Ficha 2

Programação (L.EIC009)

Objectivos

- Introdução a programação em C (continuação).
  - Arrays.
  - Strings.
  - Structs.

Recursos

Slides das aulas teóricas: HTML PDF

1

1.1

Use o esqueleto abaixo para codificar uma função `invert` e um pequeno programa de teste para a mesma. Uma chamada `invert(n, a, b)` deve copiar `n` elementos do array `a` para o array `b` mas na ordem inversa de indexação (ex. após a chamada `b` deve conter `{ 3,2,1 }` se `a` contém `{1,2,3}`).

Nota: o modificador `const` impede escritas usando `a` no corpo de `invert`.

```c
#include <stdio.h>

void invert(int n, const int a[], int b[]) {
  // ...
}

void print(int n, const int a[]) {
  for (int i = 0; i < n; i++) {
    printf("%d: %d\n", i, a[i]);
  }
}

int main(void) {
  int a[7] = { 0, 1, 2, 3, 4, 5, 6};
```

```c
  int b[7];
  invert(7, a, b);
  print(7, a);
  print(7, b);
  return 0;
}
```

Output esperado do programa:

```
0: 0
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
0: 6
1: 5
2: 4
3: 3
4: 2
5: 1
6: 0
```

1.2

Considere uma variante da função anterior em que a inversão operada o array de entrada, isto é, `invert(n, a)` deve inverter a ordem dos `n` elementos de `a`.

Que cuidado deve ter para não "perder" valores? Quantas iterações são necessárias no "ciclo de inversão"?

```c
#include <stdio.h>

void invert(int n, int a[]) {
  // ...
}

void print(int n, int a[]) {
  for (int i = 0; i < n; i++) {
    printf("%d: %d\n", i, a[i]);
  }
}

int main(void) {
```

```
  int a[7] = { 0, 1, 2, 3, 4, 5, 6};
  print(7, a);
  invert(7, a);
  print(7, a);
  return 0;
}
```

(o output esperado é o mesmo que no caso anterior)

2

Strings são arrays de tipo `char` terminadas com o valor 0, também denotado por `'\0'`. Quando usamos uma string constante, i.e. uma sequência de caracteres entre aspas, está implícita o valor `'\0'` no fim da sequência.

```
// Declarações equivalentes
char a[4] = { 'a', 'b', 'c', '\0' };
char b[4] = "abc";
```

2.1

Implemente as seguintes funções que operam sobre strings **sem usar** funções da biblioteca de C.

- `length(const char s[])` : devolve o número de caracteres em `s` (análogo à funcionalidade da função `strlen` do "header" `string.h` da biblioteca de C).
- `copy(char dst[], const char[] src)` : copia o conteúdo da string `src` para `dst` (análogo a `strcpy`). Não se esqueça de colocar o caracter terminador em `dst`.
- `concat(char dst[], const char src[])` concatena o conteúdo da string `src` no final de `dst` (análogo a `strcat`). Dará jeito usar `length` ? Não se esqueça de colocar o caracter terminador em `dst`.

Pode usar o seguinte esqueleto e programa de teste:

```
#include <stdio.h>

int length(const char s[]) {
   // ...
}
```

```c
void copy(char dst[], const char src[]) {
   // ...
}
void concat(char dst[], const char src[]) {
   // ...
}

int main(void) {
  char a[20];

  copy(a, "Hello"); puts(a); printf("%d\n", length(a));

  // Nota: "" é a string vazia (tem apenas '\0' no início)
  concat(a, ""); puts(a); printf("%d\n", length(a));
  concat(a, " world");  puts(a); printf("%d\n", length(a));
  concat(a, "!"); puts(a); printf("%d\n", length(a));

  return 0;
}
```

Output esperado do programa:

```
Hello
5
Hello
5
Hello world
11
Hello world!
12
```

2.2

Consegue usar aritmética de apontadores para as funções acima? A título de exemplo considere a seguinte implementação para `length`:

```c
int length(const char s[]) {
  const char* p = &s[0];  // ou simplesmente p = s
  while (*p != '\0') {
    p++; // avança para próxima posição
  }
  return p - &s[0]; // ou p - s
}
```

Programe variantes para `copy` e `concat` que empreguem também (apenas) aritmética de apontadores (sem usar variáveis inteiras para indexação).

