Instituto Superior de Engenharia de Lisboa

Programação II

2023/24 – 1.º semestre letivo

Exame de Época de Recurso

2024.01.22

Esta prova é formada por duas partes, I e II. A **Parte I** corresponde ao **1.º Teste Parcial**; a **Parte II** corresponde ao **2.º Teste Parcial**.

Todos os alunos recebem a prova completa. A realização do **2.º teste** tem a duração de **1 hora e 15 minutos**; se o aluno permanecer após esse tempo, realiza **exame completo**, com a duração de **2 horas e 30 minutos**.

Cada parte é cotada para 20 valores; no caso de exame completo, a classificação é a média aritmética das duas partes.

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

Parte I – Repetição do 1.º Teste Parcial

(Se realiza apenas o 2.º Teste Parcial, deve ignorar esta parte e passar de imediato à Parte II)

1. [4 valores]

Escreva a função

```
int isMirror(unsigned long value);
```

que verifica se os bits do parâmetro `value` são iguais quando lidos da direita para a esquerda e da esquerda para a direita. O retorno é 1 em caso afirmativo ou 0 no caso oposto.

Com vista à portabilidade do código, se necessitar de identificar a dimensão da palavra deve usar o operador `sizeof` e a macro `CHAR_BIT` definida em `limits.h`.

2. [8 valores]

Para a realização da **Série de Exercícios 2** foi utilizado o tipo `TagData` para armazenamento das *tags* existentes num ficheiro MP3. Uma *tag* representa uma faixa (*track*) de um álbum de música. Admita que as constantes simbólicas indicadas em maiúsculas contêm valores adequados à representação pretendida.

```
typedef struct{
  char filename[MAX_FILENAME + 1];
  char title[MAX_TITLE + 1];
  char artist[MAX_ARTIST + 1];
  char album[MAX_ALBUM + 1];
  short year;
  char comment[MAX_COMMENT + 1];
  char track;
  char genre;
} TagData;
```

Considere o troço de código que consta na caixa seguinte:

```
1   (…)
2   size_t f1( TagData a[], size_t nElem,
3              int (*action)(TagData *data, void *context),
4              void * context ){
5     size_t s = 0;
6     while(nElem--)
7       s += (*action)(a++, context);
8     return s;
9   }
10  int main(){
11    (…) // Alojamento e preenchimento do array a
12    size_t nElem = sizeof(a) / sizeof(a[0]);
13    size_t n1 = f1(/* … */);
14    char highTrack = 10;
15    size_t n2 = f1(/* … */);
16    (…)
17    return 0;
18  }
```

a) [2] Escreva a função `showAllTitles` que, quando usada, na linha 13, como um dos argumentos da função `f1`, produzirá em *standard output* a afixação do campo `title` de todas as *tags* existentes no *array* **a**. A função deve retornar um valor tal que a variável `n1` fique com o valor da contagem de todas as *tags* existentes no *array*. Transcreva a linha 13 para a folha da sua prova completando a chamada da função `f1` com os argumentos corretos.

b) [3] Escreva a função `showHighTracks` que, quando usada como consta na linha 15, produzirá em *standard output* a afixação do campo `album` de todas as *tags* que existem no *array* **a** e cuja faixa (campo `track`) é superior a um valor passado por parâmetro e contido na variável `highTrack`, iniciada na linha 14. Admita que todas as *tags* têm no campo `album` uma *string* válida, devidamente terminada, e têm o campo `track` devidamente preenchido. A função deve retornar um valor tal que a variável `n2` fique com o valor da contagem das *tags* cujo n.º da faixa é superior ao valor contido na variável `highTrack`. Transcreva a linha 15 para a folha da sua prova completando a chamada da função `f1` com os argumentos corretos.

c) [3] 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 `TagData`. Sendo uma versão da função `f1`, a função `f1a` deve seguir **de muito perto** o algoritmo da função `f1`.

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 15, onde é usada a função `f1`, no exemplo de utilização anterior, substituindo-a pela utilização da função `f1a`, igualmente aplicada ao *array* **a**.

3. [4 valores]

Pretende-se processar frases, de modo a uniformizar as palavras na forma maiúscula-minúsculas.

