Advanced Programming in C++

Winter Term 2016/17 – Assignments # 3

Hasso Plattner Institute
Computer Graphics Systems Group

Prof. Dr. Jürgen Döllner

Objectives

In the assignments you will learn how to

- implement customized memory allocation strategies;
- manage signals from the executing environment, exceptions, and error codes;
- apply smart pointers for managing dynamically allocated objects.

Task 3.1: Customized Memory Management for a Particle System

Files:, `main.cpp`, `engine.h`, `engine.cpp`, `particle.h`, `particle.cpp`, `widget.h`, and `widget.cpp`.

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. The particle system 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.

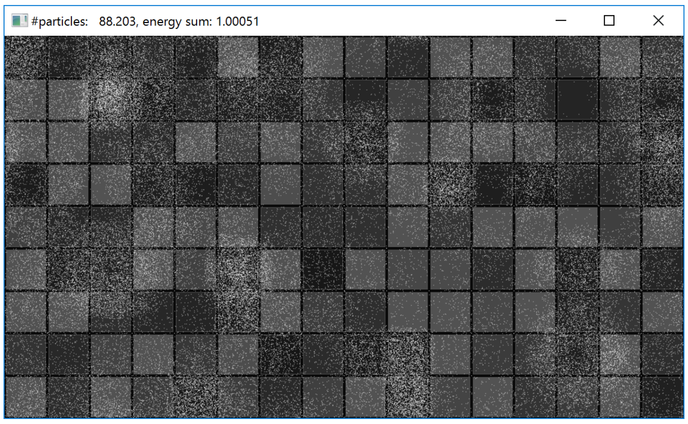
- Implement the overloaded `new` and `delete` operators for the `Particle` class.
- Manage a *static memory block* that is used for storing active particles (1.5MiB).
- Keep track of assigned and freed memory of the memory block (debug information).
- Signal unsufficing memory by returning a `nullptr`;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

You *must not* change the dynamic behavior or other parts of the source code—consider these parts as owned by another developer group.

Artifacts:

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

Example:



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

Task 3.2: Program Failure Immunity

File: `runner.cpp`, `externallibrary.h`, `externallibrary`

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 by another development team. It cannot be modified.

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: 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 3.3: Re-engineering Graph Functions by Smart Pointers

Files: `Graph.h`, `Graph.cpp`, `GraphFactory.h`, `GraphFactory.cpp`, `InstanceCounter.h`, `InstanceCounter.cpp`, `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.

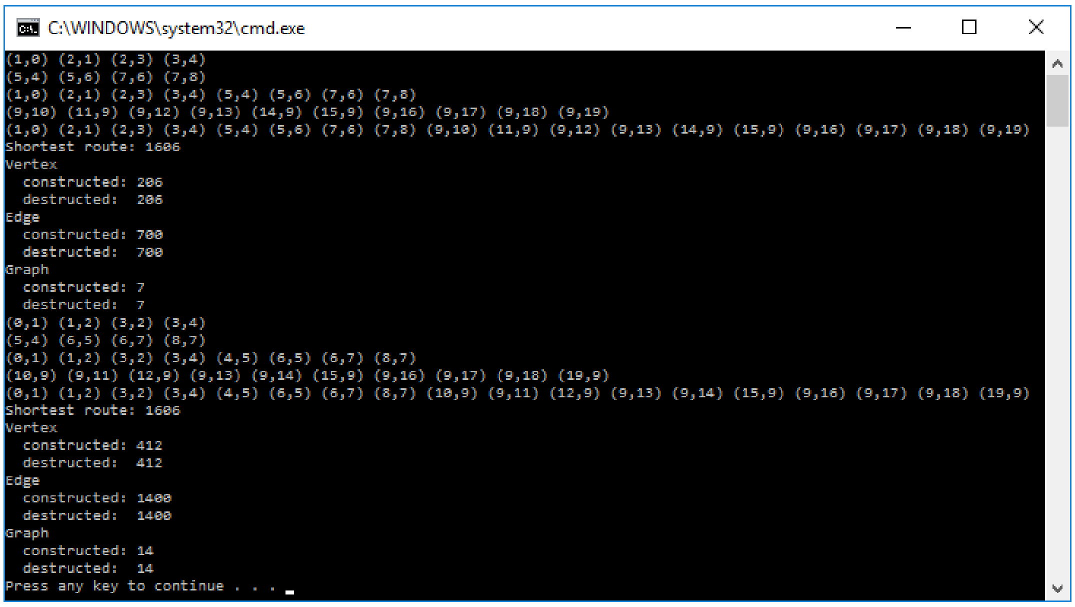
- Make sure your solution does not contain any `new` or `delete` expressions.
- Verify correct memory management using the console output of built-in instance counters.

Extended Task: Minimum Spanning Trees [1 additional exam bonus point]

Extend the graph by a function that computes the *minimum spanning tree* (MST) for a given input graph (e.g., Prim's algorithm). It returns a new graph containing the MST as a kind of "graph view", i.e., it re-uses the vertices and edges of the input graph.
The test data in the framework represents a matrix denoting the distances between all major German cities (a city corresponds to a vertex), and a distance greater zero denotes a path between two cities. Use this matrix to construct the input graph, and then compute the MST by implementing the respective functions of the `GraphFactory` and `Graph` classes. How long is the shortest route for a traveling salesman?

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

Example:



```
(1,0) (2,1) (2,3) (3,4)
(5,4) (5,6) (7,6) (7,8)
(1,0) (2,1) (2,3) (3,4) (5,4) (5,6) (7,6) (7,8)
(9,10) (11,9) (9,12) (9,13) (14,9) (15,9) (9,16) (9,17) (9,18) (9,19)
(1,0) (2,1) (2,3) (3,4) (5,4) (5,6) (7,6) (7,8) (9,10) (11,9) (9,12) (9,13) (14,9) (15,9) (9,16) (9,17) (9,18) (9,19)
Shortest route: 1606
Vertex
  constructed: 206
  destructed:  206
Edge
  constructed: 700
  destructed:  700
Graph
  constructed: 7
  destructed:  7
(0,1) (1,2) (3,2) (3,4)
(5,4) (6,5) (6,7) (8,7)
(0,1) (1,2) (3,2) (3,4) (4,5) (6,5) (6,7) (8,7)
(10,9) (9,11) (12,9) (9,13) (9,14) (15,9) (9,16) (9,17) (9,18) (19,9)
(0,1) (1,2) (3,2) (3,4) (4,5) (6,5) (6,7) (8,7) (10,9) (9,11) (12,9) (9,13) (9,14) (15,9) (9,16) (9,17) (9,18) (19,9)
Shortest route: 1606
Vertex
  constructed: 412
  destructed:  412
Edge
  constructed: 1400
  destructed:  1400
Graph
  constructed: 14
  destructed:  14
Press any key to continue . . .
```

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

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 are required to be cross-platform, that is, they compile successfully and run indifferently with respect to their input and output.

Upload Results All described artifacts are submitted to the course moodle system `https://moodle.hpi3d.de/mod/assignment/view.php?id=2741` as a zipped archive with the following naming convention: `assignment_3_matrikNr1.zip` or `assignment_3_matrikNr1_matrikNr2.zip` whether or not pair programming was applied. Compiled, intermediate, or temporary files should not be included (`*.obj, *.pdb, *.ilk, *.ncb` etc.). If pair programming is used, the results are turned in only once. The assignments #3 are due to the next tutorial on November 30th, 9:15 a.m.

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 (e.g., if one partner does not participate in the process) is a serious violation of the course collaboration policy.
- Plagiarism represents a serious violation of the course policy.