ADT Overview

A routine's performance can be judged in many ways and on many levels. In other laboratories, you describe performance using order-of-magnitude estimates of a routine's execution time. You develop these estimates by analyzing how the routine performs its task, paying particular attention to how it uses iteration and recursion. You then express the routine's projected execution time as a function of the number of data items (N) that it manipulates as it performs its task. The results are estimates of the form O(N), O(Log$N$), and so on.

These order-of-magnitude estimates allow you to group routines based on their projected performance under different conditions (best case, worst case, average case). As important as these order-of-magnitude estimates are, they are by their very nature, high-level estimates of the amount of work that needs to be done, not how much time it will take. They do not take into account factors specific to a particular environment, such as how a routine is implemented, the hardware and operating system of the computer system on which it is being run, and the kind of data being processed. If you are to accurately determine how well or poorly a given routine will perform in a particular environment, you need to evaluate the routine in that environment.

In this laboratory, you measure the performance of a variety of routines. You begin by completing a set of tools that allow you to measure execution time. Then you use these tools to measure the execution times of the routines.

You can determine a routine's execution time in a number of ways. The timings performed in this laboratory will be generated using the approach summarized next.

```
Get the current system time (call this startTime).
Execute the routine.
Get the current system time (call this stopTime).
The routine's execution time = stopTime – startTime.
```

If the routine executes very rapidly, then the difference between *startTime* and *stopTime* may be too small for your computer system to measure. Should this be the case, you need to execute the routine multiple times and divide the length of the resulting time interval by the number of repetitions, as follows:

```
Get the current system time (call this startTime).
Execute the routine m times.
Get the current system time (call this stopTime).
The routine's execution time = ( stopTime – startTime ) / m.
```

To use this approach, you must have some way to read and then save the "current system time". How the current system time is defined and how it is accessed varies from system to system. Two common methods are outlined here.

Method 1

Use a function call to get the amount of processor time that your program (or process) has used. Typically the processor time is measured in clock ticks or fractions of a second. You can use this approach on most systems. You should use this approach on multiuser or multiprocess systems where the routine you are timing is not the only program running. This is what we do in our implementation of the Timer ADT.

Method 2

Use a function call to get the current time of day. Time of day is also called "wall clock" because it refers to the current real time the way a clock on the wall would measure it; using it to measure time measures the time spent in all system processes—not just the one being timed. This method may be sufficiently accurate on single-user/single-process systems where the routine you are timing is the only program running.

In addition to acquiring and storing a point in time, you also need a convenient mechanism for measuring time intervals. The Timer ADT described below uses the familiar stopwatch metaphor to describe the timing process.

Start the timer.

...

Stop the timer.

Read the elapsed time.

C++ Concepts Overview

STL: The Standard Template Library (STL) provides implementations of common data structures and algorithms. These data structures mirror many of the data structures that you have developed in this lab book. We give a brief exposure to the STL in this lab to help make you aware of the STL and to illustrate some more advanced C++ coding techniques. For general use, we recommend that you use the STL unless there is a compelling reason not to, such as when working on an old system that does not support the STL.

Overloading pre- vs. post-increment operators: Both the pre- and post-increment operators can be overloaded. Both are unary operators with the same name (`operator++`), but a slightly different signature so that the compiler knows which one to use. They require different implementations to achieve their semantics properly. One of the learning goals of this lab is to discover that the difference between the two operators is not just a matter of semantics, but also that the performance of the two operators can differ significantly.

Templated functions: In addition to using templates to create entire generic classes, we can use the template facility to create generic functions. This is useful when you want the same functionality in one function for different data types.

C preprocessor: Sometimes typing in everything the compiler needs to see unnecessarily complicates the code, making it more difficult to write correctly and harder to understand. We use some preprocessor magic—inherited from C—to improve the code legibility. We use `#define` in *constructor.cpp* to clean up a complicated function call.

Function pointers: C++ supports the ability to use a pointer to reference functions, not just data. Dereferencing function pointers results in the execution of the referenced code. This advanced feature is very powerful and was initially used to implement many aspects of object oriented programming—such as inheritance and polymorphism—in C++ compilers.

Timer ADT

Data Items

A pair of values that denote the beginning and length of a time interval.

Structure

None

Operations

`Timer ()`

Requirements:
None

Results:
Constructor. Initializes the internal timer values so that the timer is ready to measure time.

`void start () throw ( runtime_error )`

Requirements:
The `clock` function is working correctly.

Results:
Marks the beginning of a time interval (starts the timer).

`void stop () throw ( logic_error )`

Requirements:
The beginning of a time interval has been marked.

Results:
Marks the end of a time interval (stops the timer).

`double getElapsedTime () const throw ( logic_error )`

Requirements:
The beginning and end of a time interval have been marked.

Results:
Returns the length of the time interval in seconds.

Implementation Notes

The start operation is the only operation in the lab book that throws an exception other than `logic_error`. We do this because the exception would come from an error in the system—not in the ADT or parameters to the function. Therefore, the exception is not a logical error, but a runtime error.

Noticing the runtime error illustrates an important systems programming concept: checking the error status of system calls. The clock function that C++ provides returns an error value of −1 when it doesn't work correctly. By checking for this errant value, we ensure that we don't continue a timing operation when we have no way of getting a meaningful time from the system. We throw an error to let the programmer using the ADT decide how to proceed.

Step 1: Select one of the two methods for acquiring and representing a point in time and use this method to create an implementation of the Timer ADT. Base your implementation on the class declaration from the file *Timer.h*.

Step 2: Save your implementation of the Timer ADT in the file *Timer.cpp*.

Step 3: Determine the resolution of your implementation. That is, what is the shortest time interval it can accurately measure? You must determine the correct way to report the CPU usage in seconds.

