BCC202 - Estruturas de Dados I

Aula 03: Alocação Dinâmica de Memória

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: **silvap@ufop.edu.br**

Conteúdo

Uso de memória

Visão Geral

Informalmente, podemos dizer que existem três maneiras de reservar espaço de memória para o armazenamento de informações.

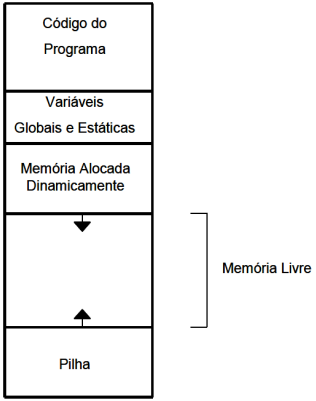
- Uso de variáveis globais e estáticas o espaço existe enquanto o programa estiver sendo executado.
- Variáveis locais (Pilha) - o espaço existe enquanto a função que declarou a variável estiver sendo executado, sendo liberado para outros usos quando a execução da função termina.
- Requisitando ao sistema em tempo de execução um espaço de memória de determinado tamanho - o espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado.

Em geral, a memória utilizada por um programa de computador é dividida em:

- Segmento de Código
- Segmento de Dados
- *Heap*
- Pilha (*stack*)

Ilustração: Uso de Memória pelo Sistema Operacional

A seguir, um esquema didático que ilustra a distribuição de memória, feita pelo sistema operacional, para um programa.



Segmento de Código

É a parte da memória que armazena o código de máquina do programa.

- É estático em tamanho e conteúdo (de acordo com o executável).
- É somente leitura.
  - As instruções do programa compilado e em execução não podem ser alteradas.

Segmento de dados

É a parte da memória que armazena as variáveis globais e as variáveis estáticas inicializadas no código do programa.

- O tamanho do segmento é calculado de acordo com os valores das variáveis definidas.
- O acesso é de leitura e escrita.
- Os valores das variáveis neste segmento podem ser alterados durante a execução do programa.

Pilha (*Stack*) - Descrição

Cada vez que uma função é chamada, o sistema operacional reserva o espaço necessário para as variáveis locais da função (incluindo seus parâmetros).

- Usa a estratégia LIFO (do inglês, *last-in-first-out*) para gerenciar a entrada/saída de dados na memória.
- Espaço pertence a pilha de execução e, quando a função termina, o espaço é desempilhado.
- Acesso é de leitura e escrita.
- O tamanho da Pilha é variável e depende do sistema operacional e compilador utilizados.

Erro Utilizar mais memória Pilha do que disponível provoca um erro de execução: *stack overflow*, o programa é abortado com erro.

Tempo de vida das variáveis - Exemplo 1

```c
#include <stdio.h>
void quad(int n) {
  n = n * n;
  printf("n = %d\n", n);
}

int main() {
  int n = 3;
  quad(n);
  printf("n = %d\n", n);
  return 0;
}
```

Quais valores serão impressos?

Alocação Automática

Ocorre quando são declaradas variáveis locais e parâmetros de funções. O espaço para a alocação dessas variáveis é reservado quando a função é invocada, e liberado quando a função termina.

Tempo de vida das variáveis - Exemplo 2

```
1  #include <stdio.h>
2  void quad(int n) {
3      n = n * n;
4      printf("n = %d\n", n);
5  }
6
7  int main() {
8    int n;
9    scanf("%d", &n);
10   if(n > 10) {
11     int x = 10;
12     quad(x);
13   } else
14       quad(n);
15   printf("n = %d\n", n);
16   return 0;
17 }
```

Quais valores serão impressos?

Heap - Memória alocada dinamicamente

É um espaço reservado para alocação dinâmica de memória dos programas.

- Memória alocada dinamicamente pode ser usada e liberada a qualquer momento.
- A linguagem C fornece funções próprias para lidar com (des)alocação dinâmica de memória.
- Acesso é de leitura e escrita.
- Essencial liberar todos os espaços alocados dinamicamente antes da finalização do programa.

Alocação Dinâmica

Conceitos de ponteiros e memória *heap*

Ponteiros

- Variáveis alocadas dinamicamente são chamadas de ponteiros ou apontadores (*pointers*), pois armazenam o endereço de memória de uma variável.
  - Número inteiro (32 ou 64 bits) indicando um endereço de memória.

