

Modularização de Código e Passagem de Parâmetros

## O que são módulos de código?

* Técnica “Dividir para conquistar”, decomposição do problema em partes menores
* Em C: módulos são denominados funções
* Funções pré-definidas em bibliotecas de linguagens de programação:
  + C: função pow() da biblioteca math.h, função strcpy() da biblioteca string.h, etc. (“Biblioteca Padrão do C” 2020)
  + C++: função pow() da biblioteca cmath, funções cin e coutda biblioteca iostream, funções setprecision() e setw() da biblioteca iomanip, etc. (“C Library” 2000)

## Justificativas para modularização de código

* Partes pequenas são mais fáceis de entender, testar e modificar: complexidade reduzida (Forbellone and Eberspächer 2005)
* Evitar duplicação de código
* Permitir reuso de código já desenvolvido e testado (Ascencio and Campos 2007)

## Função

* Seção independente de código
* Designada por um nome, seguindo as regras dos demais identificadores (variáveis, constantes, etc)
* Realiza uma tarefa específica, resolve um subproblema (Forbellone and Eberspächer 2005)
* Opcionais: aceita parâmetros e retorna um valor. A palavra **void** indica que não há retorno ou que a lista de parâmetros é vazia (o termo void indica nulo ou vazio).

**Exemplo de uma função sem parâmetros e sem retorno e sua chamada pelo main():**

**C:**

| 1  2  3  4  5  6  7  8  9  10  11  12 | #**include** <stdio.h> #**include** <stdlib.h>  **void** **soma**(){  **int** x;  x = 5 + 3;  printf("%d", x); }  **int** **main**(){  soma()); } |
| --- | --- |

**C++:**

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #**include** <iostream>  **using** **namespace** std;  void **soma**(){  **int** x;  x = 5 + 3;  cout << x; }  **int** **main**(){  soma();  } |
| --- | --- |

É importante entender que as variáveis utilizadas em uma função são restritas ao escopo da função. Ou seja, a variável x é declarada apenas quando a função soma() é chamada e deixa de existir (a posição de memória é liberada) assim que a execução da função é encerrada. Adicione a instrução

printf("%d", x); ou cout << x;

à função principal (main) e veja a mensagem que o compilador retornará.

## Argumentos de funções

Funções **podem receber valores.** Esses valores são denominados **argumentos** ou **parâmetros** formais da função.

Por meio dos parâmetros, é possível **generalizar** uma função. Considere o exemplo da função soma(). O mesmo resultado é mostrado todas as vezes, pois os valores de x são constantes, atribuídos dentro da função. Para que a função some quaisquer valores, precisamos fornecer esses valores, por meio dos parâmetros (Forbellone and Eberspächer 2005).

Na linguagem C, há duas formas de passagem de parâmetros:

* **Por valor:** uma cópia do valor é fornecida para a função, que não altera o conteúdo da variável de origem do valor. É a forma padrão (exceto para estruturas de armazenamento, como vetores e outras).
* **Por referência:** é fornecido o endereço da variável e a função chamada altera o conteúdo da variável de origem do valor.

Por enquanto, o enfoque é a **passagem de parâmetros por valor**. Veja os códigos:

**C:**

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #**include** <stdio.h> #**include** <stdlib.h>  //Retorna o dobro (int) do valor (int) fornecido por parâmetro **int** **dobro**(**int** valor){  return valor \* 2; }  **int** **main**(){  **int** v;  scanf("%d", &v);  printf("Dobro: %d\n", dobro(v));  } |
| --- | --- |

**C++:**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #**include** <iostream>  **using** **namespace** std;  //Retorna o dobro (int) do valor (int) fornecido por parâmetro **int** **dobro**(**int** valor){  **return** valor \* 2; }  **int** **main**(){  **int** v;  cin >> v;  cout << "Dobro: " << dobro(v) << endl; } |
| --- | --- |

Nesse caso, a função dobro() foi generalizada, de forma a retornar o dobro de qualquer valor fornecido por meio do parâmetro valor.

Um **argumento** é uma das formas de declaração de variáveis **no escopo de uma função**. Ele passa a existir (e é inicializado) **quando a função é chamada** e deixa de existir quando a execução da função se encerra. Seu funcionamento/utilização é o mesmo das demais variáveis locais de um módulo (Ascencio and Campos 2007).

## Funções com retorno de valor

Quando é desejado que uma função retorne um valor, o tipo do retorno deve ser especificado antes do seu nome. Se a palavra void é utilizada, significa que não há valor de retorno.

No entanto, se uma função tem um retorno de tipo especificado (int, float ou qualquer outro tipo), um valor deve ser devolvido, por meio da instrução return. (Medina and Fertig 2006). Quando a instrução return é executada, a execução da função é encerrada e o controle volta ao ponto onde essa função foi chamada. (No código acima, quando a linha 8 é executada, o controle volta à linha 17.)

A instrução return retorna um único valor e é executada uma única vez em uma função, ou seja, uma função retorna apenas um valor.

Se a função retornar um valor por meio da instrução return, o valor de retorno deve ser especificado entre parênteses (parâmetro da instrução) ou ser resultante de uma expressão (lógica ou aritmética) válida. Por exemplo:

| 1  2  3  4 | //Retorna o mínimo (int) entre dois valores (int) fornecidos por parâmetro **int** **min**(**int** a, **int** b){  **return** (a < b? a : b); } |
| --- | --- |

## Chamadas de funções com retorno

O retorno de uma função pode ser armazenado em uma variável, mostrado em uma instrução de saída, utilizado em uma expressão aritmética ou lógica.

Por exemplo, a função min() pode ser chamada das seguintes formas:

| 1  2  3  4 | menor = min(v1, v2);  total\_min += min (v1, v2);  (min(v1, v2) < 0)? menor = 0 : menor = min(v1, v2);  x = pow(min(v1, v2), 2); |
| --- | --- |

Na linha 1, o resultado da função é armazenado na variável menor; na linha 2, faz parte de uma expressão aritmética atribuída à variável total\_min; na linha 3, é avaliado em uma expressão lógica; na linha 4, o resultado é utilizado como parâmetro da função pow().

## Outros exemplos

Seguem exemplos de funções com retorno de tipos não primitivos de dados.

**Exemplo 1**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #**include** <iostream>  **using** **namespace** std;  //Converte as iniciais para maiusculas string **iniciais** (string nome){  **int** i = 0;  **while** (i < nome.length()){  **if** ((nome[i] < 65 || nome[i] > 90) && (i == 0 || nome[i - 1] == ' '))  nome[i] -= 32;  i++;  }  **return** nome; }  **int** **main**(){  string aluno;   getline(cin, aluno); //Chamadas equivalentes:  cout << iniciais(aluno) << endl; //cout << pow (2,2) << endl;  cout << aluno << endl;  aluno = iniciais(aluno); // x = pow (2,2);  cout << aluno << endl; // cout << x;   **return** 0; } |
| --- | --- |
|  |  |

**Exemplo 2**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #**include** <iostream>  **using** **namespace** std;  //Verifica se numero e par **bool** **par** (**int** n){  **return** (n % 2 == 0 ? true : false); }  **int** **main**(){  **int** num;   cin >> num;  **if** (par(num))  cout << "Numero par" << endl;  **else**  cout << "Numero impar" << endl;  **return** 0;  } |
| --- | --- |
|  |  |

quebra de página

## Bibliografia

Ascencio, Ana Fernanda Gomes, and Edilene Aparecida Veneruchi de Campos. 2007. *Fundamentos da programação de computadores: algoritmos, pascal, c/c++ e java*. 2nd ed. São Paulo, SP: Pearson Prentice Hall.

“Biblioteca Padrão do C.” 2020. Biblioteca Padrão do C. https://pt.wikipedia.org/wiki/Biblioteca\_padr%C3%A3o\_do\_C.

“C Library.” 2000. C Library. http://www.cplusplus.com/reference/clibrary/.

Feofiloff, Paulo. 2009. *Algoritmos em Linguagem C*. Rio de Janeiro: Elsevier.

Forbellone, André Luiz V., and Henri F. Eberspächer. 2005. *Lógica de Programação: A construção de algoritmos e estruturas de dados*. 3ª ed. São Paulo: Pearson.

Medina, Marco, and Cristina Fertig. 2006. *Algoritmos e programação: teoria e prática.* São Paulo: Novatec.

## Exercícios

## 

A. Escreva programas em C/C++ que contenham funções (sem parâmetros e sem retorno) com as seguintes funcionalidades:

1. Leia um número inteiro e informe se o número é ou não múltiplo de 4.
2. Leia um número inteiro, verifique e apresente se o número é par ou ímpar.
3. Leia 2 números inteiros positivos e apresente a soma dos n números existentes entre eles (incluindo os limites do intervalo) (Por exemplo, para 5 e 8: 5+6+7+8).
4. Leia o raio de uma esfera e apresente o seu volume (4 π R3/3).
5. Leia um número natural, calcule e apresente o seu fatorial.
6. Leia a base e o expoente (dois valores inteiros maiores ou iguais a zero), calcule e apresente o resultado da potência.   
   Restrição: não é permitido utilizar a função pow().

B. Reescreva as funções, generalizando-as (por meio de parâmetros e retorno).

C. Explique, com suas palavras, os conceitos/significados dos seguintes termos:

1. Funções
2. Argumentos/Parâmetros
3. Tipos de funções
4. void
5. Tipos de passagem de parâmetro (qual o padrão do C/C++?)
6. Retorno de uma função (quais os tipos de retorno?)
7. Calcular a área de um círculo: Dado o raio de um círculo, calcule sua área usando a fórmula π \* r^2.
8. Verificar se um ano é bissexto: Um ano é bissexto se for divisível por 4, mas não por 100, exceto se também for divisível por 400.
9. Converter temperatura de Celsius para Fahrenheit: Dado um valor em graus Celsius, converta-o para Fahrenheit usando a fórmula F = C \* 9/5 + 32.

## Escopo de variáveis

Uma **variável** é válida na função onde é declarada. Quando a execução de outra função é iniciada, um novo espaço de memória é alocado. O **tempo de vida** de uma variável é denominado **escopo da variável**. O escopo denota a abrangência, a visibilidade da variável conforme o módulo em que é declarada (Forbellone and Eberspächer 2005).

Considere as quatro implementações da função soma(). No primeiro caso, toda a funcionalidade está na função principal (main()).

| 1  2  3  4  5  6  7  8 | #**include** <stdio.h> // Primeiro caso: toda a função no main **int** **main**(){  **int** a, b, s;  a = b = 3;  s = a + b;  printf("%d", s); } |
| --- | --- |

No segundo caso, a função soma() não possui parâmetros e não fornece retorno. Nenhuma das variáveis que são declaradas internamente à função soma podem ser acessadas pelo main(). Elas tem **escopo local**, ou seja, existem apenas enquanto a função soma() estiver em execução.

| 1  2  3  4  5  6  7  8  9  10  11  12 | #**include** <stdio.h>  // Segundo caso: função sem retorno e sem parâmetros **void** **soma**(){  **int** a, b, s;  a = b = 3;  s = a + b;  printf("%d", s); }  **int** **main**(){  soma(); } |
| --- | --- |

No terceiro caso, os parâmetros a e b da função soma() se referem a outras posições de memória, bem distintas das posições utilizadas pelas variáveis a e b do main(). Poderiam ter nomes distintos, mas como a área de memória alocada para cada função é diferente, não há problemas em se utilizar o mesmo identificador.

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #**include** <stdio.h>  // Terceiro caso: função sem retorno e com parâmetros **void** **soma**(int a, int b){  **int** s;  s = a + b;  printf("%d", s); }  **int** **main**(){  int a, b;  a = b = 3;  soma(a, b); } |
| --- | --- |

No quarto caso, cabe considerar o retorno da função soma(): um inteiro. É o caso mais adequado para uma função desse tipo: desenvolvida para resolver um problema específico, aceita os parâmetros necessários ao cálculo e retorna o resultado, que pode ser utilizado de diversas formas pelo código que a chamou. Nesse caso, main()está apenas imprimindo o resultado da soma. Vale destacar que a variável s, utilizada pela função soma(), não é reconhecida no main() (dizemos que ela está fora do escopo da função principal).

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #**include** <stdio.h>  // Quarto caso: função com retorno e com parâmetros **int** **soma**(int a, int b){  **int** s;  s = a + b;  return (s);  }  **int** **main**(){  int a, b;  a = b = 3;  printf("%d", soma(a,b)); } |
| --- | --- |

Para testar o escopo das variáveis, considere a implementação a seguir. O valor da variável a é alterado na linha 6. Qual valor será mostrado, resultante da instrução de saída da linha 14?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #**include** <stdio.h>  // Testando o escopo das variáveis **int** **soma**(int a, int b){  **int** s;  s = a + b;  a = 5;  return (s);  }  **int** **main**(){  int a, b;  a = b = 3;  printf("%d", soma(a,b));  printf("%d", a); } |
| --- | --- |

## Protótipos e interfaces de funções

O **protótipo** de uma função contém o **tipo de retorno**, o **nome da função** e a **lista de parâmetros** (quantidade e tipo).

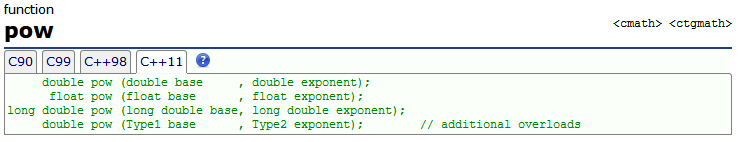
A definição dos protótipos em um código colaboram para a elegância e facilidade de uso das funções. Abaixo, segue o exemplo de um código com o protótipo da função soma(). A lista dos protótipos vem logo abaixo das diretivas de compilação (#include, #define, …), e o main()antecede as funções. Sem o uso dos protótipos, geralmente é necessário que as funções estejam implementadas antes do main() (isso pode variar com o compilador; mensagem de erro na compilação aponta o não reconhecimento da função).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #**include** <iostream>  **using** **namespace** std;  **int** **soma** (**int**, **int**); //Protótipo da função soma sem os nomes dos parâmetros  **int** **soma** (**int** a, **int** b); //Protótipo da função soma com os nomes dos parâmetros  **int** **main**(){  **int** x, y;    x = y = 3;  cout << soma(x, y) << endl; }  **int** **soma** (**int** a, **int** b){ //Implementação da função soma, após o main()  **int** s;    s = a + b;  **return** s; } |
| --- | --- |
|  |  |

A **interface** de uma função é a especificação que permite o seu uso. Para utilizar uma função, é necessário saber o **tipo de retorno** e quais **parâmetros** devem ser fornecidos (Farrell 2010).

Nos arquivos de **interfaces** das bibliotecas padrão das linguagens C e C++, as declarações das funções das bibliotecas incluem o protótipo e a documentação da função. São os arquivos de cabeçalho, que incluímos no início dos códigos-fonte, a fim de utilizarmos as funções disponíveis (diretivas #include) (Feofiloff 2009). Não precisamos saber **como** as funções são implementadas, apenas quais **parâmetros** (quantidade e tipo) devem ser fornecidos e o **tipo de retorno** que obteremos.

Abaixo, no recorte da documentação da biblioteca <cmath>/math.h., está a lista de interfaces da função pow(). Portanto, em algumas situações, há mais de uma forma de utilizar uma função. No caso da função pow(), pode-se obter retorno do tipo double, float ou long double, de acordo com os parâmetros fornecidos para a função (“C Library” 2000). Saber observar os tipos de parâmetros e de retorno é imprescindível para usar adequadamente as funções disponibilizadas pelas bibliotecas padrão.



