Assignment 1 – C Molecule Manipulation Library

Version 1.02 (last update: Jan. 22, 20:30)
Changes highlighted in yellow
Due date: Tue, Jan 31, 9:00 AM

Summary and Purpose

For this assignment, you will be writing a small C library that implements a number of functions to represent and manipulate molecules.

Deliverables

You will be submitting:

1) A file called **`mol.h`** that contains your typedefs, structure definitions and function prototypes (see below).
2) A file called **`mol.c`** that contains your function code.
3) A **`makefile`** that contains the instructions for the following targets:
    a. **`mol.o`** – a position independent (**`-fpic`**) object code file created from **`mol.c`**.
    b. **`libmol.so`** – a shared library (-shared) created from **`mol.o`**.
    c. **`clean`** – this target should delete all **`.o`**, **`.so`** and executable files.

    All compilation must be done with the **`-std=c99 -Wall -pedantic`** options and produce no warnings or errors. The **`makefile`** must correctly identify all relevant dependencies and include them as prerequisites in its rules.

You will submit all of your work via git to the School's **`gitlab`** server. (As per instructions in the labs.) Under no condition will an assignment be accepted by any other means.

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate, adjudicated using the University's Academic Integrity rules.

Setting up your work environment and using git:

Decide on your development environment. You can work on your own machine, the SoCS VM, no machine, or the SoCS ssh servers. However, you must test your code on the SoCS servers or SoCS VM as this is where it will be evaluated. If your code does not work properly on the SoCS systems, then it does not work.

```
git clone https://gitlab.socs.uoguelph.ca/2750W23/skremer/A1
```

But, use your own login ID instead of **`skremer`**.

Then, **`cd A1`**, to move to the assignment directory.

Work in the A1 directory. Use the command:

```
git add filename
```

For each file that you create as part of your code (one `.c` file, one `.h` file, and the `makefile`).

Every so often (especially if you get something working), use the command:

```
git commit -a
```

to commit your changes.

Use the command:

```
git tag -a Jan10a -m 'implemented the first two functions'
```

You will need to use a tag like "`Jan10a`" to request help with your code via the course help page.

And then make sure to use:

```
git push --all --follow-tags
```

to push everything to the server with the tag included.

If you want to make sure you know exactly what's on the server you can rename the A1 directory to A1.day1 and then use

```
git clone https://gitlab.socs.uoguelph.ca/2750W23/skremer/A1
```

to get a copy of exactly what is currently in the repo.

You can use the same command to retrieve your code on a different machine or architecture. E.g. to move your code from your laptop development to the SoCS server for testing.

Structures and typedefs used for your assignment (include these in your .h file)

```
typedef struct atom
{
  char element[3];
  double x, y, z;
} atom;
```

`atom` defines a structure that describes an atom and its position in 3-dimensional space. `element` is a null-terminated string representing the element name of the atom (e.g. Na for

sodium). **x**, **y**, and **z** are double precision floating point numbers describing the position in Angstroms (Å) of the atom relative to a common origin for a molecule.

```
typedef struct bond
{
  atom *a1, *a2;
  unsigned char epairs;
} bond;
```

bond defines a structure that represents a co-valent bond between two atoms. **a1** and **a2** are pointers to the two atoms in the co-valent bond. **epairs** is the number of electron pairs in the bond (i.e. **epairs**=2 represents a double bond). The atoms pointed to by **a1** and **a2** will be allocated and stored elsewhere, so it will never be necessary to free **a1** or **a2**.

```
typedef struct molecule
{
  unsigned short atom_max, atom_no;
  atom *atoms, **atom_ptrs;
  unsigned short bond_max, bond_no;
  bond *bonds, **bond_ptrs;
} molecule;
```

molecule represents a molecule which consists of zero or more **atom**s, and zero or more **bond**s. **atom_max** is a non-negative integer that records the dimensionality of an array pointed to by **atoms**. **atom_no** is the number of **atom**s currently stored in the array **atoms**. Note **atom_no** must never be larger than **atom_max**. You will be responsible for allocating enough memory to the **atoms** pointer. **bond_max** is a non-negative integer that records the dimensionality of an array pointed to by **bonds**. **bond_no** is the number of **bond**s currently stored in the array **bonds**. Note **bond_no** must never be larger than **bond_max**. You will be responsible for allocating enough memory to the **bonds** pointer. **atom_ptrs** and **bond_ptrs** are arrays of pointers. Their dimensionalities will correspond to the **atoms** and **bonds** arrays, respectively. These pointers in these pointer arrays will be initialized to point to their corresponding structures. E.g. **atom_ptrs[0]** will point to **atoms[0]**. Later we will sort the order of the pointers to allow for an alternate traversal (ordering) of the **atoms**/**bonds**.

```
typedef double xform_matrix[3][3];
```

xform_matrix reprents a 3-d affine transformation matrix (an extension of the 2-d affine transformation you saw in the first lab).