3

Usando o esqueleto abaixo, escreve uma função `hexstr_convert` que converta uma string contendo dígitos hexadecimais num valor de tipo `int` correspondente.

Assuma que a string de entrada `h` pode conter os caracteres `'0'` a `'9'`, `'A'` a `'F'` (correspondentes aos valores 10 a 15), ou ainda `'a'` a `'f'` (também correspondentes aos valores 10 a 15, mas usando caracteres de letras minúsculas).

```c
#include <stdio.h>

int hexstr_convert(const char h[]) {
  int r = 0;
  // ...
  return r;
}

void test(const char h[]) {
  int v = hexstr_convert(h);
  printf("\"%s\" -> %d (%X)\n", h, v, v);
}

int main(void) {
  test("A");
  test("0a");
  test("a0");
  test("18");
  test("1f");
  test("fF");
  test("ffff");
  test("AbCdEf");
  test("7fffffff");
  return 0;
}
```

Output esperado do programa acima:

```
"A" -> 10 (A)
"0a" -> 10 (A)
"a0" -> 160 (A0)
"18" -> 24 (18)
```

```
"1f" -> 31 (1F)
"fF" -> 255 (FF)
"ffff" -> 65535 (FFFF)
"AbCdEf" -> 11259375 (ABCDEF)
"7fffffff" -> 2147483647 (7FFFFFFF)
```

4

4.1

O seguinte programa ilustra uma declaração alternativa para a função `main`: `argc` representa o número de argumentos passados de fora incluindo o caminho do programa, e `argv` é um array de strings com `argc` entradas.

```c
#include <stdio.h>

int main(int argc,  char *argv[]) {
  for (int i = 0; i < argc; i++) {
    printf("Argument %d: %s\n", i, argv[i]);
  }
  return 0;
}
```

Compile e execute o programa, por ex.:

```
$ gcc e4_1.c -Wall -o e4_1
$ ./e4_1
Argument 0: ./e4_1
$ ./e4_1 a b c d e
Argument 0: ./e4_1
Argument 1: a
Argument 2: b
Argument 3: c
Argument 4: d
Argument 5: e
```

4.2

Escreva um programa que tome como argumentos (via `argc` / `argv`) uma sequência de strings codificando números de vírgula flutuante e escreva como output a soma de tipo `double` desses números. Pode usar a função `atof` declarada em `stdlib.h` para a conversão de strings em valores `double`.

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

int main(int argc,  char *argv[]) {
  double sum = 0;
  // ...
  printf("%f\n", sum);
  return 0;
}
```

Exemplo de execução:

```
$ gcc -Wall e4_2.c -o e4_2
$ ./e4_2 -1.23 1.24 -0.02 1e+02
99.990000
```

5

O seguinte fragmento define uma estrutura de dados (`struct`) para coordenadas num espaço bi-dimensional XY chamada `_coord2d` . Como pode observar a estrutura tem dois campos `x` e `y` de tipo `double` .

De seguida é definido um tipo `coord2d` que nos permite referir de forma mais sucinta ao tipo da estrutura, i.e., podemos usar `coord2d` em vez de `struct _coord2d` .

