Ficha 5 – Ponteiros e alocação dinâmica de memória

Tópicos abordados:

– Ponteiros

– Alocação dinâmica de memória

– Deteção de erros de memória com memcheck

– Exercícios

Duração prevista: 2 aulas

©2020: {miguel.negrao, vitor.carreira, patricio, miguel.frade, rui, nuno.costa, gustavo.reis, carlos.machado}@ipleiria.pt

1. Ponteiros

Um ponteiro é um tipo de dados que permite armazenar um endereço de memória. O endereço pode referir-se ao endereço de uma variável, a um bloco de memória alocado dinamicamente ou mesmo a uma função. Quando o endereço se refere a uma variável, um ponteiro permite que esta seja lida/alterada indiretamente. A passagem de parâmetros por referência é um exemplo típico da utilização de ponteiros.

1.1. Declaração

Sintaxe:

```
tipo_dados *nome_do_ponteiro;
```

Exemplos:

```
int *iptr; /* Declara um ponteiro para um inteiro */
float *fptr; /* Declara um ponteiro para um float */
char *cptr; /* Declara um ponteiro para um caráter */
```

Nota: a declaração de uma variável do tipo ponteiro apenas reserva espaço para a guardar, mas não a faz apontar para qualquer endereço de memória. É um erro utilizar um ponteiro antes de o iniciar.

1.2. Operadores especiais

Existem dois operadores especiais para operações com ponteiros, * e &, com a seguinte sintaxe:

- *<ponteiro> - devolve o conteúdo da zona de memória definida pelo <ponteiro>
- &<variável> - devolve o endereço de memória da <variável>

Apesar do operador de multiplicação e o operador * utilizarem o mesmo símbolo, não existe nenhuma relação entre eles. Note que o operador & só pode ser aplicado a variáveis. Operações tais como &10 &(x+2) são ilegais.

1.3. Iniciação e acesso

Antes de usar um ponteiro é necessário iniciar o mesmo. Caso se trate de um ponteiro para uma variável, na iniciação é utilizado o operador & ("endereço de") para colocar o ponteiro a apontar para a variável. Para mostrar o valor apontado por um ponteiro utiliza-se o operador * (operador *indirection* ou de *desreferenciação*).

As seguintes fases são fundamentais para utilizar uma variável do tipo ponteiro:

1. Declaração do ponteiro

```
int *iptr; /* ponteiro */
int zx;
```

2. Iniciação (utiliza o operador & no caso de variáveis)

```
iptr = &zx; /* O ponteiro iptr vai armazenar o endereço
            da variável zx, i.e., iptr aponta para zx */
```

3. Acesso (utiliza o operador *)

```
printf("%d", *iptr); /* Escreve o conteúdo de zx. Como iptr é um
                     ponteiro para um inteiro, *iptr é tratado
                     como se fosse um inteiro (%d) */
```

Nota: Um ponteiro deve ser sempre inicializado, sendo prática corrente inicializar um ponteiro com o valor **NULL**. **NULL** é uma constante simbólica que, quando utilizada num ponteiro, significa que o ponteiro não aponta para nada. Se for necessário mostrar o

endereço de uma variável, emprega-se o *printf* com o especificador **%p** que mostra o endereço no formato hexadecimal.

De seguida apresenta-se um exemplo mais completo.

```
#include <stdio.h>

int main(void)
{
  int *iptr = NULL, q = 20;
  iptr = &q;

  printf("%d\n", q); /* Escreve o valor 20 */
  printf("%d\n", *iptr); /* Escreve o valor apontado por iptr*/
  printf("Endereco da variavel q = %p\n", (void *) &q);
  printf("Endereco armazenado em iptr = %p\n", (void *) iptr);
  printf("Endereco da variavel iptr = %p\n", (void *) &iptr);
  return 0;
}
```

Figura 1 - Ficheiro ponteiros.c

Lab 1

Compile e execute o programa `ponteiros.c`, localizado na diretoria `lab1` (código disponível no Moodle). Analise o resultado e desenhe um diagrama representativo dos blocos de memória das variáveis *iptr* e *q*.