Function prototypes and descriptions for your assignment

```
void atomset( atom *atom, char element[3], double *x, double *y, double *z );
```

This function should copy the values pointed to by **element**, **x**, **y**, and **z** into the **atom** stored at **atom**. You may assume that sufficient memory has been allocated at all pointer addresses.

Note that using pointers for the function "inputs", **x**, **y**, and **z**, is done here to match the function arguments of **atomget**.

```
void atomget( atom *atom, char element[3], double *x, double *y, double *z );
```

This function should copy the values in the **atom** stored at **atom** to the locations pointed to by **element**, **x**, **y**, and **z**. You may assume that sufficient memory has been allocated at all pointer addresses. Note that using pointers for the function "input", **atom**, is done here to match the function arguments of **atomset**.

```
void bondset( bond *bond, atom *a1, atom *a2, unsigned char epairs );
```

This function should copy the values **a1**, **a2** and **epairs** into the corresponding structure attributes in **bond**. You may assume that sufficient memory has been allocated at all pointer addresses. Note you are not copying **atom** structures, only the addresses of the **atom** structures.

```
~~void bondget( bond *bond, atom *a1, atom *a2, unsigned char epairs );~~
void bondget( bond *bond, atom **a1, atom **a2, unsigned char *epairs );
```

This function should copy the structure attributes in **bond** to their corresponding arguments: **a1**, **a2** and **epairs**. You may assume that sufficient memory has been allocated at all pointer addresses. Note you are not copying **atom** structures, only the addresses of the **atom** structures.

```
molecule *molmalloc( unsigned short atom_max, unsigned short bond_max );
```

This function should return the address of a **malloc**ed area of memory, large enough to hold a **molecule**. The value of **atom_max** should be copied into the structure; the value of **atom_no** in the structure should be set to zero; and, the arrays **atoms** and **atom_ptrs** should be **malloc**ed to have enough memory to hold **atom_max** **atom**s and pointers (respectively). The value of **bond_max** should be copied into the structure; the value of **bond_no** in the structure should be set to zero; and, the arrays **bonds** and **bond_ptrs** should be **malloc**ed to have enough memory to hold **bond_max** **bond**s and pointers (respectively).

```
molecule *molcopy( molecule *src );
```

This function should return the address of a **malloc**ed area of memory, large enough to hold a **molecule**. Additionally, the values of **atom_max**, **atom_no**, **bond_max**, **bond_no** should be copied from **src** into the new structure. ~~Finally, the~~ The arrays **atoms**, **atom_ptrs**, **bonds** and **bond_ptrs** must be allocated to match the size of the ones in **src**. Finally, you should use **molappend_atom** and **molappend_bond** (below) to add the atoms from the **src** to the new molecule (note that this will also initialize the corresponding pointer arrays). You should re-use (i.e. call) the **molmalloc** function in this function.

```
void molfree( molecule *ptr );
```

This function should **free** the memory associated with the **molecule** pointed to by **ptr**. This includes the arrays **atoms**, **atom_ptrs**, **bonds**, **bond_ptrs**.

```
void molappend_atom( molecule *molecule, atom *atom );
```

This function should copy the data pointed to by **atom** to the first "empty" **atom** in **atoms** in the **molecule** pointed to by **molecule**, and set the first "empty" pointer in **atom_ptrs** to the same **atom** in the **atoms** array incrementing the value of **atom_no**. If **atom_no** equals **atom_max**, then **atom_max** must be incremented, and the capacity of the **atoms**, and **atom_ptrs** arrays increased accordingly. If **atom_max** was 0, it should be incremented to 1, otherwise it should be doubled. Increasing the capacity of **atoms**~~, and **atom_ptrs**~~ should be done using **realloc** so that a larger amount of memory is allocated and the existing data is copied to the new location. IMPORTANT: After **malloc**ing or **realloc**ing enough memory for **atom_ptrs**, these pointers should be made to point to the corresponding atoms in the new **atoms** array (not the old array which may have been freed).

```
void molappend_bond( molecule *molecule, bond *bond );
```