## Outros exemplos

**Exemplo 3**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #**include** <iostream> #**include** <cmath>  **using** **namespace** std;  **float** **areaQ**(**int**);  **float** **perQ**(**int**); **float** **areaT**(**int**); **float** **perT**(**int**);  //Calcula área e perímetro de um triângulo ou de um quadrado **int** **main**(){  **int** n, l;  **float** A, P;    cin >> n >> l;   **if** (n == 3){  A = areaT (l);  P = perT(l);  }  **else**{  A = areaQ (l);  P = perQ(l);  }  cout << "Area: " << A << endl;  cout << "Perímetro: " << P << endl;  **return** 0; }  **float** **areaQ**(**int** l){  **return** l \* l; }  **float** **perQ**(**int** l){  **return** l \* 4; }  **float** **areaT**(**int** l){  **return** (l \* sqrt(3))/2; }  **float** **perT**(**int** l){  **return** l \* 3; } |
| --- | --- |

quebra de página

## Bibliografia

“C Library.” 2000. C Library. http://www.cplusplus.com/reference/clibrary/.

Farrell, Joyce. 2010. Lógica e Design de Programação: Introdução. 5th ed. São Paulo: Cengage Learning.

Feofiloff, Paulo. 2009. Algoritmos em Linguagem C. Rio de Janeiro: Elsevier.

Forbellone, André Luiz V., and Henri F. Eberspächer. 2005. Lógica de Programação: A construção de algoritmos e estruturas de dados. 3ª ed. São Paulo: Pearson.

## Exercícios

1. No código do Exemplo 1 (página 5), explique a diferença entre o resultado das linhas 21 e 23. Também explique o que aconteceria se a instrução da linha 23 fosse substituída por cout << nome << endl;.

2. Considere os seguintes protótipos de funções e a implementação do main():

| **void** **bom\_dia**(); **int** **aleat**(); **int** **dobro**(**int**); **void** **mostra\_dobro**(**int**);  **int** **main**(){  **int** x, num;    bom\_dia();  printf("%d", aleat());  x = aleat();   num += dobro(x);  **if** (dobro(x) > dobro(aleat())){  mostra\_dobro(x);  }  **return** 0; } |
| --- |

Como é possível diferenciar as funções que têm retorno ou não? Como esses retornos estão sendo usados nessa implementação? E quanto aos parâmetros? Implemente as funções, conforme os protótipos, e teste a função principal.

3. O que você pode afirmar sobre as seguintes interfaces de funções?

int rand (void);

void srand (unsigned int seed);

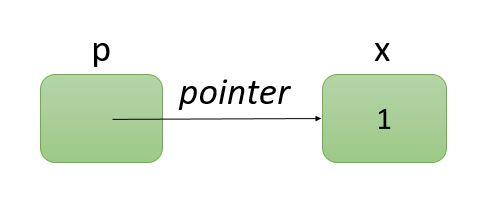
int atoi (const char \* str);

4. Escreva um programa que demonstre diferentes escopos de variáveis: global, local e bloco. O programa deve declarar uma variável global, usar essa variável dentro de uma função e modificar seu valor dentro de um bloco condicional (if) dentro dessa função.

5. Escreva um programa que inclua uma função chamada calcularAreaRetangulo que recebe dois parâmetros do tipo double representando o comprimento e a largura de um retângulo. A função deve retornar o valor da área calculada. Defina o protótipo da função antes de sua implementação e utilize-a no main.

6. Escreva um programa que inclua uma função chamada gerarNumeroAleatorio que não recebe nenhum parâmetro e retorna um número inteiro aleatório entre 1 e 100. Defina o protótipo da função antes de sua implementação e utilize-a no main.

quebra de página



Ponteiros

## Endereços de memória

A memória de um computador é dividida em bytes, e cada byte tem um **endereço**. As variáveis ocupam, conforme o seu tipo, uma sequência contígua de *bytes*, então cada variável também tem um endereço, que é o endereço do *byte* inicial da variável (Feofiloff 2009).

Em C/C++, o operador que nos permite acessar o endereço de uma variável é **&**. Portanto, dizemos que **&n** é o endereço da variável **n**. Para ver esse endereço, pode-se utilizar a instrução cout << &n;.

## Ponteiros

Algumas vezes, é necessário armazenar o endereço de outra variável. Para isso, existem variáveis próprias, chamadas **ponteiros** (Feofiloff 2009). A identificação dessas variáveis é feita pelo operador unário **\*** (cuidado, aqui os parênteses farão muita diferença para distinguir esse operador do operador binário de multiplicação).

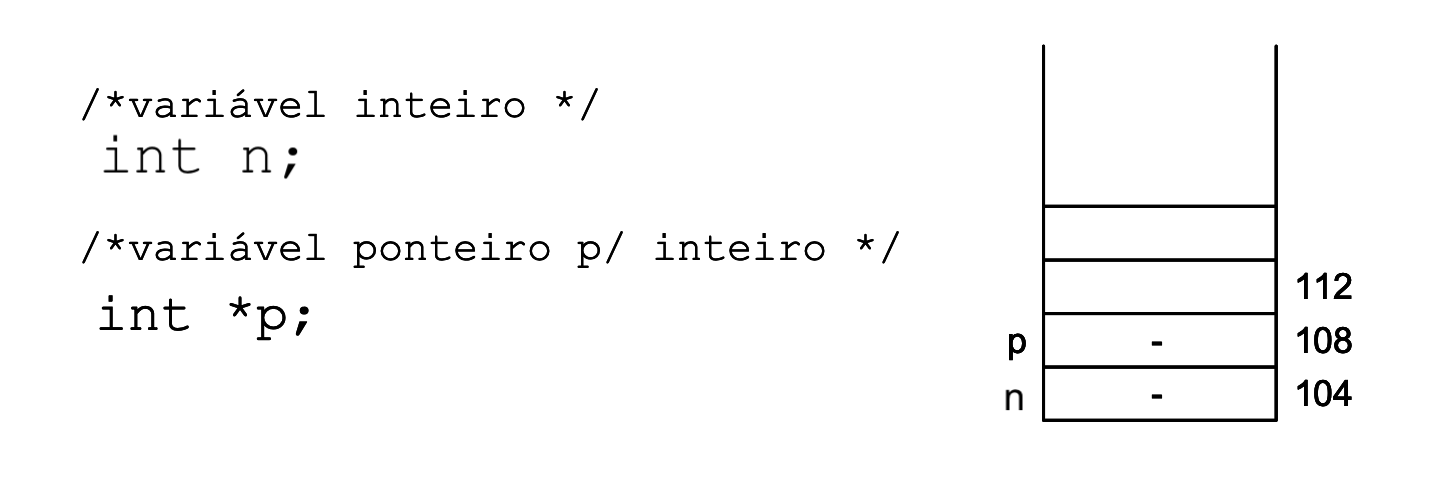
Quando escrevemos

int n;

declaramos uma variável com nome **n** que pode armazenar valores inteiros. Assim, automaticamente reserva-se um espaço na memória suficiente para armazenar valores inteiros (4 bytes). Da mesma forma, podemos declarar variáveis que servem para armazenar valores de endereços de memória. Assim, podemos escrever:

int \*p;

Nesse caso, declaramos uma variável como nome p que pode armazenar endereços de memória onde existe um inteiro armazenado. Pode-se dizer que \*p é um **alias** de n. O valor apontado é também chamado, genericamente, de **objeto** (pois nem sempre se trata de uma simples variável).



Adaptado de (Celes and Rangel 2002)

Veja o exemplo considerando **p** um ponteiro para uma variável int **n**:

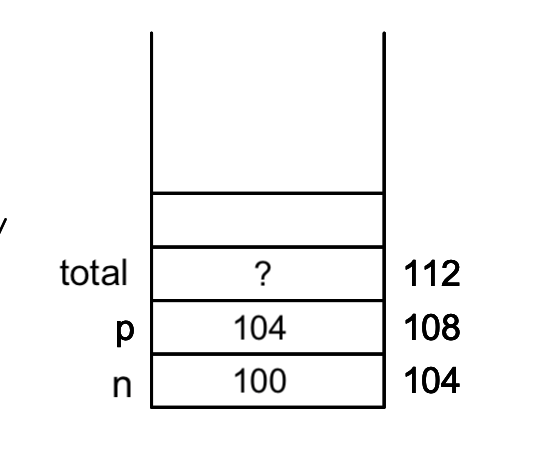
int n = 100, total = 0;

int \*p;

p = &n;//inicialização do ponteiro p

total += \*p; //total = total + \*p;

A declaração do ponteiro é feita utilizando o operador \* (int \*p;) e a atribuição é feita pela instrução p = &n;. Isto é, o operador unário **&** ("endereço de"), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. Para usar o valor que está sendo apontado por **p**, também utiliza-se o operador \* (total += \*p;). Ou seja, o operador unário **\*** ("conteúdo da variável apontada por") aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro. Segue uma representação dessa situação, com as três variáveis alocadas em um bloco de memória, sendo que os nomes das variáveis estão à esquerda e os endereços de memória à direita. Após a execução do conjunto de instruções, qual será o valor da variável total?[[1]](#footnote-0)



## Declaração e inicialização de ponteiros

Os ponteiros devem ser declarados de acordo com o tipo de valor para o qual apontarão, tanto quando se trata de tipos primitivos quanto para tipos complexos. Um ponteiro para um float será declarado pela instrução float \*taxa; para string, o ponteiro será declarado por string \*nome; para um vetor de inteiros, será int \*v[].

A inicialização de um ponteiro é feita com a constante **NULL** (em geral, equivale a zero), definida nas bibliotecas padrão de C/C++ (Feofiloff 2009).

O acesso a ponteiros não inicializados pode ser desastroso; o erro mais comum (e menos perigoso) é “Segmentation fault” na compilação, pois, em tempo de execução, valores podem ser corrompidos indevidamente. Considere o fragmento de código:

int \*p;

\*p = 43;

cout << \*p;

O ponteiro p não foi inicializado, ou seja, não é possível saber para qual posição de memória p está apontando. O valor 43 está sendo atribuído para essa posição. Eventualmente, pode-se obter o resultado esperado (43 será mostrado), mas isso será pura sorte de iniciante!

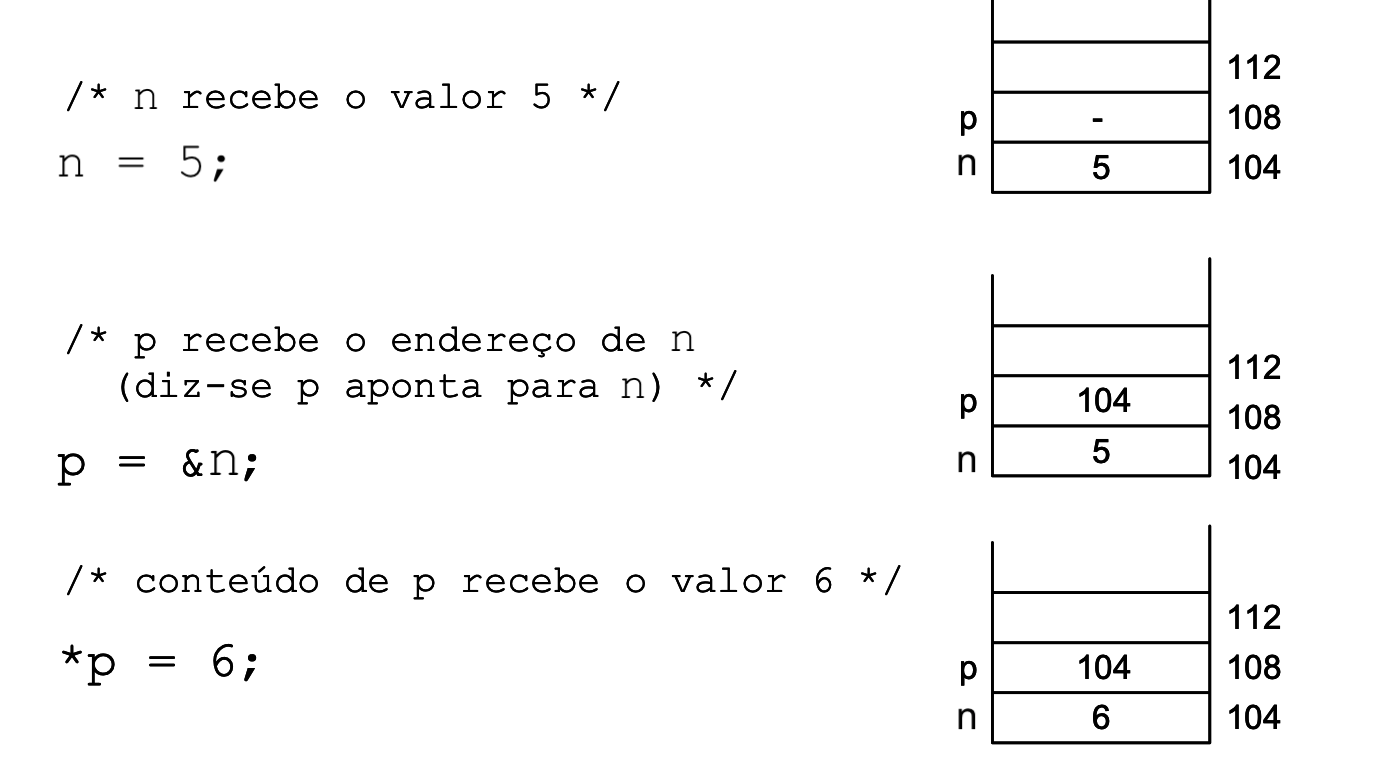
| **Atenção!**  Só podemos adicionar conteúdo a um ponteiro se ele tiver sido devidamente inicializado, isto é, ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores para o tipo de dados em questão. Nesse caso, é boa prática garantir na declaração a inicialização do ponteiro com NULL. |
| --- |

## Operações com ponteiros

Como já citado, há dois operadores próprios para ponteiros:

* **&** : operador de endereço, retorna o endereço de uma variável;
* **\*** : operador de indireção ou desreferenciação, retorna o valor apontado.

Veja o exemplo das instruções da figura a seguir, bem como a representação em um bloco de memória de acordo com cada instrução executada.



Adaptado de Celes and Rangel (2002)

No exemplo abaixo, qual será o valor mostrado pela instrução de saída na linha 6?

| 1  2  3  4  5  6  7  8 | **int** **main**(){  **int** a;  **int** \*p = NULL;  p = &a;  \*p = 2;  printf("%d", a);  **return 0**; } |
| --- | --- |

Outras operações podem ser aplicadas, tais como atribuir um ponteiro a outro, comparar o conteúdo de valores apontados, utilizar o conteúdo do valor apontado em expressões aritméticas e lógicas, entre outras. Segue um exemplo de manipulação de ponteiros de inteiros, com o teste de mesa (especificado por linha de código).