1.4. Ponteiros e tabelas

Em C existe uma relação direta entre ponteiros e tabelas (*arrays)*. Quando se declara uma tabela (vetor, *string* ou tabela de dimensão >= 2), o nome da tabela é um sinónimo do endereço do seu primeiro elemento. Passa agora a ser possível aceder aos elementos de um vetor[1] de duas maneiras diferentes:

- através dos índices: `nome_do_vetor[indice];`
- através do ponteiro: `*(nome_do_vetor + indice);`

Note que as formas de acesso apresentadas anteriormente apenas são válidas para o caso particular dos vetores. O acesso aos elementos de uma tabela com dimensão igual ou superior a dois utilizando um ponteiro engloba operações aritméticas mais complexas. A figura seguinte mostra a diferença entre as duas formas de acesso para o caso de um vetor `vet` com tamanho 10.

[1] Vetor corresponde a uma tabela unidimensional (exemplo: `int vetor[30];`)

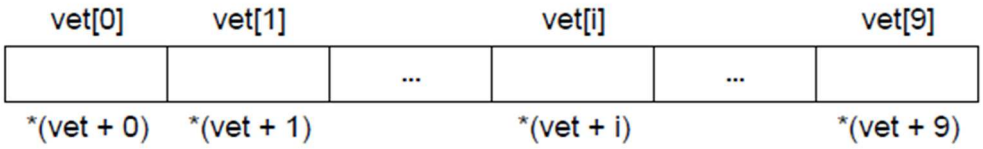

Figura 2 – Acesso a um vetor utilizando ponteiros

Para melhor entender o acesso aos elementos de um vetor utilizando um ponteiro é necessário referir as operações aritméticas permitidas envolvendo ponteiros. Em C é possível subtrair dois ponteiros (com exceção de ponteiros para o tipo `void`), assim como somar ou subtrair um inteiro com um ponteiro. Por conseguinte também são válidos o operador de incremento (++) e decremento (--). Adicionar o valor 1 a um ponteiro para um vetor faz com que o ponteiro passe a apontar para o elemento seguinte do vetor. Na realidade o operador ++ força um salto de N bytes em que N representa o **sizeof** do tipo de dados para o qual o ponteiro aponta.

Para melhor ilustrar as diferenças, apresentam-se duas versões de uma função que soma os elementos de um vetor de inteiros com *n* elementos.

```
int somatorio_v1(int v[], int n){
   int i, soma = 0;
   for (i = 0; i < n; i++)
      soma += v[i];
   return soma;
}
```

Figura 3 – Somatório dos elementos de um vetor usando índices

```
int somatorio_v2(int *v, int n){
   int i, soma = 0, *ptr;
   ptr = v;
   for (i = 0; i < n; i++) {
      soma += *ptr;
      ptr++;
   }
   return soma;
}
```

Figura 4 – Somatório dos elementos de um vetor usando ponteiros

Comparando as duas versões podemos constatar que os seguintes protótipos são equivalentes:

```
void somatorio_v1(int v[], int n);
void somatorio_v2(int *v, int n);
```

Do mesmo modo, se reparar nos protótipos de algumas funções que manipulam *strings*, por exemplo a função `strcmp`, pode-se constatar que os seguintes protótipos também são equivalentes:

```
int strcmp(char s1[],char s2[]);
int strcmp(char *s1,char *s2);
```

Compreende-se agora porque é necessária a utilização de uma função própria em vez do operador == para comparar duas *strings*, uma vez que o nome da variável é um ponteiro para o primeiro caráter. Senão, veja-se o seguinte exemplo:

