CSE310 Project 1: Makefile, Modular Design, File I/O, Pointers, Dynamic Memory Allocation, Command Line Interpretation
Posted: 02/18/2021, Updated: 02/23/2021, Due: 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 heap data structure (not all heap 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 have been posted as of 02/23/2021. **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 `struct` that contains a field named `key`, which is of type `int`. In later assignments, you will add other fields to `ELEMENT`, without having to change the functions. 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` (of type `**ELEMENT`). `H` will be pointing to an array of `capacity + 1` of pointers of type `*ELEMENT`. Note that the size of `HEAP` should be equal to `12`, regardless of the `capacity` or the `size` of the heap. In other words, `sizeof(HEAP)` should always return `12`.

The functions that you are required to implement are:

- `Initialize(n)` which creates an object of type `HEAP` with `capacity` n, `size` 0, and `H` points to a dynamically allocated array of $n + 1$ pointers. It then returns a pointer to this object. This function requires you to perform dynamic memory allocation, given the demand n.

- `printHeap(heap)` which prints out the information of the heap pointed to by `heap`, 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 `stdin` and feeds to the main program:

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

The main program should react to each of the above command in the following way.

S: On reading **S**, the program

1. Writes the following line to `stdout`:

   `COMMAND: S`

   where `S` is the character `S`.
2. Stops.

C: On reading **C n**, the program

1. Writes the following line to `stdout`:

   `COMMAND: C n`

   where `C` is the character `C` and `n` is replaced by the **value** of `n`.
2. Calls a function in the `heap` module to create a heap with `capacity` equal to **n** and `size` equal to 0, and return a pointer to this heap object to the caller.
3. Waits for the next command from `stdin`.

R: On reading **R**, the program

1. Writes the following line to `stdout`:

`COMMAND: R`

where `R` is the character `R`.

2. Opens the file "HEAPinput.txt" in read mode. If the file is not opened successfully, writes the following line to `stdout`:

   `Error: cannot open file for reading`

   and waits for the next command from `stdin`.

3. Reads in the first integer `n` from the file opened.
   If `heap` is `NULL` or `heap->capacity` is smaller than `n`, writes the following line to `stdout`:

   `Error: heap overflow`

   and waits for the next command from `stdin`.

4. 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 `heap->H[j]` points to this `ELEMENT`, for $j = 1, 2, \ldots, n$.

5. Waits for the next command from `stdin`.

P: On reading **P**, the program

1. Writes the following line to `stdout`:

   `COMMAND: P`

   where `P` is the character `P`.

2. If `heap` is `NULL`, writes the following line to `stdout`:

   `Error: heap is NULL`

   and waits for the next command from `stdin`.

3. Writes the information of the heap pointed to by `heap` to `stdout`. Refer to the posted test cases for the output format.

4. Waits for the next command from `stdin`.

W: On reading **W**, the program

1. Writes the following line to `stdout`:

   `COMMAND: W`

where `W` is the character `W`.

2. Opens the file "HEAPout.txt" in write mode. If the file is not opened successfully, writes the following line to `stdout`:

   `Error: cannot open file for writing`

   and waits for the next command from `stdin`.

3. If `heap` is `NULL`, writes the following line to `stdout`:

   `Error: heap is NULL`

   and waits for the next command from `stdin`.

4. Writes the information of the heap pointed to by `heap` to the file "HEAPoutput.txt". "HEAPoutput.txt" should have exactly the same format as "HEAPinput.txt".

5. Waits for the next command from `stdin`.

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. Refer to the posted test cases for the exact format of "HEAPinput.txt".

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 implementationfile(s), indicating the file name(s) and line number(s) where dynamic memory allocation is implemented.

(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.

NOTE: For test case #5, the file `output.txt` was produced by the following `bash` script:

```
./run < input.txt > output.txt
cat HEAPoutput.txt >> output.txt
```

We use this to check the content of your output file.

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.

```
typedef struct TAG_ELEMENT{
    int key;
}ELEMENT;

typedef ELEMENT *ElementT;

typedef struct TAG_HEAP{
    int capacity;  /* max size of the heap */
    int size;      /* current size of the heap */
    ElementT *H;  /* pointer to pointers to elements */
}HEAP;
```

```
#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
```