Escreva a função

```
char *firstUpper( char *str );
```

que modifica a *string* indicada por `str`, de modo a formatar cada palavra com o primeiro carater em maiúscula e os restantes em minúscula, mantendo os separadores existentes. Retorna o endereço de início da primeira palavra. Se não houver palavras, retorna a localização do terminador. Considere palavra qualquer sequência de carateres que não sejam separadores.

Propõe-se que use as primitivas seguintes, declaradas no *header file* normalizado `ctype.h`.

```
int isspace(int c); // True se c é ' ' (espaço), '\t' (tab), '\n', '\r', '\v' ou '\f';
int toupper(int c); // Maiúscula de c;
int tolower(int c); // Minúscula de c.
```

4. [4 valores]

Considere o conjunto de módulos fonte com o conteúdo representado na tabela:

Ficheiro	Conteúdo

base.c	`#include "base.h"`
	`double base( Solid *rp ){`
	`  return rp->width * rp->depth;`
	`}`

volume.c	`#include "base.h"`
	`#include "volume.h"`
	`static double area( Solid *sp ){`
	`  return base( sp );`
	`}`
	`double volume( Solid *sp ){`
	`  return area( sp ) * sp->height;`
	`}`

main.c	`#include <stdio.h>`
	`#include "volume.h"`
	`int main(){`
	`  Solid s;`
	`  scanf( "%lf%lf%lf", &s.width, &s.depth, &s.height );`
	`  printf( "Volume: %lf\n", volume( &s ) );`
	`  return 0;`
	`}`

Admita que existem também os *header files* `solid.h`, com a definição do tipo `Solid`, bem como `base.h` e `volume.h`, que incluem `solid.h` e cada um contém as assinaturas das funções públicas do módulo fonte (`.c`) respetivo.

a) [2] Considerando a execução dos comandos «`gcc -c *.c`» e «`nm *.c`», apresente a lista de símbolos de todos os módulos compilados, e a respetiva classificação com a convenção da ferramenta `nm` (p. ex.: T, *public text*; t, *private text*; U, *undefined*).

Indique ainda, por cada símbolo com classificação U, se é resolvido pela biblioteca normalizada ou por algum dos outros módulos, e, neste caso, qual deles.

b) [2] Tendo em conta as dependências existentes, escreva um *makefile* que produza, de forma eficiente, o executável com o nome "`main`" a partir dos módulos fonte (`.c`) estritamente necessários.

Parte II – Repetição do 2.º Teste Parcial

Em todos os exercícios desta parte, por simplificação, assuma que o **alojamento dinâmico** é **sempre bem sucedido**, não ocorrendo falta de memória de *heap*.

5. [9 valores]

Pretende-se armazenar, em alojamento dinâmico, informação sobre bilhetes de avião. A informação de cada voo é obtida a partir de uma linha de texto iniciada pelo código de reserva, seguida pelos códigos normalizados com 3 carateres dos aeroportos de partida e chegada separados por um hífen (‘–’) e por uma breve descrição textual. Os campos são separados entre si pelo caráter *pipe* (‘|’).

Exemplos: “`12345|LIS-FNC|Lisboa -> Funchal`”

“`23456|FNC-PDL|Origem: Funchal, Destino: Ponta Delgada`”

a) [3] Escreva a função

```
char *break_flight(char *text, int *code, char *start, char *end);
```

destinada a separar e identificar os dados de um voo, contidos na *string* indicada por `text`, criando uma réplica da descrição, em alojamento dinâmico, e afetando as variáveis indicadas por `code`, `start` e `end`, respetivamente, com os valores do código de reserva, aeroporto de partida e aeroporto de chegada. Em caso de sucesso, retorna a réplica da *string* de descrição criada; se ocorrer insucesso, devido a conteúdo de `text` incompleto, retorna NULL. Neste caso, deve terminar sem deixar espaço inadequadamente alojado.

