Systems Engineering and Data Processing in C++

Summer Term 2021

Hasso Plattner Institute
Computer Graphics Systems Group

Prof. Dr. Jürgen Döllner
and Willy Scheibel

Assignments #2: STL & Algorithms, Signals, and Memory

Objectives

In these assignments you will learn how to:

- use the standard library to formulate efficient algorithms by means of code clarity (and size);
- manage signals from the executing environment, exceptions, and error codes;
- implement customized memory allocation strategies; and
- apply smart pointers for managing dynamically allocated objects.

Rewards

For each successfully solved task you and your partner (optional) gain one bonus exam point. A serious shortcoming in fulfilling the requirements of a task will result in no bonus point regarding this task. You can gain at most 4 bonus points through exercise sheet.

Submission

All described artifacts has to be submitted to the course moodle system[1] as a zipped archive. The naming convention includes the assignment number and your *personal assessment numbers* with following naming convention: `assignment_1_paNr1.zip` or `assignment_1_paNr1_paNr2.zip` whether or not pair programming was applied. Compiled, intermediate, or temporary files should not be included. If pair programming is used, the results are turned in only once. The assignments #2 are due to the next tutorial on Mai 20th, 9:15 a.m. GMT+2.

Presentation

Feel free to present any of your solutions during the Zoom meeting in the upcoming tutorial. If you want to present your solution of a task, please send the source code beforehand via e-mail to willy.scheibel@hpi.de.

General Instructions

Pair Programming On these assignments, you are encouraged (not required) to work with a partner, provided you practice pair programming. Pair programming "is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test." One partner is driving (designing and typing the code) while the other is navigating (reviewing the work, identifying bugs, and asking questions). The two partners switch roles every 30-40 minutes, and on demand, brainstorm.

Cross-Platform The assignments can be solved on all major platforms (i.e., Windows, Linux, macOS). The evaluation of submitted assignments can be carried out on any of these platforms. All results should not use platform-specific dependencies. Platform-specific build and macro code is permitted.

Violation of Rules a violation of rules results in grading the affected assignments with 0 points.

- Writing code with a partner without following the pair programming instructions listed above represents a serious violation of the course collaboration policy.
- Plagiarism represents a serious violation of the course policy.

[1] https://moodle.hpi.de/course/view.php?id=174

Task 2.1: RegEx Counter (Code Golf)

File: `regex-filter/regex-filter.cpp`, `regex-filter/emails.txt`.

Description: Extend the framework program (`regex-filter.cpp`) that already reads a file into memory by a command line parameter that gets interpreted as a regular expression (regex). This regular expression should get used to find matches in the lines of the file. Lines that do not match the regex are not considered further. The matching part of each line should be extracted and counted. The output of the program should list all matched parts and their number of occurrences. Further, add a command line option to switch to case insensitive regex matching. When using case insensitive matching, the matching part should be printed as its lowercase version.

More specifically, ensure that your program...

- takes one command-line argument as regex,
- provides a command-line switch to enable case-insensitive regex matching,
- discards all unmatched lines from the file input,
- collects and counts the match within each line,
- outputs all matches with their number of occurrences.

Example:

Example contents of the file.

```
norman.kluge@hpi.de
willy.scheibel@hpi.de
lukas.wagner@hpi.de
jan.vollmer@hpi.de
```

Example behavior of your program.

```
# ./regex-filter "lu.*@(hpi|student.*).de"
hpi: 2
student.hpi.uni-potsdam: 3
```

Competition: This is a competitive task with respect to source code length. If you want to compete in the open competition submit your isolated source code of this task via e-mail to willy.scheibel@hpi.de until Mai 18th. Your source code is checked for conformity with respect to the expected output. If the source code behaves as expected, we measure the code length using `wc -c` and subtract the number of new lines with `wc -l`, i.e., counting the number of characters of the whole `cpp` file.

This challenge belongs to the sports of *code golf*[2]. You are allowed to change the whole source code of the file, including already present code, as long as the output is in accordance to the expected behavior. You may use C constructs and other familiar and unfamiliar constructs of C++ to reduce your code in size. The compilation command must not include compile-time definitions that shortens your source code.

