Instituto Superior de Engenharia de Lisboa

Programação II

Ano letivo 2022/23

Teste de época especial

2023.09.09

Nota importante: Valoriza-se a escrita de código que inclua comentários esclarecedores da implementação seguida e que contenha indentação legível.

1. [5 valores]

Considere o troço de código constante da caixa seguinte. Na linha 1 daquele troço de código deverá inserir o algarismo das unidades do seu número de aluno no local indicado. Por exemplo: se o seu número de aluno fosse o 12345, aquela linha ficaria assim: `#define ALG 5`
Nota importante: a utilização do algarismo errado penalizará significativamente a avaliação de algumas alíneas deste grupo.

```
1  #define ALG algarismo_das_unidades_do_seu_número_de_aluno
2  #include <stdio.h>
3  #include <stdlib.h>
4  int * f1( int * a, size_t size, size_t i ) {
5    for(  ; i < size - 1 ; i++)
6      a[i] = a[i+1];
7    return realloc(a, i * sizeof(int));
8  }
9  int * f2( int * a, size_t *size, int (*cond)(const int *) ) {
10   size_t k;
11   for(k=0; k < *size; )
12     if(cond(&a[k]))
13       a = f1(a, (*size)--, k);
14     else
15       k++;
16   return a;
17 }
18 int f3(const int *e) {
19   return *e == ALG;
20 }
```

No troço de programa seguinte consta um exemplo de utilização da função `f1`.

```
...  int *a1, *a2, size, n;
...  (…) // Preenchimento do array dinâmico a1.
30
31   a2 = f1( a1, size, n );
32
33
```

a) [1] Apresente, correta e completamente preenchido, o cabeçalho descritivo da função `f1` seguindo o formato do cabeçalho que se apresenta na caixa seguinte, que está preenchido (como exemplo) para a função `memmove`.

```
/*--------------------------------------------------------------------------------

Nome da função: memmove


Descrição: Esta função copia n bytes da memória apontada por src para a memória apontada
por dest. As áreas de memória podem sobrepor-se: a cópia ocorre como se os bytes a copiar
fossem primeiro copiados para um array temporário, não sobreposto às áreas de memória
apontadas por src e dest, e fossem depois copiados para a memória apontada por dest.


Parâmetros:

void *dest: referência para a memória para onde é realizada a cópia.

const void *src: referência para a memória de onde é realizada a cópia.

size_t n: número de bytes a copiar.


Retorno:

void: não tem tipo de retorno.

--------------------------------------------------------------------------------*/
```

b) [2] Admita que imediatamente antes da execução da linha 31 o *array* referenciado por **a1** e as variáveis **size** e **n** contêm os valores a seguir representados:

Array **a1:**

14	6	7	1	11	8

size: 6 **n:** 2

Represente, justificando com clareza, o *array* referenciado por **a2** imediatamente após a execução da linha 31.

Admita agora que, noutra execução do programa, imediatamente antes da execução da linha 31 o *array* referenciado por **a1** e as variáveis **size** e **n** contêm os valores a seguir representados:

Array **a1:**

15	3	5	2	4	7	6	8	13	17	12	11

size: 6 **n:** 0

Tenha presente que o *array* **a1** representado acima tem 12 elementos, mas a variável `size` tem, para este caso, o valor 6, e que é este valor que é passado para a função `f1` na linha 31.

Represente, justificando com clareza, o *array* referenciado por **a2** imediatamente após a execução da linha 31.

Nota: a ausência de justificação dos valores produzidos no *array* referenciado por **a2** penalizará significativamente a avaliação desta alínea.

c) [2] Admita que a função `f2` é chamada com a seguinte linha de código,

```
a2 = f2(a1, &size, f3);
```

sabendo-se que imediatamente antes da execução desta linha de código o *array* referenciado por **`a1`** e a variável **`size`** contêm os valores a seguir representados:

Array **`a1`**:

2	9	1	3	0	4	7	8	6	5	7	1	8	-3	5	-9	-5	3	0	6	2	-1	9	4

`size:` `24`

Represente, justificando com clareza, o *array* referenciado por **`a2`** imediatamente após a execução daquela linha de código.

Admita agora que a função `f2` é chamada sobre ao *array* **a1** acima representado, usando-se a mesma linha de código indicada no início desta alínea, mas desta vez com a variável `size` contendo o valor 10. Represente, justificando com clareza, o *array* referenciado por **`a2`** imediatamente após a execução daquela linha de código.