Sugere-se que use as funções `strtok` e `atoi` da biblioteca normalizada.

```
char *strtok(char *string, char *delimiters);
int atoi(char *string);
```

b) [3] Os voos são registados em elementos, alojados dinamicamente, com o tipo seguinte:

```
typedef struct{
  int code;           // Código de reserva
  char start[3+1];    // Aeroporto de partida
  char end[3+1];      // Aeroporto de chegada
  char *desc;         // Descrição do voo (string alojada dinamicamente)
} Flight;
```

Escreva a função

```
Flight *create_flight(char *text);
```

destinada a criar, em alojamento dinâmico, um elemento `Flight` preenchido com os dados obtidos da *string* `text`. A função retorna o endereço do elemento criado ou NULL, em caso de insucesso. Neste caso, a função não deve deixar espaço inadequadamente alojado. Deve utilizar a função `break_flight`.

c) [3] Pretende-se adicionar à descrição o tempo estimado para o voo, devendo este ser colocado entre parêntesis retos (‘`[`’ ‘`]`’).

Escreva a função

```
void append_flight(Flight *flight, char *duration);
```

que modifica o campo `desc` da estrutura indicada por `flight`, adicionando, no final da *string* existente, o texto indicado por `duration`. Deve usar a função de biblioteca adequada para assegurar o espaço necessário ao novo conteúdo da *string*.

Exemplo:

No voo com a descrição “`Lisboa -> Funchal`”, depois de invocada a função `append_flight` com a estrutura e o tempo estimado de voo “`2 horas e 10 min`”, a *string* do campo `desc` deverá conter “`Lisboa -> Funchal [2 horas e 10 min]`”.

6. [5 valores]

Pretende-se suportar, numa lista simplesmente ligada, a criação e acesso de um conjunto de elementos com o tipo `Flight`, definido no anteriormente. A lista é ordenada crescentemente pelo código de reserva. Considere o nó de lista representado pelo tipo seguinte.

```
typedef struct listFlight{
  Flight *flight;
  struct listFlight *next;
} ListFlight;
```

Assumindo que existem listas criadas, com o formato especificado, escreva a função

```
Flight *listRemove(ListFlight **headAddr, int code);
```

destinada a encontrar e retirar de uma lista, cujo ponteiro cabeça é indicado por `headAddr`, um elemento identificado pelo código de reserva `code`.

Em caso de sucesso, retorna o endereço do elemento retirado da lista; deve deixar este elemento ativo, para utilização, e eliminar o nó de lista onde se encontrava o seu acesso. Se o código pesquisado não existir, não modifica a lista e retorna NULL.

7. [6 valores]

Pretende-se organizar os elementos do tipo `Flight` em subconjuntos acedidos através de uma árvore binária de pesquisa. Os subconjuntos são suportados por listas do modelo especificado no exercício anterior. A árvore é ordenada pelo aeroporto de chegada (campo `end`, o qual é reproduzido no nó de árvore), alfabeticamente crescente de `left` para `right`.

Considere o nó de árvore binária representado pelo tipo seguinte.

```
typedef struct bstFlight{
  char end[3+1];
  ListFlight *subset;
  struct bstFlight *left, *right;
} BstFlight;
```

a) [4] Escreva a função

```
void bstInsert(BstFlight **rootAddr, Flight *flight);
```

que insere no subconjunto adequado, pertencente a uma árvore, o elemento indicado por `flight`, previamente alojado e preenchido. O parâmetro `rootAddr` representa o endereço do ponteiro raiz da árvore binária. Se ainda não existir o subconjunto adequado para o elemento, deve ser criado um nó para o suportar.

Para inserir o voo na lista ligada, deve utilizar a função `listInsert` que se assume já existir, com a assinatura seguinte.

```
void listInsert(ListFlight **headAddr, Flight *flight);
```

b) [2] Admita que existe uma árvore binária com 15 nós, perfeitamente balanceada, e é chamada a função `bstInsert`, com um voo, podendo já existir na árvore, ou não, outros voos com o mesmo destino. Indique e justifique, tendo em conta a possibilidade de chamada recursiva, as quantidades mínima e máxima de chamadas à função `bstInsert` para realizar esta inserção. Nos seus cálculos, deve incluir a chamada à função `bstInsert` a partir do programa de aplicação.