Artifacts:

a) `regex-filter.cpp` : Source code of the solution.

Objectives: Regex, standard library.

Task 2.2: Program Failure Immunity

File: `runner/runner.cpp`, `runner/externallibrary.h`, `runner/externallibrary*{dll,dylib,so}`.

Description: The framework program (`runner.cpp`) performs computations. Much is not known of these computations as they are provided by an external library whose source code is owned

[2] "Code golf is a type of recreational computer programming competition in which participants strive to achieve the shortest possible source code that implements a certain algorithm." (https://en.wikipedia.org/wiki/Code_golf)

by another development team. It cannot be modified and the developers haven't bothered to document potential erroneous behavior.

Modify the framework program to establish a kind of "immunity" to prevent all forms of unexpected terminations:

- The program may terminate if it receives `SIGKILL` or `SIGSTOP` signals.
- The `SIGINT` signal should terminate the program at the second occurrence using `exit(int)`.
- Other signals defined by C and C++ should get ignored if permitted by the standard. Otherwise, the program should be terminated using `quick_exit(int)`.
- The program should not terminate in the case of other signals or thrown exceptions (those declared by C and C++ are sufficing).
- The program may abort its current computation but must continue with the next processing.
- The program should terminate gracefully without any error (the message "So long, and thanks for all the fish." is shown).
- Add signal handler(s) and corresponding behavior.
- Add exception handler(s).
- Test your solution with a first command line argument greater or equal to 1 000 000 (be careful: your colleague already tested this and reported massive system resource consumption).
- Collect all types of issued signals and exceptions that occurs during the execution in `errors.txt`.

Artifacts:

a) `runner.cpp` : Source code of the solution.

b) `errors.txt` : List of occurring signals and exceptions.

Objectives: Signal handling, exception handling, error codes.

Task 2.3: Customized Memory Management for a Particle System

Files:, `particles/particle.h`, `particles/particle.cpp`, `particles/engine.h`, `particles/engine.cpp`, `particles/main.cpp`, `particles/particles.pro`, `particles/widget.cpp`, `particles/widget.h`.

Description: Given a framework that implements a particle system. In each time step it creates new particles, updates existing ones, and destroys those that reached their end of life. Due to its dynamic behavior, we do not know how many particles are created or destroyed in each time step. However, the particle system itself is not the main subject of this assignment—it is a representative of systems that dynamically allocate and deallocate a large number of objects at run-time.

The framework uses the standard memory allocation by the global `new` and `delete` operators. You should implement a new customized memory management strategy (all instances of `Particle` have the same size); one reason could be that this system is going to be part of an embedded system.

- Implement the overloaded `new` and `delete` operators for the `Particle` class.
- Manage a *static memory block* that is used for storing active particles (1.5MiB).
- Keep track of assigned and freed memory of the memory block (debug information).
- Signal insufficient memory by returning a `nullptr`; the particle system handles this case properly.

You *must not* change the dynamic behavior or other parts of the source code.

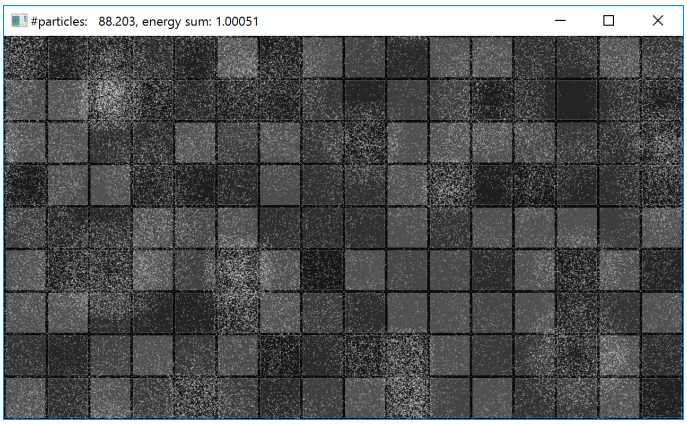
Hints: The framework depends on the Open Source toolkit Qt with its fifth major release (e.g., Qt 5.12). The project file `particles.pro` can be used to generate a platform-specific Makefile with the following command:

```
# qmake particles.pro
```

Artifacts:

a) `particle.h`, `particle.cpp` : Source code of the solution.