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char s1[] = "Ola", s2[] = "Ola";
   if (s1 == s2)
      printf("ponteiros apontam para o mesmo sitio\n");
   else
      printf("Ponteiros diferentes\n");
   if (strcmp(s1, s2) == 0)
      printf("As strings sao iguais\n");
   return 0;
}
```

Figura 5 – Comparação de strings

Lab 2

Compile e execute o programa `comparastrings.c`, localizado na diretoria `lab2`. Altere o programa a fim de imprimir o endereço das variáveis `s1` e `s2`.

1.5. Ponteiros e estruturas

A linguagem C disponibiliza um operador especial chamado ponteiro para membro (`->`) que permite aceder aos membros de uma estrutura a partir de uma variável do tipo ponteiro. Por exemplo, na listagem abaixo mostrada, a linha de código `ptr->pratica = 15;` acede ao campo `pratica` da variável `nota_so` através do ponteiro `ptr`. Note-se que o acesso `ptr->pratica` é equivalente a `(*ptr).pratica`.

```c
#include <stdio.h>

typedef struct nota {
   float teorica;
   float pratica;
} nota_t;

int main(void){
   nota_t nota_so;
   nota_t *ptr = NULL;

   ptr = &nota_so;
   nota_so.teorica = 10;
   ptr->pratica = 15;/* o mesmo que: (*ptr).pratica = 15; */

   printf("Teorica: %.3f\n", ptr->teorica);
   printf("Pratica: %.3f\n", ptr->pratica);
   return 0;
}
```

Figura 6 – Operador ponteiro para membro de uma estrutura

2. Uso de memória estática

Até ao momento todas as variáveis declaradas numa função usam alocação estática. Quando uma variável local é declarada, o espaço em memória necessário para a representar é alocado numa zona de memória denominada ***stack*** (pilha). Assim que uma função termina, a parte da pilha afeta à função (*stack frame*) é libertada invalidando todas as variáveis locais. Retornar endereços para variáveis locais é um erro bastante frequente e grave dado que assim que a função termina o endereço da variável local deixa de ter qualquer significado.

A alocação estática apresenta as seguintes limitações:

- É necessário majorar em tempo de compilação o tamanho da estrutura de dados pretendida (e.g. no caso das tabelas é necessário majorar o número de elementos);
- Não é possível construir estruturas de dados dinâmicas cujo tamanho varia durante a execução (e.g. listas ligadas);
- A pilha possui um tamanho máximo muito inferior à memória total disponível;

2.1. Variable Length Arrays (VLA)

A norma C99 (1999) introduziu o conceito de *Variable Length Array* (VLA), que possibilita a declaração e uso de um vetor estático com um número variável de argumentos. A norma C99 não esclarece, contudo, o segmento de memória – pilha ou *heap* – que o compilador deve usar para implementar um VLA.

A Figura 7 lista o programa **vla.c** que demonstra o uso de vetores de tamanho variável. Concretamente, o código declara o tamanho do vetor `float vla_vector` da função `vla_func` dependente do parâmetro `num_elems`, que por sua vez depende do argumento passado ao programa através da linha de comando. O espaço para o vetor `vla_vector` é reservado apenas em tempo de execução.

Importa mencionar que os VLA apresentam uma grande fragilidade: caso o tamanho solicitado em tempo de execução seja maior do que o disponível ocorre um erro de execução. Em virtude disso e também do pouco suporte para VLA em compiladores de C, a norma C11 (2011) tornou os VLA opcionais. Deste modo, o seu uso é fortemente desaconselhado pois para além dos problemas de robustez, podem originar problemas de portabilidade do código. Esta funcionalidade é apenas referida para alertar para a sua existência, já que alguns utilizadores por vezes usam VLA sem que se apercebam desse facto.

Note-se que no compilador `gcc` é necessário usar a opção `-std=c99` quando se pretende que o código seja compilado de acordo com a norma C99.

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
/*
 * compile: gcc -Wall -W -std=c99 vla.c -o vla.exe
 */

void vla_func(int num_elems){
  float vla_vector[num_elems];

  printf("Using VLA with %d elements\n", num_elems);
  int i;
  for(i=0;i<num_elems;i++){
    vla_vector[i] = 1.0 * i;
  }
  /* Print 1st and last element of array */
  printf("[0] => %.2f\t", vla_vector[0]);
  printf("[%d] => %.2f\n", num_elems-1, vla_vector[num_elems-
1]);
}

int main(int argc, const char *argv[]){
  int num_elms;
  if( argc == 1 ){
    fprintf(stderr,"ERROR: no arguments given\n");
    fprintf(stderr,"%s <num_elements>\n", argv[0]);
    exit(1);
  }
  num_elms = atoi( argv[1] );
  if( num_elms <= 0 ){
    fprintf(stderr, "ERROR: invalid number of elements %d "
```

```
      "(num_elements needs to be positive)\n", num_elms);
    exit(2);
  }
  vla_func(num_elms);
  return 0;
}
```

