

Ponteiros (parte 2)

Prof. Paulo Henrique Pisani

outubro/2021

Tópicos

- Alocação estática vs Alocação dinâmica (vetores)
- Aritmética de ponteiros
- Vetores como parâmetro e como retorno de função
- Buffer overflow

Alocação estática vs Alocação dinâmica

(vetores)

Alocação dinâmica

- Para alocar memória dinamicamente, podemos usar o **malloc**:

```
void* malloc( size_t size );
```

- Para liberar a memória, usamos o **free**:

```
void free( void* ptr );
```

Alocação dinâmica

- Para alocar memória, podemos usar o **malloc**:

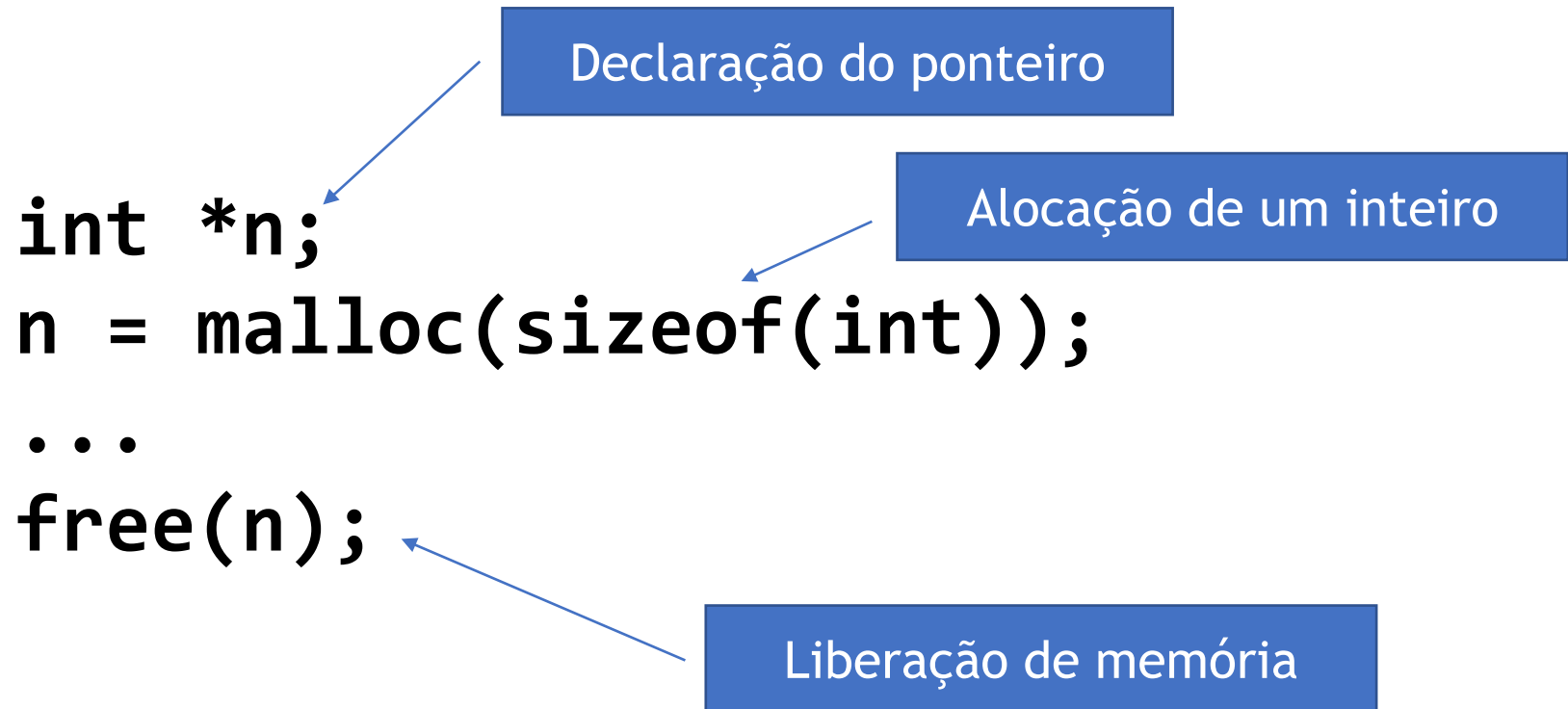
```
void* malloc( size_t size );
```

- Para liberar a memória, usamos o **free**:

