

## Alguns Tópicos de Linguagem C (parte II)

---

1. Recorde a definição de número complexo  $z \in \mathbb{C}$ . Um número complexo  $z$  tem a forma  $a + bi$ , onde  $a, b \in \mathbb{R}$ . Os valores  $a$  e  $b$  representam, respectivamente, as partes real e imaginária de  $z$ .

O seguinte cabeçalho em C (ficheiro com extensão `.h`) define um novo tipo `complex` que pode ser usado para implementar uma biblioteca de funções que realizam as operações sobre complexos de forma simplificada. A lista completa dos tipos destas funções (a API) é também incluída neste ficheiro (`complex.h`):

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

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

/* definition of the complex 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 *);
```

Considere ainda o ficheiro `use_complex.c` que faz uso da API acima definida para criar números complexos e manipulá-los.

```

#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;
}

```

Finalmente, implemente cada uma das funções da API num ficheiro à parte: `complex_new.c`, `complex_add.c`, ..., `complex_re.c`, `complex_im.c`, por exemplo:

```

#include <stdlib.h>
#include "complex.h"

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

#include "complex.h"
complex* complex_add(complex* z, complex* w){
    return complex_new(z->x + w->x, z->y + w->y);
}

```

```

#include "complex.h"
complex* complex_sub(complex* z, complex* w){
    /* to complete ... */
}

#include "complex.h"
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);
}

#include "complex.h"
complex* complex_div(complex* z, complex* w){
    /* to complete ... */
}

#include "complex.h"
complex* complex_conj(complex* z){
    /* to complete ... */
}

#include <math.h>
#include "complex.h"
double complex_mod(complex* z){
    return sqrt( z->x * z->x + z->y * z->y );
}

#include <math.h>
#include "complex.h"
double complex_arg(complex* z){
    return atan2(z->y,z->x);
}

#include "complex.h"
double complex_re(complex* z){
    return z->x;
}

#include "complex.h"
double complex_im(complex* z){
    /* to complete ... */
}

```

Para executar o exemplo, compilamos primeiro a API e construímos uma biblioteca `libcomplex.a` que será usada pelo programa principal:

```
$ gcc -Wall -c complex_*.c
$ ar -rc libcomplex.a complex_*.o
$ ar -t libcomplex.a      // usar o comando "ar" para ver o conteúdo
$ nm libcomplex.a         // o comando "nm" também permite fazê-lo
```

e compilamos depois o programa principal `use_complex.c` indicando ao compilador (linker) que deve usar a biblioteca `libcomplex.a` (`-lcomplex`) situada no directório corrente (`-L.`):

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

Note que também foi incluída a biblioteca matemática `-lm`, necessária para funções como `atan2` e `sqrt`, usadas em `complex.c`.

**2.** Repita o exercício dos números complexos mas agora criando uma biblioteca dinâmica, executando os seguintes comandos:

```
$ gcc -c -Wall -fPIC complex_*.c
$ gcc -shared -o libcomplex.so complex_*.o
$ nm libcomplex.so
```

A opção `-fPIC` indica ao compilador que deve gerar código binário que possa ser colocado em qualquer posição na memória, e.g., as instruções de salto são feitas sempre usando endereços relativos. A opção `-shared` indica ao compilador que a biblioteca resultante vai ser um *shared object*, com extensão `.so`. Depois de criada a biblioteca, esta é usada da mesma forma que uma biblioteca estática:

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

Dependendo do sistema operativo que estiver a usar poderá ter também de executar o comando:

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

Para a biblioteca ser encontrada.

**3.** Considere o seguinte ficheiro `vector.h` que contém a definição de um tipo `vector`, representando um vector em  $\mathbb{R}^3$ :

```
/* 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*);

```

Considere ainda o ficheiro `use_vector.c` que faz uso da API acima definida para criar e manipular vectores.

```

#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;
}

```

Na mesma linha do exercício anterior (uma função por ficheiro), escreva uma implementação para a API apresentada. Compile as funções e crie bibliotecas `libvector.a` e `libvector.so`. Compile o programa `use_vector.c` com as bibliotecas e execute-o.

4. Considere o ficheiro `list.h` contendo a definição do tipo `list`, uma lista ligada de inteiros.

```
/* 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 *);
```

Considere ainda a seguinte implementação parcial da API. Complete-a colocando uma função por ficheiro. Compile depois as funções e crie bibliotecas `liblist.a` e `liblist.so`.

```
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 ... */
}

```

Finalmente, escreva um ficheiro `use_list.c` que crie uma ou mais listas e as manipule utilizando todas as funções da API.

**5.** Neste exercício não reescreva ficheiros do exercício **1**. Crie um directório à parte e resolva o exercício aí.

O código que se segue apresenta uma implementação alternativa ao exercício **1** para uma biblioteca que opera sobre números complexos:

```

#include "complex.h"
complex complex_new(double x, double y) {
    complex z;
    z.x = x;
    z.y = y;
    return z;
}

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

#include "complex.h"
complex complex_sub(complex z, complex w){
    complex r;

```



```

    r.x = z.x - w.x;
    r.y = z.y - w.y;
    return r;
}

#include "complex.h"
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;
}

#include "complex.h"
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;
}

#include "complex.h"
complex complex_conj(complex z){
    complex r;
    r.x = z.x;
    r.y = -z.y;
    return r;
}

#include <math.h>
#include "complex.h"
double complex_mod(complex z){
    return sqrt(z.x * z.x + z.y * z.y);
}

#include <math.h>
#include "complex.h"
double complex_arg(complex z){
    return atan2(z.y, z.x);
}

#include "complex.h"
double complex_re(complex z){
    return z.x;
}

```

```
#include "complex.h"
double complex_im(complex z){
    return z.y;
}
```

Note que as funções da API recebem agora argumentos do tipo `complex` e não `complex*`. A API é usada no seguinte exemplo (ficheiro `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 = %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;
}
```

Com base no código aqui apresentado, escreva também o ficheiro `complex.h` (nota: é parecido com o `complex.h` do exercício 1), compile a biblioteca e o exemplo e verifique que obtém resultados iguais aos da implementação apresentada no exercício 1.

Qual a diferença fundamental entre as duas APIs?