Instituto Superior de Engenharia de Lisboa

Programação II

2020/21 – 2.º semestre letivo

Teste – época de recurso

2021.07.26

1. [4,5 valores] Considere as definições das funções `f1`, `f2`, e `f3`. Na linha 1 do troço de código seguinte 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 da alínea c) 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 das funções `f1`, `f2`, e `f3`.

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

a) [1] Comente o código da função `f1` e apresente, correta e completamente preenchido, o respetivo cabeçalho descritivo seguindo o formato do 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) [1] 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

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) [1] Admita agora 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	-3	1	3	0	4	7	8	6	-5	7	1	8	9	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.

d) [1,5] Considere que se pretende agora realizar uma função `f1a` com a mesma funcionalidade da função `f1` mas com maior generalidade, que possa operar sobre *arrays* de outros tipos e não apenas *arrays* de elementos do tipo `int`. Sendo uma versão da função `f1`, a função `f1a` deve seguir **de muito perto** o algoritmo da função `f1`.

Sem utilizar qualquer função da biblioteca normalizada, escreva a definição da função `f1a`, adicionando ou removendo parâmetros se necessário. Escreva a definição da função com uma indentação correta, usando obrigatoriamente comentários.

Rescreva a linha de código onde é usada a função `f1`, no exemplo de utilização anterior, substituindo-a pela utilização da função `f1a`, igualmente aplicada ao *array* referenciado por **a1**.

Nota: desaconselha-se veementemente a escrita da definição desta função sem a utilização de comentários porque penalizará significativamente a avaliação desta alínea.

2. [4 valores] Pretende-se usar uma representação alternativa de números inteiros usando uma estrutura composta pelos seguintes campos:

```
typedef struct num
{
  int isPositive;          // true  - Indica se o número é positivo (ou zero)
                           // false - Indica se o número é negativo
  unsigned int value;      // Valor absoluto do número
  char *desc;              // String alojada dinamicamente,
                           // contendo a descrição do número
                           //             Exemplo: 174 ou -174 --> "OneSevenFour"
                           //                       89 ou  -89 --> "EightNine"
                           //                                0 --> "Zero"
} NUM;
```

Suponha ainda que existe o *array* **arrDigits** definido globalmente com os seguintes valores:

```
const char *arrDigits[] = { "Zero", "One", "Two",   "Three", "Four",
                            "Five", "Six", "Seven", "Eight", "Nine" };
```

a) [1] Escreva a função

```
int num_compare( NUM *v1, NUM *v2 );
```

que compara dois números armazenados na estrutura anteriormente definida, retornando um valor negativo, zero ou um valor positivo, respetivamente, se o valor numérico armazenado em **v1** for inferior, igual ou superior a **v2**.

b) [2] Escreva a função

```
char *get_num_description( unsigned int n );
```

que cria dinamicamente uma *string* e a preenche com a descrição (em palavras) do valor do parâmetro **n**, usando as *strings* já definidas no *array* **arrDigits**. A função deverá fazer sucessivas concatenações (re)alojando memória sempre que seja necessário concatenar uma nova palavra à descrição, recorrendo para tal à função **realloc**.

```
void *realloc(void *ptr, size_t size);
```

Exemplos:

```
get_num_description(174) --> "OneSevenFour"
get_num_description(74)  --> "SevenFour"
get_num_description(5) --> "Five"
get_num_description(0) --> "Zero"
```

Sugestão: Caso entenda, poderá supor já definida a função:

```
unsigned int reverse_int(unsigned int n);
```

que devolve um inteiro com os dígitos invertidos (Ex: 123 --> 321)

c) [1] Escreva a função

```
NUM *get_num( signed int n );
```

que cria e preenche uma nova estrutura capaz de representar o valor numérico enviado para a função no parâmetro **n**. A estrutura deverá ser iniciada com o valor absoluto de **n**, informação sobre o seu sinal e a descrição do número em palavras.