This function should operate like that **molappend_atom** function, except for bonds.

```
void molsort( molecule *molecule );
```

This function should sort the **atom_ptrs** array in place in order of increasing **z** value. I.e. **atom_ptrs[0]** should point to the atom that contains the lowest **z** value and **atom_ptrs[atom_no-1]** should contain the highest **z** value. It should also sort the **bond_ptrs** array in place in order of increasing "**z** value". Since **bond**s don't have a **z** attribute, their **z** value is assumed to be the average **z** value of their two **atom**s. I.e. **bond_ptrs[0]** should point to the bond that has the lowest **z** value and **bond_ptrs[atom_no-1]** should contain the highest **z** value.Hint: use **qsort**.

```
void xrotation( xform_matrix xform_matrix, unsigned short deg );
```

This function will ~~allocate, compute, and return~~ set the values in an affine transformation matrix, **xform_matrix**, corresponding to a rotation of **deg** degrees around the *x*-axis. ~~This matrix must be **free**d by the user when no longer needed.~~

```
void yrotation( xform_matrix xform_matrix, unsigned short deg );
```

This function will ~~allocate, compute, and return~~ set the values in an affine transformation matrix, **xform_matrix**, corresponding to a rotation of **deg** degrees around the *y*-axis. ~~This matrix must be **free**d by the user when no longer needed.~~

```
void zrotation( xform_matrix xform_matrix, unsigned short deg );
```

This function will ~~allocate, compute, and return~~ set the values in an affine transformation matrix, **`xform_matrix`**, corresponding to a rotation of **`deg`** degrees around the *z*-axis. ~~This matrix must be freed by the user when no-longer needed.~~

```
void mol_xform( molecule *molecule, xform_matrix matrix );
```

This function will apply the transformation **`matrix`** to all the **`atoms`** of the **`molecule`** by performing a vector matrix multiplication on the **`x`**, **`y`**, **`z`** coordinates.

Your code should **`malloc`** only as much memory as required in the function descriptions. All **`malloc`** return values must be checked before accessing memory. If **`malloc`** returns a **`NULL`** value, the return value of the calling function should also be **`NULL`**.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will be updated with some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the SoCS servers. If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. If your code works on one machine/environment but not another, you code is incorrect. Correct code will work consistently across all machines/environments. We will NOT test your code on YOUR machine/environment.

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.

Nightmare mode

Students who want an extra challenge for zero extra grades can complete the additional following requirements. There are no extra or bonus remarks for students who complete the assignment in “Nightmare mode”. The experience gained is its own reward. This is only recommended for students who have completed the other parts of the assignment.

Students who complete all assignments in Nightmare mode, may get a small token of recognition at the end of the course.

```
typedef struct rotations
{
  molecule *x[72];
  molecule *y[72];
  molecule *z[72];
} rotations;
```

The structure **`rotations`** represents the rotations of a given **`molecule`** around the *x*, *y* and *z* axes in 5 degree increments. I.e. element **`x[3]`** of the structure is a pointer to a **`molecule`** which is equivalent the given **`molecule`** rotated by 15 degrees around the x axis.

```
rotations *spin( molecule *mol );
```

This function will allocate memory for a rotations structure, create molecules using molcopy applied to the provided mol, and add their pointers to the x, y, and z members of the rotations structure. Each molecule will be rotated by an angle equal to 5 times the array index, and it will be sorted.

```
void rotationsfree( rotations *rotations );
```

This function will free the memory associated with a rotations structure. This includes freeing each of the 216 molecules included within the structure in addition to the structure itself.

Git

You must submit your **`.c`**, **`.h`** and **`makefile`** using **`git`** to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A1 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, nor allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

atomset	3
atomget	3
bondset	3
bondget	3
molmalloc	3
molcopy	3
molfree	3
molappend_atom	3
molappend_bond	3
molsort	3
xrotation	3
yrotation	3
zrotation	3
mol_xform	3
style	4
makefile	4
Total	50