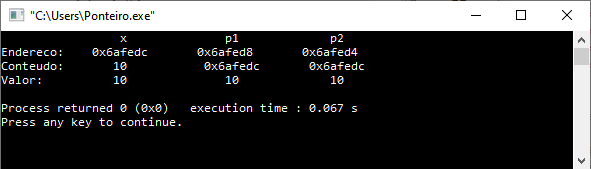
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **int** **main**(){  **int** m = 5, n = 6, \*p = NULL, \*q = NULL;  p = &m;  \*p += 5; // \*p = \*p +5;  q = p;  \*q += 5; // \*q = \*q + 5;  q = &n;  **if** (\*p > \*q)  \*q \*= 2; // \*q = \*q \* 2;  \*p = \*q;  cout << m << n << endl;  cout << \*p << \*q << endl; } |
| --- | --- |

| Linha de execução | **m** | **n** | **p** | **q** | **(\*p > \*q)?** | **Saída** |
| --- | --- | --- | --- | --- | --- | --- |
| &1000 | &1004 | &1008 | &1012 |
| 2 | ~~5~~ | ~~6~~ | ~~NULL~~ | ~~NULL~~ |  |  |
| 3 |  |  | 1000 |  |  |  |
| 4 | ~~10~~ |  |  |  |  |  |
| 5 |  |  |  | ~~1000~~ |  |  |
| 6 | ~~15~~ |  |  |  |  |  |
| 7 |  |  |  | 1004 |  |  |
| 8 |  |  |  |  | V |  |
| 9 |  | 12 |  |  |  |  |
| 10 | 12 |  |  |  |  |  |
| 11 |  |  |  |  |  | 12 12 |
| 12 |  |  |  |  |  | 12 12 |

Algumas vezes, pode ser necessário mostrar o endereço de uma variável. Com a função printf, é necessário usar o formato %p. Veja os exemplos a seguir, em C e C++, em que são mostrados os endereços, valores e referências de três variáveis (um inteiro e dois ponteiros). A tela de execução segue os códigos.

| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <stdio.h>  int main(){  int x = 10;  int \*p1, \*p2; //declaração de dois ponteiros inteiros  p1 = &x; //atribui a p1 o endereço de x  p2 = p1; //atribui a p2 o endereço que p1 está apontando  printf("\t\t x\t\t p1\t\t p2\n");  printf("Endereco: \t%p\t%p\t%p\n", &x, &p1, &p2);  printf("Conteudo: \t %d\t\t%p\t%p \n", x, p1, p2);  printf("Valor: \t\t %d\t\t %d\t\t %d\n", x, \*p1, \*p2); } |
| --- | --- |

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #**include** <iostream> #**include** <iomanip>  **using** **namespace** std;  **int** **main**(){  **int** x = 10;  **int** \*p1, \*p2; //declaração de dois ponteiros inteiros  p1 = &x; //atribui a p1 o endereço de x  p2 = p1; //atribui a p2 o endereço que p1 está apontando  cout << setw(18) << "x" << setw(16) << "p1" << setw(15) << "p2" << endl;  cout << "Endereco:" << setw(12) << &x << setw(15) << &p1 << setw(15) << &p2 << endl;  cout << "Conteudo:" << setw(9) << x << setw(19) << p1 << setw(15) << p2 << endl;  cout << "Valor:" << setw(12) << x << setw(16) << \*p1 << setw(15) << \*p2 << endl; } |
| --- | --- |



quebra de página

## Bibliografia

Celes, W., and L. Rangel. 2002. “Funções.” Instituto de Computação. http://www.ic.uff.br/~cbraga/ed/apostila/ed04-funcoes.pdf.

Feofiloff, Paulo. 2009. Algoritmos em Linguagem C. Rio de Janeiro: Elsevier.

## Exercícios

A. Considerando os fragmentos de código, responda (explicando os valores que serão mostrados[[2]](#footnote-1), e não mostrando a tela de execução ou copiando os valores):

1. Quais serão os valores impressos pelas instruções das linhas 4 e 5?

| 1  2  3  4  5 | int i, \*p\_contador;  i = 5;  p\_contador = &i;  printf("\n Valor de i: %d", \*p\_contador);  printf("\n Valor de i: %d", p\_contador); |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 4 a 6?

| 1  2  3  4  5  6 | **int** conta = 1;  float salario = 40000.0;  char resposta = 'S';  **printf**("O endereco de conta ' %x\n", &conta);  **printf**("O endereco de salario ' %x\n", &salario);  **printf**("O endereco de resposta ' %x\n", &resposta); |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 6 a 8?

| 1  2  3  4  5  6  7  8 | int num, valor;  int \*p;  num = 55;  p = &num;  valor = \*p;  printf ("%d\n", valor);  printf ("%p\n", p);  printf ("%d\n", \*p); |
| --- | --- |

B. Considere as seguintes instruções:

| a.  b.  c.  d.  e. | \*int pti;  i = \*pti;  pti = &i;  int pti;  int \*pti; |
| --- | --- |

1. Qual a instrução correta para a declaração de um ponteiro de inteiros?
2. Qual instrução atribui o valor para a uma variável do tipo ponteiro?
3. Há outras instruções válidas? Quais? O que elas fazem?

C. Considere o fragmento de código:

| 1  2  3 | int \*pti;  int i = 10;  pti = &i; |
| --- | --- |

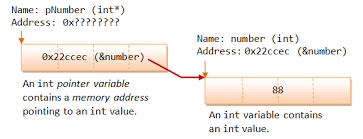
Qual afirmativa é falsa?

1. pti armazena o endereço de i.
2. \*pti é igual a 10.
3. Ao incluir a instrução \*pti = 20; na linha 4, i passará a ter o valor 20.
4. Ao alterar o valor de i, \*pti será modificado.
5. pti tem valor 10.

D. Supondo duas variáveis de ponto flutuante (numero1 e numero2), e que a primeira tenha sido inicializada com o valor 7.3, escreva instruções que executem as seguintes ações:

1. Declare a variável fP como um ponteiro para uma variável do tipo float.
2. Atribua o endereço da variável numero1 à variável de ponteiro fP.
3. Mostre o valor do objeto apontado por fP.
4. Atribua o valor do objeto apontado por fP à variável numero2.
5. Mostre o valor de numero2.
6. Mostre o endereço de numero1. (Se usar printf, use o formato %p.)
7. Mostre o endereço armazenado em fP. Use o especificador de formtato %p.   
   O valor impresso é o mesmo que o endereço de numero1?

quebra de página



Passagem de parâmetros por valor e por referência

## Passagem de parâmetros por valor

Os argumentos passados quando uma função é chamada são copiados para os parâmetros. Tome como exemplo o código abaixo. A função soma() espera dois parâmetros do tipo int. Na linha 13, ela está sendo chamada com os valores de x e y, que são copiados para os parâmetros a e b. Qualquer alteração que seja feita nos valores de a e b não irá alterar o conteúdo das variáveis x e y.

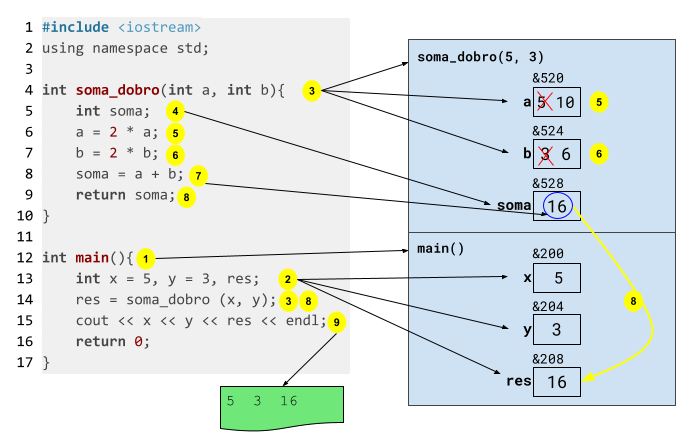
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #**include** <stdio.h>  //Retorna a soma inteira de dois inteiros fornecidos como parâmetros **int** **soma**(**int** a, **int** b){  **int** s;  s = a + b;  **return** (s); }  **int** **main**(){  **int** x, y;  x = y = 3;  printf("%d", soma(x, y));  } |
| --- | --- |

A passagem de parâmetros por valor garante a integridade do conteúdo das variáveis na função chamadora (no exemplo, as variáveis x e y no main).

Veja outro exemplo, em que a função soma\_dobro() faz o processamento a partir dos parâmetros passados por valor (adaptado de Ascencio e Campos (2007)).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #**include** <iostream>  using namespace std;  **int** **soma\_dobro** (**int** a, **int** b){  **int** soma;  a = 2 \* a;  b = 2 \* b;  soma = a + b;  **return** soma; }  **int** **main** (){  **int** x = 5, y = 3, res;  res = soma\_dobro (x, y);  cout << x << y << res << endl;  **return** 0; } |
| --- | --- |

A figura abaixo representa a passagem de parâmetros por valor, conforme o código acima. Quando a linha 14 é executada, os valores das variáveis x e y são copiados para as variáveis a e b (que estão no escopo da função soma\_dobro()). Depois disso, os valores de a e b são multiplicados por 2 (linhas 6 e 7), e somados (linha 8). O resultado da soma é devolvido à função main() (linha 9), sendo atribuído à variável res (linha 14).



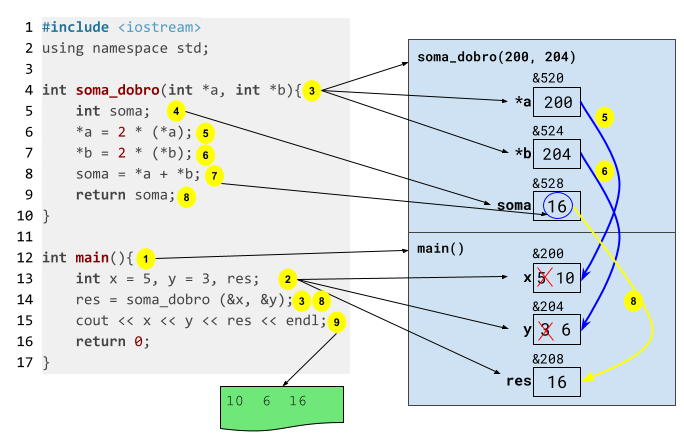
No entanto, em algumas situações, é necessário que uma ou mais variáveis sejam alteradas. Ainda, vale lembrar que uma função retorna apenas **um** valor. Então, como é possível que uma função altere o conteúdo de uma variável fornecida como parâmetro?

## Passagem de parâmetros por referência

É possível fornecer o endereço de uma ou mais variáveis para que uma função altere seu conteúdo. Veja a implementação da função soma\_dobro() com parâmetros por referência.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #**include** <iostream>  using namespace std;  **int** **soma\_dobro** (**int** \*a, **int** \*b){  **int** soma;  \*a = 2 \* (\*a);  \*b = 2 \* (\*b);  soma = \*a + \*b;  **return** soma; }  **int** **main** (){  **int** x = 5, y = 3, res;  res = soma\_dobro (&x, &y);  cout << res << endl;  **return** 0; } |
| --- | --- |

A figura abaixo representa graficamente o que acontece durante a execução do código apresentado. Quando a função soma\_dobro() é chamada (linha 14), são passados como parâmetros para a função os endereços de memória ocupados pelas variáveis x e y (isso é feito pelo operador **&** que obtém o endereço de memória de uma variável), ou seja, conforme a representação, os valores 200 (endereço ocupado por x) e 204 (endereço ocupado por y). Dessa maneira, os valores armazenados pelos ponteiros a e b (no escopo da função) são, respectivamente, 200 e 204. Nas linhas 6 e 7, os valores apontados por a e b (5 e 3) são multiplicados por 2, por meio da **referência** aos endereços de memória 200 e 204, sendo que os resultados serão armazenados nas mesmas variáveis que estão sendo apontadas. Dessa forma, no endereço 200 o valor 5 será sobrescrito pelo valor 10, e no endereço 204, 3 será substituído por 6. Na linha 8, é realizada a soma dos valores que estão nos endereços especificados por a e b. O resultado da soma é devolvido à função main() (linha 9), sendo atribuído à variável res (linha 14). Quando a função soma\_dobro() é encerrada, todos o espaço alocado para a execução da função é liberado, e as variáveis a, b e soma deixam de existir. Entretanto, as alterações decorrentes das multiplicações feitas são mantidas, pois cada alteração fez referência a endereços de memória que estavam fora da área destinada à função chamada (soma\_dobro()). Assim, após a execução da função soma\_dobro(), o valor de x será 10 e o de y será 6, mantendo as alterações feitas pela função.

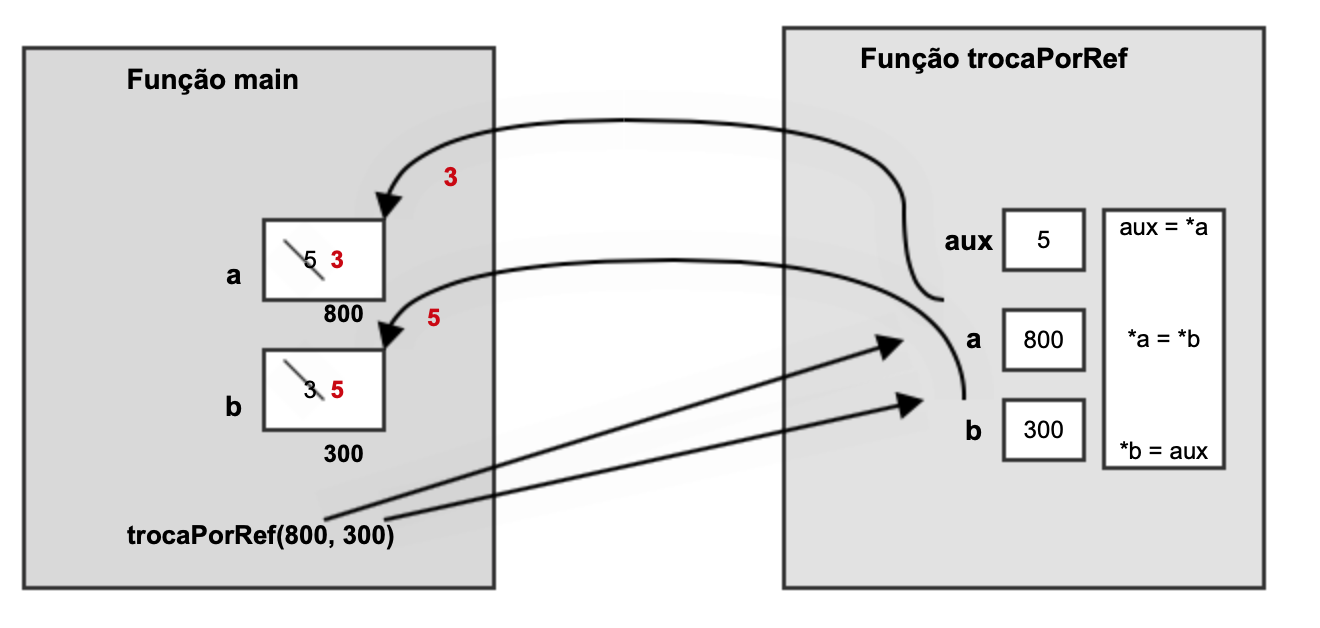


Agora, considere uma função para fazer a troca de valor de duas variáveis. Se os parâmetros forem passados por valor, quando a execução retornar à função chamadora, nenhuma alteração terá sido feita - ou seja, a funcionalidade não terá sido executada.

Para fazer a troca dos valores, então, é necessário que a função tenha acesso às posições de memória das variáveis que devem ser trocadas. Segue o código das duas funções, com parâmetros por valor (que mantém o conteúdo das variáveis passadas por parâmetro) e por referência (que altera o conteúdo das variáveis passadas por parâmetro).

A função trocaPorVal() recebe os valores de a e b (5 e 3), troca os conteúdos das variáveis (linhas 5 a 7), mas essa alteração não tem impacto nas variáveis do main() fornecidas como parâmetro. O resultado mostrado pela execução da linha 21 (após a chamada da função trocaPorVal()) confirma isso. Distintamente, a função trocaPorRef() recebe como parâmetros os endereços das variáveis a e b. Assim, a função tem acesso a posições de memória das variáveis alocadas no main(), podendo alterar seus conteúdos. Ao receber os endereços por parâmetro, a função trocaPorRef() declara duas variáveis do tipo ponteiro em seu escopo, inicializadas com os endereços fornecidos na chamada da função. A figura abaixo ilustra a passagem de parâmetros para a função trocaPorRef(). Ao executar a atribuição do conteúdo da variável apontada por a (linha 13), o valor da variável a (no escopo do main(), representado pelo quadro à esquerda) é alterado. O mesmo acontece quando a linha 14 é executada, em que o valor armazenado temporariamente em aux é atribuído ao endereço de memória armazenado no ponteiro b (no escopo da função) à variável b (no escopo do main()).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #**include** <stdio.h>  **void** **trocaPorVal** (**int** a, **int** b){  **int** aux;  aux = a;  a = b;  b = aux; }  **void** **trocaPorRef** (**int** \*a, **int** \*b){  **int** aux;  aux = \*a;  \*a = \*b;  \*b = aux; }  **int** **main**(){  **int** a = 5, b = 3;  printf("A: %d, B: %d\n", a, b);  trocaPorVal(a, b);  printf("A: %d, B: %d\n", a, b);  trocaPorRef(&a, &b);  printf("A: %d, B: %d\n", a, b);  **return** 0; } |
| --- | --- |



## 

## Outros exemplos

**Exemplo 4**

Considere um sistema de fidelidade, em que um cliente obtém um selo para cada compra acima de um valor estabelecido. Ao atingir o décimo selo, o cliente obtém 50% de desconto no valor da compra e a contagem dos selos volta a zero. Então, a função fidelidade deve ser capaz de alterar a variável responsável por armazenar os selos. Observe o código.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #**include** <stdio.h> #**define** VAL\_COMPRA\_FIDELIDADE 500  **int** **fidelidade** (**int** \*conta, **float** valor){  **if** (valor >= VAL\_COMPRA\_FIDELIDADE){  (\*conta)++;  **if** ((\*conta) == 10){  \*conta = 0;  **return** 50;  }  }  **return** 0; }  **void** **main**(){  **int** cliente, selos;  **float** valor, pagar;   scanf("%d%f", &selos, &valor);  pagar = valor - (valor \* fidelidade(&selos, valor))/100;  printf("Tem %d selos. Total a pagar: %.2f", selos, pagar); } |
| --- | --- |

A função fidelidade recebe dois parâmetros:

* int \*conta: por referência, é o endereço da variável inteira que armazena a quantidade de selos;
* float valor: por valor, é o valor da compra, para verificar se o valor mínimo foi atingido para adicionar um selo.

