Purpose

The purpose of this assignment is to continue to have you work on an array of structures and File I/O. **No input re-direction is allowed for the data sets, the input files must be read programmatically using File I/O.**

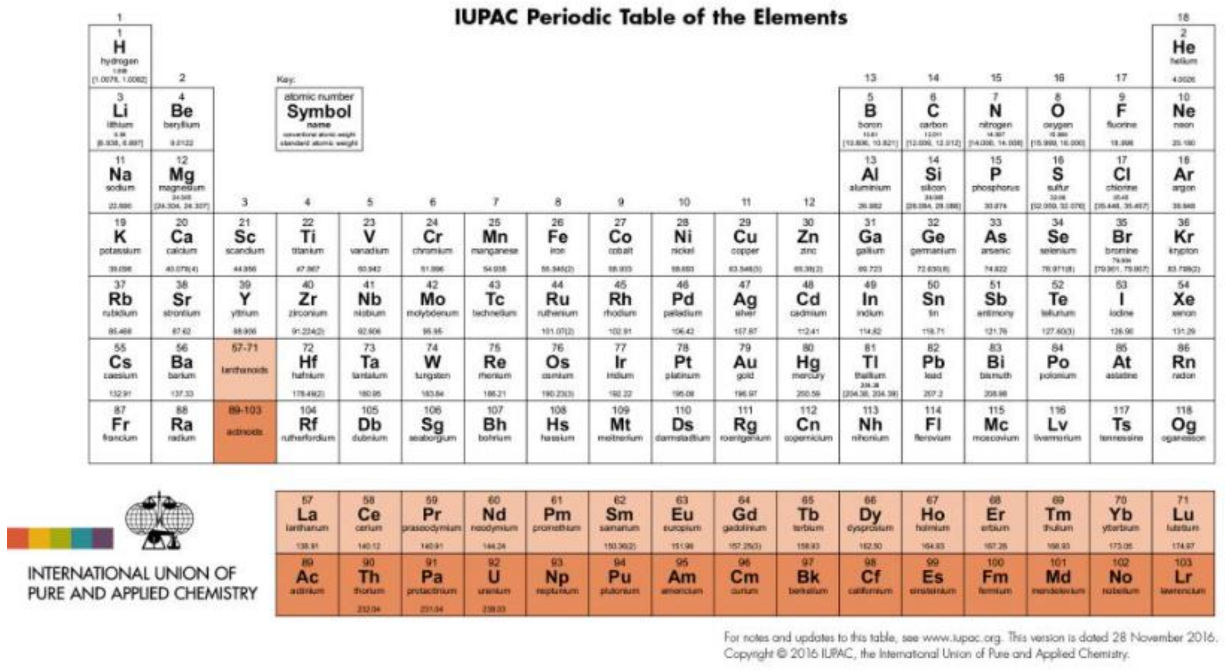
The only user input (from the keyboard or standard input) are two input data file names and the atomic numbers that we want to search for once our data structure contains all of the data.

atomic_elements → input file containing atomic elements with number and weight
other_properties → input file containing melting and boiling points of elements

Scenario

The periodic table, or periodic table of elements, is a tabular arrangement of the chemical elements, ordered by their atomic number, electron configuration, and recurring chemical properties, whose structure shows periodic trends. Generally, within one row (period) the elements are metals to the left, and non-metals to the right, with the elements having similar chemical behaviors placed in the same column. Table rows are commonly called periods and columns are called groups. Six groups have accepted names as well as assigned numbers: for example, group 17 elements are the halogens; and group 18 are the noble gases. Also displayed are four simple rectangular areas or blocks associated with the filling of different atomic orbitals.

Each element in the periodic table has several properties but often we are not able to collect all their most commonly used properties together in one data set.


IUPAC Periodic Table of the Elements
INTERNATIONAL UNION OF PURE AND APPLIED CHEMISTRY
For notes and updates to this table, see www.iupac.org. This version is dated 28 November 2016.
Copyright © 2016 IUPAC, the International Union of Pure and Applied Chemistry.

Problem

Our data for each element in the periodic table may come from different sources. Let's take some of these characteristics of each element and see if we are able to put them together under one data structure so that we are able to make queries against them more easily.

We have two input files that we need to work with:

1) atomic_elements → input file containing atomic elements with number and weight
2) other_properties → input file containing melting and boiling points of elements

The only common item that appears in both data files is the name of the atomic element. Our first task is to be able to set up an array of structures from the first data set that contains the atomic element name, its abbreviation, its atomic number and atomic weight as four different items in one record. After we are done setting up this array we want to be able to add to the properties of these atomic elements with the additional information about melting, boiling points and density (if applicable) for each atomic element.

Once our array is completely set up with all the data we should be able to perform queries by specifying an atomic number and coming up with the details of that atomic element. Input is accepted continuously (integer values) to search by atomic number until the value 0 is entered by the user.

When you load the second data file you will notice that several values for melting point, boiling and density have the value -9999. This simply means that the particular property for that atomic element is either not applicable or unknown. While it's ok for you to load those values into your array, when you report on it when you query individual elements you must print those values as "Unknown"

Details

1) Set up a structure in your header file (.h file) to be able to store all the individual properties of an atomic element namely:
   Name, Symbol, Atomic Number, Atomic Weight, Melting Point in Celsius, Boiling Point in Celsius, and Density. In your main() function, create a variable which is an array of structures called periodic_table of an appropriate size.
2) Load the data from the first data file into the array of structures in a function called read_periodic_table()
3) Write a function called search_periodic_table() which will return the index/subscript of the array that holds the data for the element you are looking for, by name. If the atomic element is not found return the value -1.
4) Loading data from the second file is a bit more involved.Since the second data file contains just the name of the atomic element (along with melting and boiling points), as you read every line from this file, you must search the existing array (updated in Step 2 above) using the function created in Step 3 above and update that particular array element with the melting and boiling points. Continue to do this until the end of the second data file. All of this must be done in a function called update_periodic_table().
5) Print the entire periodic_table array table in a function called print_periodic_table()
6) Now we are ready to search the periodic_table by atomic number and print the properties of the element if it exists in the table. If the element does not exist print a suitable error message. You will need to modify the search_periodic_table() function written in Step 3 above to accommodate searching by atomic number.
7) In your main() function you will need to write a loop prompting for the user to enter an atomic number, call the search_periodic_table() function which will return the index/subscript of the array that holds the element, and print the results in the main() function. You must loop until 0 is entered for an atomic number, at which point the program stops. Make sure you print "Unknown" for any values that are equal to -9999 for their property. If the search function returns -1 it just means that the atomic number entered by the user is invalid, print an appropriate error message in this case.

Input

The input data is coming from two data files present in your local directory. You should read the data into your array. Your array should be capable of holding a maximum of 150 records.

Output

All output is on standard output (programmed output to a file is not needed).

The output must first print out data stored in the array after reading the data from both files. As the user repeatedly searches for an atomic element by specifying the atomic number, the matching record details (or a message indicating record was not found) must be printed. Your output does not have to look exactly like mine but it should be as complete and easy to read. Make sure you indicate "Unknown" values when you encounter the value -9999 for any property for a specific atomic element.

Requirements

- The header file with the structure definition, and some constants is given to you. You should not modify this file since I'm going to testing your functions with my own main() function.
- The structure is called atomic_element_t defined in your header file
- You must use at least the following functions, proto-type declarations of these functions are given below (each of these proto-type declarations are given in the header file supplied):

  ```
  int read_periodic_table(char filename[], atomic_element_t e[]);
  void print_periodic_table(atomic_element_t element[], int size);
  void update_periodic_table(char filename[], atomic_element_t e[], int size);
  int search_periodic_table(int search_by, char name[], int number, atomic_element_t
  e[], int size);
  ```

  Note: The search periodic_table() function must be capable of searching either by atomic name or by atomic number (only the relevant argument i.e. either name or number will be used based on the search_by value).

- The program must be split into multiple files:
  - periodic_table.h is the header file
  - periodic_tabel.c is the file with the main function, which sets up the array of structures, prints out the table and accepts user input to search for atomic elements
  - periodic_table_functions.c is the file that contains all the other functions.

- All function definitions must have a block comment describing what they do.

- Work on the program using the modularity of the functions, for example write a program that reads the data and prints it out. Once you have these working you can move on to the other functions

- As always test your program carefully and follow good programming style.