**Rust Programming Lab #5 6th September 2022**

**Arrays and Vectors**

**Exercise 1**

Arrays and vectors have similar purposes – the management of structures representing objects in the real or virtual world that your program is manipulating. Already you have seen points and triangles. For this lab, you should create a different geometric structure: you may choose any simple structure, e.g. circles, rhomboids, rectangles, polygons or …. The brave among you can try to model some 3D structures, *e.g.* a prism, but this is not required.

Then you should create (a) an array of your structure and (b) a vector of the same structure. Then you should compare the time your program takes to (i) add ***N*** structures to your array or your vector and (ii) access the array or vector to extract an element. As you have been shown in lectures, an element of an array can be accessed simply using an index. However, elements can be added to vectors by **push** and extracted by both an index and the **pop** function. So for vectors, you should measure three times: add (by **push**), access by direct access **[]** and **pop**.

Example code to report the time taken for specific operations is shown in the **vectime.rs** module. Note that the example has a parameter ***N*** which allows you to time a large number of operations: Rust’s timer provides nanosecond resolution, but the time it returns will vary considerably from operation to operation, so you must time a large number of operations and average to estimate the time per operation. These variations are caused, in part, by interruptions from the operating system. In a modern multi-tasking OS, it is doing many things in the background, *e.g.* refreshing those fancy icons on your screen 😉, checking the network, *etc.*, so it ‘steals’ some cycles and generally interferes with accurate timing! In your report, show a sample of the times that your program reported – 3 measurements will give you an estimate of the variation that you can expect for ***any*** time measured. You *should wrap your timing code in a loop* to make the measurements.

For dynamic vectors, adding elements with **push** will cause the vector code to add more space when the current space is exhausted. After you have timed the additions, add code to the loop that adds elements to the vector with **push** and find out what your implementation does to add more space. Use the **capacity()** function to find out the current capacity. Find (a) what your implementation does, *i.e.* how much extra space it allocates, when the current space is used up and (b) the effect on the time to add new elements with **push()** on the initially allocated space set with **with\_capacity().** You should compare the effect of an initially very small allocation and an allocation that was at least as large as the expected capacity, *i.e.* when there will be no re-allocations after the initial allocation. You can use the example in the lecture slides to guide this experiment.

A simple way to measure the time uses this function

**fn time\_diff\_nsecs( t0:Instant ) -> f64 {**

**let duration = t0.elapsed();**

**let d\_nsecs = duration.as\_nanos();**

**d\_nsecs as f64**

**}**

which calculates time (in nanoseconds) from some ‘instant’, ***t*0**. Call the **Instant::now()** library function to capture the time (or ‘Instant’), **t0**, at the start of the code you wish to time.

*Note for the curious:* the **Instant** module contains **opaque** structures, specified as **struct Instant {\_},** so the details of the structure are hidden from you and only revealed using functions, *e.g.* **as\_nanos()**, that unpack the hidden structure. Using

**println!(“Duration {:?}”,duration);**

will only print a string of information, but not the actual elements of the **struct.** This presumably allows portability across different operating systems and architectures, where the access to times uses different strategies.

Add more timing loops to access the struct’s from (a) the array and (b) the vector – use the one your created in the previous section. For a vector, remember that **pop()** will produce a **Result** (an enumerated type) and you need to add a **match** to select whether the result is **Some(x)** (where **Some(*x*)**contains the popped value) or **None** (i.e. there was some error – for a vector, most likely an empty vector.

Thanks to your holiday last week, you now know how to make iterators which access every element of an array or a vector in turn. They are convenient because you do not need to keep track of the actual capacity of a dynamic structure, *e.g.* a vector. Add two loops that time access to your array or vector. Don’t forget to fill the vector again after popping – **pop** removes the last element, so your vector will be empty after the timing loop!!

For a vector, you need to put an **&** in front of the vector to make the iterator

**for p in &v { … }**

*(this due to Rust’s ownership rules .. a delight in store for you soon 😊!)*

***or*** you can add an explicit iterator

**for p in v.iter() { … }**

Finally, make a short report summarizing your results, *i.e.* which access method is faster.

Show your code to the TA and run your program to produce some reasonably readable output (*i.e.* comment out any debugging output first – including debug output from the example code 😊), so that the TA can sign off your answers. Of course, you can put all debugging code in **eprint!(..)** or **eprintln!(..)** statements and capture the ‘clean’ output in a file:

**cargo run >out.txt**

**Fill in a sample of your results and your summary in the attendance record and hand it in.**

**Website: kris.kmitl.ac.th/clinic/Courses/Rust/**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attendance** | **01286120** | **Elementary Systems Programming** | **6 Sep 2022** |

|  |  |  |
| --- | --- | --- |
| **Name (Thai script\*)** |  | **Student ID** |
| **(Latin characters -  as you enrolled)** |  |
| **\****Please write clearly: practice for one farang who is trying to improve* **😉** | | |

|  |  |  |
| --- | --- | --- |
| **Check 1** | Sketch the **struct** of the structure you chose | TA |
|  | |
| **Check 2** | Array times | TA |
| Array size used:  Average time to add/element  Average time to access/element | |
| **Check 3** | Vector times | TA |
| Vector size used:  Average time to build/element  Initial capacity (at least 3 values, including max) Average time  Average time to access/element  Same for all initial capacities? | |
| **Check 4** | Vector building | TA |
| List capacities as they were changed | |
| **Check 5** Access times using iterators  Array  Vector | | TA |
| **Check 6** Short summary of your results. | | TA ( \_ / 5) |