Memória *Heap*

- A memória alocada dinamicamente faz parte de uma área da memória chamada heap.
  - Basicamente, o programa aloca e desaloca porções de memória do *heap* durante a execução.

Liberação de memória

- A memória deve ser liberada após o término de seu uso.
- Este trabalho deve ser feito por quem fez a alocação explicitamente (ao contrário das variáveis alocadas automaticamente).

Funções da Biblioteca Padrão

- Existem funções, presentes na biblioteca padrão stdlib, que permitem alocar e liberar memória dinamicamente.

Alocar A função para alocar memória é *malloc*(...).

- Recebe como parâmetro o número de *bytes* que se deseja alocar.
- Retorna o endereço inicial da área de memória alocada.

Liberar A função para liberar um espaço de memória alocado dinamicamente é *free*(...).

- Recebe um endereço de memória que tenha sido alocado dinamicamente.
- O espaço de memória depois de liberado não **deve** ser acessado.

malloc

Para ficarmos independentes de compiladores e máquinas, usamos o operador *sizeof*(...). Assim, teríamos a seguinte sintaxe:

sizeof(...)

```
1  <tipo>* nome_variavel = malloc(sizeof(<tipo>));
2
```

malloc

- A função *malloc* é usada para alocar espaço para armazenarmos valores de qualquer tipo.
- *malloc* retorna um ponteiro genérico, para um tipo qualquer, representado por *void*$*$.
- é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (*cast*).

Assim, teríamos a seguinte sintaxe:

cast

```
1  <tipo>* nome_variavel = (<tipo>*) malloc(sizeof(<tipo>));
```

Exemplo

```
1  ...
2  int* var;
3  var = (int*) malloc(sizeof(int));
4  ...
5
```

Alocação Dinâmica de Vetores

A seguir, a sintaxe para alocação dinâmica de um vetor:

Sintaxe

```
1  ...
2  <tipo>* nome_variavel;
3  nome_variavel = (<tipo>*) malloc(<n>*sizeof(<tipo>));
4  /*n eh o numero de elementos do vetor*/
5  ...
6
```

Exemplo

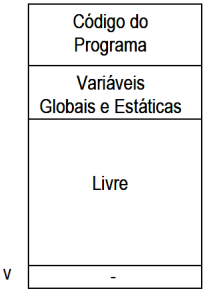
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main ( void ) {
5    int *v;
6    v = (int*) malloc(10*sizeof(int));
7    if(v==NULL) { /* Qual a relevancia desta verificação?*/
8      printf("Memoria insuficiente.\n");
9      return 1;
10   ...
11   return 0;
12 }
13
```

Ilustração: Alocação Dinâmica de Memória

```
1   ...
2   int* v = (int *) malloc(10*sizeof(int));
3   ...
```

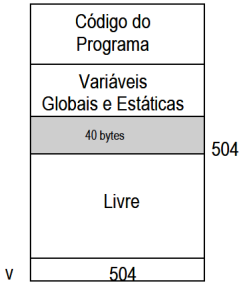
1 - Declaração: int *v

Abre-se espaço na pilha para o ponteiro (variável local)



2 - Comando: v = (int *) malloc (10*sizeof(int))

Reserva espaço de memória da área livre e atribui endereço à variável



free

- Para liberar um espaço de memória alocado dinamicamente, usamos a função *free*, que tem a sintaxe a seguir:

```
1 <tipo>* nome_variavel = (<tipo>*)malloc(sizeof(<tipo>));
2 free(nome_variavel);
3 nome_variavel = NULL; /* Seria necessário? */
```

Memory Leak

Quando a memória alocada dinamicamente não é liberada, há *"vazamento de memória"* ou *memory leak*.

- Em programas que manipulam grandes quantidades de dados, a memória pode se esgotar.

Exemplo

```
1  ...
2  int* var;
3  var = (int*) malloc(sizeof(int));
4  free(var);
5    ...
6
```

Alocação (Liberação) Dinâmica de um Vetor

Exemplo

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

int main ( void )
{
  int n;
  float *v;
  scanf("%d", &n);
  v = (float*) malloc(n*sizeof(float)); /* alocação dinamica */
  if (v==NULL) {
    printf("Memoria insuficiente.\n");
    return 1;
  }
  for (int i = 0; i < n; i++)
    scanf("%f", &v[i]);
  free(v); /* libera memória */
  return 0;
}
```

realloc

A linguagem C oferece ainda um mecanismo para re-alocarmos um vetor dinamicamente.

