Some Topics on the C Language (part II)

---

1. Recall the definition of the complex number $z \in \mathbb{C}$ as $x + yi$, where $x, y \in \mathbb{R}$. The values x and y represent, respectively, the real and imaginary parts of z.

The following C header file (a file with extension `.h`) defines a new datatype called `complex` that can be used to implement a library of functions that operate on complex numbers. The list of such functions and their types (the library's Application Programmer's Interface or API) is also provided in this file (`complex.h`):

```
/* definition of new type complex */

typedef struct {
  double x;
  double y;
} complex;

/* definition of the complex library API */

complex* complex_new(double, double);
complex* complex_add(complex *, complex *);
complex* complex_sub(complex *, complex *);
complex* complex_mul(complex *, complex *);
complex* complex_div(complex *, complex *);
complex* complex_conj(complex *);
double   complex_mod(complex *);
double   complex_arg(complex *);
double   complex_re(complex *);
double   complex_im(complex *);
```

Consider also the file `use_complex.c` that makes use of the above API to create complex numbers and to manipulate them.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include "complex.h"

int main(int argc, char** argv) {

  complex* z1 = complex_new(-2.16793, 5.23394);
  complex* z2 = complex_new( 1.12227, 2.52236);

  complex* z3 = complex_add(z1, z2);
  complex* z4 = complex_sub(z1, z2);
  complex* z5 = complex_mul(z1, z2);
  complex* z6 = complex_div(z1, z2);

  double  x1 = complex_mod(z1);
  double  x2 = complex_re(z1);
  double  x3 = complex_im(z3);

  printf("z1 = %f + %fi\n", z1->x, z1->y);
  printf("z2 = %f + %fi\n", z2->x, z2->y);
  printf("z3 = %f + %fi\n", z3->x, z3->y);
  printf("z4 = %f + %fi\n", z4->x, z4->y);
  printf("z5 = %f + %fi\n", z5->x, z5->y);
  printf("z6 = %f + %fi\n", z6->x, z6->y);
  printf("x1 = %f\n", x1);
  printf("x2 = %f\n", x2);
  printf("x3 = %f\n", x3);

  return 0;
}
```

Finally, the file `complex.c` provides the implementation of the API, i.e., the implementation of all functions listed in `complex.h`:

```
#include <stdlib.h>
#include <math.h>

#include "complex.h"


/*
 * implementation of the Complex API
 */
```

```
complex* complex_new(double x, double y) {
  complex* z = (complex*) malloc(sizeof(complex));
  z->x = x;
  z->y = y;
  return z;
}

complex* complex_add(complex* z, complex* w){
  return complex_new(z->x + w->x, z->y + w->y);
}

complex* complex_sub(complex* z, complex* w){
  /* to complete ... */
}

complex* complex_mul(complex* z, complex* w){
  return complex_new(z->x * w->x - z->y * w->y,
                     z->x * w->y + z->y * w->x);
}

complex* complex_div(complex* z, complex* w){
  /* to complete ... */
}

complex* complex_conj(complex* z){
  /* to complete ... */
}

double   complex_mod(complex* z){
  /* to complete ... */
}

double   complex_arg(complex* z){
  return atan2(z->y,z->x);
}

double   complex_re(complex* z){
  return z->x;
}

double   complex_im(complex* z){
  /* to complete ... */
```

```
}
```

To run the example, we first compile the API and build a library as an *archive* (extension `.a`) as `libcomplex.a` that will be used by the main program:

```
$ gcc -Wall -c complex.c
$ ar -rc libcomplex.a complex.o
$ ar -t  libcomplex.a
```

finally, we compile the main program `use_complex.c` informing the compiler (actually the linker) that it should use code from the library `libcomplex.a` (`-lcomplex`) located in the current directory (`-L.`):

```
$ gcc -Wall use_complex.c -o use_complex -L. -lcomplex -lm
```

Note also that C's math library was also included `-lm`, as function in it such as `atan2` and `sqrt`, are used in the implementation of `complex.c`.

2. Repeat the above exercise but now building and using a dynamic library, by running the following commands:

```
$ gcc -c -Wall -fPIC -o complex.o complex.c
$ gcc -shared -o libcomplex.so complex.o
```

Option `-fPIC` informs the compiler that it should generate position independent code. This is important because the dynamic library will be loaded into memory when the program is already running (hence the dynamic adjective) in addresses that are not known a priori by the compiler. Option `-shared` indicates to the compiler that the resulting library should be created as a *shared object* (extension `.so`), as `libcomplex.so`. After being created, the library is used in much the same way as its static version to compile the main program:

```
$ gcc -Wall use_complex.c -o use_complex -L. -lcomplex
$ ./use_complex
```

Depending on the operating system you are using, you may also need to run the command:

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

so that the library may be found by the operating system.

3. Consider the header file `vector.h` as follows, containing the definition of a type `vector`, that represents a 3D vector $\in \mathbb{R}^3$:

```
/* definition of new type vector */

typedef struct {
  double x;
  double y;
  double z;
} vector;

/* definition of the vector API */

vector* vector_new(double, double, double);
vector* vector_add(vector*, vector*);
vector* vector_sub(vector*, vector*);
vector* vector_scale(double, vector*);
vector* vector_vprod(vector*, vector*);
double  vector_sprod(vector*, vector*);
double  vector_mod(vector*);
```

As in the previous exercise, consider a file `use_vector.c` that uses the “vector” API.

