //**************************************************************************
2.  // File            [ tm_usage.cpp ]
3.  // Author          [ littleshamoo ]
4.  // Synopsis        [ functions to calculate CPU time and memory usage ]
5.  // Date            [ Ver 3.0 started 2010/12/20 ]
6.  //**************************************************************************
7.  #include <sys/resource.h> // for getrusage()
8.  #include <sys/time.h>         // for gettimeofday()
9.  #include <cstdio>
10. #include <cstring>
11. #include <cstdlib>
12.
13. #include "tm_usage.h"
14.
15. using namespace std;
16. using namespace CommonNs;
17.
18. TmUsage::TmUsage() {
19.       tStart_.uTime    = 0;          tStart_.sTime    = 0;          tStart_.rTime    = 0;
20.       tStart_.vmSize = 0;         tStart_.vmPeak = 0;         tStart_.vmDiff = 0;
21.       pStart_.uTime    = 0;          pStart_.sTime    = 0;          pStart_.rTime    = 0;
22.       pStart_.vmSize = 0;         pStart_.vmPeak = 0;         pStart_.vmDiff = 0;
23. }
24.
25. TmUsage::~TmUsage() {}
26.
27.
28. bool TmUsage::totalStart() {
29.       return checkUsage(tStart_);
30. }
31.
32. bool TmUsage::periodStart() {
33.       return checkUsage(pStart_);
34. }
35.
36. bool TmUsage::getTotalUsage(TmStat &st) const {
37.       if (!checkUsage(st))
38.             return false;
39.       st.uTime    -= tStart_.uTime;
40.       st.sTime    -= tStart_.sTime;
41.       st.rTime    -= tStart_.rTime;
42.       st.vmDiff = st.vmSize - tStart_.vmSize;
43.       st.vmPeak = st.vmPeak > tStart_.vmPeak ? st.vmPeak : tStart_.vmPeak;
44.       return true;
45. }
```

```
46.
bool TmUsage::getPeriodUsage(TmStat &st) const {
47.         if (!checkUsage(st))
48.             return false;
49.         st.uTime    -= pStart_.uTime;
50.         st.sTime    -= pStart_.sTime;
51.         st.rTime    -= pStart_.rTime;
52.         st.vmDiff = st.vmSize - pStart_.vmSize;
53.         st.vmPeak = st.vmPeak > pStart_.vmPeak ? st.vmPeak : pStart_.vmPeak;
54.         return true;
55.     }
56.
57.     //…. (details skipped)
```

tm_usage.cpp

If you want to know the details, you can read this *tm_usage.cpp* file and see how we obtain the run time and memory of the program. But if you are not interested, you can only read the *.h* header file. This is why we want to separate the implementation *(.cpp)* file and the header *(.h)* file.

2.3 Compile library

To compile the *tm_usage* library, please type:

```
make lib
```

```
1.  ############################################################
2.  # File          [Makefile]
3.  # Author        [sleepyoala]
4.  # Synopsis      [an example Makefile to generate tm_usage package]
5.  # Date          [Ver. 1.0 started 2010/02/23]
6.  ############################################################
7.
8.  # CC and CFLAGS are varilables
9.  CC = g++
10. CFLAGS = -c -g
11. # -c option ask g++ to compile the source files, but do not link.
12. # -g option is for debugging
13.
14. AR = ar
15. ARFLAGS = rcv
16.
17. lib: libtm_usage.a
18.
19. libtm_usage.a: tm_usage.o
20.         $(AR) $(ARFLAGS) libtm_usage.a tm_usage.o
21.
22. tm_usage.o: tm_usage.h tm_usage.cpp
23.         $(CC) $(CFLAGS) tm_usage.cpp
24.
25. # clean all the .o and executable files
26. clean:
27.         rm -rf *.o *.a
```

makefile (for tm_usage library)

Line 22-23: compile the object file *tm_usage.o* from *tm_usage.cpp* and *tm_usage.h*

Line 19-20: archive *tm_usage.o* into a static library file *libtm_usage.a*. Please note that library must start with *lib* and ends with *.a*.

Line 17: this small library has only one object file. In a big library, more than one objective files can be archived into a single *lib*.a* file like this

```
ar  rcv  libx.a  file1.o  [file2.o  …]
```

In a big project, the libraries are usually pre-compiled by library developers so you usually do not compile the library by yourself.

3. Merge Sort

After the library is ready, please change directory to *src*.

