SCSSE, University of Wollongong
CSCI124 Applied Programming
Autumn 2015

Assignment 2 (Due 11:59 PM Sun 26th April)

Objectives

To implement an array based sorting algorithm.
To work using a generic type T.
To write a makefile which makes two programs requiring different options.

Getting Started - Dual Pivot Quicksort

Ordinary quicksort uses a single pivot to divide an array into two parts: values less than or equal to pivot values greater than pivot

For this quicksort, we will divide the array into three parts using two pivots: values less than pivot1, values between pivots 1 and 2, values greater than pivot2.

A description of the algorithm, with pseudocode and some pictures, can be found in the pdf file named dualPivotQuicksort.pdf. You can also find a visualisation of the algorithm with some pseudocode from http://tinyurl.com/dpquicksort which works provided A[left] is less than A[right].

Read through the pdf and look at the visualisation to understand the algorithm.

Part 1 - Two Programs(3 marks)

Look through the source files provided, get an idea of what is being done. Each file should start with a block comment describing its purpose.

- Add an appropriate block comment at the start of each file that does not already have one.

There are two programs to make: **sortints** and **sortmoves**. The first program just sorts a small array of integers. The second sorts a dataset from a file (which contains the direction, date, time, and number of people entering or exiting a building in half-hour intervals), in ascending order according to the number of people moving at that time.

They can already be compiled directly on the commandline using:

```
g++ main1.cpp sort.cpp -o sortints
g++ main2.cpp movement.cpp sort.cpp -D MOVEMENT -o sortmoves
```

you should not need to alter the source files for this step.

- Write a makefile named **makefile** to make the two programs. It should run using the command `make` and should only recompile/relink those files which need it.

Writing the targets to make sortints should be relatively straightforward. Making sortmoves is trickier. It requires the compiler option -D MOVEMENT whenever sort.h is included. The trouble occurs because we are conditionally compiling a typedef in sort.h, to set T to be either an int or a Move struct. So sortmoves requires MOVEMENT to be defined when compiling sort.cpp, but sortints requires MOVEMENT to *not* be defined!

To fix this, you will need to make 2 different targets which compile sort.cpp with different options; for example, named sort1.o and sort2.o. When linking to form the programs, the first without (-D MOVEMENT), should be linked with main1.o, and the second with (-D movement) should be linked with main2.o and movement.o.

To ensure both programs are made when only **make** is typed, place, at the top of your makefile, a target named **all**, requiring the two files **sortints** and **sortmoves**, and having no commands. The rest of the makefile should contain all the targets necessary to compile all the necessary components and the two programs. Don't forget to include a clean target.

Part 2 - Dual Pivot Quicksort(3 marks)

If you first compile and run the programs, you may notice that neither of program actually sorts anything. This is because the function

```
void dualPivotQuicksort(T array[], int left, int right, int &ncomp, int& nswap)
```

is currently just a stub.

- Implement the dualPivotQuicksort function according to the pseudocode given in dualPivotQuicksort.pdf.

You will notice that array elements are of type T, while indices are of type int. The sort should be able to work with any type for which the less than and greater than comparisons work.

You will find it easier to use **sortints** for initial tests that your implementation is working. Once it is, compile and run **sortmoves**, and use **less** or **head** and **tail** to check that `sorted_movement.txt` contains the building movement data sorted in ascending order by count.

Part 3 - Performance(2 marks)

- If you haven't done it already, increment ncomps for each comparison and nswaps for each swap. Run both programs and record the number of comparisons and swaps in a text file named `peformance.txt`

  Currently, we are just using the first and last elements (in order) as pivot1 and pivot2. This has the same problem as standard quicksort, as pivot1 might be the smallest element, and pivot2 might be the largest. A better solution is to call a function which examines 5 elements, for example: first, one_quarter, middle, three_quarters, and last. The 2nd smallest element should be swapped into the first position, while the 2nd largest element should be swapped into the last position.

- Write the function to determine the pivots and swap them into place. Call it at the start of each partitioning phase.

- Check that you are still sorting everything correctly. Run both programs and append the number of comparisons and swaps in your improved sort a text file named `peformance.txt`.

Submission instructions

Submit the files for your assignment main1.cpp main2.cpp movement.cpp sort.cpp makefile performance.txt.
`submit -u <username> -c CSCI124 -a 2 main1.cpp main2.cpp movement.cpp sort.cpp makefile perform`
from the directory containing your file, where `<username>` is your SOLS username.

1. Late submissions will be marked with a 25% deduction for each day.
2. Submissions more than three days late will not be marked, unless an extension has been granted.
3. If you need an extension apply through SOLS, if possible **before** the assignment deadline.
4. Plagiarism is treated seriously. If we suspect any work is copied, all students involved are likely to receive zero for the entire assignment.