- Em tempo de execução, podemos verificar que a dimensão inicialmente escolhida para um vetor tornou-se insuficiente (ou excessivamente grande), necessitando um redimensionamento.
- A função *realloc* da *stdlib* nos permite re-alocar um vetor, preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação.

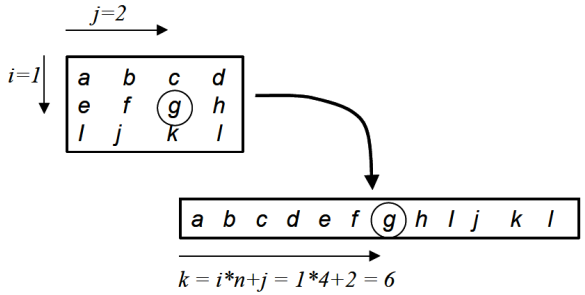
Sintaxe:

```
1  ...
2  /*n, é o novo tamanho do vetor.*/
3  nome_var= (<tipo>*) realloc(nome_var, n*sizeof(<tipo>));
4  ...
```

Matriz representada por um vetor simples

A matriz pode ser representada por um vetor simples.

- As primeiras posições do vetor para armazenar os elementos da primeira linha,
- seguidos dos elementos da segunda linha, e assim por diante.



Matriz Representada por um vetor simples

Com esta estratégia, a alocação da matriz recai numa alocação de vetor que tem $nl * nc$ elementos, onde nl e nc representam as dimensões da matriz.

Exemplo

```
1  float *mat; /* matriz representada por um vetor */
2  ...
3  // nl, nc -> número de linhas e número de colunas
4  mat = (float*) malloc(nl * nc * sizeof(float));
5  ...
6
```

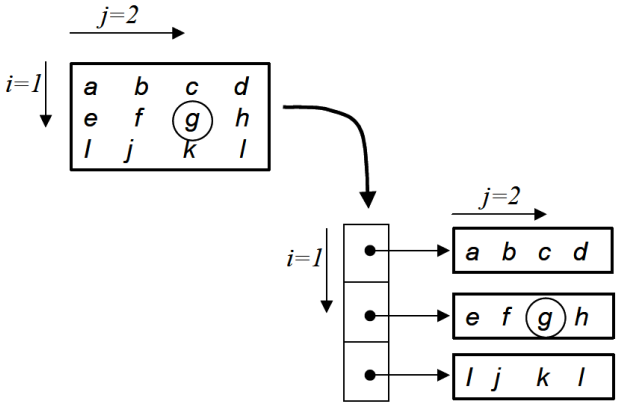
Necessário usar a notação v[i * n + j] para acessar os elementos.

Matriz Representada por um Vetor de Ponteiros

- Cada linha da matriz é representada por um vetor independente.
- A matriz é então representada por um vetor de vetores, ou vetor de ponteiros.
- Cada elemento armazena o endereço do primeiro elemento de cada linha.

Matriz Representada por um Vetor de Ponteiros

A figura a seguir ilustra o arranjo da memória nesta estratégia.



Matriz Representada por um Vetor de Ponteiros

A alocação da matriz agora é mais elaborada, conforme codificado a seguir.

```
1  ...
2  /* matriz representada por um vetor de ponteiros */
3  float **mat;
4  ...
5  mat = (float**) malloc(m*sizeof(float*));
6  for (int i=0; i<m; i++)
7    /*cada linha é um vetor (ponteiro)*/
8    mat[i] = (float*) malloc(n*sizeof(float));
9  ...
```

Como é codificada a liberação de memória da matriz alocada dessa forma?

Liberação de Memória da Matriz (Vetor de Ponteiros)

A liberação do espaço de memória ocupado pela matriz também exige a construção de um laço.

- "Liberar as partes para depois liberar o todo".

```
1  float **mat;
2  ...// alocar a matriz dinamicamente
3  for (i=0; i<m; i++)
4    free(mat[i]);
5  free(mat);
6  ...
```

Alocação Dinâmica de Tipos Estruturados

Considere um programa que manipule pontos no plano cartesiano, sendo cada ponto formado por coordenadas x e y.

