1. Ei mitään constructorijuttuja
2. Ei mitään constructorijuttuja
   1. Copy inner
   2. Copy outer
   3. Destr. inner
   4. Destr. Outer
3. asd
   1. Assigment inner
   2. Destr inner
   3. Copy inner
   4. Copy outer
   5. Destr outer
   6. Destr inner

# Comparing performance of inserting different kinds of elements at the end of different C++’s STL data containers

In this exercise I compared the performance of four different kind of elements inserted in different STL data containers. The elements were

1. int
2. a simple struct containing only one literal property ( unsigned int ) and with defined constructor, copy constructor, destructor and assignment operator
3. a struct containing one literal property ( unsigned int ) and one object ( another struct that has only one literal property, unsigned int ) and with defined constructor, copy constructor, destructor and assignment operator. The struct member isn’t initialized with the keyword new and is destroyed with the scope
4. a struct containing one literal property ( unsigned int ) and one object t( another struct that has only one literal property, unsigned int ) and with defined constructor, copy constructor, destructor and assignment operator. The struct member is initialized with the keyword new, hence needing manual deleting too.

These different scenarios with different elements will be referred as A, B, C or D respectively. The used data structures were (from STL) vector, list, forward-list, deque, stack and queue. The tests were copied from the course’s teacher’s tests by only changing the elements in any data container.

Different methods of inserting elements weren’t considered in these tests. However, insertion methods were consistent between different tests, so they are at least somewhat comparable.

## Results

From table 1 we can compare the averages of different scenarios. Scenario A with the ‘simplest’ element ( int ) has been chosen to be the baseline against which other scenarios are compared to. We notice increased expenses in every scenario compared to baseline A. This is to be expected, as our objects will always make more calls to the memory manager and have more ‘lifecycle calls’. We also learn that in scenario C, where another struct is being initialized with the keyword ‘new’ inside the element, is easily the most expensive element type. Dynamic memory allocation (C) seems to be noticeably more expensive than static (D). Comparing B and D we notice a slight increase in cost in D. D in comparison has the extra statically allocated object member in comparison, which does still require initialization, so higher expenses make sense.

In table 2 we compare each scenarios performance averages against each scenario’s vector’s performance respectively. We notice that while the overall costs (table 1) were in descending order from C, D, B to A, here we can notice that in comparison the cost of using different data containers from vectors gets more expensive in reverse order. **We can deduce that the increased cost of using a more complex element makes the differences between the costs of different data containers less significant. This same comparison could probably be generalized to other operations in addition to inserting.**

The measurements (that can be found from github) show that vector has a more expensive call always in the first time it’s run. This is due to its memory being dynamically allocated several times on the first run, but having enough capacity to avoid further allocations in the following runs. As such the results here can be seen basically as if the vector’s memory is being pre-allocated (as 1 non-pre-allocated run against 19 ‘pre-allocated’ runs doesn’t weight much). With this amount of elements (1 000 000) pre-allocations seems to double the speed of vector operations in all but A scenario. Curiously pre-allocating doesn’t seem to affect vector’s performance when the element is int. When we clear the capacity (vector<T>(v).swap(v)) each run, each consequent run is as fast/slow as the first run in non-clearing runs.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Avg % diff vs A | |  |  |  |  |
|  | Vector | List | FL | Deque | Stack | Queue |
| A | 1 | 1 | 1 | 1 | 1 | 1 |
| B | 1.315 | 1.076 | 1.104 | 1.089 | 1.080 | 1.071 |
| C | 3.376 | 1.906 | 2.116 | 2.049 | 2.082 | 2.038 |
| D | 1.719 | 1.242 | 1.327 | 1.378 | 1.295 | 1.348 |

Table 1. Percentual differences between average performances of different elements in different containers with A being the baseline. A: int. B: Struct with no inner struct. C: Struct with another struct initialized with ‘new’. D: Struct with another struct not initialized with ‘new’.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Averages against vector of each test | | | |  |  |
|  | Vector | List | FL | Deque | Stack | Queue |
| A | 1.000 | 2.428 | 1.878 | 2.315 | 2.208 | 2.422 |
| B | 1.000 | 1.986 | 1.577 | 1.916 | 1.814 | 1.972 |
| C | 1.000 | 1.370 | 1.177 | 1.405 | 1.361 | 1.462 |
| D | 1.000 | 1.754 | 1.450 | 1.856 | 1.664 | 1.900 |

Table 2. Comparing averages of each scenario against the performance of vector structure of with each respective element.

Github: