Exercise 8 - Structures

Objective

This practical session aims to illustrate the importance of careful structure design in C, highlighting the potential savings in memory that can be achieved with relatively little effort. The session also looks at how structures can be read and written to disk in text and binary format.

Reference Material

This session is based entirely on material in the Structures chapter. This practical session is located in the following directory:

	Microsoft Windows	*Linux*
Directory:	***c:\qacadv\struct***	***~/qacadv/atruct***
Solution directory:	***c:\qacadv\struct\Solution***	***~/qacadv/struct/Solution***

Overview

A chemist with a rudimentary understanding of C structures developed a database of chemical elements and some of their properties.

On Microsoft Windows open **make_db.sln, on Linux** change your current working directory and **make make_db**.

Take a look at **elem.h**, which defines the chemist's data structure, `ELEMENTDATA`. The structure was created over a period of time. The programmer thought that new structure members always had to be placed at the end of the structure. As a result, the structure is quite disorganised and there are `chars` and `doubles` scattered liberally:

Type	Name	Description
`char`	`first`	First character of element name
`double`	`rmm`	Relative Molecular Mass (i.e. "Atomic Weight")
`char`	`second`	Second character of element name
`double`	`mp`	Element's melting point in degrees Kelvin
`char`	`bcc`	Set to 'y' if Block Centred Cubic arrangement of atoms
`char`	`cubic`	Set to 'y' if Cubic arrangement
`char`	`fcc`	Set to 'y' if Face Centred Cubic arrangement
`double`	`bp`	Element's boiling point in degrees Kelvin
`char`	`hcp`	Set to 'y' if Hexagonal Close Packed arrangement
`char`	`hex`	Set to 'y' if Hexagonal Planar arrangement of atoms
`char`	`mon`	Set to 'y' if Monoclinic arrangement

`char`	`ortho`	Set to 'y' if Orthorhombic arrangement
`char`	`tetra`	Set to 'y' if Tetrahedral arrangement
`char`	`rhombic`	Set to 'y' if Rhombic arrangement

Note that some elements, for instance Carbon, have *several* valid atomic arrangements: diamond has a tetrahedral arrangement whereas graphite has a hexagonal plane arrangement. Because of this, it was not possible to have a single field representing the atomic arrangement

Practical Outline

1. The information for various chemical elements is held in text format in **elements.txt**, which was generated using a simple text editor.

   Now take a look at the program **make_db.c**. The program reads textual data from **elements.txt** one line at a time, and writes the data in binary format to an output file.

   Build and run this program. The program gives you two options:

   1 Build binary file
   2 Query struct size

   Select option 1, and choose a name such as **elements.rec** for the binary output file. What do you notice about the size of **elements.rec** versus **elements.txt** (the idea of binary files is that they're small)?

   Try to predict the size that an `ELEMENTDATA` structure would have. Then write the body of the `Query()` function in **make_db.c**, so that it prints the `sizeof` one of these structures. Build and run the program again, and this time select option 2 to query the struct size (this calls your `Query()` function). Did you predict the correct struct size? Can you see where (and why) the compiler has inserted padding into the structure?

   Take another look at the `ELEMENTDATA` structure defined in **elem.h**. Working in this file, define a new structure template `NEWELEMENTDATA` to hold the same data but in a more space-efficient manner. There is enormous scope for improvement.

   Consider rearranging members to fill some of the padding. Also consider replacing bytes containing 'y' or 'n' values with something taking much less storage (Hint: about eight times less). It is also worthwhile examining values stored in the `rmm`, `mp` and `bp` fields to see if a data type smaller than `double` would suffice.

   Extend your `Query()` function to determine the size of this new format `NEWELEMENTDATA` structure template. Build and run the program, and query the size of the `NEWELEMENTDATA` structure.

2. Now modify your **make_db.c** so that it writes data to a binary file in the new format (`NEWELEMENTDATA`) rather than the old format (`ELEMENTDATA`). The number of changes you will need to make are surprisingly few:

   - change all usage of `ELEMENTDATA` to `NEWELEMENTDATA`.
   - modify `ReadTextualRec()` to take into account the new format of your element record.
   - if you have changed the way the atomic arrangement flags are stored in the element record, you will also have to change the way `ScanFlags()` sets these arrangement flags in your record.

   Note in particular that there is no need to modify the function `WriteBinaryRec()` because this function performs a "dumb" `fwrite()` to write a whole record in one operation, rather than writing the constituent data members individually.

   Build and run the program, and select option 1 to generate a binary file containing the new-format element records. Notice the improvement in file size that has been achieved.

Optional

3. **On Microsoft Windows** open **disp_db.sln, on Linux** stay in the same directory.

   Using the (empty!) file called **disp_db.c**, write a simple program to read the records from the new binary file, one at a time, and displays the information for each record in the following format:

Element name:	U
Rel. molecular mass:	238.03
Melting point:	1405.40 Kelvin
Boiling point:	4091.00 Kelvin
Rhombic	Tetrahedral

   Hint: write separate functions `ReadBinaryRec()` and `DispBinaryRec()` - use `fread()` to read the individual records from the binary file.