Figura 7 – Exemplo com *variable length array* (VLA)

Lab 3

Compile e execute o programa `vla.c`, localizado na diretoria `lab3`, das seguintes formas, comentando os resultados:

a) ./vla 1(RTA: OK)0

b) ./vla 100 (RTA: OK)

c) ./vla -1 (RTA: detetado pelo programa que termina a execução)

d) ./vla 999999999 (RTA: dá *segmentation fault*)

3. Alocação dinâmica de memória

A alocação dinâmica permite alocar um bloco de memória numa zona de memória denominada ***heap***. A alocação dinâmica apresenta as seguintes vantagens:

- É possível indicar o tamanho pretendido efetuando assim um uso mais racional da memória;
- Permite criar estruturas de dados dinâmicas cujo tamanho varia durante a execução;
- O bloco de memória permanece alocado até que seja explicitamente destruído;
- No limite, é possível alocar um valor próximo da memória disponível (salvo se forem impostos limites por processo/utilizador);

3.1. Alocação de memória

Para alocar um bloco de memória utiliza-se a função ***malloc()*** definida na biblioteca **stdlib.h**:

```
#include <stdlib.h>
void *malloc(size_t size);
```

A função recebe por parâmetro o tamanho do bloco de memória a criar (em bytes) e devolve, em caso de sucesso, um ponteiro para o bloco alocado. Em caso de erro, a função devolve **NULL**. Note que o bloco de memória **não** é iniciado a zero.

Por omissão, o Linux segue uma abordagem otimista na alocação da memória. De facto, um valor diferente de **NULL** não é sinónimo de que toda a memória requisitada foi

alocada. Se o sistema ficar sem memória livre um ou mais processos poderão ser terminados abruptamente. Para mais informações, consulte **man malloc**.

Diferentes tipos de dados possuem diferentes tamanhos. O mesmo tipo de dados pode ter um tamanho em bytes consoante a arquitetura/compilador. Assim, um inteiro pode ocupar: 2, 4 ou mesmo 8 bytes. Para efeitos de portabilidade, o operador **sizeof** permite calcular o tamanho de uma expressão. A expressão pode ser um tipo de dados ou uma variável. Por exemplo, **sizeof(int)** devolve o número de bytes necessário para armazenar um inteiro.
O código seguinte aloca dinamicamente uma variável do tipo inteiro.

```
#include <stdlib.h>
int *ptr = malloc(sizeof(int));
```

O operador **sizeof** também pode receber como parâmetro uma variável. Neste caso, o operador devolve o tamanho do tipo de dados da expressão avaliada.

```
#include <stdio.h>
int main(void) {
   char *ptr = NULL;
   printf("sizeof(*ptr)=%zu\n", sizeof(*ptr));
   printf("sizeof(ptr) =%zu\n", sizeof(ptr));
   return 0;
}
```

Figura 8 – Operador sizeof

Lab 4
Compile e execute o programa `sizeof.c`, localizado na diretoria `lab4`, e comente os resultados.