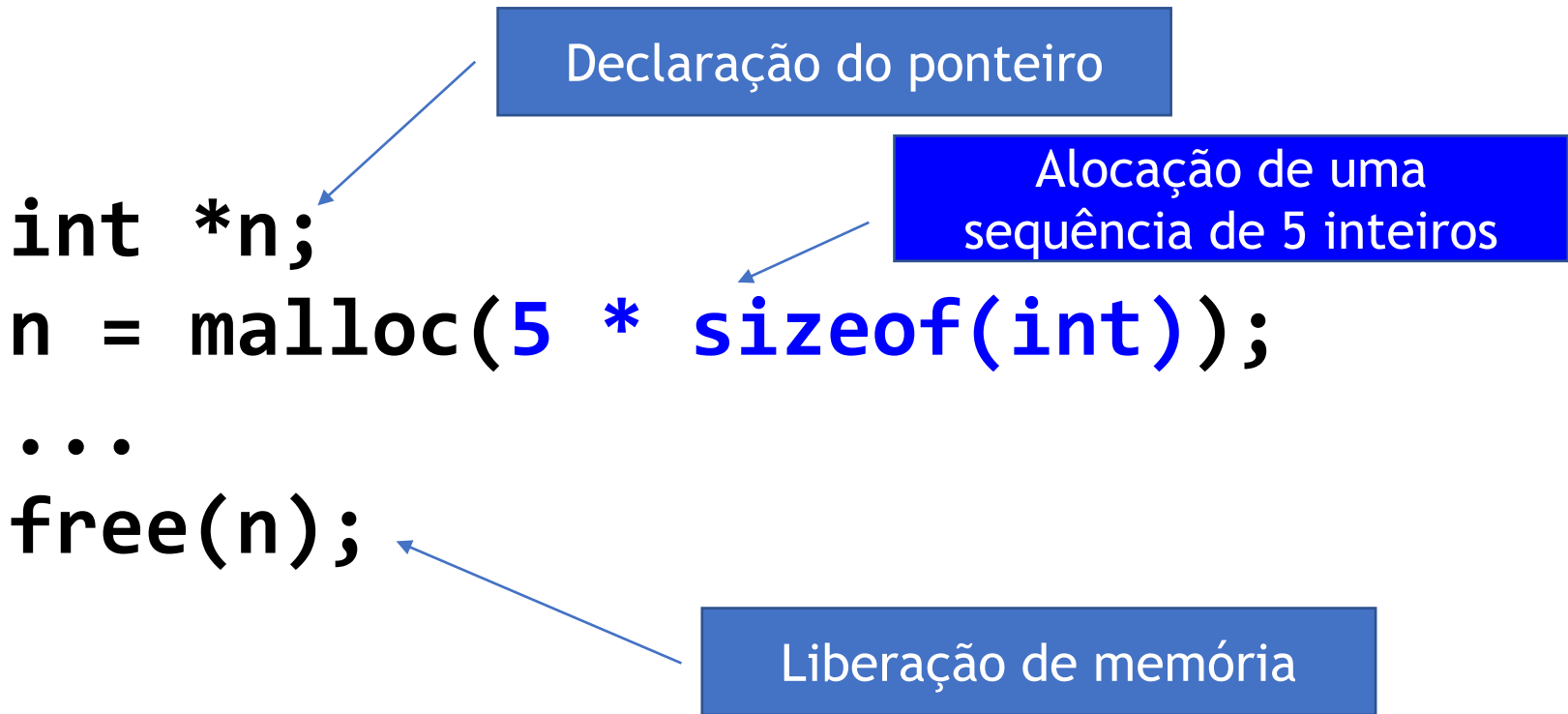
```
void free( void* ptr );
```

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

Alocação dinâmica



Alocação dinâmica (vetores)



Alocação dinâmica (vetores)

```
#include<stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    int vetor[n];

    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    return 0;
}
```

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

int main() {
    int n;
    scanf("%d", &n);

    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    free(vetor);

    return 0;
}
```


Lembre-se de sempre liberar a memória alocada!

```
int *n;  
n = malloc(sizeof(int));  
...  
free(n);
```

`free(n);`

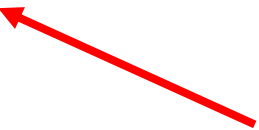
```
#include<stdio.h>
#include<stdlib.h>
```

```
int main() {
    int n;
    scanf("%d", &n);

    int *vetor = malloc(sizeof(int) * n);
    if (vetor == NULL) {
        printf("Erro na alocao.\n");
        return -1;
    }
    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    free(vetor);

    return 0;
}
```



Importante: Não há garantia que a memória seja alocada! Em caso de erro, é retornado o ponteiro NULL (internamente é o valor zero)

Aritmética de ponteiros

Aritmética de ponteiros

- Podemos utilizar alguns operadores aritméticos sobre ponteiros:

a) ++

b) --

c) +

d) -

Aritmética de ponteiros

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

```
int main() {
    char *ptr = malloc(sizeof(char));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00740D00

?