```
cd ../src
```

3.1 mergeSort.h

Let's look at the coding style of the *mergeSort.h* header file. It provides an interface of the *MergeSort* class. The actual implementation is in another file, *mergeSort.cpp*. This separation provides the user a convenient way to quickly know how to use the class without actually bothering the implementation details.

```
1.  // ************************************************************************
2.  // File          [mergeSort.h]
3.  // Author        [Deitel How to program C++, 5ed. Ch 20, Fig. 20.05]
4.  // Synopsis      [The header file for a MergeSort Class]
5.  // Modify        [2010/02/21 CM Li]
6.  // ************************************************************************

7.  #ifndef _MERGESORT_H_
8.  #define _MERGESORT_H_

9.  #include <vector>
10. using std::vector;

11. // MergeSort class definition
12. class MergeSort
13. {
14.   public:
15.         MergeSort( int ); // constructor initializes vector
16.         void sort(); // sort vector using merge sort
17.         void displayElements() const; // display vector elements
18.   private:
19.         int size; // vector size
20.         vector< int > data; // vector of ints
21.         void sortSubVector( int, int ); // sort subvector
22.         void merge( int, int, int, int ); // merge two sorted vectors
23.         void displaySubVector( int, int ) const; // display subvector
24. }; // end class SelectionSort

25. #endif
```

mergeSort.h

Line 1-6: title of this *.h* file. Provide author and a short synopsis.

Line 7-8: these two lines are *compiler preprocessor* that prevent a redundant inclusion of *mergeSort.h*. Note that *_MERGESORT_H_* start and end with underscores '_'. This naming tells us that this word is not a regular variable.

Line 12: Define the class *MergeSort.* Please note that the class name starts with a capital letter. Each subsequent word also starts with a capital letter.

Line 14-17: public member functions. Please note that the function names start with small letters. Each subsequent word also starts with a capital letter. Do add comments to describe the function briefly so that users know how to call these functions.

Line 17: *displayElements()* is a constant so that this function cannot change the values of the

object.

Lines 18-23: define private variables and functions. Variables have no parenthesis '()' but functions do.

3.2 mergeSort.cpp

This file actually implements the *MergeSort* class. You are encouraged to read the implementation and compare it with the pseudo code of our textbook. Note that the *vector* is a *container* that is defined in *STL* (*standard template library*). Using STL helps you to quickly develop your project without worrying about the data structure implementation. Please refer to our reference reading for more information about STL.

```
1.  // ****************************************************************************
2.  // File           [mergeSort.cpp]
3.  // Author         [Deitel How to program C++, 5ed. Ch 20, Fig. 20.06]
4.  // Synopsis       [The implementation of the MergeSort Class]
5.  // Modify         [2010/02/21 CM Li]
6.  // ****************************************************************************
7.
8.  #include <iostream>
9.  using std::cout;
10. using std::endl;
11.
12. #include <vector>
13. using std::vector;
14.
15. #include <cstdlib> // prototypes for functions srand and rand
16. using std::rand;
17. using std::srand;
18.
19. #include <ctime> // prototype for function time
20. using std::time;
21.
22. #include "mergeSort.h" // class MergeSort definition
23.
24. // constructor fill vector with random integers
25. MergeSort::MergeSort( int vectorSize )
26. {
27.    size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
28.    srand( time( 0 ) ); // seed random number generator using current time
29.
30.    // fill vector with random ints in range 10-99
31.    for ( int i = 0; i < size; i++ )
32.       data.push_back( 10 + rand() % 90 );
33. } // end MergeSort constructor
34.
35. // split vector, sort subvectors and merge subvectors into sorted vector
36. void MergeSort::sort()
37. {
38.    sortSubVector( 0, size - 1 ); // recursively sort entire vector
39. } // end function sort
40.
41. // recursive function to sort subvectors
42. void MergeSort::sortSubVector( int low, int high )
43. {
44.    // test base case; size of vector equals 1
45.    if ( ( high - low ) >= 1 ) // if not base case
46.       {
47.          int middle1; // calculate middle of vector
48.          int middle2; // calculate next element over
49.          /*TODO : assign middle1 and middle2
50.
51.
52.          */
53.          // output split step
54.
55.         #ifdef _DEBUG_ON_
56.          cout << "split:     ";
57.          displaySubVector( low, high );
58.          cout << endl << "               ";
59.          displaySubVector( low, middle1 );
60.          cout << endl << "               ";
61.          displaySubVector( middle2, high );
```

