CSE310 Project 1: Makefile, Modular Design, File I/O, Pointers, Dynamic Memory Allocation, Command Line Interpretation
Posted: Thursday, 02/18/2021, Due: Monday, 03/01/2021

This is your first programming project. It should be implemented in C++, on a Linux platform such as general.asu.edu. Your program will be graded on Gradescope, which uses a Linux platform. You will perform modular design, provide a Makefile to compile various modules to generate the executable file named run. Among other things, you need to have

1. a main program, which coordinates all other modules;
2. a module that provides utility services including command line interpretation;
3. a module that implements the max-heap data structure (not all the functions yet);
4. a Makefile which compiles all modules and link them into the executable.

For each module other than the main program, you should have a **header file** which specifies the data structures and the prototypes of the functions in the module, and an **implementation file** which implements all of the functions specified in the header file. **All programs will be compiled and graded on Gradescope**. If your program works well on general.asu.edu, there should not be much problems. You will need to submit it electronically on Gradescope via the link on Canvas. Submission details will be posted shortly. **If your program does not compile and work on Gradescope, you will receive 0 on this project**.

You need to define the following data types.

- ELEMENT is a data type that contains a field named key, which is of type int. Note that ELEMENT should not be of type int.
- HEAP is a data type that contains three fields named capacity (of type int), size (of type int), and H (an array of type ELEMENT with index ranging from 0 to capacity).

The functions that you are required to implement are:

- Initialize(n) which returns an object of type HEAP with capacity n and size 0. This function requires you to perform dynamic memory allocation, given the demand n.
- printHeap(heap) which prints out the heap information, including capacity, size, and the key fields of the elements in the array with index going from 1 to size.

You should implement a module that takes the following commands from the key-board and feeds to the main program:

- **S**
- **C n**
- **R**
- **W**
- **P**

On reading **S**, the program stops.

On reading **C n**, the program (1) creates a heap with `capacity` equal to **n**, and `size` equal to 0, and (2) waits for the next command. Note that dynamic memory allocation is required here.

On reading **R**, the program (1) opens the file "HEAPinput.txt" in read mode; (2) reads in the first integer n in the file; (3) reads in the next n integers $key_1, key_2, \ldots, key_n$ from the file, dynamically allocates memory for an `ELEMENT`, sets it `key` to key_j, and let the pointer `heap->Elements[j]` points to this `ELEMENT`; and (4) waits for the next command.

On reading **P**, the program (1) writes the current heap information to the screen, and (2) waits for the next command. The output should be in the same format as in the file `HEAPinput.txt`, proceeded by the heap capacity.

On reading **W**, the program (1) opens the file "HEAPoutput.txt" for writing, (2) writes the current heap size and the keys of the elements into "HEAPoutput.txt", and (3) waits for the next command. The file "HEAPoutput.txt" should be in the same format as the file `HEAPinput.txt`.

The file HEAPinput.txt is a text file. The first line of the file contains an integer n, which indicates the number of array elements. The next n lines contain n integers, one integer per line. These integers are the key values of the n array elements, from the first element to the nth element.

Grading policies: (Sample test cases are posted on Canvas.) All programs will be graded on Gradescope. If your program does not compile and execute on Gradescope, you will receive 0 for this project. So start working today, and do not claim "my program works perfectly on my PC, but I do not know how to use Gradescope."

(5 pts) You should provide a Makefile that can be used to compile your project on Gradescope. The executable file should be named run. If your program does not pass this step, you will receive 0 on this project.

(5 pts) Modular design: You should have a file named util.cpp and its corresponding header file util.h, where the header file defines the prototype of the functions, and the implementation file implements the functions. You should have a file named heap.cpp and its corresponding header file heap.h. This module implements (some of) the heap functions.

(5 pts) Documentation: You should provide a README.txt file that clearly specifies the file and line numbers where dynamic memory allocation is used.

(5 pts) Your program should use dynamic memory allocation correctly. You should document this clearly both in the README.txt file and in the implementation file(s).

(50 pts) You will earn 5 points for each of the 10 posted test cases your program passes on Gradescope.

(10 pts) You will earn 5 points for each of the 2 unposted test cases your program passes on Gradescope.

You should try to make your program as robust as possible. A basic principle is that your program can complain about bad input, but should not crash.

As an aid, the following is a partial program for reading in the commands from the keyboard. You need to understand it and to expand it.

```
#include "util.h"
//=============================================================================
int nextCommand(int *n, int *f)
{
  char c;
  while(1){
    scanf("%c", &c);
    if (c == ' ' || c == '\t' || c == '\n'){
        continue;
    }
    if (c == 'S'){
        break;
    }

    if (c == 'C' || c == 'c'){
        scanf("%d", n);
        break;
    }
    if (...){
    ...
    }
  }
  return c;
}
//===============================================================================
```

The following is a partial program that calls the above program.

```
//===============================================================================
#include <stdio.h>
#include <stdlib.h>
#include "util.h"

int main()
{
    // variables for the parser...
    char c;
    int i, v;
    while(1){
        c = nextCommand(&n, &f);
        switch (c) {
            case 's':
            case 'S': printf("COMMAND: %c\n", c); exit(0);

            case 'c':
```

```
            case 'C': printf("COMMAND: %c %d\n", c, n);
                      heap = heapInit(n);
                      break;

            case 'r':
            case 'R': printf("COMMAND: %c\n", c);
                      ifile = fopen("HEAPinput.txt", "r");
                      if (!ifile){
                      }
                      fscanf(ifile, "%d", &n);
                      ...

            default: break;
        }
    }
    exit(0);
}
//================================================================================
```

The following is a partial Makefile.

```
EXEC = run
CC = g++
CFLAGS = -c -Wall

# $(EXEC) has the value of shell variable EXEC, which is run.
# run depends on the files main.o util.o heap.o
$(EXEC) :main.o util.o heap.o
# run is created by the command g++ -o run main.o util.o
# note that the TAB before $(CC) is REQUIRED...
        $(CC) -o $(EXEC) main.o util.o heap.o

# main.o depends on the files main.h main.cpp
main.o:main.h main.cpp
# main.o is created by the command g++ -c -Wall main.cpp
# note that the TAB before $(CC) is REQUIRED...
```

```
        $(CC) $(CFLAGS) main.cpp

util.o  :util.h util.cpp
        $(CC) $(CFLAGS) util.cpp

heap.o  :heap.h heap.cpp
        $(CC) $(CFLAGS) heap.cpp

clean   :
        rm *.o
```