Caso a compra tenha sido maior do que o valor mínimo, a quantidade de selos é atualizada (linha 06). Se for o décimo selo do cliente (condição da linha 07), a variável que armazena a quantidade de selos volta a zero (linha 08) e é retornado o valor do desconto (50%) (linha 09) que deve ser aplicado à compra. Nos demais casos, o retorno da função é zero (linha 12).

Quando o controle de execução volta ao main (linha 20), é atribuído o conteúdo da variável pagar e impressa a quantidade de selos, de acordo com o valor devidamente atualizado da variável selos, que pode ter sido alterada na função fidelidade().  
 Veja o teste de mesa para o exemplo, com a chamada da função fidelidade(), o acesso à variável selos e o retorno da função.



**Exemplo 5**

Lembra da função para padronizar um nome, deixando as iniciais em maiúsculas e as demais letras em minúsculas? Pois esse é um bom exemplo de como a passagem de parâmetro por referência pode ser útil. Veja o código. A função iniciais() recebe como parâmetro um ponteiro para uma string, alterando o conteúdo da variável na função chamadora sem atribuir o retorno da função.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #**include** <iostream>  **using** **namespace** std;  **void** **iniciais** (string\* nome){  **int** i = 0;  **while** (i < (\*nome).length()){  **if** (i == 0 || (\*nome)[i - 1] == ' '){  **if** ((\*nome)[i] > 96 && (\*nome)[i] < 123)  (\*nome)[i] -= 32;  }  **else**{  **if** ((\*nome)[i] > 64 && (\*nome)[i] < 91)  (\*nome)[i] += 32;  }  i++;  } }  **int** **main**(){  string aluno;  getline(cin, aluno);  iniciais(&aluno);   cout << aluno << endl;  **return** 0;  } |
| --- | --- |

quebra de página

## Bibliografia

Ascencio, Ana Fernanda Gomes, and Edilene Aparecida Veneruchi de Campos. 2007. *Fundamentos da programação de computadores: algoritmos, pascal, c/c++ e java*. 2nd ed. São Paulo, SP: Pearson Prentice Hall.

## Exercícios

A. Analise os seguintes fragmentos de código (fazendo testes de mesa, e não executando os códigos e copiando os resultados).

1. Quais valores serão mostrados nas instruções de saída das linhas 2, 6 e 12?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | void exibe\_e\_altera(**int** **x**, **int** **y**, **int** z){  **printf**("Valores originais de x, y e z: %d %d %d\n", **x**, **y**, z);  **x** += 100;  **y** += 100;  z += 100;  **printf**("Valores finais de x, y e z dentro da funcao: %d %d %d\n", **x**, **y**, z); }  **int** main(){  **int** a = 1, b = 2, c = 3;  exibe\_e\_altera(a, b, c);  **printf**("Valores finais em main de a, b e c: %d %d %d\n", a,b,c);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 2, 6 e 12?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | void exibe\_e\_altera(**int** \*n1, **int** \*n2, **int** \*n3) {  **printf**("Valores originais na funcao - 1o, 2o, e 3o: %d %d %d\n",\*n1, \*n2, \*n3);  \*n1 += 100;  \*n2 += 100;  \*n3 += 100;  **printf**("Valores finais na funcao - 1o, 2o e 3o: %d %d %d\n", \*n1, \*n2, \*n3); }  **int** main() {  **int** a = 1, b = 2, c = 3;  exibe\_e\_altera(&a, &b, &c);  **printf**("Valores finais em main 1o, 2o e 3o: %d %d %d\n", a, b ,c);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 7 e 9?

| 1  2  3  4  5  6  7  8  9  10  11 | **void** **muda\_primeiro**(**int** \*primeiro, **int** segundo){  \*primeiro = segundo; }  **int** **main**() {  **int** a = 0, b = 5;  printf("Valor de a: %d - valor de b: %d\n", a, b);  muda\_primeiro(&a, b);  printf("Valor de a: %d - valor de b: %d\n", a, b);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 11, 13, 14 e 16?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | void por\_valor(**int** a, **int** b, **int** c) {  a = b = c = 0; }  void por\_ref(**int** \*a, **int** \*b, **int** \*c) {  \*a = \*b = \*c = 0; }  **int** main () {  **int** **x** = 2, **y** = 4, z = 6;  **printf**("\nAntes de chamar por\_valor: x = %d, y = %d, z = %d", **x**, **y**, z);  por\_valor(**x**, **y**, z);  **printf**("\nDepois de chamar por\_valor: x = %d, y = %d, z = %d", **x**, **y**, z);  **printf**("\nAntes de chamar por\_ref: x = %d, y = %d, z = %d", **x**, **y**, z);  por\_ref(&**x**, &**y**, &z);  **printf**("\nDepois de chamar por\_ref: x = %d, y = %d, z = %d", **x**, **y**, z);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 3 e 9?

| 1  2  3  4  5  6  7  8  9  10  11 | **void** **incrementar**(**int** \*num) {  \*num += 10;  printf("\nnum --> %d ", \*num); }  **int** **main**(){  **int** i = 1;  incrementar(&i);  printf("\ni --> %d ", i);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes das instruções das linhas 4, 5, 11 e 12?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | void calc(**int** \*a, **int** b) {  \*a \*= 2;  b += 5;  **printf**("\na --> %d ", \*a);  **printf**("\nb --> %d ", b); }  **int** main(){  **int** **x** = 4, **y** = 2;  calc(&**x**, **y**);  **printf**("\nx --> %d ", **x**);  **printf**("\ny --> %d ", **y**);  **return** 0; } |
| --- | --- |

1. Quais valores serão resultantes da instrução da linha 12?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | void soma(**int** **x**, **int** **y**, **int** \*z){  **int** i;  **for** (i=**x**; i<=**y**; i++)  \*z += i; }  **int** main(){  **int** a, b, **s** = 0;  scanf("%d%d", &a, &b);  **if** (a<=b)  soma(a, b, &**s**);  **printf**("\nA soma de %d e %d --> %d ", a, b, **s**);  **return** 0;  } |
| --- | --- |

B. Considerando o fragmento de código abaixo, é possível efetuar a leitura da variável z? Explique.

|  | float z;  float \*pz;  pz = &z;  **scanf("%f",** pz); |
| --- | --- |

quebra de página

****

Passagem de vetores   
por parâmetro

## Parâmetros, memória e processamento

Para relembrar, parâmetros são alocados em memória da mesma forma que variáveis locais. Quando há passagem de parâmetros por valor, os valores são copiados da função chamadora para o escopo da função que será executada. Isso implica em uma duplicação do uso de memória enquanto a função chamada estiver em execução. Para os parâmetros de tipos primitivos, essa implicação não é significativa, tanto em termos de memória quanto de tempo de processamento para realizar a cópia dos valores. No entanto, pense quando um volume maior de dados precisa ser fornecido para uma função… Eis aqui outro aspecto em favor da passagem de parâmetros por referência: ao passar como parâmetro apenas o endereço de memória, espaço de armazenamento e tempo de processamento são poupados.

## Passagem de vetores por parâmetro

Conceitualmente, conhecemos vetores como um conjunto de elementos do mesmo tipo. Ao declararmos um vetor, uma série de posições contíguas de memória são alocadas estaticamente, a fim de armazenar seus *n* elementos. Na prática, o identificador do vetor é um ponteiro para o primeiro elemento dessa estrutura (v[0]).

Vetores são exceções ao modo como são fornecidos parâmetros a uma função. Se o nome do vetor é, intrinsecamente, um ponteiro, os operadores de endereço (&) e desreferenciação (\*) são desnecessários em algumas situações.

Há três formas de passagem de vetores por parâmetro:

* Vetor sem dimensão: indicando que o parâmetro é um vetor (v[]), e passando o tamanho desse vetor como outro argumento para a função;
* Vetor do mesmo tipo e tamanho do declarado na função chamadora: indicando o vetor e seu tamanho no mesmo parâmetro, sendo que o tamanho foi previamente definido em uma cláusula #define;
* Vetor como ponteiro: indicando que é uma referência ao vetor (\*v), e passando o tamanho desse vetor como outro argumento para a função.

Veja no código abaixo a implementação da função inicializa() seguindo essas três formas.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | //Inicializa com zeros o vetor fornecido como //parâmetro (v[]), de acordo com o tamanho informado **void** **inicializa\_1** (**int** v[], **int** n){  **int** i;  **for** (i = 0; i < n; i++)  v[i] = 0; }  //Inicializa com zeros o vetor fornecido como //parâmetro (v[N]), de acordo com o tamanho definido (N) **void** **inicializa\_2** (**int** v[N]){  **int** i;  **for** (i = 0; i < N; i++)  v[i] = 0; }  //Inicializa com zeros o vetor fornecido como //parâmetro (\*v), de acordo com o tamanho informado **void** **inicializa\_3** (**int** \*v, **int** n){  **int** i;  **for** (i = 0; i < n; i++)  v[i] = 0; }  //Chamadas das diferentes funções inicializa() **int** **main**(){  **int** v[N];   inicializa\_1(v, N);  inicializa\_2(v);  inicializa\_3(v, N); } |
| --- | --- |

Na chamada das funções, segue a variação dos parâmetros das funções, fornecendo o nome do vetor (v) (linha 30) e, quando necessário, o valor para o tamanho do vetor (N, conforme a cláusula #define) (linhas 29 e 31).

As mesmas formas de passagem de parâmetro são válidas para outras funções. Veja as implementações das funções preenche() e mostra().

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | //Inclui valores no vetor até que um zero seja fornecido. **void** **preenche**(**int** \*v){  **int** i = 0;  **do**{  scanf("%d", &v[i]);  i++;  } **while** (v[i-1] != 0); }  //Mostra os elementos diferentes de zero que constam no vetor. **void** **mostra** (**int** \*v){  **int** i = 0, j = 0;  **while**((v[i] != 0) && (i < N)) {  printf("%d\n", v[i]);  i++;  } } |
| --- | --- |

Para os argumentos de uma função, há duas sintaxes equivalentes de declaração de vetores:

int s[];

int \*s;

um ponteiro para um tipo, ou um vetor de tamanho não especificado do mesmo tipo, são a mesma coisa (em se tratando de argumentos de função) (Arefin 2006).

## O operador const

O operador const é um qualificador de tipo, aplicado a uma variável, faz com que ela não tenha seu valor alterado (*read only*) (Wikipedia 2020). É possível declarar uma string, por exemplo, para que seu valor não seja alterado:

const string aula = “Programacao de Computadores II”;

Da mesma forma, const faz com que uma variável que tenha sido passada por parâmetro (por referência, é claro) não seja alterada dentro da função que a recebe. O modificador deve ser incluído na definição do parâmetro, de acordo com a sintaxe:

bool par (int const \*n);

Assim, o conteúdo da variável apontada por n não terá seu valor alterado pela função par(). Atenção, enquanto (int const \*n) garante que o conteúdo apontado de n não será alterado, (const int \*n) garante que o conteúdo de n (o endereço para onde n aponta) não seja alterado (Wikipedia 2020).

Considerando que a passagem de vetores por parâmetro ocorre sempre por referência, as funções que recebem os vetores são sempre capazes de alterar seu conteúdo. No entanto, há situações em que uma função não deve alterar o conteúdo do vetor. Nesses casos, const deve ser usado. Observe a função valores(), que retorna a quantidade de elementos do vetor.

| 1  2  3  4  5  6  7  8  9  10 | //Retorna a quantidade de valores diferentes de zero no vetor. **int** **valores** (**int const** \*v){  **int** i = 0;  **while** (v[i] != 0){  //v[i] = -1;  i++;  }  **return** (i); } |
| --- | --- |

Imagine que, por engano, a instrução da linha 6 tenha sido implementada! Ou seja, todos os elementos serão substituídos por -1! Todos os elementos originais do vetor serão perdidos! Com o uso de const na definição do parâmetro, o compilador retornará um erro, não permitindo que o vetor seja alterado dentro da função.

**Um exemplo com string**

Considerando que strings são vetores de caracteres, a passagem de strings por parâmetro se vale das mesmas condições/restrições que os argumentos com vetores.

Como exemplo, vamos usar ponteiros em uma função que calcula o comprimento de um vetor de caracteres. Lembre-se de que, por convenção, todos os vetores de caracteres são encerrados com o caracter '\0' (Arefin 2006). No código abaixo, estão implementadas três funções, com formas distintas de declaração de argumentos para string.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #**include** <iostream>  **using** **namespace** std;  **int** **lenght** (**char** s[]){  **int** n = 0;  **while** (s[n] != '\0')  n++;  **return**(n); }  **int** **lenght\_1** (**char** \*s){  **int** n = 0;  **while** (s[n] != '\0')  n++;  **return**(n); }  **int** **lenght\_2** (string s){  **int** n = 0;  **while** (s[n] != '\0')  n++;  **return**(n); }  **int** **main**(){  cout << lenght("String como parametro") << endl;  cout << lenght\_1("String como parametro") << endl;  cout << lenght\_2("String como parametro") << endl; } |
| --- | --- |

A primeira função (linha 5) usa o vetor de caracteres sem tamanho definido (char s[]) , a segunda usa um ponteiro para uma variável do tipo char (char \*s), e a terceira usa uma string (string s). Importante notar que a forma de chamada das três funções é a mesma, está sendo fornecida uma string, sem que tenha sido criada uma variável para armazená-la no main().

quebra de página

## Bibliografia

Arefin, Ahmed S. 2006. *Art of Programming Contest*. 2nd ed. Dhaka, Bangladesh: Gyankosh Prokashoni.

Wikipedia. 2020. “const (computer programming).” Wikipedia. https://en.wikipedia.org/wiki/Const\_(computer\_programming).

## Exercícios

A. Implemente funções para as seguintes funcionalidades:

1. Os dois parâmetros recebidos devem ser devolvidos ordenados (para os parâmetros a e b, o menor dos valores deve ser armazenado em a e o maior em b). Caso sejam passados valores iguais, a ordem da resposta entre eles não importa.
2. Um valor double deve ser decomposto em uma parte inteira e uma parte fracionária. (Para obter somente a parte inteira do número use o modificador int. Por exemplo, se na função foi usada uma variável ponteiro denominada \*int\_part e uma variável float x para receber o número, use a seguinte expressão para obter a parte inteira: \*int\_part = (int) x;.)