3.2. Libertar a memória

Ao contrário das linguagens autogeridas (Java, C#, etc) que possuem um *garbage collector*, em C é necessário libertar **explicitamente** todos os blocos alocados dinamicamente. Para libertar um bloco utiliza-se a função ***free()***:

```
#include <stdlib.h>
void free(void *ptr);
```

A função recebe por parâmetro um ponteiro com o endereço do bloco a libertar. Exemplo:

```
free (ptr);
ptr = NULL;
```

Boas práticas de programação:

- Por cada ***malloc*** deve existir um ***free***;
- Após libertação o ponteiro deve ser colocado a **NULL.**

3.3. Alocação dinâmica de "strings"

Em C, uma *string* é um vetor de carateres terminado com caráter '\0' (zero). Quando as *strings* são alocadas dinamicamente, há que ter o cuidado especial de adicionar +1, correspondendo ao '`\0`', ao número de carateres que se pretende que a *string* possa suportar.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "debug.h"

int main(void) {
  char str_estatica[] = "ola mundo";
  char *clone = NULL;

  clone = malloc(strlen(str_estatica) + 1);

  if (clone == NULL)
    ERROR(1, "Nao foi possivel alocar a memoria para a string");

  strncpy(clone, str_estatica, strlen(str_estatica) + 1);
  printf("Clone=%s\n", clone);

  free(clone);
  clone = NULL;


  return 0;
}
```

Figura 9 – Alocação dinâmica de strings

Note que **sizeof(char)** é sempre 1, mas por questões de clareza é comum encontrar código que explicitamente usa o operador **sizeof** na alocação de *strings*. De acordo com o exemplo anterior teríamos:

```
clone = malloc((strlen(str_estatica) + 1)*sizeof(char));
```

3.4. Alocação dinâmica de vetores

Para alocar dinamicamente um vetor usa-se a função ***malloc()*** passando como parâmetro um inteiro do tipo `size_t` correspondente ao tamanho em bytes do tipo do vetor (normalmente obtida via ***sizeof()***) multiplicado pelo número de elementos no vetor:

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

int main(void) {
   int n = 0, *vetor = NULL, i;

   printf("Indique o número de elementos pretendido?");
   scanf("%d", &n);

   if (n <= 0){
       return 1;
   }
   vetor = malloc(sizeof(int)*n);
   if (vetor == NULL) {
       ERROR(1, "malloc() failed");
   }
   for (i = 0; i < n; i++){
      vetor[i] = 0;
   }
   /* código que use o vetor */

   free(vetor);
   vetor = NULL;
   return 0;
}
```

Figura 10 – Alocação dinâmica de vetores

3.5. Erros de execução típicos

No ambiente Linux, quando ocorre um erro de execução que termina abruptamente um processo, é ativado o mecanismo de *core dump*. O nome advém da escrita (*dumping*) da memória principal (*core memory*) num ficheiro chamado `core`. Este ficheiro é útil para, posteriormente, utilizando ferramentas para o efeito, verificar o estado do sistema na altura da ocorrência do erro, nomeadamente a memória afeta ao processo. Em muitos sistemas Linux, para que a escrita do ficheiro *coredump* seja efetivada, torna-se necessário que opção *coresize* do *ulimit* da *shell*, que indica o tamanho máximo de um ficheiro core, esteja ativa. Essa ativação pode ser feita especificando-se um valor elevado na opção –c do `ulimit`, conforme o exemplo que se segue:

```
ulimit -c unlimited
```

Para mais informações, consulte as instruções que acompanham o Lab 4.