```c
struct _coord2d {
  double x;
  double y;
};
typedef struct _coord2d coord2d;
```

Em alternativa podemos também definir `coord2d` como:

```c
typedef struct {
  double x;
  double y;
} coord2d;
```

Tenha em conta as seguintes operações sobre coordenadas:

Operação	Descrição	Função a implementar
Soma	$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$	`coord2d_add`
Subtração	$(x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2)$	`coord2d_sub`
Multiplicação por escalar	$k \times (x, y) = (k \times x, k \times y)$	`coord2d_mul`

Complete o seguinte programa que lê 2 coordenadas `a` e `b` a partir de 4 valores dados como argumentos ao programa, ou então assume as coordenadas `a=(3.5,5.0)` e `b=(-1.75, -2.5)` por omissão e imprime a coordenada correspondente a `0.5 a + b - (1,2)`.

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

struct _coord2d {
  double x;
  double y;
};
typedef struct _coord2d coord2d;

void coord2d_print(const coord2d* c) {
  printf("(%f,%f)\n", c->x, c->y);
}

void coord2d_add(coord2d* r, const coord2d* c1, const coord2d* c2) {
  // ...
}

void coord2d_sub(coord2d* r, const coord2d* c1, const coord2d* c2) {
  // ...
}

void coord2d_mul(coord2d* r, double k, const coord2d* c) {
  // ...
}

int main(int argc, char* argv[]) {
  // Initialization with default values
  coord2d a = { 3.5, 5 };
  coord2d b = { .x = -1.75, .y = -2.5 };

  if (argc == 5) {
```

```
    a.x = atof(argv[1]);
    a.y = atof(argv[2]);
    b.x = atof(argv[3]);
    b.y = atof(argv[4]);
  }

  coord2d r;
  coord2d_mul(&r, 0.5, &a);
  coord2d_add(&r, &r, &b);

  coord2d c = { 1, 2 };
  coord2d_sub(&r, &r, &c);
  coord2d_print(&r);
  return 0;
}
```

Exemplos de execução

(sem argumentos)

```
$ ./ex5
(-1.000000,-2.000000)
```

(com argumentos)

```
$ ./ex5 0 0 0 0
(-1.000000,-2.000000)
```

```
$ ./ex5 2 4 0 0
(0.000000,0.000000)
```

```
$ ./ex5 1 1 0 0
(-0.500000,-1.500000)
```

```
$ ./ex5 0 0 1 1
(0.000000,-1.000000)
```

```
$ ./ex5 1.25 -1.25 -1.25 1.25
(-1.625000,-1.375000)
```

6

Considere a seguinte estrutura de dados `stack` para suportar um TAD (tipo abstracto de dados) para uma "stack" (pilha) de valores inteiros com capacidade limitada a um número máximo de elementos (`MAX_ELEMENTS` , no caso com valor 5):

```
#define MAX_ELEMENTS 5

typedef struct {
  int elements[MAX_ELEMENTS]
  int size;
} stack;
```

6.1

Pretende-se que um item do tipo `stack` represente uma stack (pilha) onde: os elementos colocados na stack estão nas posições `0` a `size-1` do array `elements` , sendo o topo da stack dado pelo conteúdo de `elements[size-1]` .

Por exemplo o seguinte programa imprime uma stack com elementos 0, 1, e 2 (do fundo para o topo da tack; 2 é o valor no topo da stack):

```
#include <stdio.h>

#define MAX_ELEMENTS 5

typedef struct {
  int elements[MAX_ELEMENTS];
  int size;
} stack;

void stack_print(const stack* s) {
  printf("[");
  for (int i = 0; i < s->size; i++) {
    printf(" %d", s->elements[i]);
  }
  printf(" ] (%d elements)\n", s->size);
}

int main(void) {
  stack s = {
    .elements = { 0, 1, 2},
    .size = 3
  };
  stack_print(&s);
  return 0;
```

```
}
```

Compile e execute o programa. Deverá obter como output:

```
[ 0 1 2 ] (3 elements)
```

6.2

Para que a estrutura de dados acima seja realmente útil, precisamos de definir convenientemente funções em linha com a usual disciplina LIFO ("Last-In, First-Out") de uma stack.