B. A função has\_zero() deve retornar true se um dos elementos do vetor tem valor zero e false se todos os elementos são diferentes de zero. Infelizmente, essa função contém um erro. Encontre o erro e mostre como arrumá-lo.

| 1  2  3  4  5  6  7  8  9 | **bool** **has\_zero**(**int** a[], **int** n) {   **int** i;  **for** (i = 0; i < n; i++)   **if** (a[i] == 0)   **return** true;   **else**   **return** false;  } |
| --- | --- |

C. Implemente funções (e suas chamadas no main()), de acordo com as três formas de passagem de vetores por parâmetro, que cumpram as seguintes funcionalidades:

1. Retorne o maior dentre os valores armazenados em um vetor de inteiros.
2. Retorne a média dos elementos de um vetor de inteiros.
3. Retorne o número de elementos positivos armazenados em um vetor de inteiros.
4. Insira n valores em um vetor e apresente-o na função main() (n deve ser fornecido como parâmetro.)
5. Encontre o maior e o menor número em um vetor de inteiros e os apresente na função main() (é necessária a passagem de parâmetros por referência).

D. Analise os seguintes fragmentos de código (fazendo testes de mesa, e não executando os códigos e copiando os resultados).

1. Qual o valor resultante da instrução da linha 16?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | void soma(**int** **x**, **int** **y**, **int** z, **int** \*ss){  **int** i;  **for** (i=**y**; i<=z; i++) {  **if** (i%x == 0)  \*ss += i;  } }  **int** main(){  **int** a, b, c, **s**=0;  **do** {  **printf**("digite 3 números, sendo o primeiro maior ou igual a 1: \n");  scanf("%d%d%d", &a, &b, &c);  } **while** (a <= 0);  soma(a, b, c, &**s**);  **printf**("\nA soma --> %d ", **s**);  **return** 0; } |
| --- | --- |

1. Qual a resposta resultante da condição da linha 19?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **int** verifica(**int** n) {  **int** n1, n2, **s**;  n1 = n / 100;  n2 = n % 100;  **s** = n1 + n2;  **printf**("\nSoma %d ", **s**);  **if** (**s**\***s** == n)  **return** 1;  **else**   **return** 0; }  **int** main() {  **int** num, **x**;  **do** {  scanf("%d", &num);  } **while** (num < 1000 || num > 9999);  **x** = verifica(num);  **if** (**x** == 1)  **printf**("\nNumero --> %d possui a caracteristica \n", num);  **else**   **printf**("\nNumero --> %d nao possui ", num);  **return** 0; } |
| --- | --- |

1. Quais os valores mostrados pela instrução da linha 14?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int verifica(int n) {  int i, q = 2;  **for** (i=2; i<=(n/2); i++)  **if** (n % i == 0)  q++;  **return** q; }  int main() {  int num;  **do** {  scanf("%d", &num);  **if** (num != 0)  printf("\nnr --> %d possui %d divisores \n", num, verifica(num));  } **while** (num != 0);  **return** 0; } |
| --- | --- |

quebra de página

1. Qual o valor resultante da instrução da linha 15? Explique.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **void** **soma**(**int** x, **int** y, **int** z, **int** \*ss){  **int** i;  **for** (i = y; i <= z; i++) {  **if** (i % x == 0)  \*ss += i;  } }  **int** **main**(){  **int** a, b, c, s = 0;  **do** {  cin >> a >> b >> c; //O primeiro valor deve ser maior ou igual a 1  } **while** (a <= 0);  soma(a, b, c, &s);  cout << "Soma = " << s << endl; } |
| --- | --- |

1. Qual a resposta resultante da condição da linha 19?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **int** verifica(**int** n) {  **int** n1, n2, **s**;  n1 = n / 100;  n2 = n % 100;  **s** = n1 + n2;  **printf**("\nSoma %d ", **s**);  **if** (**s**\***s** == n)  **return** 1;  **else**   **return** 0; }  **int** main() {  **int** num, **x**;  **do** {  scanf("%d", &num);  } **while** (num < 1000 || num > 9999);  **x** = verifica(num);  **if** (**x** == 1)  **printf**("\nNumero --> %d possui a caracteristica \n", num);  **else**   **printf**("\nNumero --> %d nao possui ", num);  **return** 0; } |
| --- | --- |

1. Quais os valores mostrados pela instrução da linha 14?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int verifica(int n) {  int i, q = 2;  **for** (i=2; i<=(n/2); i++)  **if** (n % i == 0)  q++;  **return** q; }  int main() {  int num;  **do** {  scanf("%d", &num);  **if** (num != 0)  printf("\nnr --> %d possui %d divisores \n", num, verifica(num));  } **while** (num != 0);  **return** 0; } |
| --- | --- |

quebra de página

**Passagem de matrizes por parâmetro**

Quando se trata de passar uma matriz por parâmetro, valem as mesmas observações que para os vetores, pois o nome atribuído quando uma matriz é declarada é um ponteiro para o primeiro elemento da matriz.

No entanto, a dimensão que se refere às colunas **deve** ser especificada no parâmetro da função:

| **void** **preenche\_matriz**(**int** m[][coluna]); |
| --- |

Observe o exemplo abaixo, em que as funções acessam/manipulam a matriz m.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #**include** <iostream> #**define** L 4 #**define** C 3  **using** **namespace** std;  **void** **preenche**(**int** m[][C]){  **for** (**int** i = 0; i < L; i++)  **for** (**int** j = 0; j < C; j++)  m[i][j] = 0; }  **void** **mostra**(**int** m[L][C]){  **for** (**int** i = 0; i < L; i++){  **for** (**int** j = 0; j < C; j++)  cout << m[i][j] << " ";  cout << endl;  } }  **void** **altera**(**int** m[][C]){  **int** lin, col;  cin >> lin >> col;  if (lin < L && col < C)  m[lin][col] = 1; }  **int** **main**(){  **int** m[L][C];  preenche(m);  mostra(m);  altera(m);  mostra(m); } |
| --- | --- |

É preciso muita atenção para que os limites da matriz não sejam ultrapassados, pois nenhuma verificação é feita nesse sentido. Duas boas práticas para evitar essa situação são o uso das cláusulas define (exemplo acima) e a passagem dos limites da matriz como parâmetros (exemplo a seguir).

| 1  2  3  4  5  6  7  8  9  10 | **void** **preenche** (**int** m[][C], **int** lin, **int** col){  **for** (**int** i = 0; i < lin; i++)  **for** (**int** j = 0; j < col; j++)  m[i][j] = 0; }  **int** **main**(){  **int** m[L][C];  preenche(m, L, C); } |
| --- | --- |

****

Registros (structs)

## Conjuntos de dados heterogêneos

Vetores e matrizes possibilitam que um conjunto de dados de mesmo tipo seja manipulado, por meio de um só identificador. O acesso a cada elemento do conjunto ocorre por meio de índices.

No entanto, há situações em que é preciso tratar um conjunto de dados em que cada elemento é de um tipo de dado diferente. Por exemplo, as informações de um aluno: é necessário nome, registro acadêmico (RA), curso, série. Cada um desses elementos é representado por um tipo diferente de dado.

Nessas situações, é preciso definir uma nova estrutura de dado (também chamada de registro, ou struct em C/C++). Um registro consiste em um conjunto de variáveis distintas, possivelmente de tipos de dados diferentes, alocadas em um bloco de memória sob um mesmo identificador. As variáveis da estrutura são chamadas campos ou membros (Feofiloff 2009) (“C++ Language” 2000).

## Definição e declaração de registros

Um registro é definido da seguinte forma: a lista de variáveis é incluída no bloco da instrução **struct**, entre as chaves. O identificador do registro conclui o bloco.

| 1  2  3  4  5  6 | **struct** {  **int** var1;  **float** var2;  string var3;  **char** var4; } novaStruct; |
| --- | --- |

É possível dar um nome para a struct, permitindo que registros sejam criados a partir desse nome. No fragmento de código abaixo, os registros x e y (linha 8) foram declarados a partir da struct novaStruct.

**Em C**

| 1  2  3  4  5  6  7  8 | **struct** **novaStruct** {  **int** var1;  **float** var2;  string var3;  **char** var4; };  struct novaStruct x, y; |
| --- | --- |

**Em C++**

| 1  2  3  4  5  6  7  8 | **struct** **novaStruct** {  **int** var1;  **float** var2;  string var3;  **char** var4; };  novaStruct x, y; |
| --- | --- |

Em geral, os registros são criados utilizando a instrução typedef. A instrução **typedef** associa um nome à estrutura (tipo de dado) definida pelo programador (esse nome é chamado *alias*, ou apelido). Com isso, o identificador pode ser usado para que registros desse novo tipo sejam declarados, inclusive ponteiros.

| 1  2  3  4  5  6 | **typedef** **struct** {  **int** var1;  **float** var2;  string var3;  **char** var4; } NovaStruct; |
| --- | --- |

Assim, NovaStruct representa a definição da estrutura associada a ela. Agora esse identificador NovaStruct pode ser usado para declarar uma variável desse tipo de estrutura. Por exemplo:

**NovaStruct id1, id2;**

O escopo da definição e declaração dos registros segue o padrão da linguagem. Portanto, se é desejável que várias funções utilizem uma struct, ela deve ser definida como global.

**Uso de registros**

Considere o exemplo para os dados de uma conta bancária, para a qual é preciso armazenar o número da conta, tipo de conta, nome do titular e saldo. A estrutura, denominada Conta, está definida no bloco das linhas 4 a 9 (em C) e 1 a 6 (em C++).

**Em C**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **#include <stdio.h> #include <string.h>**  **typedef struct {  int id;  char tipo;  char titular[25];  float saldo; } Conta;**  **int main(){  Conta novaConta;  novaConta.id = 12345;  novaConta.tipo = 'C';  strcpy(novaConta.titular, "Paulo Figueira de Lima");  novaConta.saldo = 100.00;  printf("%d %s \n ", novaConta.id,novaConta.titular);  printf("%f", novaConta.saldo);  return 0; }** |
| --- | --- |

**Em C++**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **typedef struct {  int id;  char tipo;  string titular;  float saldo; } Conta;  int main(){  Conta novaConta;  novaConta.id = 12345;  novaConta.tipo = 'C';  novaConta.titular = "Paulo Figueira de Lima";  novaConta.saldo = 100.00;  cout << novaConta.id << " " << novaConta.titular << " ";  cout << novaConta.saldo << endl;  return 0; }** |
| --- | --- |

Cada um dos campos é acessado de forma independente, por meio do identificador utilizado para declarar a variável do tipo estrutura (nesse exemplo, novaConta foi declarada nas linhas 12 e 9) e do identificador de cada campo da estrutura. Os dois identificadores são separados por um ponto (.), que é o operador de associatividade. Portanto, novaConta.id se refere ao campo id do registro novaConta, do tipo Conta. Os códigos também mostram a atribuição de valores a cada um dos campos da estrutura em C e C++ (respectivamente, linhas 13 a 16 e linhas 10 a 13), e a impressão dos conteúdos (linhas 17 e 18 em C, e 14 e 15 em C++), sempre utilizando a notação variável\_estrutura.campo.

Assim como um vetor, quando um registro é declarado, um bloco de memória sequencial é alocado; portanto, a inicialização pode ser feita da mesma forma, seguindo a ordem dos campos:

**Conta novaConta = {12345, ‘C’, “Paulo Figueira de Lima”, 100.00};**

Ainda, o conteúdo de uma variável pode ser copiada para outra:

**Conta outraConta;**

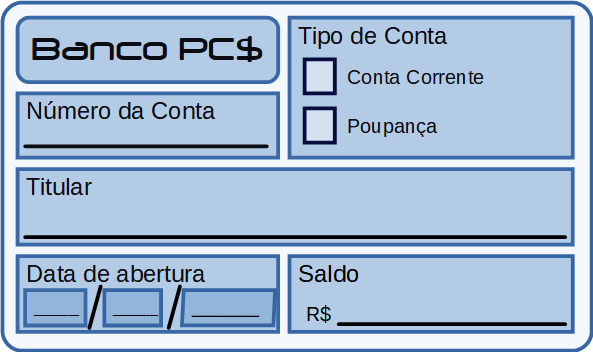
**outraConta = novaConta;**

## Composição de estruturas

Uma estrutura pode conter outras estruturas. Considere, por exemplo, que data é composta por três campos distintos (dia, mês e ano).



Há várias situações em que pode ser necessário armazenar a data. No contexto da conta bancária, é preciso armazenar a data da abertura da conta.



A estrutura Data está definida nas linhas 1 a 5, e sendo utilizada no campo abertura (linha 11) da estrutura Conta.

**Em C++**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **typedef struct{  int dia;  int mes;  int ano; } Data;  typedef struct{  int id;  char tipo;  string titular;  Data abertura;  float saldo; } Conta;  int main(){  Conta novaConta = {12345, 'C', "Paulo Figueira de Lima", 05, 02, 2021, 100.00};  novaConta.abertura.dia = 04;  cout << fixed << setprecision(2);  cout << novaConta.id << " " << novaConta.titular << endl;  cout << "Cliente desde " << novaConta.abertura.dia << "/";  cout << novaConta.abertura.mes << "/" << novaConta.abertura.ano;  cout << endl << "Saldo: R$" << novaConta.saldo << endl;  return 0; }** |
| --- | --- |

**Em C**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | **typedef struct {  int dia;  int mes;  int ano; } Data;  typedef struct {  int id;  char tipo;  char titular[25];  Data abertura;  float saldo; } Conta;  int main () {  Conta novaConta = {12345, 'C', "Paulo Figueira de Lima", 05, 02, 2021, 100.00};  printf("%d - %s", novaConta.id, novaConta.titular);  printf("\nCliente desde %d/%d/%d", novaConta.abertura.dia, novaConta.abertura.mes,novaConta.abertura.ano);  printf("\nSaldo: R$ %f", novaConta.saldo); }** |
| --- | --- |

**Atividade proposta**

Para a conta bancária, o campo tipo armazena o tipo de conta, que pode ser Poupança (representada por ‘P’) ou Conta Corrente (representada por ‘C’). O tipo não está sendo mostrado nas informações que estão sendo impressas; inclua o tipo de conta, por extenso, antes do número da conta (id).  
Crie outra variável do tipo Conta e preencha os campos, fazendo a leitura dos dados.

**Ponteiros para registros**

A declaração de ponteiros para registros requer que um *alias* tenha sido associado à struct. Então, caso isso ainda não tenha sido feito, ao declarar o ponteiro, será necessário utilizar a instrução typedef, da seguinte forma:

**typedef struct novaStruct \*ptr;**

Se a estrutura já tiver sido associada a um alias em sua definição, o ponteiro pode ser declarado seguindo o padrão (utilizando apenas o *alias*):

**NovaStruct \*ptr;**

Utilizando ponteiros, o modo de acessar os campos de um registro pode ser feito com o uso de um operador próprio (operador seta) (->), que simplifica a notação para acesso aos campos do registro referenciados por um ponteiro.

Considerando novaData um ponteiro para um registro da struct Data, as instruções

**(\*novaData).dia = 15;**

**novaData->dia = 15;**

são equivalentes. O operador seta reduz a possibilidade de acessos indevidos, caso os parênteses sejam usados incorretamente.

| **Atenção!**  A instrução (\*novaData).dia é bem diferente da instrução \*novaData.dia. Sem os parênteses, segundo a ordem de precedência, o ponteiro estará se referindo ao campo dia, o que pode resultar em um acesso indevido, pois não sabemos se o campo dia é ou não um ponteiro. |
| --- |

**Registros como parâmetros**

Assim como outras variáveis, os registros também podem ser utilizados como parâmetros de funções. É possível passar o registro todo ou apenas um dos campos (como uma simples variável).

Se é necessário que a função atualize um dos campos de um registro, é mais adequado que apenas esse campo seja passado como parâmetro e o novo valor seja devolvido pela instrução return.