3. [5 valores] Pretende-se alterar significativamente a forma como as análises clínicas são registadas. De acordo com a nova tendência, o valor ótimo de uma determinada análise passa a ser o valor zero, isto é, aquele que não está, nem em falta, nem em excesso. O valor associado a cada análise pode assim variar entre um valor inteiro negativo e um valor positivo, sendo registado numa estrutura do tipo **myRecord** onde, além do valor (**value**), se regista também o nome da análise realizada (**item**).

   O conjunto de análises realizadas por um indivíduo será armazenado numa lista ligada ordenada de forma crescente pelo valor que for indicado para cada um dos itens em análise.

```
typedef struct my_record{
    char *item;              // nome do item a registar
    NUM  *value;              // valor associado
    struct my_record *next;  // ligação em lista
} myRecord;

typedef struct{
    myRecord *data; // aponta o primeiro elemento da lista (menor valor)
} myList;
```

   a) [1] Escreva a função

```
myRecord *get_new_record(int value, char* item_name);
```

   que cria um novo registo do tipo **myRecord**, sem ligação a qualquer outro registo, a partir dos valores enviados à função.

   b) [1] Escreva a função

```
int record_compare(myRecord *ptr1, myRecord *ptr2);
```

   que compara dois registos do tipo **myRecord**, com vista à sua ordenação, retornando um valor negativo, zero ou um valor positivo, respetivamente, se o registo indicado por **ptr1** for considerado inferior, igual ou superior ao indicado por **ptr2**.

   A ordenação resultante desta função deverá ser feita por valor crescente no campo **value** dos elementos **myRecord** apontados. Caso os valores sejam iguais, o desempate será realizado pela descrição, de forma crescente. Deverá usar obrigatoriamente a função **num_compare**, especificada no exercício 2.

   c) [2] Escreva a função

```
void list_insert(myList *list, myRecord *rec,
                 int (*compare)( myRecord *, myRecord *));
```

   que insere o novo registo apontado por **rec** de forma ordenada na lista **list**. A comparação entre elementos deverá ser realizada usando a função passada no parâmetro **compare**, a qual tem especificação compatível com a da alínea anterior.

   d) [1] Escreva a função

```
void list_print(myList *list);
```

   que apresenta, em *standard output*, o conjunto de elementos presentes na lista indicada por **list**. Para cada elemento deverão ser apresentados os campos: descrição do elemento; valor associado (com sinal, exceto se for 0); valor numérico descritivo dos dígitos (com indicação explícita de sinal no caso dos negativos).

   Exemplo:

```
Lymphocytes     -321 : ThreeTwoOne [MINUS]
Albumin         -21  : TwoOne [MINUS]
Basophils       -21  : TwoOne [MINUS]
Hemoglobin      -12  : OneTwo [MINUS]
Cholesterol      0   : Zero
Protein          0   : Zero
Hematocrit      +12  : OneTwo
Glucose         +13  : OneThree
```

4. [3,5 valores] Pretende-se construir um programa para apoiar o acesso a ficheiros de áudio, contendo faixas de música. A estrutura proposta é baseada numa árvore binária de pesquisa, formada por nós do tipo **`TNode`**, que faculta o acesso aos descritores das músicas. Os descritores, com o tipo **`Music`**, contêm o nome do artista, o título da obra e o nome completo do ficheiro de áudio. Este é representado numa *string* que inclui, na parte inicial, o caminho de localização do ficheiro no disco do computador.

```
typedef struct{
    char artist[MAX_ART_NAME+1]; // nome do artista
    char title[MAX_TITLE+1];     // título da obra musical
    char *fileName; // nome completo do ficheiro; alojamento dinâmico
} Music;

typedef struct tNode{
    struct tNode *left, *right; // ponteiros de ligação na árvore
    Music *data;// acesso ao descritor da obra musical
} TNode;
```

A ordenação da árvore é pelo campo `title` dos descritores referenciados, alfabeticamente crescente da esquerda para a direita.

a) [2] Escreva a função

```
Music *tAddReplace( TNode **rp, char *a, char *t, char *fn );
```

que adiciona a informação de uma música à estrutura de dados, ou atualiza-a se já existir. O parâmetro **`rp`** (*root pointer*) representa o endereço do ponteiro raiz da árvore binária. Os parâmetros **`a`**, **`t`** e **`fn`** indicam a informação a colocar, respetivamente, nos campos **`artist`**, **`title`** e **`filename`** do descritor da música.