Aritmética de ponteiros

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

```
int main() {
    char *ptr = malloc(sizeof(char));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00740D00
00740D01

Incrementou apenas uma unidade!



Aritmética de ponteiros

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

```
int main() {
    int *ptr = malloc(sizeof(int));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00750D00

?

Aritmética de ponteiros

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

```
int main() {
    int *ptr = malloc(sizeof(int));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00750D00
00750D04

Incrementou 4 unidades!



Aritmética de ponteiros

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

int main() {
    int *ptr1 = malloc(sizeof(int));
    char *ptr2 = malloc(sizeof(char));
    double *ptr3 = malloc(sizeof(double));

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    ptr1 = ptr1 + 3;
    ptr2 = ptr2 + 3;
    ptr3 = ptr3 + 3;

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    return 0;
}
```

O que será
impresso no
segundo printf?

Saída

00BF0D00 00BF0D20 00BF0D30
?

Aritmética de ponteiros

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

int main() {
    int *ptr1 = malloc(sizeof(int));
    char *ptr2 = malloc(sizeof(char));
    double *ptr3 = malloc(sizeof(double));

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    ptr1 = ptr1 + 3;
    ptr2 = ptr2 + 3;
    ptr3 = ptr3 + 3;

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    return 0;
}
```

O que será
impresso no
segundo printf?

Saída

```
00BF0D00 00BF0D20 00BF0D30
00BF0D0C 00BF0D23 00BF0D48
```

Aritmética de ponteiros

- O efeito das operações aritméticas sobre os ponteiros depende de como foram declarados!


```
int *ptr1 = malloc(sizeof(int));  
char *ptr2 = malloc(sizeof(char));  
double *ptr3 = malloc(sizeof(double));
```


```
ptr1 = ptr1 + 1; —————> Incrementa 4 (int = 4 bytes)  
ptr2 = ptr2 + 1; —————> Incrementa 1 (char = 1 byte)  
ptr3 = ptr3 + 1; —————> Incrementa 8 (double = 8 bytes)
```

Usando aritmética de ponteiros em vetores

- Podemos usar aritmética de vetores para acessar posições de um vetor:

```
int *vetor = malloc(sizeof(int) * 10);
```

```
*(vetor + 0) = 80            vetor[0] = 80
```

```
*(vetor + 4) = 507        vetor[4] = 507
```

Usando aritmética de ponteiros em vetores

- Podemos usar aritmética de vetores para acessar posições de um vetor:

```
int *vetor = malloc(sizeof(int) * 10);
```

```
*(vetor + 0) = 80   $\longleftrightarrow$   vetor[0] = 80  
*(vetor + 4) = 507  $\longleftrightarrow$   vetor[4] = 507
```

Importante! Coloque parênteses! Assim a aritmética de ponteiros é realizada antes de dereferenciar com *

Usando aritmética de ponteiros em vetores

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

int main() {
    int *ptr = malloc(sizeof(int) * 10);

    int i;
    for (i = 0; i < 10; i++)
        *(ptr + i) = i*i;

    for (i = 0; i < 10; i++)
        printf("%d ", ptr[i]);
    printf("\n");
    free(ptr);

    return 0;
}
```

O que será impresso?

Usando aritmética de ponteiros em vetores

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

int main() {
    int *ptr = malloc(sizeof(int) * 10);

    int i;
    for (i = 0; i < 10; i++)
        *(ptr + i) = i*i;

    for (i = 0; i < 10; i++)
        printf("%d ", ptr[i]);
    printf("\n");
    free(ptr);


    return 0;
}
```

O que será impresso?

0 1 4 9 16 25 36 49 64 81

Vetores como
parâmetro e como
retorno de função

Passagem de vetor como parâmetro

- Já vimos que vetores são passados por referência: 

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
           v[0], v[1], v[2]);

    return 0;
}
```

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

```
#include <stdio.h>

void muda_valor(int *vetor) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Há diferença na saída dos dois programas?

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

```
#include <stdio.h>

void muda_valor(int *vetor) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Há diferença na saída dos dois programas?

```
90
90 507 300
```

```
90
90 507 300
```

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    *vetor = 90;
    *(vetor+1) = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

```
#include <stdio.h>

void muda_valor(int *vetor) {
    *vetor = 90;
    *(vetor+1) = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Há diferença na saída dos dois programas?