```
62.             cout << endl << endl;
63.             #endif
64.
65.             /*TODO : recursive function call to split vector in half; sort each half (recursive calls)
66.                    // first half of vector
67.                      // second half of vector
68.             */
69.
70.             // merge two sorted vectors after split calls return
71.             merge( low, middle1, middle2, high );
72.          } // end if
73.       } // end function sortSubVector
74.
75.       // merge two sorted subvectors into one sorted subvector
76.       void MergeSort::merge( int left, int middle1, int middle2, int right )
77.       {
78.          int leftIndex = left; // index into left subvector
79.          int rightIndex = middle2; // index into right subvector
80.          int combinedIndex = left; // index into temporary working vector
81.          vector< int > combined; // working vector
82.
83.          // output two subvectors before merging
84.          #ifdef _DEBUG_ON_
85.          cout << "merge:      ";
86.          displaySubVector( left, middle1 );
87.          cout << endl << "                ";
88.          displaySubVector( middle2, right );
89.          cout << endl;
90.          #endif
91.
92.           /*TODO : merge vectors until reaching end of either
93.          while ( leftIndex <= middle1 && rightIndex <= right )
94.             {
95.                // place smaller of two current elements into result
96.                // and move to next space in vector
97.
98.             } // end while
99.
100.         if ( leftIndex == middle2 ) // if at end of left vector
101.            {
102.             // copy in rest of right vector
103.
104.            } // end if
105.         else // at end of right vector
106.            {
107.             // copy in rest of left vector
108.
109.            } // end else
110.          */
111.
112.          /*TODO : copy values back into original vector
113.
114.         */
115.
116.         // output merged vector
117.            #ifdef _DEBUG_ON_
118.         cout << "                ";
119.         displaySubVector( left, right );
120.         cout << endl << endl;
121.         #endif
122.      } // end function merge
123.
124.      // display elements in vector
125.      void MergeSort::displayElements() const
126.      {
127.         displaySubVector( 0, size - 1 );
128.      } // end function displayElements
129.
130.      // display certain values in vector
131.      void MergeSort::displaySubVector( int low, int high ) const
132.      {
133.         // output spaces for alignment
134.         for ( int i = 0; i < low; i++ )
135.            cout << "      ";
136.
137.         // output elements left in vector
138.         for ( int i = low; i <= high; i++ )
139.            cout << " " << data[ i ];
140.      } // end function displaySubVector
```

```
141.
142.
```

mergeSort.cpp

Lines 55-63: These lines are debug messages. These lines are compiled only if the *_DEBUG_ON_* is defined when we invoke *g++* ; otherwise, these lines are NOT compiled for speed optimization.

4. Main

4.1 global.h

This file contains the definition of global variables and constants. The constants are named in all capital letters (such as *VECTOR_SIZE*) to remind the programmer this is not a regular variable. It is a good habit to define global variables and constants in a head file, instead of the *.cpp* files, for easy maintenance.