Se o título, indicado por **`t`**, não existir na árvore, deve ser adicionado um novo nó para o armazenar. Considere a inserção nas folhas. Se já existir uma música com o título indicado, devem ser atualizados os outros campos do respetivo descritor. No caso de se deixar de utilizar algum espaço de alojamento dinâmico, este deve ser adequadamente libertado.

A função **`tAddReplace`** retorna o endereço do descritor criado ou existente com o título indicado.

b) [1,5] Pretende-se também adicionar à estrutura de dados uma *hash table* para facultar o acesso aos descritores das músicas a partir dos nomes dos artistas. Admita que a resolução de colisões é feita usando listas ligadas. O propósito da consulta à *hash table* é: dado o nome de um artista, encontrar um conjunto referências para os descritores de todas as músicas desse artista. A estrutura de dados para armazenar os conjuntos de referências é ao seu critério.

Escreva a definição dos tipos necessários para construir a *hash table* especificada. Admita que a dimensão da tabela é passada como parâmetro na sua construção. Escreva comentários nos campos das estruturas, de modo a clarificar o seu significado.

5. [3 valores] Considere um conjunto de módulos escritos em linguagem C, **comp1.c**, **comp2.c**, **comp3.c**, **inout.c**, **sortPair.c** e **sort.c**, compilados individualmente com o comando "**gcc -c *.c**".

   Na caixa abaixo apresenta-se o resultado do comando "**nm *.o**" que mostra, a partir dos módulos compilados, as listas de símbolos, classificados com as abreviaturas: **T**, *public text*; **t**, *private text*; **U**, *undefined*.

```
comp1.o:
0000000000000000  T compare1

comp2.o:
0000000000000000  T compare2

comp3.o:
0000000000000000  T compare3

inout.o:
                  U free
                  U scanf
0000000000000041  T print
                  U printf
0000000000000000  T read
                  U realloc
0000000000000069  T recycle

sortPair.o:
0000000000000022  T sortPair
0000000000000000  t swap

sort.o:
0000000000000000  T sort
                  U sortPair
```

   Considere também que há um módulo de aplicação, **aplic.c**, contendo a função **main** seguinte

```
int main( int arc, char **argv ){
    int n = 0;
    Item *a = read( &n );
    sort( a, n, compare1 );
    print( a, n );
    recycle( a, n );
    return 0;
}
```

   Admita que são criados e usados no processo de compilação os *header files* **comp1.h**, **comp2.h**, **comp3.h**, **inout.h**, **sortPair.h** e **sort.h**, cada um contendo as assinaturas das funções públicas do módulo fonte (**.c**) com o respetivo prefixo. Existe também o *header file* **item.h**, com a definição do tipo **Item**, o qual é incluído em todos os módulos fonte.

   a) [0,5] Apresente a lista de símbolos produzida por "**nm aplic.o**" e a respetiva classificação (**t**, **T** ou **U**).

   b) [0,5] Diga, justificando, se há funções definidas com o atributo **static**.

   c) [1] Escreva o *header file* **inout.h**, tendo em conta o controlo de inclusão múltipla e considerando assinaturas das funções compatíveis com a utilização na função **main**. Indique, justificando, quais os módulos de código fonte (**.c**) onde o *header file* **inout.h** deve ser incluído.

   d) [1] Com base nos símbolos listados, identifique os módulos que devem ser ligados ao módulo de aplicação **aplic.o** para gerar o executável. Escreva um *makefile* que produza, de forma eficiente, o executável com o nome "**aplic**" a partir dos módulos fonte (**.c**) necessários.