DUE DATE: **Mon, Oct 6 2014 BY 5:00 PM (EST)**

Heterogeneous and Cloud Computing

Homework #4

Max Points: 30

Note: If you are using your personal machine then it is best to install a Linux virtual machine on your computer and use the Linux virtual machine.

You should save and rename this document using the naming convent `MUid_Homework4.docx` (example: `raodm_Homework4.docx`).

Objective: The objective of this exercise is to gain some familiarity with:

- Pointer operations.
- Observing information via the GNU debugger `gdb`.
- Troubleshooting memory errors using `valgrind`.
- Developing an OpenCL kernel

Submission: Once you have completed this exercise, upload:

1. This MS-Word document (duly filled with the necessary information) named and sa as a PDF file using the convention *`MUid_Homework4`*`.pdf` (exam `raodm_Homework4.pdf`)
2. The C++ program completed as part of this exercise named with the convent *`MUid`*`_Homework4.cpp`.
3. The OpenCL C++ program developed as part of this homework named with convention `MUid_Homework4_PartB.cpp`.

Fill in answers to all of the questions as directed. For some of the questions that require outp to be indicated, you can simply copy-paste appropriate text from the shell/PuTTY window i this document. You are expected refer to `LinuxEnvironment.pdf` document available Handouts folder off Niihka. You may discuss the questions with your instructor.

Name: **Weiguo Xia**

Grading Rubric:

This is a graduate course and consequently the expectations in this course are higher. Accordingly, the C++ program submitted for this homework must be operational in order to qualify for earning a full score.

NOTE: Program that do not compile, have methods longer than 25 lines, or just some skeleton code will be assigned zero score.

-5 points: The methods are not implemented concisely and efficiently by using good programming practices and suitable algorithms. Error conditions don't need to be handled and you can just throw an exception.

-1 point: For each style violations as reported by the C++ style checker (a slightly relaxed version from Google Inc. automatically downloaded by `Makefile` from Niihka)

-3 points: If the program does not include suitable comments at appropriate points in each method to elucidate flow of thought/logic in each method.

-1 point: For each warning message generated by g++ when compiling with `–Wall -Wextra` (report all warnings) flag.

-1 point: For each difference in outputs between your program and expected output.

Part A: Gaining Experience with pointers and tools [10 points]

Task #A.1: Copy and review supplied code

Expected time for completion 5 minutes

For this exercise you are supplied with a starter code with three methods (that you are expected to implement) and a `main` method that runs the tests:

- `print`: Method to print all entries in a given list of strings. See sample output for format.
- `genNum`: Method to convert numbers to strings and store it in a given list.
- `freeList`: Method to free entries (as needed) in the given list.

Task #A.2: Implement the `print` method

Expected time for completion 15 minutes

Background

Pointers play an important role in C/C++ program. They map directly to various memory accessing modes that are supported by modern microprocessors. Understanding pointers is crucial for developing sophisticated interfaces (possibly across multiple languages) between software and also with hardware.

Exercise

In this task you are expected to implement the print method to print all the entries in the list. A sample output is shown below:

```
$ ./raodm_hw4
Creating and printing fixed list of entries:
list[0, 0x7ffff769f8a0]: one
list[1, 0x7ffff769f8a8]: two
list[2, 0]: null
list[3, 0x7ffff769f8b0]: three
list[4, 0]: null
```

Task #A.3: Using the GNU debugger
Expected time for completion 15 minutes

Background on gdb
The GNU debugger is a standard tool on Linux distributions that provides a convenient approach to troubleshooting issues in programs. It also provides a convenient interface for analyzing core files generated when programs crash.

Exercise
In this part of the exercise you are expected to use the GNU debugger to obtain runtime information and fill-in the table below in the following manner:

1. Compile the program with debugging information using the `-g` flag (if you have not already compiled with the `-g` flag).
2. Start the GNU debugger from `emacs` using the main menu options `Tools`→`Debugger`.
    a. In the mini buffer at the bottom ensure that the actual executable that you would like to debug is correctly selected.
3. Set a breakpoint in the first line of the `print` method.
4. Run the program and when the breakpoint is hit, use suitable GBD commands to populate the table below. You can check your values with your neighbor to see if the values are close/consistent or they are significantly different than yours.
5. Once you have recorded the necessary observation you can quit out of GDB.

Task	**Data displayed by gdb**
Print value of variable `list` [gdb-command: `print list`]	(const StrList * const) 0x7fffffffe340
Print the size of `list` [gdb-command: `print list->size()`]	5
Print the object pointed to by `list`	{<std::_Vector_base<std::basic_string<char, std::char_traits<char>, std::allocator<char> >*, std::allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> >*> >> = {_M_impl = {<std::allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> >*>> = {<__gnu_cxx::new_allocator<std::basic_string<char, std::char_traits<char>, std::allocator<char> >*>> = {<No data fields>}, <No data fields>}, _M_start = 0x6060a0, _M_finish = 0x6060c8, _M_end_of_storage = 0x6060c8}}, <No data fields>}