Definição de um novo tipo Ponto

```
1   ...
2   typedef struct ponto Ponto; /*definição do novo tipo Ponto*/
3   struct ponto{
4     int x;
5     int y;
6   };
7   ...
8   /* forma alternativa
9   typedef struct{
10    int x;
11    int y;
12  }Ponto;
13  */
14
```

Alocação Dinâmica de Tipos Estruturados

Alocando variáveis do tipo *Ponto* dinamicamente:

```
1  int x, y;
2  ...
3  Ponto * pt = (Ponto*) malloc(sizeof(Ponto));/* alocacao dinamica */
4  if(pt == NULL){ /* Boa prática! */
5    printf("Memória Insuficiente");
6    exit(1);
7  }
8  ...
9  /*inicialização dos elementos da estrutura*/
10 pt->x = x;
11 pt->y = y;
12 ...
13 free(pt); /* liberar memória */
14 ...
```

- Considerando que precisamos alocar diversas variáveis do tipo Ponto, como refatorar essa implementação?

Alocação Dinâmica de Tipos Estruturados

Podemos definir uma função para alocar dinamicamente um ponto e inicializar seus elementos.

```
1   ...
2   /*Protótipo da Função de Alocação*/
3   Ponto* alocarPonto(int, int); /*retorna um ponto devidamente alocado*/
4   ...
5   /*Implementação da Função*/
6   Ponto* alocarPonto(int x, int y){
7     Ponto * pt = (Ponto*) malloc(sizeof(Ponto));
8     if(pt == NULL){/*Boa prática!*/
9       printf("Memória Insuficiente");
10      exit(1);
11    }
12    pt->x = x;
13    pt->y = y;
14    return pt;
15  } /*retorna um ponto devidamente alocado*/
16    ...
```

O que deve ser feito com toda memória alocada dinamicamente?

Alocação Dinâmica de Tipos Estruturados

De maneira análoga, podemos definir uma função para liberar dinamicamente a memória alocada para uma variável do tipo Ponto.

Função para Liberar a Memória Dinamicamente Alocada

```
1  ...
2  /*Protótipo da Função de Liberação de Memória*/
3  void liberarPonto(Ponto*);
4  ...
5  Ponto* liberarPonto(Ponto* pt){
6    free(pt); /*liberando a memória alocada em tempo de execução*/
7            return pt;
8  }
9  ...
```

Alocação Dinâmica de Tipos Estruturados

Agora, podemos reutilizar a função toda que quisermos alocar variáveis do tipo *Ponto*.

Reutilizando a Função

```
1   ...
2   int main() {
3     int x, y;
4     Ponto* ponto_central;
5     ...
6     ponto_central = alocarPonto(x,y);
7     ....
8     ponto_central = liberarPonto(ponto_central);
9     ....
10    return 0;
11  }
12
```

Modularização

Visão Geral

- *ponto.h*: protótipos das funções que manipulam variáveis do tipo *Ponto*.
- *ponto.c*: implementação dos protótipos das funções que manipulam variáveis do tipo *Ponto*.
- *main.c*: reutiliza as funções que manipulam variáveis do tipo *Ponto*.

Mais detalhes, na próxima aula.

Erros Comuns

Erros Comuns

- Esquecer de alocar memória e tentar acessar o conteúdo da variável.
- Copiar o valor do ponteiro ao invés do valor da variável apontada.
- Esquecer de desalocar memória.
  - A memória será desalocada apenas no encerramento do programa, o que pode ser um grande problema em *loops*.

    Ocasiona "desperdício de memória", que pode causar falha do sistema.
- Tentar acessar o conteúdo da variável depois de desalocá-la.

Exemplo I

Considero o tipo *Ponto* previamente definido.

```
1   ...
2   int main() {
3     Ponto* pt_ponto;
4     Ponto ponto;
5     ponto.x = 2;
6     ponto.y = 4;
7     printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
8     pt_ponto = liberarPonto(pt_ponto);
9     return 0;
10  }
11
```

Onde está o erro? Quais são as possíveis soluções?