Example:



Objectives: Customized new and delete operators, class-specific memory management strategy.

Task 2.4: Re-engineering Graph Functions by Smart Pointers

Files: `graph/Graph.h`, `graph/Graph.cpp`, `graph/GraphFactory.h`, `graph/GraphFactory.cpp`, `graph/InstanceCounter.h`, `graph/InstanceCounter.cpp`, `graph/graph-test.cpp`.

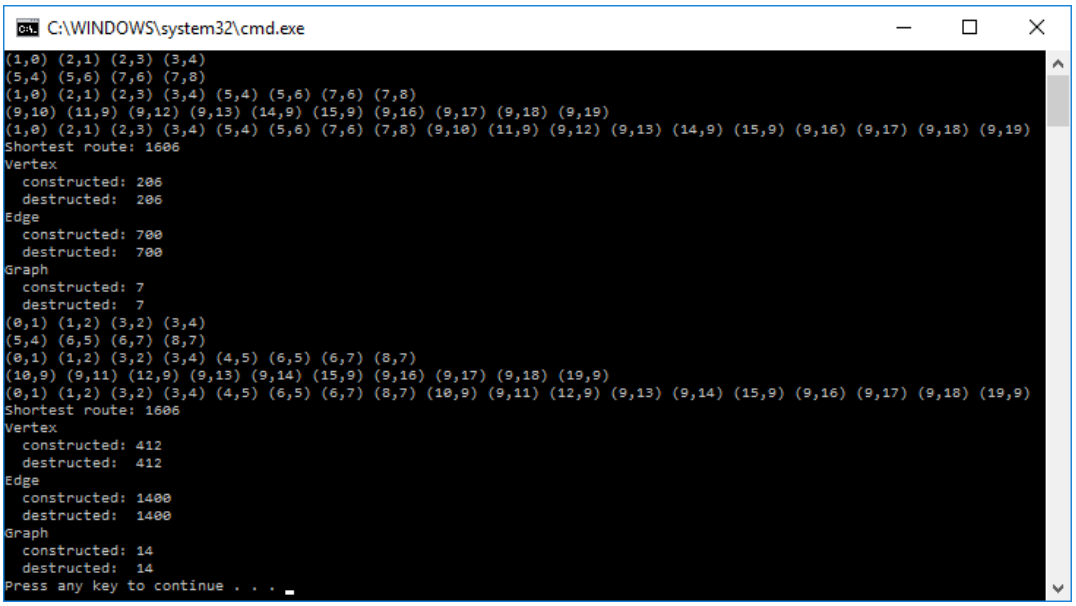
Description: The framework contains an implementation for graphs. A *graph* consists of a set of *vertices* and a set of *edges*. Each edge denotes an undirected link between exactly two vertices. A vertex is identified by its unique *id*. A factory class is provided that generates linear graphs, tree-like graphs, circular graphs, and random graphs. A key operation provided by the graph class is the *merge* operation: If two graphs are merged, their sets of vertices are merged (removing duplicates) as well as their sets of edges (duplicates are eliminated, too).

Unfortunately, the current implementation uses raw pointers for graphs and their components, which are all dynamically allocated. Memory leaks are hard to control if this library is used. For that reason, you should re-engineer the library based on smart pointers. With the smart pointers, a clear memory management should be grounded. The dynamic creation of graphs, vertices, and edges cannot be changed due to application requirements (assume that vertices and edges may have additional data and may be specialized by subclasses).

- Re-engineer the `Vertex`, `Edge`, `Graph`, and `GraphFactory` classes to use smart pointers.
- Re-engineer the built-in test case to use smart pointers.
- Think of an object-ownership model for the system to select a matching smart pointer in the appropriate places.
- Make sure your solution does not contain any `new` or `delete` expressions.
- Verify correct memory management using the console output of built-in instance counters.

Artifacts: `Graph.h`, `Graph.cpp`, `GraphFactory.h`, `GraphFactory.cpp`, `graph-test.cpp` : Source code of the solution.

Example:



Objectives: C++ smart pointers, object-ownership.