What is the output of the gdb command `print *((std::string*) xxx)`, where `xxx` is the number/address/first-entry in `list`. Displayed by the previous command.	one
Use `gdb` to print value of the mathematical expression 0x16 + 0123 + 5 [gdb-command: `print 0x16 + 0123 + 5`]	110
Print the current stack trace [gdb-command: `where`]	#0 print (list=0x7fffffffe340) at xiaw_Homework4.cpp:14 #1 0x000000000040167c in main (argc=1, argv=0x7fffffffe4a8) at xiaw_Homework4.cpp:43
Print the location about the `main` method [gdb-command `print main`]	{int (int, char **)} 0x401516 <main(int, char**)>
Similarly print location of the `print` method	{void (const StrList * const)} 0x4012fd <print(std::vector<std::string*, std::allocator<std::string*> > const*)>
Print the information about the CPU's instruction pointer register [gdb-command: `print info rip`]. Do you observe correlation between its value and the information on the `print` method?	0x7fffffffe340

Task #A.4: Troubleshooting programs under `valgrind`

Expected time for completion 15 minutes

Background on `valgrind`

Valgrind (pronounced val-grinned) is a collection of tools built using a common runtime instrumentation framework. Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools. Any x86 program can be analyzed using valgrind. It is a handy tool for detecting and fixing complex memory issues in programs, particularly those that involve dynamic memory.

Exercise

In this part of the exercise you are expected to troubleshoot the operation of the supplied program to obtain the necessary outputs shown below, in the following manner:

1. Uncomment the line of code `#define EX_PART2`. Compile and run the program as shown below. Observe that the output is different than the expected output shown further below in this document.

```
$ ./raodm_Homework4 5
```

2. Now, run the program under valgrind (in `emacs` you can use the main menu option `Tools`Shell command…) as shown below:
3. Study the first `valgrind` error. The `valgrind` errors are in the form of a stack trace giving you information about
    a. The location where the error occurred
    b. The location where the memory was created/deallocated.
4. Using `valgrind` error messages and information about the behavior of the supplied program fix the issue with the `genNum` method.
5. Once you have generated the desired output, ensure the memory is correctly deleted by suitably implementing the `freeList` method.
6. Run the final program under `valgrind` and verify that there are no errors or no leaked memory (outputs highlighted below).

Sample output

```
$ valgrind ./raodm_Homework4 5
==22761== Memcheck, a memory error detector
==22761== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==22761== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==22761== Command: ./Homework4 5
==22761==
Creating and printing fixed list of entries:
list[0, 0x7ff0000d0]: one
list[1, 0x7ff0000d8]: two
list[2, 0]: null
list[3, 0x7ff0000e0]: three
list[4, 0]: null
Generating and printing list of entries:
list[0, 0x4c3b2f0]: 0
list[1, 0x4c3b3f0]: 1
list[2, 0x4c3b4f0]: 2
list[3, 0x4c3b600]: 3
list[4, 0x4c3b6b0]: 4
==22761==
==22761== HEAP SUMMARY:
==22761==     in use at exit: 0 bytes in 0 blocks
==22761==   total heap usage: 21 allocs, 21 frees, 500 bytes allocated
==22761==
==22761== All heap blocks were freed -- no leaks are possible
==22761==
==22761== For counts of detected and suppressed errors, rerun with: -v
==22761== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Task #A.5: Run a different executable under valgrind

Recollect that `valgrind` can be used to validate and verify any program. Try `valgrind` on the following one-liner python program (copy-pasted from this document into a file named `hello.py`):

```
print “hello world”
```

Once you have saved the above one-liner into a file named `hello.py`, run the python VM under valgrind using the following command:

```
$ valgrind python hello.py
```

Contrast the output with output from your C++ program. Are there any possibly lost memory? What are the implications?

Part B: OpenCL program for vector operations [20 points]

Program requirement

Convert the given implementation of four vector operations (add, subtract, multiply, and divide) to functionally similar OpenCL implementation and assess effectiveness of the OpenCL implementation.

Supplied Starter Code

The following files are supplied for reference implementation to be used for testing the OpenCL implementations:

- `vector_operations.h` [Do not modify]: Header file declaring the operators to be implemented as OpenCL kernels.
- `vector_tester.cpp` [Do not modify]: Tester code to create two vectors and call the corresponding operators with it.
- `vector_operations.cpp` [Modify]: Reference implementation for verifying OpenCL implementations.

Sample output

Expected sample outputs from the OpenCL implementation are shown below:

```
$ ./vector_opencl 20 +
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

```
$ ./vector_cpu 10 -
0 0 0 0 0 0 0 0 0 0
```

```
$ ./vector_cpu 20 \*
1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

```
$ ./vector_cpu 20 /
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Part C: Submit files to Niihka

Once you have completed this exercise, upload:

1. This MS-Word document (duly filled with the necessary information) named with the convention *`MUid_Homework4`*`.pdf`. Save the document as a PDF file to upload it to Niihka.
2. The C++ program developed as part of this exercise named with the convention *`MUid`*`_Homework4.cpp`.
3. The OpenCL C++ program developed as part of this homework named with the convention `MUid_Homework4_PartB.cpp`.