Exemplo I - Solução 1

Considero o tipo *Ponto* previamente definido.

```
1   ...
2   int main() {
3     Ponto* pt_ponto;
4     Ponto ponto;
5     ponto.x = 2;
6     ponto.y = 4;
7     pt_ponto = &ponto;
8     printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
9     return 0;
10  }
11
```

Passar o endereço de uma variável já alocada para a variável de ponteiro.

Exemplo I - Solução 2

Considero o tipo *Ponto* previamente definido.

```
1   ...
2   int main() {
3     Ponto* pt_ponto = (Ponto*) malloc(sizeof(Ponto));
4     pt_ponto->x = 8;
5     pt_ponto->y = 16;
6     printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
7             free(pt_ponto);
8     return 0;
9   }
10
11
```

Alocar a variável de ponteiro dinamicamente.

Exemplo I - Solução 3

Considero o tipo *Ponto* previamente definido.

```
1  ...
2  int main() {
3    Ponto* pt_ponto = alocarPonto(8, 16);
4    printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
5            pt_ponto = liberarPonto(pt_ponto);
6    return 0;
7  }
8
9
```

Alocar a variável de ponteiro dinamicamente.

Exemplo II

Considero o tipo *Ponto* previamente definido.

```
1   ...
2   int main() {
3     Ponto* pt_ponto;
4     pt_ponto = (Ponto*) malloc(sizeof(Ponto));
5     pt_ponto->x = 8;
6     pt_ponto.y = 16;
7     printf("(%d,%d)\n", pt_ponto.x, pt_ponto->y);
8     pt_ponto = liberarPonto(pt_ponto);
9     return 0;
10  }
11
```

Onde estão os erros?

Exemplo II - Solução

Considero o tipo *Ponto* previamente definido.

```
1   ...
2   int main() {
3     Ponto* pt_ponto;
4     pt_ponto = alocarPonto(0, 0);
5     pt_ponto->x = 8;
6     pt_ponto->y = 16;
7     printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
8     pt_ponto = liberarPonto(pt_ponto);
9     return 0;
10  }
11
```

Exemplo III

Considero o tipo *Ponto* previamente definido.

```
1   /* protótipo para função alocar ponto*/
2   void alocarPonto_falha(Ponto*, int, int);
3
4   /*implementação do protótipo*/
5   void alocarPonto_falha(Ponto* pt, int x, int y) {
6     pt = (Ponto*) malloc(sizeof(Ponto));
7     pt->x = x;
8     pt->y = y;
9   }
10
11  int main() {
12    Ponto* pt_ponto;
13    alocarPonto_falha(pt_ponto, 8,8);
14    printf("(%d,%d)\n", pt_ponto->x, pt_ponto->y);
15    pt_ponto = liberarPonto(pt_ponto);
16    return 0;
17  }
18
```

Exemplo III - Solução

- Fazer a passagem da variável de ponteiro por referência.
- Retornar uma variável de ponteiro devidamente alocada (Slide 34).

Exemplo III - Solução

Passagem por Referência para Alocação de Memória

```
1  /* Protótipo da Função Alternativa de Alocação */
2  void alocarPontoAlt(Ponto**, int, int);
3
4  /* Implementação da Função */
5  void alocarPontoAlt(Ponto** pt, int x, int y) {
6    *pt = (Ponto*) malloc(sizeof(Ponto));
7    (*pt)->x = x;
8    (*pt)->y = y;
9  }
10
11 int main() {
12   Ponto* pt_ponto;
13   alocarPonto_falha(&pt_ponto, 8, 8);
14   printf("(%d,%d)\n", pt_ponto->x,pt_ponto->y);
15   pt_ponto = liberarPonto(pt_ponto);
16   return 0;
17 }
```

Exemplo IV

```
1   ...
2   int main() {
3     int* var = (int*) malloc(sizeof(int));
4     var[2] = 4;
5     ....
6     free(var);
7     ....
8     return 0;
9   }
10
```

Onde está o erro?