Continuando o exemplo da conta bancária: considere a função atualizaSaldo. Ela recebe como parâmetro o saldo e a taxa, e retorna o valor atualizado. Quando a função é chamada (linha 8), o conteúdo do campo novaConta.saldo é copiado para o parâmetro saldo. O valor retornado pela função é sobreposto ao conteúdo do campo novaConta.saldo.

| 1  2  3  4  5  6  7  8  9  10  11 | **float atualizaSaldo(float saldo, float taxa){  return (saldo + saldo \* taxa); }  int main(){  Conta novaConta = {12345, 'C', "Paulo Figueira de Lima", 05, 02, 2021, 100.00};  cout << endl << "Saldo: R$" << novaConta.saldo << endl;  novaConta.saldo = atualizaSaldo(novaConta.saldo, 0.01);  cout << "Saldo atualizado: R$" << novaConta.saldo << endl;  return 0; }** |
| --- | --- |

Quando é necessário fornecer todo o registro para uma função, é possível fazer passagem de parâmetro por valor ou por referência. Na passagem por valor, as alterações feitas na função não alteram o conteúdo do registro.

A função imprime exemplifica esse tipo de passagem de registro por valor. Todo o conteúdo do registro novaConta (no escopo do main()) é copiado para o parâmetro conta (no escopo da função imprime()).

**Em C++**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **void imprime (Conta conta){  cout << fixed << setprecision(2);  cout << conta.id << " " << conta.titular << endl;  cout << "Cliente desde " << setfill('0') << setw(2) << conta.abertura.dia << "/";  cout << setfill('0') << setw(2) << conta.abertura.mes << "/";  cout << conta.abertura.ano << endl;  cout << "Saldo: R$" << conta.saldo << endl; }  int main(){  Conta novaConta = {12345, 'C', "Paulo Figueira de Lima", 05, 02, 2021, 100.00};  imprime(novaConta);  return 0; }** |
| --- | --- |

**Em C**

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **void imprime (Conta conta){  printf("%d %s \n", conta.id, conta.titular);  printf("Cliente desde %d/%d/%d \n", conta.abertura.dia , conta.abertura.mes, conta.abertura.ano);  printf("Saldo: R$ %f", conta.saldo); }   int main(){  Conta novaConta = {12345, 'C', "Paulo Figueira de Lima", 05, 02, 2021, 100.00};  imprime(novaConta);  return 0; }** |
| --- | --- |

Ao copiar todos os dados do registro, é produzida uma **sobrecarga** (de memória e tempo de processamento; imagine um registro com vários campos, campos com tipos de dados compostos ou até mesmo outros registros). Uma forma de evitar isso é utilizar a passagem de parâmetros por referência. Além de reduzir as demandas de recursos computacionais, os parâmetros por referência possibilitam às funções alterar os campos do registro (se for necessário).

A passagem de parâmetros por referência é feita usando **ponteiros**. Ao chamar a função, o valor fornecido deve ser o endereço do registro que será passado por parâmetro. No escopo da função, o tratamento é para o ponteiro do registro.

**Em C++**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **void preenche (Conta \*conta){  cin >> conta->id;  cin >> conta->tipo;  cin.get();  getline(cin, conta->titular);  cin >> conta->abertura.dia >> conta->abertura.mes >> conta->abertura.ano;  cin >> conta->saldo; }  int main(){  Conta novaConta;  preenche(&novaConta);  imprime(novaConta);  return 0; }** |
| --- | --- |

**Em C**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **void preenche (Conta \*conta){  scanf("%d%\*c", &conta->id);  scanf("%c%\*c", &conta->tipo);  scanf("%79[^\n]",conta->titular);  scanf("%d%\*c%d%\*c%d%\*c", &conta->abertura.dia, &conta->abertura.mes, &conta->abertura.ano);  scanf("%f%\*c", &conta->saldo); }  int main(){  Conta novaConta;  preenche(&novaConta);  imprime(novaConta);  return 0; }** |
| --- | --- |

**Vetores de Registros**

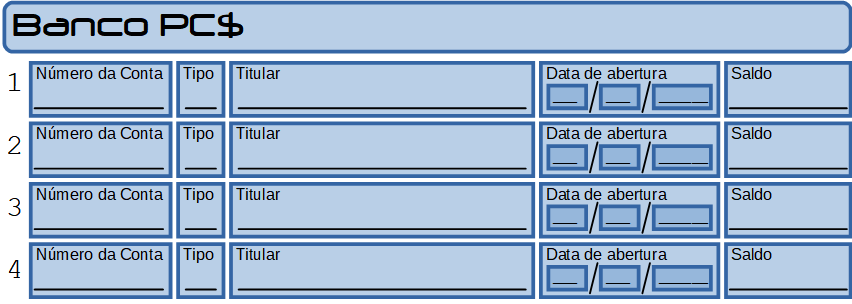
E quanto é necessário um conjunto de registros? Uma das possibilidades é o uso de vetores de registros. Uma das formas de declarar um vetor é usando o *alias* da struct:

**NovaStruct vetReg[100];**

Cada elemento do vetor é constituído por um registro. O acesso a cada registro é feito por meio do índice, e então especificado o campo:

**vetReg[i].var1 = 324;**

Considere, continuando o exemplo da conta bancária, a implementação de uma agência, com um conjunto de contas.



Utilizando a struct já definida, o vetor é declarado (linha 32) e utilizado da forma usual. As funções preenche() e imprime(), tratam um elemento por vez, por meio de um ponteiro (Conta \*conta) para uma das posições do vetor. Para chamar essas funções, é necessário passar o endereço do elemento do vetor (&agencia[N]) (linhas 34 e 39). Com essa implementação, é possível incluir contas enquanto a resposta para a pergunta “Continua? S/N” for ‘S’ ou ‘s’.

**Em C++**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | **int preenche (Conta \*conta){  cout << "Numero da conta: ";  cin >> conta->id;  cout << "Tipo: ";  cin >> conta->tipo;  cin.get();  cout << "Titular: ";  getline(cin, conta->titular);  cout << "Data de abertura: ";  cin >> conta->abertura.dia >> conta->abertura.mes >> conta->abertura.ano;  cout << "Saldo: ";  cin >> conta->saldo;  return 1; }  void imprime (Conta \*conta){  cout << fixed << setprecision(2);  if (conta->tipo == 'C')  cout << "Conta Corrente: ";  else  cout << "Conta Poupanca: ";  cout << conta->id << endl << "Titular: " << conta->titular << endl;  cout << "Cliente desde " << setfill('0') << setw(2) << conta->abertura.dia;  cout << setfill('0') << "/" << setw(2) << conta->abertura.mes << "/";  cout << setfill('0') << setw(4) << conta->abertura.ano << endl;  cout << "Saldo: R$" << conta->saldo << endl; }  int main(){  int N = 0;  char continua;  Conta agencia[1000];  do{  N += preenche(&agencia[N]);  cout << "Continua? S/N ";  } while (cin >> continua && continua == 'S' || continua == 's');  cout << endl << "Banco PC$" << endl;  for (int i = 0; i < N; i++){  imprime(&agencia[i]);  cout << endl;  }  return 0; }** |
| --- | --- |

**Em C**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | **int preenche (Conta \*conta){  printf("Numero da conta: ");  scanf("%d%\*c", &conta->id);  printf("Tipo: ");  scanf("%c%\*c", &conta->tipo);  printf("Titular: ");  scanf("%79[^\n]",conta->titular);  printf("Data da abertura ");  scanf("%d%\*c%d%\*c%d%\*c",&conta->abertura.dia, &conta->abertura.mes, &conta->abertura.ano);  printf("Saldo ");  scanf("%f%\*c", &conta->saldo);  return 1; }  void imprime (Conta \*conta){  if (conta->tipo == 'C')  printf("Conta Corrente: ");  else  printf("Conta Poupanca: ");   printf("%d %s", conta->id, conta->titular);  printf(" Cliente desde %d/%d/%d", conta->abertura.dia , conta->abertura.mes, conta->abertura.ano);  printf(" Saldo: R$ %f", conta->saldo);   }  int main(){  int N = 0;  char continua;  Conta agencia[1000];  do{  N += preenche(&agencia[N]);  printf("Continua? S/N ");  scanf("%c%\*c", &continua);  } while (continua == 'S' || continua == 's');  printf("Banco PC$\n");  for (int i = 0; i < N; i++){  imprime(agencia[i]);  printf("\n");  }  return 0;  }** |
| --- | --- |

quebra de página

## Bibliografia

“C++ Language.” 2000. C++ Tutorials. http://www.cplusplus.com/doc/tutorial/.

“C Library.” 2000. C Library. http://www.cplusplus.com/reference/clibrary/.

Feofiloff, Paulo. 2009. Algoritmos em Linguagem C. Rio de Janeiro: Elsevier.

## 

## Exercícios

A. Implemente uma agenda, a partir da definição de novas estruturas:

Horário: composto de hora e minutos;

Data: composto de dia, mês e ano;

Compromisso: composto de data, horário e texto que descreve o compromisso.

B. Para representar um vetor no R3, considere a estrutura:

| **struct Vetor{   float x;  float y;   float z;  };** |
| --- |

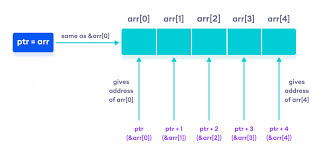
Implemente um programa que calcule a soma de dois vetores.

C. Faça um programa que faça operações simples de números complexos:

1. Crie e leia dois números complexos z e w, compostos por parte real e parte imaginária.
2. Implemente funções que calculem a soma, a subtração e o produto entre dois números complexos.

D. Escolha um jogo de cartas, baseado em um “baralho tradicional” (cada carta tem seu naipe e seu valor) ou “super trunfo” (cada carta possui um conjunto de atributos). Implemente a parte de distribuição (sorteio) de cartas para 2 jogadores, considerando que cada jogador irá receber 5 cartas. Exiba na tela as cartas que cada um dos jogadores recebeu.

quebra de página

****

Aritmética de ponteiros

## Ponteiros em C/C++

Ponteiros em C são variáveis que guardam endereços, que, em essência, são valores numéricos. Portanto, operações aritméticas são válidas para esse tipo de variável.

Considere que ptr é um ponteiro para um inteiro, que está apontando para o endereço 1000. Assuma que um inteiro ocupa 32 bits. Ao executar a instrução

**ptr++;**

a variável ptr estará apontando para o endereço 1004, porque é o endereço do próximo inteiro a partir do endereço 1000 (4 bytes a mais do que o endereço inicial), sem nenhum impacto para o valor que está sendo apontado por ptr. Se o ponteiro ptr for para uma variável do tipo char, depois da instrução de adição o conteúdo de ptr será 1001, pois o próximo caractere estará no endereço 1001 (Tutorials Point 2021).

| **Atenção!**  Enquanto a instrução ptr++ faz com que o conteúdo do ponteiro seja alterado, passando a apontar para outra posição, a instrução (\*ptr)++ adiciona valor ao conteúdo apontado por ptr. Muita atenção ao uso dos parênteses. Sem eles, seguindo a ordem de precedência dos operadores, primeiro o ponteiro será incrementado e só então retornado o valor que está sendo apontado por ele.  **int n[] = {2, 4, 6, 8};**  **int \*ptr;**  **ptr = n;**  **(\*ptr)++; //o primeiro elemento do vetor passará a ser 3**  **ptr++; //ptr passa a apontar para o segundo elemento do vetor (no caso, para o 4)** |
| --- |

## Operações aritméticas e relacionais em ponteiros

Os endereços podem ser manipulados por instruções aritméticas de **adição** e **subtração**, tanto unárias (ptr++ ou ptr- -) quanto utilizando valores **inteiros** (ptr + 2 ou ptr + i) (sempre valores inteiros).

Considerando que os ponteiros são declarados para um tipo específico de dado, a adição e a subtração de seu valor (endereço) está relacionada a esse tipo; por isso, o acréscimo de um ponteiro para um inteiro soma 4 ao endereço, enquanto para um ponteiro para um caractere soma apenas 1. Já, no caso de um ponteiro para um registro, o valor acrescido ou subtraído será a quantidade de bytes necessários para armazenar um registro, conforme a struct definida.

O nome de um vetor é um ponteiro constante, portanto não pode ser incrementando. Mas um ponteiro para um vetor pode; então, pode-se percorrer um vetor por meio de um ponteiro, incrementando o ponteiro (cuidado para respeitar o tamanho do vetor, caso contrário serão acessadas posições de memória que não contém elementos do vetor).

**Operadores relacionais** também podem ser usados para ponteiros de mesmo tipo de dados. No exemplo, a condição de parada na linha 12 é que p seja menor ou igual a u. Essa condição também poderia ser substituída por (p < &v[N]).

Para percorrer os elementos de um vetor, o exemplo utiliza dois ponteiros, sendo p inicializado com o endereço do primeiro elemento (linha 10) e u com endereço do último (linha 11). Enquanto o ponteiro p for menor ou igual a u (ou seja, apontar para uma posição anterior), o valor apontado por p é mostrado (linha 13) e p é incrementado (linha 14), apontando para o elemento seguinte.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **#include <iostream> #define N 5  using namespace std;  int main(){  int v[] = {1, 2, 3, 5, 7};  int \*p, \*u;   p = &v[0]; // ou p = v;  u = &v[N-1];  while (p <= u){  cout << \*p << endl;  p++;  } }** |
| --- | --- |

Enquanto a instrução p++ muda o conteúdo do ponteiro, fazendo-o apontar para outra variável (ou elemento do vetor), a instrução \*(p + 1) acessa o próximo elemento, sem alterar o conteúdo do ponteiro. No exemplo abaixo, qual será o conteúdo do vetor v, após executado o trecho de código das linhas 11 a 13?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **#include <iostream> #define N 5  using namespace std;  int main(){  int v[] = {1, 2, 3, 5, 7};  int \*p;   p = v;**  **for (int i = 0; i < N; i++){**  **\*(p + i) += 10; //ou v[i] += 10;**  **} }** |
| --- | --- |

## Operações inválidas

Muito cuidado! Somar ponteiros entre si não é uma operação válida (p + u, por exemplo). Outras operações aritméticas, como multiplicação e divisão também são inválidas.

Valores que não sejam inteiros também não podem ser somados ou subtraídos de ponteiros (p + 2.0, por exemplo).

**Atividade proposta**

Explique a diferença entre as seguintes instruções:

1. p++;
2. (\*p)++;
3. \*(p++);

## Um exemplo

Segue o exemplo de uma função que preenche um vetor de registros (parte do Exercício D do capítulo de Structs). A primeira implementação usa índice (linhas 7 a 9), enquanto a segunda usa ponteiro (linhas 6 a 8), que é acrescido a cada elemento adicionado ao vetor. Para usar o índice, é necessário passar o vetor como parâmetro (ponteiro constante), enquanto no segundo caso, o parâmetro é um ponteiro.

| 1  2  3  4  5  6  7  8  9  10  11  12 | **void cria\_baralho(Carta baralho[]){  string naipes[4] = {"Espadas", "Paus ", "Ouros ", "Copas "};  string valores[13] = {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};  int k = 0;  for (int i = 0; i < 4; i++){  for (int j = 0; j < 13; j++){  baralho[k].naipe = naipes[i];  baralho[k].valor = valores[j];  k++;  }  } }** |
| --- | --- |

| 1  2  3  4  5  6  7  8  9  10  11 | **void cria\_baralho(Carta \*baralho){  string naipes[4] = {"Espadas", "Paus ", "Ouros ", "Copas "};  string valores[13] = {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};  for (int i = 0; i < 4; i++){  for (int j = 0; j < 13; j++){  baralho->naipe = naipes[i];  baralho->valor = valores[j];  baralho++;  }  } }** |
| --- | --- |

## O operador sizeof

Ao utilizar a aritmética de ponteiros, o programa considera o tipo de dado para onde o ponteiro aponta (conforme a declaração da variável ponteiro) para somar a quantidade de bytes compatível com o tipo e assim acessar a próxima posição de memória.