3.5.1. Segmentation Fault

O erro de execução *segmentation fault* resulta do acesso a uma zona de memória não alocada ou que não está mapeada no processo. As causas mais frequentes para um erro desta natureza são:

1. Utilizar um ponteiro sem ter alocado memória para o mesmo

```
int *ptr;
/* ... */
ptr[0] = 1;
```

2. Invocar a função ***free*** com um ponteiro inválido, i.e., um ponteiro que não aponta para nenhum bloco de memória

```
int *ptr;
free(ptr);
```

3. Libertar, erradamente, duas vezes o mesmo bloco de memória (*double free*)

```
int *ptr = malloc(sizeof(int));
free(ptr);
free(ptr);
```

4. Não reservar espaço para o caráter de terminação nas *strings*

```
char ola[] = "ola mundo";
char *str = malloc(strlen(ola));
/* Deveria ser: malloc(strlen(ola) + 1);
```

5. No contexto das *strings*, utilizar o operador **sizeof** em vez de ***strlen***

```
char *ola = "ola mundo";
char *str = malloc(sizeof(ola)+1);
```

Considere o seguinte código fonte:

```
#include <stdio.h>

int main(void) {
   long *vect_not_initialized=NULL;
   unsigned int i=0;

   for(i=0;i<10;i++){
     vect_not_initialized[i]=i*100;
   }
}
```

Figura 11 - Código fonte com erro de execução "*Core Dumped*"

Neste caso, o vetor *vect_not_initialized* foi declarado, mas não alocado. Quando for executada a instrução de atribuição ocorre um erro do tipo *Segmentation Fault*, produzindo um *core dump*.

Existem algumas ferramentas que permitem ao programador descartar os erros mais comuns. Uma dessas ferramentas é o *Valgrind* (http://valgrind.org). Encontra-se disponível nos repositórios um interface gráfico para o *Valgrind* chamado **Valkyrie** (http://valgrind.org/downloads/guis.html) cuja utilização é vivamente recomendada.

Lab 5

1. Compile e execute o programa `main.c`, localizado na diretoria `lab5`, e execute-o de seguida. Comente o resultado. Verifique se foi criado um ficheiro *core*. Note: nas distribuições recentes, por omissão, os ficheiros *core* não são criados, sendo necessário fazer uso do comando *ulimit*, conforme anteriormente explicado (`ulimit -c unlimited`).
2. Usando o *gdb* e o ficheiro *core* determine em que linha ocorreu o erro de tipo *Segmentation Fault* (`gdb ./prog core`).
3. Faça a mesma verificação correndo o programa diretamente no *gdb* (`gdb ./prog, run`).

Lab 6

1. Compile e execute o programa `main.c`, localizado na diretoria `lab6`**.** Execute o programa normalmente e comente o resultado.
2. Corra novamente o programa através do **Valkyrie** e verifique os erros detectados. Experimente desativar a *flag* -g na *makefile*. Compile e execute novamente. Comente os resultados.
3. Utilizando o seguinte alias para a ferramenta *memcheck*, volte a executar o programa e comente os resultados:

```
$ alias memcheck='valgrind --track-origins=yes --tool=memcheck --leak-check=full'
$ memcheck ./prog
```

3.6.

3.7. Matrizes dinâmicas

Quando se faz uso de matrizes, a notação bidimensional [x][y] é muito conveniente para conferir maior clareza ao algoritmo. Tal notação está disponível quando se faz uso de uma matriz estática, isto é, uma matriz cujas dimensões são conhecidas aquando da compilação do código (*compile time*). Vejamos o seguinte código fonte:

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

```
int main(int argc, char* argv[]) {
   int matrix[10][5];
   int i, j;
   for(i=0;i<10;i++){
      for(j=0;j<5;j++){
         matrix[i][j] = i * 5 + j;
         printf("matrix[%d][%d]=%d | ", i, j, matrix[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

Figura 12 – Preenchimento de uma matriz estática com notação [i][j]

Na linguagem C, a memória alocada é sempre unidimensional, isto é, é um conjunto de octetos com apenas uma dimensão. Deste modo, será possível implementar uma matriz dinâmica? A resposta é afirmativa. Para o efeito atenda-se ao seguinte código fonte que transforma as coordenadas bidimensionais **[row][col]** na coordenada unidimensional **[row * num_cols + col]**.