Exemplo IV

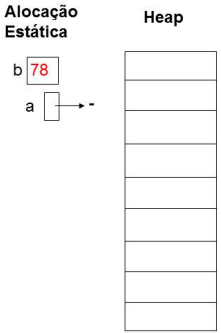
Considerações

"Ver um ponteiro NÃO necessariamente equivale a ver um vetor".

```
1  int* var = (int*) malloc(sizeof(int));
2  int* vetor_int = (int*) malloc (4 * sizeof(int));
3  int** vetor_pt_int = (int**) malloc(4*sizeof(*int));
4
```

Exemplo V

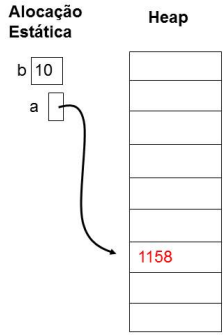
```
1    int *a, b;
2    :
3    :
4    b = 10;
5    a = (int*) malloc(sizeof(int));
6    *a = 20;
7    a = &b;
8    free(a);
```



Ao executar a linha *1*, **a** e **b** estarão em um espaço de endereçamento estático, e seu valor será aquele anteriormente armazenado na memória.

Exemplo V

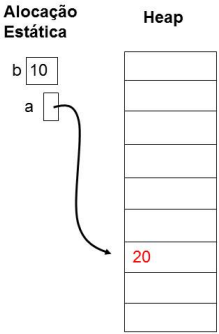
```
1    int *a, b;
2    :
3    :
4    b = 10;
5    a = (int*) malloc(sizeof(int));
6    *a = 20;
7    a = &b;
8    free(a);
```



Ao executar as linhas *4* e *5* o conteúdo de **b** passa a ser 10 e **a** apontará para um endereço do espaço dinâmico, que conterá o valor armazenado anteriormente na memória.

Exemplo V

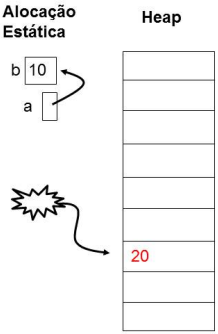
```
1    int *a, b;
2    :
3    :
4    b = 10;
5    a = (int*) malloc(sizeof(int));
6    *a = 20;
7    a = &b;
8    free(a);
```



Ao executar a linha *6* o endereço para onde **a** aponta passa a armazenar o valor 20.

Exemplo V

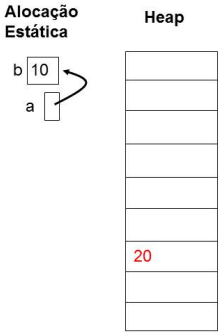
```
1    int *a, b;
2    :
3    :
4    b = 10;
5    a = (int*) malloc(sizeof(int));
6    *a = 20;
7    a = &b;
8    free(a);
```



Na linha *7*, **a** passa a apontar para **b**. Agora ninguém referencia mais o endereço apontado anteriormente por **a**. A utilização da função `free` na linha *8*, está incorreto. O objetivo seria liberar a memória que foi inicialmente alocada para **a**. No entanto, **a** não aponta mais para este endereço de memória.

Exemplo V

```
1    int *a, b;
2    :
3    :
4    b = 10;
5    a = (int*) malloc(sizeof(int));
6    *a = 20;
7    a = &b;
8    free(a);
```



Para corrigir este problema, as linhas *7* e *8* deveriam ser invertidas. Primeiro a memória alocada dinamicamente é desalocada e depois `a` passa a apontar para `b`.

Considerações Finais

Conclusão

- Implementar um programa com apenas alocação automática (ou estática) pode ser ineficiente em termos de uso de memória.
- Toda memória alocada em tempo de execução deve ser também liberada.
- Toda variável de ponteiro deve receber de fato um endereço válido de memória antes de ser acessada.
- Nem toda variável de ponteiro refere-se a um vetor.

Tipo Abstrato de Dado (TAD).

Bibliografia

Bibliografia

Os conteúdos deste material, incluindo figuras, textos e códigos, foram extraídos ou adaptados do livro-texto indicado a seguir:


Celes, Waldemar and Cerqueira, Renato and Rangel, José
Introdução a Estruturas de Dados com Técnicas de Programação em C.
Elsevier Brasil, 2016.
ISBN 978-85-352-8345-7.

Exercícios

Exercício 01

Implemente uma função que recebe um vetor de números reais e tenha como valor de retorno um novo vetor, alocado dinamicamente, com os elementos do vetor original em ordem inversa. A função deve ter como retorno o valor do ponteiro alocado, conforme protótipo a seguir:

```
1 float* reverso(int n, float* v);
```

Faça uma função *main*() para testar sua função. Não esqueça de liberar a memória alocada em tempo de execução.