```
1.  // ************************************************************************
2.  // File           [global.h]
3.  // Author         [CM Li]
4.  // Synopsis       [define global variables]
5.  // Modify
6.  // ************************************************************************
7.
8.  #ifndef _GLOBAL_H_
9.  #define _GLOBAL_H_
10.
11. #define VECTOR_SIZE 10    // define the vector size
12.
13. #endif
```

global.h

3.2 main.cpp

This file is the main function of the project. Because most jobs are defined in the library and the class, the main file itself is quite simple.

```
1.  //************************************************************************
2.  // File           [main.cpp]
3.  // Author         [Deitel How to program C++, 5ed. Ch 20, Fig. 20.07]
4.  // Synopsis       [The main program of this demo]
5.  // Modify         [2010/02/21 CM Li]
6.  //************************************************************************
7.
8.  #include <iostream>
9.  using std::cout;
10. using std::endl;
11.
12. #include "global.h"    // glabl variables
13. #include "mergeSort.h" // class MergeSort definition
14. #include "tm_usage.h" // the tm_usage library
15.
16. int main()
17. {
18.         CommonNs::TmUsage tmusg;
19.         CommonNs::TmStat stat;
20.         tmusg.periodStart();
21.

22.       // create object to perform merge sort
23.       MergeSort sortVector( VECTOR_SIZE );
24.
25.       cout << "Unsorted vector:" << endl;
26.       sortVector.displayElements(); // print unsorted vector
27.       cout << endl << endl;
```

```
28.
29.         sortVector.sort(); // sort vector
30.
31.         cout << "Sorted vector:" << endl;
32.         sortVector.displayElements(); // print sorted vector
33.         cout << endl;
34.
35.         tmusg.getPeriodUsage(stat);
36.         cout <<"user time:" << stat.uTime / 1000000.0 << "s" << endl; // print period user time in seconds
37.         cout <<"system time:" << stat.sTime / 1000000.0 << "s" << endl; // print period systemtime in seconds
38.         cout <<"user+system time:" << (stat.uTime + stat.sTime) / 1000000.0 << "s" << endl;
39.         return 0;
40.     } // end main
```

main.cpp

4.3 Compile the debug version

Please type the following commands,

```
make demo_dbg
cd ../bin
./demo_dbg
```

Please go to the *src* directory and read the sample makefile.