```
#include <stdio.h>
#include <stdlib.h>

#include "vector.h"

int main(int argc, char** argv) {
  vector* v1 = vector_new(-5.1, 2.3, 3.6);
  vector* v2 = vector_new( 1.6, 7.6, -4.2);

  vector* v3 = vector_add(v1, v2);
  vector* v4 = vector_sub(v1, v2);
  vector* v5 = vector_scale(-9.2, v2);
  vector* v6 = vector_vprod(v1,v2);
  double  x1 = vector_sprod(v1, v2);
  double  x2 = vector_mod(v6);

  printf("v1 = (%f, %f, %f)\n", v1->x, v1->y, v1->z);
  printf("v2 = (%f, %f, %f)\n", v2->x, v2->y, v2->z);
  printf("v3 = (%f, %f, %f)\n", v3->x, v3->y, v3->z);
  printf("v4 = (%f, %f, %f)\n", v4->x, v4->y, v4->z);
```

```
  printf("v5 = (%f, %f, %f)\n", v5->x, v5->y, v5->z);
  printf("v6 = (%f, %f, %f)\n", v6->x, v6->y, v6->z);
  printf("x1 = %f\n", x1);
  printf("x2 = %f\n", x2);

  return 0;
}
```

Write an implementation for the API in a file `vector.c`, compile it and build a library `libvector.a`. Compile the program `use_vector.c` with the library and run it.

4. Consider the file `matrix.h` that contains the definition of type `matrix`, representing a $N \times M$ matrix of floating point values.

```
/* definition of new type matrix */

typedef struct {
  int n;
  int m;
  double* vals;
} matrix;

/* definition of the matrix API */

matrix* matrix_new(int, int);
matrix* matrix_new_random(int, int, double, double);
void    matrix_print(matrix*);
double  matrix_get(int, int, matrix*);
void    matrix_set(int, int, double, matrix*);
matrix* matrix_add(matrix *, matrix *);
matrix* matrix_sub(matrix *, matrix *);
matrix* matrix_mul(matrix *, matrix *);
matrix* matrix_trans(matrix *);
```

Consider the file `matrix.c` that contains a partial implementation for the API. Complete it.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "matrix.h"
```

```
/* implementation of the matrix API */

matrix* matrix_new(int n, int m) {
  matrix* u = (matrix*) malloc(sizeof(matrix));
  u->n = n;
  u->m = m;
  u->vals = (double*) malloc ((u->n * u->m) * sizeof(double));
  return u;
}

matrix* matrix_new_random(int n, int m, double min, double max) {
  matrix* u = (matrix*) malloc(sizeof(matrix));
  u->n = n;
  u->m = m;
  u->vals = (double*) malloc ((u->n * u->m) * sizeof(double));

  int i, j;
  double range = max - min;
  double div   = RAND_MAX / range;
  for(i = 0; i < u->n; i++)
    for(j = 0; j < u->m; j++)
      matrix_set(i, j, min + (rand() / div), u);
  return u;
}

void matrix_print(matrix* u) {
  /* to complete ... */
}

double matrix_get(int i, int j, matrix* u){
  return *(u->vals + i * u->m + j);
}

void matrix_set(int i, int j, double val, matrix* u){
  /* to complete ... */
}

matrix* matrix_add(matrix* u, matrix* v){
  int i, j;
  matrix* w = matrix_new(u->n, u->m);
  for (i = 0; i < u->n; i++ )
    for (j = 0; j < u->m; j++ )
```

```
      matrix_set(i, j, matrix_get(i, j, u) + matrix_get(i, j, v), w);
  return w;
}

matrix* matrix_sub(matrix* u, matrix* v){
  /* to complete ... */
}

matrix* matrix_mul(matrix* u, matrix* v){
  /* to complete ... */
}

matrix* matrix_trans(matrix* u){
  /* to complete ... */
}
```

Write a file `use_matrix.c` with a `main()` function where you create some matrices and use the API to perform operations on them.