2. [5,5 valores]

Pretende-se implementar uma estrutura de dados para armazenar registos de informação com o tipo seguinte:

```
typedef struct {
    char *name;          // nome: string; alojar dinamicamente
    int age;
} Info;
```

Considere o armazenamento dos registos numa lista ligada, formada por elementos com o tipo `Elem`. A lista é ordenada pelo campo `name` dos registos armazenados, alfabeticamente crescente.

```
typedef struct elem {
    struct elem *next;   // ligação em lista
    Info  *reg;          // aponta registo armazenado
} Elem;
```

Nos exercícios considere, por simplificação, que nunca ocorre falta de memória para alojamento dinâmico.

a) [1,5] Escreva a função

```
Info *buildInfo( char *name, int age );
```

que constrói e retorna um registo de informação. Deve alojar dinamicamente a memória necessária e preencher com os dados passados nos parâmetros correspondentes ao nome dos campos.

b) [2] Escreva a função

```
int listInsert( Elem **headPtr, Info *reg );
```

que se destina a inserir numa lista, cujo ponteiro para a cabeça é indicado por `headPtr`, o registo de informação indicado por `reg`. Este registo foi previamente alojado e preenchido. Não são armazenados elementos com o mesmo nome. A função retorna 1, se inseriu com sucesso, ou 0, se já existia um registo com o campo `name` idêntico, mantendo-se inalterada a lista.

c) [2] Escreva a função

```
int listGetAge( Elem *head, char *name, int *agePtr );
```

que procura, na lista indicada por `head`, um registo com o nome indicado por `name` e obtém o valor do respetivo campo `age`, depositando-o na variável indicada pelo parâmetro de saída `agePtr`. A função retorna 1, em caso de sucesso, ou 0, se não existir nenhum registo com o campo `name` pretendido.

3. [5,5 valores]

Pretende-se implementar uma versão diferente, com a mesma funcionalidade especificada no exercício anterior, usando uma árvore binária de pesquisa em vez da lista ligada.

Mantém-se o tipo `Info`, definido anteriormente, e a função que o constrói. O tipo `Node` seguinte representa os nós da árvore binária. Esta é ordenada pelo campo `name` dos registos armazenados, alfabeticamente crescente de `left` para `right`.

```
typedef struct node {
    struct node *left, *right;     // ligações na árvore
    Info  *reg;                    // aponta registo armazenado
} Node;
```

Nos exercícios considere, por simplificação, que nunca ocorre falta de memória para alojamento dinâmico.

a) [2] Escreva a função

```
int treeInsert( Node **rootPtr, Info *reg );
```

que se destina a inserir numa árvore, cujo ponteiro para a raiz é indicado por `rootPtr`, o registo de informação indicado por `reg`. Este registo foi previamente alojado e preenchido. Não são armazenados elementos com o mesmo nome. A função retorna 1, se inseriu com sucesso, ou 0, se já existia um registo com o campo `name` idêntico, mantendo-se inalterada a árvore.

b) [2] Escreva a função

```
int treeGetAge( Node *root, char *name, int *agePtr );
```

que procura, na árvore indicada por `root`, um registo com o nome indicado por `name` e obtém o valor do respetivo campo `age`, depositando-o na variável indicada pelo parâmetro de saída `agePtr`. A função retorna 1, em caso de sucesso, ou 0, se não existir nenhum registo com o campo `name` pretendido.

c) [1,5] Escreva a função

```
void treeDelete( Node * root );
```

que elimina a árvore com a raiz `root`, libertando toda a memória de alojamento dinâmico sob o seu controlo.

4. [4 valores]

Considere um conjunto de módulos escritos em linguagem C, compilados individualmente com os comandos:

```
gcc -c m_*.c
gcc -c app.c
```

Na caixa ao lado apresenta-se, relativamente aos módulos compilados, as invocações do comando `nm` e as respetivas listas de símbolos, classificados com as abreviaturas: `T`, *public text*; `t`, *private text*; `U`, *undefined*.

```
nm m_*.o
m_a.o:
0000000000000013 T fa
                 U fd
0000000000000000 t fc

m_b.o:
0000000000000018 T fb
                 U fc
m_c.o:
0000000000000000 T fc

m_d.o:
                 U fc
0000000000000000 T fd

nm app.o
                 U fa
0000000000000000 t fc
0000000000000022 T main
```

Admita que são criados e usados no processo de compilação os *header files*, `m_a.h`, `m_b.h`, `m_c.h` e `m_d.h`, cada um contendo as assinaturas das funções públicas do módulo fonte "`.c`" com o respetivo prefixo.

a) [1] Tendo em conta o controlo de inclusão múltipla e sabendo que a função `fa` tem um parâmetro do tipo `int` e não produz valor de retorno, escreva o *header file* `m_a.h`. Indique, justificando, quais os módulos de código fonte "`.c`" onde este deve ser incluído.

b) [2] Com base nos símbolos listados, identifique os módulos que devem ser ligados ao módulo de aplicação `app.o` para gerar o executável. Escreva um *makefile* que produza, de forma eficiente, o executável com o nome "`app`".

c) [1] Após a produção do executável obtém-se a sua lista de símbolos com o comando "`nm app`". Indique, justificando, quantos são os símbolos com o nome "`fc`" existentes nesta lista.