Compilation Directions

When measuring the performance of your routines, you must tell the compiler to generate efficient code. By default, most compilers generate machine code that is similar to the C++ code you write. Usually, this is not the fastest way to execute on a machine. Instead, you should use a "compiler flag" that directs the compiler to generate the most efficient machine code it can. Ask your instructor for directions on how to perform this for your compiler.

Testing

Step 1: Write a test program that allows you to test the accuracy of your implementation of the Timer ADT by measuring time intervals of known duration. Our implementation of the timer returns different values based on the operating system

Step 2: Compile your implementation of the Timer ADT in the file *Timer.cpp* and the test program in the file *test13.cpp*.

Step 3: Prepare Test Plan 13-1 for your implementation of the Timer ADT. Your test plan should cover intervals of various lengths, including intervals at or near the resolution of your implementation.

Step 4: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Measurement and Analysis Exercise 1

In this exercise, you examine the performance of the searching routines in the file *search.cpp.*

Step 1: Use the program in the file *search.cpp* to measure the execution times of the `linearSearch`, `binarySearch`, and `STLSearch` classes.

This program begins by generating an ordered list of integer keys (`keyList`) and a set of keys to search for in this list (`searchSet`). It then measures the amount of time it takes to search for the keys using the specified routines and computes the average time per search.

The constant `numRepetitions` controls how many times each search is executed. Depending on the speed of your system, you may need to use a value of `numRepetitions` that differs from the value given in the test program. **Before continuing, experiment or check with your instructor to determine the value of `numRepetitions` you should use in order to obtain meaningful timing data.**

Step 2: Fill in Timings Table 13-2 with the observed execution times of the `linearSearch`, `binarySearch`, and `STLSearch` routines for each combination of the three test categories and the three values of `numKeys` listed in the table.

Step 3: Plot your results in a spreadsheet. The resulting graph should show time in seconds on the *Y* axis vs. the number of items in the list on the *X* axis. Submit the graph as directed by your instructor.

Step 4: Fill in your answer to Question 1 on the Analysis Exercise 1 worksheet: "How well do your measured times conform to the order-of-magnitude estimates given for the `linearSearch` and `binarySearch` routines?"

Step 5: Fill in your answer to Question 2 on the Analysis Exercise 1 worksheet: "Using the code in the file *search.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the `STLSearch` routine. Briefly explain your reasoning behind this estimate."

Measurement and Analysis Exercise 2

In this exercise, you examine the performance of the set of sorting routines in the file *sort.cpp*.

Step 1: Use the program in the file *sort.cpp* to measure the execution times of the `selectionSort`, `quickSort`, and STL `sort` routines.

This program begins by generating a list of integer keys (`keyList`). It then measures the amount of time it takes to sort this list into ascending order using the specified routine.

The constant `numRepetitions` controls how many times each sort is executed. Depending on the speed of your system, you may need to use a value of `numRepetitions` that differs from the value given in the test program. **Before continuing, experiment or check with your instructor to determine the value of `numRepetitions` in *sort.cpp* you should use in order to obtain meaningful timing data.**

Step 2: Fill in Timings Table 13-3 with the observed execution times of the `selectionSort`, `quickSort`, and STL `Sort` routines for each combination of the three test categories and the three values of `numKeys` listed in the table.

Step 3: Plot your results in a spreadsheet. The resulting graph should show time in seconds on the *Y* axis vs. the number of items in the list on the *X* axis. Submit the graph as directed by your instructor.

Step 4: Fill in your answer to Question 1 on the Analysis Exercise 2 worksheet: "How well do your measured times conform with the order-of-magnitude estimates given for the `selectionSort` and `quickSort` routines?"

Step 5: Fill in your answer to Question 2 on the Analysis Exercise 2 worksheet: "Using the code in the file *sort.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the STL `sort` routine. Briefly explain your reasoning behind this estimate.

Measurement and Analysis Exercise 3

In this exercise, you measure the performance of common C++ actions. We are going to explore the performance implications of different ways of using constructors and increment operators in this lab.

Step 1: The program in *constructor.cpp* lets you test the performance implications of putting a constructor inside a `for` loop versus just prior to the loop. Compile *constructor.cpp*, *Timer.cpp*, and *TestVector.cpp* together to generate a test program.

Step 2: Using the program that you generated in Step 1, measure the time it takes to construct/initialize `int`, `double`, `vector`, and `TestVector` both just prior to and inside a loop. Record the results in Table 13-4.

Step 3: Fill in your answer to Question 1 on the Analysis Exercise 3 worksheet: "For each data type, how do your measured times for the constructor just before the loop compare to the times for the constructor inside the loop? What might explain any observed differences?"

Step 4: The program in *increment.cpp* lets you test the performance implications of the post-increment operator compared with the pre-increment operator. Compile *increment.cpp*, *Timer.cpp*, and *TestVector.cpp* together to generate a test program.

Step 5: Using the program that you generated in Step 4, measure the times recorded for the pre-increment and post-increment tests for `int`, `double`, and `TestVector`. Record the results in Table 13-5.

Step 6: Fill in your answer to Question 2 on the Analysis Exercise 3 worksheet: "For each data type, how do your measured times for the pre-increment operator compare to the times for the post-increment operator? What might explain any observed differences?"

Analysis Exercise 1

You are given another pair of searching routines. Both routines have order-of-magnitude execution time estimates of O(N). When you measure the actual execution times of these routines on a given system using a variety of different data sets, you discover that one routine consistently executes five times faster than the other. How can both routines be O(N), yet have different execution times when they are compared using the same system and the same data?

Analysis Exercise 2

Why might the authors of the STL choose a search implementation that has the big-O performance that you observed?