```
#include <stdio.h>
#include <stdlib.h>
/**
 * allocates a matrix num_rows and num_cols and treats it as
 * a one dimension structure
 */
int main(int argc, char* argv[]) {
  if( argc != 3 ){
    fprintf(stderr, "expecting two arguments\n");
    fprintf(stderr, "%s <num_rows> <num_cols>\n", argv[0]);
    exit(1);
  }

  int num_rows = atoi(argv[1]);
  if( num_rows <= 0 ){
    fprintf(stderr, "Expecting a positive number for <num_rows>"
        " (got '%s')\n", argv[1]);
    exit(2);
  }
  int num_cols = atoi(argv[2]);
  if( num_cols <= 0 ){
    fprintf(stderr, "Expecting a positive number for <num_cols>"
        " (got '%s')\n", argv[2]);
    exit(3);
  }

  /* Allocate the memory */
  size_t mem_len = num_rows * num_cols * sizeof(int);
  int *matrix_ptr = malloc(mem_len);
  if( matrix_ptr == NULL ){
    fprintf(stderr, "Can't allocate %zu bytes for "
        "%d x %d matrix\n", mem_len, num_rows, num_cols);
    exit(4);
  }
```

```
  /* Iterate over the matrix */
  int row, col;
  int idx_row_col;
  for(row=0; row < num_rows; row++){
    for(col=0; col < num_cols; col++){
      idx_row_col = row * num_cols + col;
      matrix_ptr[idx_row_col] = row;
    }
  }
  /* Show the matrix */
  for(row=0; row < num_rows; row++){
    for(col=0; col < num_cols; col++){
      idx_row_col = row * num_cols + col;
      printf("[%d][%d]=%d | ", row, col,
          matrix_ptr[idx_row_col]);
    }
    printf("\n");
  }
  free(matrix_ptr);
   return 0;
}
```

Figura 13 – Preenchimento de uma matriz dinâmica com notação unidimensional

Embora o código esteja funcional, a necessidade de conversão da notação bidimensional para a notação unidimensional torna o código menos claro. A pergunta é pois: consegue-se usar a notação bidimensional com uma matriz dinâmica? Sim. Contudo, para tal é necessário proceder a uma estrutura auxiliar, nomeadamente um vetor de ponteiros. O número de elementos deste ponteiro deve ser igual ao número de linhas da matriz, estando cada ponteiro a apontar para a respetiva linha. Assim, o ponteiro de índice [0] aponta para o início da 1ª linha, o ponteiro de índice [1] aponta para o início da 2ª linha e assim sucessivamente. O uso de uma matriz dinâmica de inteiros com suporte para notação bidimensional é mostrado no código seguinte.

```
#include <stdio.h>
#include <stdlib.h>
/**
 * Allocates a matrix num_rows and num_cols and treats it as
 * a two dimension structure
 */
int main(int argc, char* argv[]) {
  if( argc != 3 ){
    fprintf(stderr, "expecting two arguments\n");
    fprintf(stderr, "%s <num_rows> <num_cols>\n", argv[0]);
    exit(1);
  }

  int num_rows = atoi(argv[1]);
  if( num_rows <= 0 ){
    fprintf(stderr, "Expecting a positive number for <num_rows>"
```

```
        " (got '%s')\n", argv[1]);
    exit(2);
  }
  int num_cols = atoi(argv[2]);
  if( num_cols <= 0 ){
    fprintf(stderr, "Expecting a positive number for <num_cols>"
        " (got '%s')\n", argv[2]);
    exit(3);
  }

  /* Allocate the memory */
  size_t mem_len = num_rows * num_cols * sizeof(int);
  int *matrix_ptr = malloc(mem_len);
  if( matrix_ptr == NULL ){
    fprintf(stderr, "Can't allocate %zu bytes for %d x %d
matrix\n",
        mem_len, num_rows, num_cols);
    exit(4);
  }