```

#include <stdio.h>

void muda_valor(int vetor[]) {
    *vetor = 90;
    *(vetor+1) = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}

```

```

#include <stdio.h>

void muda_valor(int *vetor) {
    *vetor = 90;
    *(vetor+1) = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}

```

Há diferença na saída dos dois programas?

90
90 507 300

90
90 507 300

Retorno de vetor por função

- Qual a melhor forma de retornar um vetor?

```
int[] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

A

```
int[n] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

B

Retorno de vetor por função

- Qual a melhor forma de retornar um vetor?

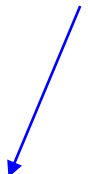
```
int[] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

```
int* cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

Nenhuma das duas formas está correta!

Retorno de vetor por função

- Para retornar um vetor, precisamos retornar seu ponteiro:



```
int* cria_vetor(int n) {  
  
    // Implementacao da funcao  
  
}
```

Retorno de vetor por função

Qual a saída
deste programa?

```
#include <stdio.h>

int* cria_vetor(int n) {
    int vetor[n];

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

Retorno de vetor por função

Função retornou ponteiro para variável local!



Qual a saída deste programa?

Segmentation fault (core dumped)

```
#include <stdio.h>

int* cria_vetor(int n) {
    int vetor[n];

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;


    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

Não retorne ponteiro para variável local!



```
int* cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

Retorno de vetor por função

Qual a saída deste programa?

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

```
int* cria_vetor(int n) {
    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");
    free(v);

    return 0;
}
```

Retorno de vetor por função

Qual a saída deste programa?

1 2 3 4 5

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

```
int* cria_vetor(int n) {
    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");
    free(v);

    return 0;
}
```

Buffer overflow

Cuidados com uso de ponteiros

- É necessário ter cuidado com o uso de ponteiros!
- Quando usamos um vetor, temos que ter atenção aos limites dele!
- Por exemplo, o seguinte código tem comportamento imprevisível quando *i* passa de 9:

```
double vetor[10];  
int i;  
for(i = 0; i <= 10; i++)  
    printf("%d", i);
```



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

```
void funcao_protegida() {
    // Funcao que soh pode ser executada por usuario autorizado
    printf("[FUNCAO_PROTEGIDA]\n");
}
```

```
int main(void) {
    char *senha = malloc(sizeof(char) * 5);
    int *ok = malloc(sizeof(int));
    *ok = 0;
    printf("senha=%p ok=%p\n", senha, ok);

    printf("Digite a senha: \n");
    gets(senha);

    if(strcmp(senha, "1234")) { // retorna 0 se sao strings iguais
        printf ("Senha ERRADA !\n");
    } else {
        printf ("Senha CORRETA !\n");
        *ok = 1;
    }

    if(*ok) funcao_protegida();

    free(senha);
    free(ok);
    return 0;
}
```

Este programa usa o **gets**. Caso gets escreva mais do que 5 valores no vetor senha, o comportamento passa a ser imprevisível!

Buffer overflow

- Neste programa, podemos evitar o buffer overflow usando: **fgets(senha, 5, stdin)** ao invés do **gets**;
- Esse exemplo é interessante para mostrar como alguém poderia explorar um programa mal protegido (veja alguns exemplos de entradas e saída no próximo slide).

Buffer overflow

Veja que quando a senha é grande o suficiente, o endereço de memória alocado para “ok” é alterado para um valor diferente de zero.

A senha está errada, mas a função protegida foi chamada.

```
senha=0x562b94b71260 ok=0x562b94b71280
Digite a senha:
123
Senha ERRADA !
```

```
senha=0x55f9cb095260 ok=0x55f9cb095280
Digite a senha:
1234
Senha CORRETA !
[FUNCAO_PROTEGIDA]
```

```
senha=0x556a89fe7260 ok=0x556a89fe7280
Digite a senha:
12345
Senha ERRADA !
```

```
senha=0x55ca20470260 ok=0x55ca20470280
Digite a senha:
123456789012345678901234567890123
Senha ERRADA !
[FUNCAO_PROTEGIDA]
```

Exercício 1

- Escreva uma função que receba uma string e retorne outra string com todos os caracteres invertidos;
- Por exemplo, para “abcde” deve retornar “edcba”.

Exercício 2

- Escreva uma função que receba um vetor e retorne outro vetor sem o primeiro elemento.

Exercício 3

- Crie um programa que leia um vetor de n números; Este vetor deve ser passado para uma função recursiva que irá contar quantos números são primos.
- Não use colchetes! Portanto, será preciso usar malloc e aritmética de ponteiros.

Bibliografia básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2. ed. Rio de Janeiro, RJ: Campus, 2002.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo, SP: Prentice Hall, 2005.
- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.

Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 1994.
- TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.