Se for desejável saber quantos bytes cada tipo de dado ou estrutura ocupa, pode-se utilizar o **operador sizeof()**. Esse operador retorna, em *bytes*, o tamanho de uma variável ou de um tipo de dado (incluindo structs).

No código abaixo, estão sendo mostrados os tamanhos das variáveis dos tipos nativos (linhas 10 a 13) e de alguns exemplos, como a struct data (linha 14) e os vetores letra, numero e fds (linhas 15 a 17).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **struct data{**  **int dia, mes, ano;**  **};**  **int main(){**  **char letra[5] = {'a', 'e', 'i', 'o', 'u'};**  **int numero[5] = {30, 12, 67, 13, 41};**  **struct data fds[2] = {20, 02, 2021, 21, 02, 2021};**  **cout << "Tamanho do tipo char: " << sizeof(char) << endl;**  **cout << "Tamanho do tipo int: " << sizeof(int) << endl;**  **cout << "Tamanho do tipo float: " << sizeof(float) << endl;**  **cout << "Tamanho do tipo double: " << sizeof(double) << endl;**  **cout << "Tamanho da struct data: " << sizeof(data) << endl;**  **cout << "Tamanho do vetor letra: " << sizeof(letra) << endl;**  **cout << "Tamanho do vetor numero: " << sizeof(numero) << endl;**  **cout << "Tamanho do vetor fds: " << sizeof(fds) << endl;**  **return 0;**  **}** |
| --- | --- |

quebra de página

## Bibliografia

“C++ Language.” 2000. C++ Tutorials. http://www.cplusplus.com/doc/tutorial/.

Feofiloff, Paulo. 2009. Algoritmos em Linguagem C. Rio de Janeiro: Elsevier.

Tutorials Point. 2021. “C - Pointer arithmetic.” Tutorials Point. https://www.tutorialspoint.com/cprogramming/c\_pointer\_arithmetic.htm.

## Exercícios

A. Ao se utilizar um vetor como parâmetro para uma função, qual informação está sendo passada à função? Assinale a opção correta e explique.

(a). Uma cópia de todos elementos do vetor

(b). Uma cópia do primeiro elemento do vetor

(c). O endereço do primeiro elemento do vetor

(d). O endereço de todos os elementos do vetor

(e). Nenhuma das opções anteriores

B. Qual o valor de y após a execução do programa?

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **int main(){   int y, \*p, x;   y = 0;   p = &y;   x = \*p;   printf ("x = %d\n", x);   x = 4;   (\*p)++;   printf ("p = %d\n", \*p);   x--;   printf ("x = %d\n", x);   (\*p) += x;   printf ("y = %d\n", y);   return(0);  }** |
| --- | --- |

C. Sabe-se que o valor de uma variável ou expressão do tipo vetor é o endereço do elemento zero do vetor. Seja a[] um vetor qualquer, independente de tipo e tamanho, e pa um ponteiro para o mesmo tipo de a[].   
Responda V ou F:

( ) Após a atribuição pa = &a[0], pa e a possuem valores idênticos, isto é, apontam para o mesmo endereço.

( ) A atribuição pa = &a[0]; pode ser escrita como pa = a;.

( ) a[i] pode ser escrito como \*(a+i).

| 1  2  3  4  5  6  7  8  9 | **int main() {   int i, \*pa;   int a[] = {7, 9, 1, 10, 15};   pa = a;   for (i = 0; i < 5; i++) {   printf("\npa = %d", \*(a+i));   }   return 0;  }** |
| --- | --- |

D. O que está errado no código abaixo?

| 1  2  3  4  5  6 | **int main() {   int x, \*p;   x = 10;   \*p = x;   return 0;  }** |
| --- | --- |

E. Considerando o código abaixo, qual o valor apresentado na linha 7 (pb - pa)? Explique o resultado apresentado.

| 1  2  3  4  5  6  7  8  9 | **int main() {   float \*pa, \*pb;   float vet[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};   pa = &vet[0];   pb = &vet[9];   printf("pa = %d pb = %d", pa, pb);   printf("\npb - pa = %d", pb - pa);  return 0;  }** |
| --- | --- |

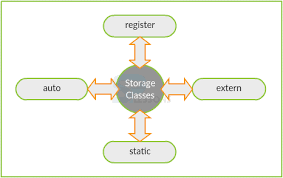
F. Analise o fragmento de código e identifique os valores finais de pc1, pc2, pc3 e pc4. Assuma que a variável x está armazenada no endereço 1000 de memória e que uma variável do tipo caracter ocupa 1 byte.

| 1  2  3  4  5  6  7 | **char x = 0;  char \*pc1, \*pc2, \*pc3, \*pc4;**  **pc1 = &x;  pc2 = pc1++;  pc3 = pc2 + 4;  pc4 = pc3 - 5;** |
| --- | --- |

G. Usando aritmética de ponteiros, implemente:

1. Uma função que receba um vetor de inteiros e os endereços de duas variáveis inteiras, onde devem ser armazenados os elementos mínimo e máximo do vetor.
2. Uma função que, a partir de dois vetores de tamanhos distintos, retorne um único vetor resultante da junção dos dois vetores iniciais.
3. Uma função que separe um vetor em dois, sendo um com os elementos pares e outro com os ímpares.

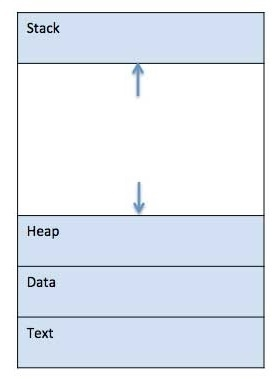
quebra de página

****

Classes de armazenamento

## Layout de memória de programas C/C++

Para um programa em linguagem C/C++, o bloco de memória disponível é organizado em segmentos (Thakur 2019), tal como ilustrado abaixo.



Cada segmento tem funções distintas, a saber (Thakur 2019) (Kangralkar 2021):

* ***Stack***: contém os dados temporários, tais como os parâmetros das funções, os endereços de retorno (endereço para onde o controle de execução deve retornar quando completar a execução de uma função) e as variáveis locais. O tamanho desse segmento é variável, avançando de um endereço mais alto para um endereço menor;
* ***Heap***: é o segmento de memória que é alocada dinamicamente para um processo durante sua execução. A Heap é a memória de armazenamento dinâmico, onde são endereçadas as variáveis alocadas pelas pelas funções new() e malloc(). Esse segmento também tem tamanho variável, e os endereços avançam de um mais baixo para um endereço mais alto;
* ***Text***: inclui a atividade corrente, representada pelo valor do contador de programa (*Program Counter*) e os conteúdos dos registradores do processo. Trata-se da área de memória onde estão as instruções de código (executável). É uma área de tamanho fixo;
* ***Data***: esse segmento contém as variáveis globais e estáticas. Contém duas seções: **data** e **bss**. A seção .data (também chamada *Initialized Data Segment*), usada no armazenamento das variáveis, tem tamanho fixo. A seção .bss (“*block started by symbol*”) (também chamada *Uninitialized Data Segment*), que também é estática, mas contém um buffer para os armazenamentos que são declarados tardiamente no programa. Variáveis globais não inicializadas no programa também são alocadas nessa seção. As variáveis alocadas nesse segmento são inicializadas em zero (por padrão).

Um programa em C suporta três tipos de alocação de memória (“Alocação de memória”, n.d.):

* **Estática**: variáveis globais ou estáticas, são geralmente alocadas em *Data*;
* **Automática**: 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, são geralmente alocadas em *Stack*;
* **Dinâmica**: blocos de memória requeridos para armazenar dados (pelas funções new() ou malloc()); o controle das áreas alocadas dinamicamente é manual ou semi-automático (o programador é responsável por liberar as áreas alocadas dinamicamente). A alocação dinâmica geralmente usa o segmento *Heap*.

## Classes de armazenamento em C/C++

A linguagem C disponibiliza quatro classes de armazenamento distintas: auto, register, static e extern. Em C++, além dessas, há também a classe mutable. Cada uma dessas classes é detalhada a seguir (“Storage Classes in C++” 2021). Como a classe mutable é aplicada no contexto de programação orientada a objetos, não será abordada neste material.

### Classe auto

A classe auto é a classe de armazenamento padrão para todas as variáveis locais. Não precisa ser explicitada na declaração da variável.

Variáveis da classe auto tem duração de armazenamento automático, ou seja, enquanto a variável estiver em seu escopo, seu valor é mantido.

### Classe register

A classe register é usada para definir uma variável que deve ser armazenada em um registrador, e não na RAM. Isso significa que a variável tem um tamanho máximo igual ao tamanho do registrador (usualmente uma palavra) e não tem o operador unário ‘&’ aplicado a ele (pois não possui uma posição de memória alocada).

As variáveis register possuem duração automática. A declaração que sugere que a alocação seja feita em um registrador é feita da seguinte forma:

**register int tam;**

Essa classe deve ser usada para variáveis que precisam de acesso rápido, tais como contadores. Deve-se observar que definir uma variável como register não garante que a variável será armazenada em um registro; significa que ela pode ser armazenada em um registro dependendo das restrições de hardware e implementação.

### Classe static

A classe de armazenamento static instrui o compilador a manter uma variável local durante toda a execução de um programa, ao invés de criar e destruir cada vez que entra ou sai do escopo. Além disso, declarar variáveis static permite que os valores sejam mantidos entre as chamadas de função.

Embora variáveis globais mantenham o armazenamento estático, se for utilizado o modificador static para variáveis desse tipo, o escopo da variável será restrito ao arquivo (programa) em que foi declarada.

Variáveis numéricas da classe static, quando não recebem um valor inicial explicitamente, são inicializadas em zero.

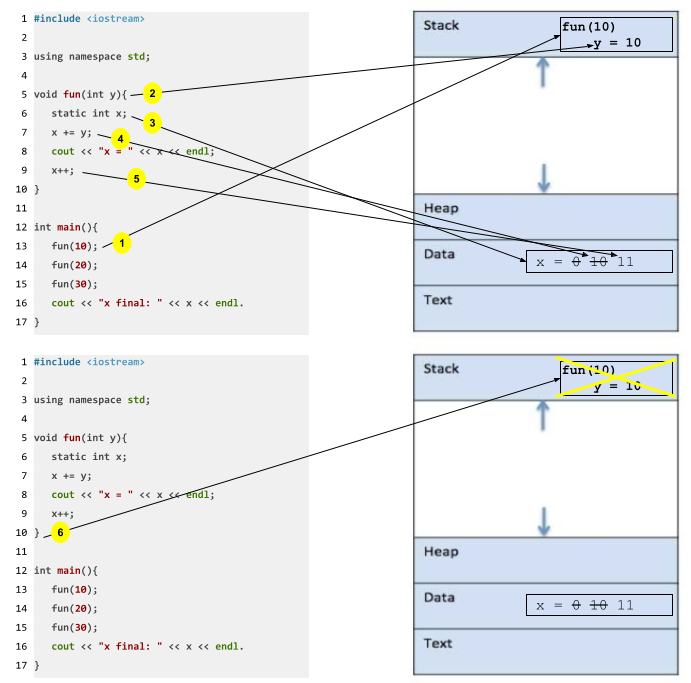
**Exemplo**

A variável x, declarada como sendo da classe static (linha 6) (dentro da função fun()), será mantida durante toda a execução do programa.

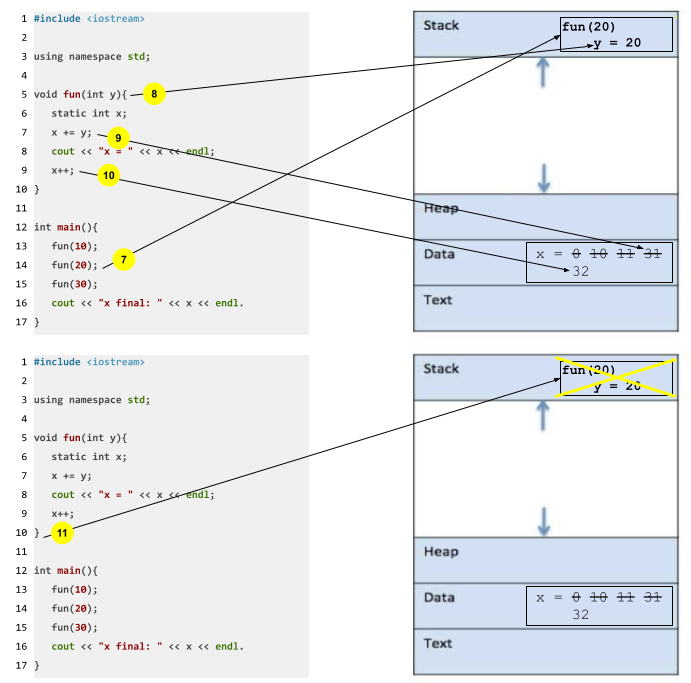
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **#include <iostream>  using namespace std;  void fun(int y){  static int x;  x += y;  cout << "x = " << x << endl;  x++; }  int main(){  fun(10);  fun(20);  fun(30);  cout << "x final: " << x << endl. }** |
| --- | --- |

**Representação de alocação de variáveis automáticas e estáticas**

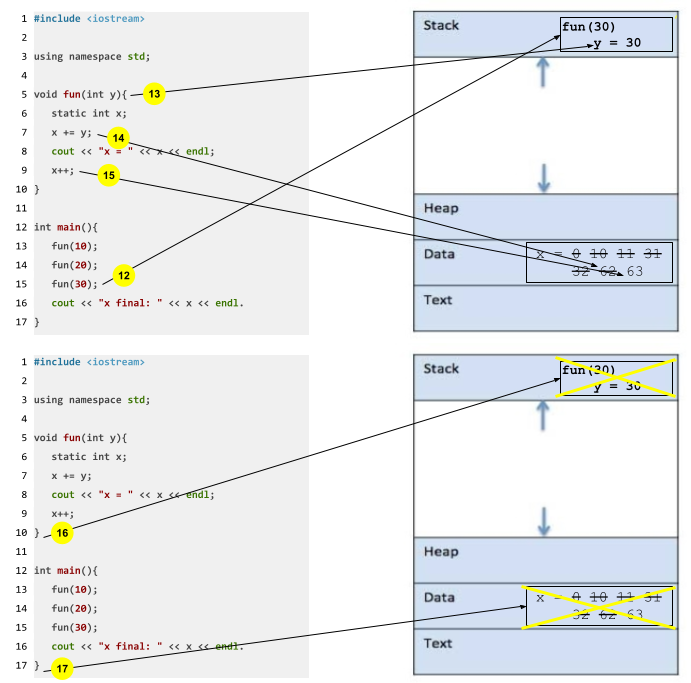
**Chamada da função fun(10)**



**Chamada da função fun(20)**



**Chamada da função fun(30)**

****

### Classe extern

A classe de armazenamento extern possibilita que uma variável global seja visível a TODOS os programas. Variáveis extern não podem ser inicializadas como todas as outras; a variável é armazenada em uma posição que foi previamente definida no código de origem.

Quando há múltiplos arquivos e são definidas variáveis ou funções globais, que serão usadas por outros programas, então a classe extern deve ser usada para declarar variáveis e funções em outros programas. O modificador extern é comumente usado quando há dois ou mais arquivos de programas compartilhando as mesmas variáveis ou funções globais.

**Exemplo**

Considere dois arquivos, main.cpp e troca.cpp. Em main.cpp, está declarada a variável global n (linha 5), e a função int troca\_val(), definida em outro arquivo, como assinalado pelo modificador extern (linha 6) . Em troca.cpp, a declaração extern int n (linha 5) diz q a variável n, declarada em outro arquivo (nesse exemplo, em main.cpp), será usada.

main.cpp

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **#include <iostream>  using namespace std;  int n; extern int troca\_val();  int main(){  n = 10;  cout << "Main: " << n << endl;  troca\_val();  cout << "Main de novo: " << n << endl; }** |
| --- | --- |

troca.cpp

| 1  2  3  4  5  6  7  8  9  10 | **#include <iostream>  using namespace std;  extern int n;  int troca\_val(){  n = 432;  cout << "Troca: " << n << endl; }** |
| --- | --- |

**Atividade proposta**

Faça testes de mesa dos códigos de exemplo das classes de armazenamento static e extern.

quebra de página

## Bibliografia

“Alocação de memória.” n.d. Alocação de memória. Accessed 03 09, 2021. https://www.inf.ufpr.br/roberto/ci067/10\_aloc.html.