  /* Allocate the memory for the auxiliary vector of pointers */
  size_t vect_len = sizeof(int*) * num_rows;
  int **matrix_vect_ptr = (int**) malloc( vect_len );
  if( matrix_vect_ptr == NULL ){
    fprintf(stderr, "Can't allocate %zu bytes for %d vector\n",
        mem_len, num_rows);
    exit(5);
  }
  int row, col;
  /* Create the connection from the vector to the main block */
  for(row=0;row < num_rows; row++){
    matrix_vect_ptr[row] = &(matrix_ptr[row*num_cols]);
  }

  /* Iterate over the matrix, with the [row][col] notation */
  for(row=0; row < num_rows; row++){
    for(col=0; col < num_cols; col++){
      matrix_vect_ptr[row][col] = row * num_cols + col;
    }
  }
  /* Show the matrix */
  for(row=0; row < num_rows; row++){
    for(col=0; col < num_cols; col++){
      printf("[%d][%d]=%d | ", row, col,
          matrix_vect_ptr[row][col]);
    }
    printf("\n");
  }
   free(matrix_vect_ptr);
   free(matrix_ptr);
   return 0;
}
```

Figura 14 – Preenchimento de uma matriz dinâmica com notação didimensional

4. Exercícios

Na resolução dos exercícios utilize as macros MALLOC e FREE para o auxiliar nas operações de gestão de memória. As macros encontram-se definidas no ficheiro "memory.h" e as respetivas funções no ficheiro "memory.c". Doravante deverá utilizar estas macros para as operações de alocação e libertação de memória. Exemplo do seu uso:

```
  int *matrix_ptr = MALLOC(mem_len);
  if( matrix_ptr == NULL ){
    exit(4);
  }
  ...
  FREE(matrix_ptr);
}
```

A macro FREE possui como vantagem adicional a atribuição do valor NULL ao ponteiro libertado.

1. Com base no template de projeto, elabore o programa **matrix** que permite o teste de um conjunto de funções para a multiplicação de matrizes criadas dinamicamente. As funções relacionadas com matrizes deverão ser colocadas num módulo composto pelos ficheiros **matrix.c** e **matrix.h**. O ficheiro **main.c** irá conter o código para testar as funções desenvolvidas nas alíneas seguintes:

a) Adicione ao projeto a função **matrix_new**. Como o nome sugere, esta função deve criar uma matriz dinâmica de *float* de *n_rows* por *n_cols*. A função deve devolver um vetor de ponteiros que permite a manipulação com notação bidimensional da matriz. Passos para a resolução da alínea (válido para as restantes alíneas):

- Definição do protótipo da função (ficheiro **matrix.h**)
- Implementação do código da função (ficheiro **matrix.c**)
- Elabore código de teste à função (ficheiro **main.c**)

b) Adicione ao projeto a função **matrix_delete** que deve libertar convenientemente a memória alocada pela matriz devolvida pela função da alínea anterior;

c) Adicione ao projeto a função **matrix_fill** que deve preencher a matriz dinâmica passada por parâmetro com um determinado valor;

d) Adicione ao projeto a função **matrix_print** que deve escrever para a saída padrão os valores da matriz dinâmica passada por parâmetro;

e) Adicione ao projeto a função **matrix_mul** que deve receber por parâmetro duas matrizes e efetuar a sua multiplicação. Note que o resultado da multiplicação é constituído por 3 valores: a matriz resultante, o número de linhas e o número de colunas da matriz resultante. A multiplicação só deverá ser realizada se as matrizes forem compatíveis;

f) Recorrendo às funções definidas nas alíneas anteriores, altere a função main a fim de proceder à multiplicação de duas matrizes (compatíveis) cujas respetivas dimensões são definidas por parâmetros passados na linha de comando. Os valores das matrizes a multiplicar deve ser 2.0 para a matriz da esquerda e 4.0 para a matriz da direita.

2. Resolva o exercício anterior, utilizando uma estrutura que permite representar de forma completa uma matriz. Compare as duas resoluções.

5. Bibliografia

"Understanding and Using C Pointers - Core Techniques for Memory Management", Richard M. Reese, O'Reilly, 2013.