Considere as seguintes funções a implementar:

Função	Descrição
`void stack_init(stack* s)`	Inicializa uma stack vazia `s`.
`int stack_size(const stack* s)`	Devolve o número de elementos guardados em `s`.
`bool stack_is_empty(const stack* s)`	Indica se `s` está vazia.
`bool stack_is_full(const stack* s)`	Indica se `s` está cheia.
`bool stack_push(stack* s, int v)`	Adiciona valor `v` ao topo da stack `s` caso haja espaço e retorna `true` nesse caso. Se não houver espaço, `s` não é modificada e a função deve devolver `false`.
`bool stack_pop(stack* s, int* v)`	Caso `s` não esteja vazia, retira valor do topo da stack `s`, devolve-o em `v`, e finalmente retorna `true`. Se `s` estiver vazia, `s` não deverá ser modificada e a função deve devolver `false`.

Use o seguinte esqueleto para implementar e testar as funções:

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

#define MAX_ELEMENTS 5

typedef struct {
  int elements[MAX_ELEMENTS];
  int size;
} stack;

void stack_init(stack* s) {
  s -> size = 0;
}

// A IMPLEMENTAR --->
int stack_size(const stack* s) {
  // ...
}

bool stack_is_empty(const stack* s) {
  // ...
}

bool stack_is_full(const stack* s) {
  // ...
}

bool stack_push(stack* s, int v) {
  // ...
}

bool stack_pop(stack* s, int* v) {
  // ...
}
// <--- A IMPLEMENTAR

// CÓDIGO / PROGRAMA DE TESTE -->

void stack_print(const stack* s) {
  printf("[");
  for (int i = 0; i < s->size; i++) {
    printf(" %d", s->elements[i]);
  }
  printf(" ] (%d elements; empty: %d; full: %d)\n",
    stack_size(s),
```

```
    stack_is_empty(s),
    stack_is_full(s));
}

int main(void) {
  stack s;
  stack_init(&s);
  stack_print(&s);
  for (int i = 0; i <= MAX_ELEMENTS; i++) {
      bool r = stack_push(&s, i * 100);
      if (r) {
        printf("value pushed\n");
      } else {
        printf("value not pushed - stack is full!\n");
      }
      stack_print(&s);
  }

  for (int i = 0; i <= MAX_ELEMENTS; i++) {
      int v;
      bool r = stack_pop(&s, &v);
      if (r) {
        printf("value popped: %d\n", v);
      } else {
        printf("value not popped - stack is empty!\n");
      }
      stack_print(&s);
  }
  return 0;
}
```

Output esperado do programa (em fragmentos):

- Stack inicialmente vazia, depois de chamada a `stack_init()`:

```
[ ] (0 elements; empty: 1; full: 0)
```

- Várias chamadas a `stack_push` até que stack fica cheia (última chamada a `stack_push()` já não tem sucesso):

```
value pushed
[ 0 ] (1 elements; empty: 0; full: 0)
value pushed
[ 0 100 ] (2 elements; empty: 0; full: 0)
value pushed
[ 0 100 200 ] (3 elements; empty: 0; full: 0)
```

```
value pushed
[ 0 100 200 300 ] (4 elements; empty: 0; full: 0)
value pushed
[ 0 100 200 300 400 ] (5 elements; empty: 0; full: 1)
value not pushed - stack is full!
[ 0 100 200 300 400 ] (5 elements; empty: 0; full: 1)
```

- Várias chamadas a `stack_pop()` até que stack fica vazia (última chamada a `stack_pop()` já não tem sucesso):

```
value popped: 400
[ 0 100 200 300 ] (4 elements; empty: 0; full: 0)
value popped: 300
[ 0 100 200 ] (3 elements; empty: 0; full: 0)
value popped: 200
[ 0 100 ] (2 elements; empty: 0; full: 0)
value popped: 100
[ 0 ] (1 elements; empty: 0; full: 0)
value popped: 0
[ ] (0 elements; empty: 1; full: 0)
value not popped - stack is empty!
[ ] (0 elements; empty: 1; full: 0)
```