```
1.      # CC and CFLAGS are varilables
2.      CC=g++
3.      CFLAGS =    -c
4.      # -c option ask g++ to compile the source files, but do not link.
5.      # -g option is for debugging version
6.      # -O2 option is for optimized version
7.      DBGFLAGS = -g -D_DEBUG_ON_
8.      OPTFLAGS = -O2
9.
10.
11.     # DEBUG Version
12.     demo_dbg      : mergeSort_dbg main_dbg
13.                             $(CC) $(DBGFLAGS) mergeSort.o main.o -ltm_usage -L../lib -o ../bin/demo_dbg
14.     main_dbg                :   ../lib/tm_usage.h    global.h main.cpp
15.                             $(CC) $(CFLAGS)    main.cpp      -I../lib
16.     mergeSort_dbg           : mergeSort.cpp mergeSort.h
17.                             $(CC) $(CFLAGS) $(DBGFLAGS) mergeSort.h mergeSort.cpp
18.
19.     # optimized version
20.     demo_opt      : mergeSort_opt main_opt
21.                             $(CC) $(OPTFLAGS) mergeSort.o main.o -ltm_usage -L../lib -o ../bin/demo_opt
22.     main_opt                :   ../lib/tm_usage.h    global.h main.cpp
23.                             $(CC) $(CFLAGS)    main.cpp      -I../lib
24.     mergeSort_opt: mergeSort.cpp mergeSort.h
25.                             $(CC) $(CFLAGS) $(OPTFLAGS) mergeSort.h mergeSort.cpp
26.
27.     # clean all the .o and executable files
28.     clean:
29.                     rm -rf *.o demo
```

makefile

Lines 11-17: Compile the debug version when we type '`make demo_dbg`'. This version invokes options '-g' (for DDD debugger) and also '-D_DEBUG_ON_' to enable the printing of arrays in *mergeSort.cpp*.

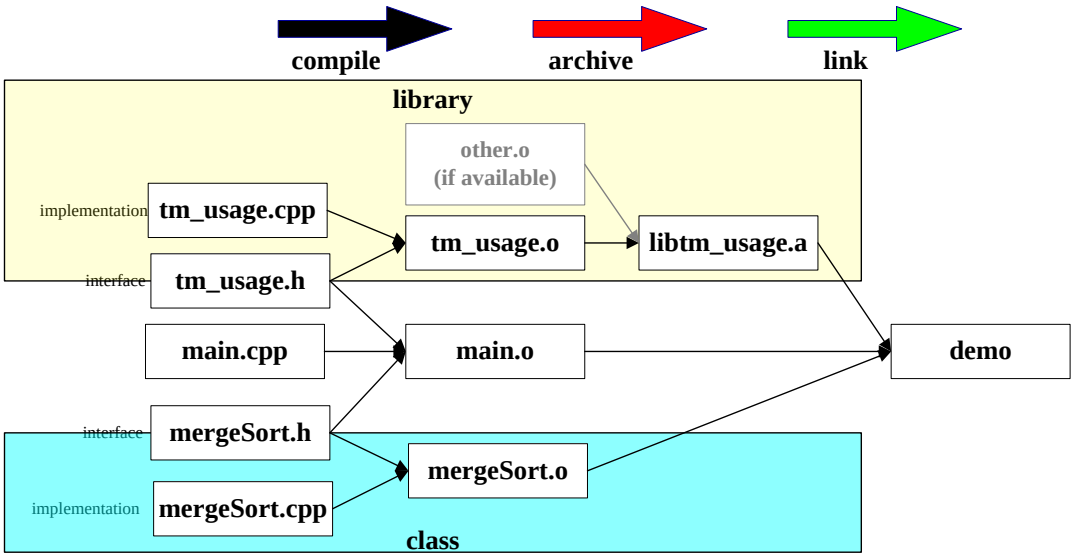
Lines 19-25: Compile the optimization version when we type '`make demo_opt`'. This version invokes options '-O2' for speed improvement. Also '_DEBUG_ON_' is not defined to disable the printing of arrays in *mergeSort.cpp*.

4.4 Compile the optimized version

Please type the following commands,

```
cd ../src
make demo_opt
cd ../bin
./demo_opt
```

We can summarize the compilation of the whole project in this figure.



compilation flow of this project

5. Input File Format

Go to the input directory and see the input file by typing the following.

```
cd ../inputs
vim 5.case1.in
```

The first two lines are comments. The first line shows the number of data points and the second line shows the format. Lines 3 to 7 lists five data points, with a index and the number to be sorted. The file `5.case1.in` contains five numbers

```
# 5 data points
# index number
0 16
1 13
2 0
3 6
4 7
```

The output file(`*.out`) is actually the same as the input file except that the numbers are sorted in *increasing* order. For example, `5.case1.out` is like:

```
# 5 data points
# index number
0 0
```

```
1 6
2 7
3 13
4 16
```

6. Exercise (NO need to submit)

Please finish all TODOs in *.cpp* to run your merge sort correctly. In the command line, you are required to follow this format:

```
./demo_opt  <input_file_name> <output_file_name>
```

Instead of generating random numbers, you are required to read in the numbers from an external file, and then output your results to another external file. For example,

```
./demo_opt  ../inputs/5.case1.in  ../outputs/5.case1.out
```

6. References

about STL:

H. Deiltel, *How to Program C++*, 5ed. Prentice Hall.

CPLUSPLUS http://www.cplusplus.com/reference/stl/

C++ STL Tutorial http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html

SGI STL Programmer's Guide http://www.sgi.com/tech/stl/

About Coding Style

http://geosoft.no/development/cppstyle.html

H. Sutter, *C++ coding standards, 101 rules, guidelines and best practice*, Addison Wesley.