5. Consider the file `list.h` that contains a definition of a type `list`, representing a linked list of integers.

```
/* definition of new type list */

typedef struct anode {
  int val;
  struct anode* next;
} node;

typedef struct  {
  int size;
  node* first;
} list;

/* definition of the list API */

node* node_new(int, node*);
list* list_new();
list* list_new_random(int, int);
void  list_add_first(int, list *);
void  list_add_last(int, list *);
int   list_get_first(list *);
int   list_get_last(list *);
```

```
void  list_remove_first(list *);
void  list_remove_last(list *);
int   list_size(list *);
void  list_print(list *);
```

Consider the file `list.c` that contains a partial implementation of the aforementioned API. Complete it.

```
#include <stdio.h>
#include <stdlib.h>

#include "list.h"


/* implementation of the List API */

node* node_new(int val, node* p) {
  node* q = (node*)malloc(sizeof(node));
  q->val = val;
  q->next = p;
  return q;
}

list* list_new() {
  list* l = (list*) malloc(sizeof(list));
  l->size = 0;
  l->first = NULL;
  return l;
}

list* list_new_random(int size, int range) {
  list* l = list_new();
  int i;
  for(i = 0; i < size; i++)
    list_add_first(rand() % range, l);
  return l;
}

void  list_add_first(int val, list *l) {
  /* to complete ... */
}

void  list_add_last(int val, list *l) {
```

```
  node* p = node_new(val, NULL);
  if (l->size == 0) {
    l->first = p;
  }else{
    node* q = l->first;
    while (q->next != NULL)
      q = q->next;
    q->next = p;
  }
  l->size++;
}

int   list_get_first(list *l) {
  /* assumes list l is not empty */
  return l->first->val;
}

int  list_get_last(list *l) {
  /* to complete ... */
}

void  list_remove_first(list *l) {
  /* assumes list l is not empty */
  node* p = l->first;
  l->first = l->first->next;
  l->size--;
  /* free memory allocated for node p */
  free(p);
}

void  list_remove_last(list *l) {
  /* to complete ... */
}

int   list_size(list *l) {
  /* to complete ... */
}

void list_print(list* l) {
  /* to complete ... */
}
```

Write a file `use_list.c` that creates one or more lists and that uses the functions of the

API to manipulate them.

6. The code that follows presents an alternative implementation of exercise **1** for a library that operates on complex numbers (file `complex.c`):

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "complex.h"

char complex_buf[100];

complex complex_new(double x, double y) {
  complex z;
  z.x = x;
  z.y = y;
  return z;
}

char*  complex_print(complex z) {
  if (z.y == 0)
    sprintf(complex_buf, "%f", z.x);
  else if (z.x == 0)
    sprintf(complex_buf, "%fi", z.y);
  else if (z.y > 0)
    sprintf(complex_buf, "%f+%fi", z.x, z.y);
  else
    sprintf(complex_buf, "%f%fi", z.x, z.y);
  return complex_buf;
}


complex complex_add(complex z, complex w){
  complex r;
  r.x = z.x + w.x;
  r.y = z.y + w.y;
  return r;
}

complex complex_sub(complex z, complex w){
  complex r;
  r.x = z.x - w.x;
  r.y = z.y - w.y;
```

```
  return r;
}

complex complex_mul(complex z, complex w){
  complex r;
  r.x = z.x * w.x - z.y * w.y;
  r.y = z.x * w.y + z.y * w.x;
  return r;
}

complex complex_div(complex z, complex w){
  complex r;
  double d = w.x * w.x + w.y * w.y;
  r.x = (z.x * w.x + z.y * w.y) / d;
  r.y = (- z.x * w.y + z.y * w.x) / d;
  return r;
}

complex complex_conj(complex z){
  complex r;
  r.x = z.x;
  r.y = -z.y;
  return r;
}

double  complex_mod(complex z){
  return sqrt(z.x * z.x + z.y * z.y);
}

double  complex_arg(complex z){
  return atan2(z.y, z.x);
}

double  complex_re(complex z){
  return z.x;
}

double  complex_im(complex z){
  return z.y;
}
```

This new API is used in file `use_complex.c`:

```
#include <stdio.h>
```

```
#include "complex.h"

int main(int argc, char** argv) {
  complex z1 = complex_new(-2.16793, 5.23394);
  complex z2 = complex_new( 1.12227, 2.52236);

  complex z3 = complex_add(z1, z2);
  complex z4 = complex_sub(z1, z2);
  complex z5 = complex_mul(z1, z2);
  complex z6 = complex_div(z1, z2);

  double  x1 = complex_mod(z1);
  double  x2 = complex_re(z1);
  double  x3 = complex_im(z3);

  printf("z1 = %s\n", complex_print(z1));
  printf("z2 = %s\n", complex_print(z2));
  printf("z3 = %s\n", complex_print(z3));
  printf("z4 = %s\n", complex_print(z4));
  printf("z5 = %s\n", complex_print(z5));
  printf("z6 = %s\n", complex_print(z6));
  printf("x1 = %f\n", x1);
  printf("x2 = %f\n", x2);
  printf("x3 = %f\n", x3);

  return 0;
}
```

Based on the code here presented, write the corresponding header file `complex.h`, compile the library and main program and check that you get similar results to those of exercise **1**. Pay close attention to the code and make a diagram (draw it!) that shows how the memory is used in the *heap* and *stack* during the execution of both APIs. What is the main different between the two?