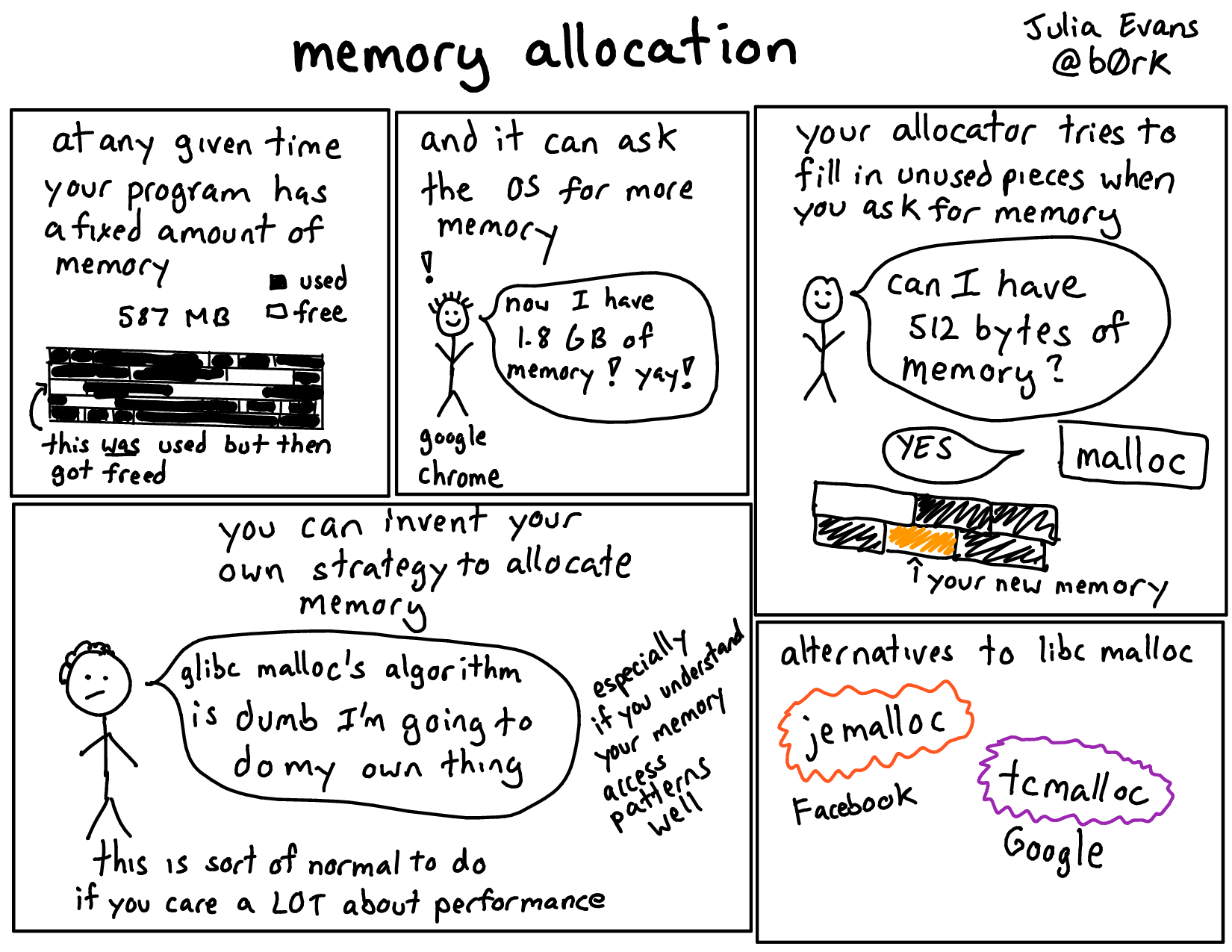
Kangralkar, N. 2021. “Memory Layout of C Programs.” GeeksforGeeks. https://www.geeksforgeeks.org/memory-layout-of-c-program/.

“Storage Classes in C++.” 2021. Tutorials Point. https://www.tutorialspoint.com/cplusplus/cpp\_storage\_classes.htm.

Thakur, Nishtha. 2019. “Memory Layout of C Programs.” Tutorials Point. https://www.tutorialspoint.com/memory-layout-of-c-programs.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | **#include <stdio.h> #include <stdlib.h>  //Exemplo de variável de escopo global int resultado;  //Exemplo de escopo de parâmetros formais int soma (int a, int b){  return a+b; }  //Exemplo de uso de variável global void soma2 (int a, int b){  resultado = a + b; }  //Exemplo de variáveis locais e de escopo de bloco int produto (int a, int b){  int p = a, i;  //for (int i = 1; i <= b; i++)  for (i = 1; i < b; i++)  p += a;  return p; }  //Exemplo de escopo de identificador (seguido por ':') com escopo de bloco (label dos case) void epar (int n){  int r;  r = n % 2;  switch (r){  case 0: printf("Eh par\n");  break;  case 1: printf("Eh impar\n");  break;  } }  //Demonstração de variável estática (local). Padrão é inicialização com zero. void fun(int y){  static int x;   x += y;  printf("x = %d\n",x);  x++; }  int main(void){  fun(10);  fun(20);  fun(30);  printf("%d\n", soma (27,48));  soma2(13, 37);  printf("%d\n", resultado);  epar(produto(3, 5));  return 0; }** |
| --- | --- |

quebra de página

****

Alocação dinâmica de memória

## Alocação estática de memória

O modo padrão de declaração de variáveis declara variáveis estáticas. Toda a memória necessária é alocada antes que a execução do programa seja iniciada. Essas variáveis não podem ter sua capacidade de armazenamento alterada enquanto o programa estiver sendo executado (“Dynamic Memory” 2020) (“C++ Dynamic Memory” 2021).

Isso também é válido para os parâmetros das funções; quando a função é chamada, um novo bloco de memória é alocado para essa função e, tanto os parâmetros formais quanto as variáveis locais da função são alocadas e persistem da mesma forma até que a execução da função seja concluída. Um vetor de 100 posições continuará com essa capacidade até que o programa ou a função em que foi declarado tenham suas execuções encerradas.

No entanto, algumas linguagens de programação disponibilizam instruções que permitem a alocação de memória em tempo de execução, à medida que novos espaços de armazenamento são necessários, ou seja, de forma dinâmica.

## Alocação dinâmica de memória em C++

Diferente da maior parte das técnicas vistas até aqui, a alocação dinâmica em C e C++ são diferentes. Cada linguagem possui operadores próprios, com algumas diferenças peculiares.

Em C++, o operador responsável pela alocação é o operador new, que retorna o endereço da variável alocada ou do primeiro elemento (no caso de um vetor).

**ptr = new int;**

**ptr = new int[N];**

No caso do vetor, o sistema aloca dinamicamente espaço para os N elementos inteiros e retorna o endereço do primeiro elemento, que é atribuído à variável ptr (que é um ponteiro). A partir do ponteiro, os demais elementos podem ser acessados, tanto pela sintaxe de índice (ptr[i]) quanto utilizando aritmética de ponteiros (ptr+i).

Uma das principais diferenças entre a alocação estática e dinâmica de um vetor é que, no caso de uma declaração estática, o tamanho do vetor é constante, enquanto com o operador new, em qualquer momento durante a execução a memória pode ser alocada, utilizando qualquer valor variável como tamanho (“Dynamic Memory” 2020).

A memória utilizada dinamicamente é alocada na memória heap, que é limitada e, em determinado momento, pode estar cheia. Então, não há garantia de que todas as requisições do operador new sejam atendidas pelo sistema (“Dynamic Memory” 2020). C++ fornece dois mecanismos para verificar se a alocação foi bem sucedida:

* com exceção: mais complexo, se a exceção não for tratada, a execução do programa é encerrada, é a forma padrão da linguagem;
* sem exceção: precisa ser explicitado como parâmetro (nothrow) do operador new. Caso a alocação não seja bem sucedida, new retorna nullptr e não encerra a execução do programa.

**ptr = new(nothrow) int[N];**

Ao usar o mecanismo sem exceções, para evitar falhas, antes de continuar a execução do programa, deve-se verificar se foi possível alocar o bloco de memória, avaliando o conteúdo do ponteiro que recebe o retorno da função.

| 1  2  3  4  5  6 | **int v\*; v = new(nothrow) int[100]; if (v == nullptr){  cout << "Falha de alocacao de memoria" << endl;  return -1; }** |
| --- | --- |

Outra opção é testar o sucesso da instrução, sem a constante nullptr (que não é reconhecida em algumas plataformas), e que resulta em um código mais enxuto (“C++ Dynamic Memory” 2021), da seguinte forma:

| 1  2  3  4  5 | **int v\*; if (!(v = new(nothrow) int[100])){**  **cout << "Falha de alocacao de memoria" << endl;  return -1; }** |
| --- | --- |

Embora o mecanismo de tratamento de exceções seja geralmente usado e resulte em códigos mais curtos, por enquanto vamos utilizar o parâmetro nothrow por sua simplicidade.

**Atividade proposta**

Tente obter a mensagem de erro de alocação de memória. Você sabe qual o tamanho do heap do sistema que você utiliza? Aumente o valor de n a cada execução para testar.

| 1  2  3  4  5  6  7  8  9  10  11 | **int \*v, n;**  **cin >> n; if (!(v = new(nothrow) int[n])){  cout << "Falha de alocacao de memoria" << endl;  return -1; } for (int i = 0; i < n; i++){  v[i] = i;  cout << v[i] << " "; } return 0;** |
| --- | --- |

## Liberando memória dinâmica em C++

Se o bloco foi alocado para atender uma demanda de execução, quando ele não for mais necessário deve ser liberado, para que outras requisições new possam ser executadas. A instrução que libera um bloco alocado para uma variável é delete, ou delete[] se for um vetor (“Dynamic Memory” 2020).

**delete ptr;**

**delete[] ptr;**

No exemplo abaixo, está sendo alocado um vetor dinamicamente (linha 4), conforme o tamanho fornecido (linha 3). A alocação é verificada e, caso não tenha sido bem sucedida, o programa é finalizado (linha 6), retornando um valor diferente, que pode ser tratado pela função chamadora. Se a execução ocorrer conforme o esperado, o bloco de memória é liberado (linha 12) e o programa é encerrado normalmente.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **int main(){  int \*v, n;  cin >> n;  if (!(v = new(nothrow) int[n])){  cout << "Falha de alocacao de memoria" << endl;  return -1;  }  for (int i = 0; i < n; i++)  v[i] = i\*2;  for (int i = 0; i < n; i++)  cout << \*(v + i) << " ";  delete[] v;  return 0; }** |
| --- | --- |

## Evitando ponteiros pendentes

O operador delete libera o bloco alocado, permitindo que aqueles endereços sejam usados por uma nova requisição new. No entanto, ainda é possível acessar o endereço de memória guardado no ponteiro. A fim de evitar usos indevidos dos chamados *dangling pointers[[3]](#footnote-2)*, é necessário atribuir NULL ao ponteiro depois que a instrução delete for executada.

O código abaixo permite testar essa situação, na linha 10 ainda é possível acessar a posição apontada por n, enquanto a tentativa de acessar o valor na linha 12 (após a atribuição de NULL ao ponteiro) resulta em erro de execução (Segmentation fault).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **int main(){**  **int \*n;**  **if (!(n = new(nothrow) int)){**  **cout << "Falha de alocacao de memoria" << endl;**  **return -1;**  **}**  **\*n = 432;**  **cout << endl << \*n;**  **delete n;**  **cout << endl << \*n;**  **n = NULL;**  **cout << endl << \*n;**  **return 0;**  **}** |
| --- | --- |

**Atividade proposta**

Repita o teste de atribuição de NULL após a liberação de memória para o caso da alocação dinâmica de um vetor.

## Alocação dinâmica de memória em C

Enquanto em C++ os operadores para alocação dinâmica são nativos, em C é necessário utilizar funções definidas na biblioteca <cstdlib.h> (“Dynamic Memory” 2020). São elas (Feofillof 2018) (CPDEE/UFMG 1999):

**void \* malloc(int qty\_bytes\_alloc);**

**void \* calloc(int qty, int size);**

**void realloc(void \* pointer, int new\_size);**

**free (void \* pointer);**

### Malloc

É a função malloc que realiza a alocação de memória. Deve-se informar para a função a quantidade de bytes para alocação. A função irá retornar, se existir memória suficiente, um endereço que deve ser colocado em uma variável do tipo ponteiro.

A declaração int v[10]; é equivalente a:

**int \*v;**

**v = malloc(10 \* sizeof(int \*));**

Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço de memória inicial da área da memória alocada. Perceba que a função malloc retorna um endereço e portanto esta informação deve ser armazenada em um ponteiro. Observe o uso do operador sizeof().

Além disso, devemos salientar que a função malloc é usada para alocar espaço para armazenar valores de qualquer tipo. Por esse motivo, malloc retorna um ponteiro genérico, para um tipo qualquer, representado por void\*, que deve ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição.

No entanto, é comum fazer a conversão utilizando o operador de molde de tipo (cast). O comando para a alocação do vetor de inteiros fica então:

**v = (int\*) malloc (10\*sizeof(int));**

Assim, o ponteiro v aponta para o endereço inicial de uma área contínua de memória para 10 inteiros. O operador sizeof() obtém o tamanho em bytes do tipo int e o operador de cast (int\*), converte o ponteiro genérico retornado por malloc para um ponteiro do tipo int\*. É importante fazer o tratamento de erro adequado quando for retornado o valor nulo.

**v = (int\*) malloc (10\*sizeof(int));**

**if (!v){**

**printf(“Memoria insuficiente.”);**

**exit(1);**

**}**

**Exemplo**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **#include <stdio.h>**  **#include <stdlib.h> /\* Para usar malloc() \*/ int main (void) {  int \*v;  int a=2;   v = (int \*) malloc(a\*sizeof(int));   if (!v)  {  printf ("\*\* Erro: Memoria Insuficiente \*\*");  exit;  }**  **return 0;**  **}** |
| --- | --- |

### Calloc

Ao invés de se alocar uma quantidade de bytes através da função malloc, pode-se usar a função calloc e especificar a quantidade de bloco de um determinado tamanho.

**v = (int\*) calloc (10, sizeof(int));**

**Exemplo**

| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **#include <stdio.h>**  **#include <stdlib.h> /\* Para usar malloc() \*/ int main (void) {  int \*v;  int a = 2;   v = (int \*) calloc(a, sizeof(int));   if (!v) {  printf ("\*\* Erro: Memoria Insuficiente \*\*");  exit;  }**  **return 0; }** |
| --- | --- |

### Realloc

Às vezes é necessário redimensionar uma área alocada. Para isto deve-se usar a função realloc. Deve-se passar para ela o ponteiro retornado pelo malloc e a indicação do novo tamanho. A realocação de memória pode resultar na troca de blocos na memória. Podíamos então realocar o vetor do exemplo anterior para 40 posições.

**v = (int \*)realloc (v, 40\*sizeof(int));**

**Exemplo**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **#include <stdio.h>**  **#include <stdlib.h> /\* Para usar malloc() \*/ int main (void) {  int \*v;  int a;  a = 30;   v = (int \*) malloc(a \* sizeof(int));   if (!v){  printf ("\*\* Erro: Memoria Insuficiente \*\*");  exit;  }  /\* O tamanho de v deve ser modificado, por algum motivo ... \*/  a = 100;  v = realloc (v, a\*sizeof(int));**  **return 0; }** |
| --- | --- |

### Free

Quando não se deseja mais uma área alocada, deve-se liberá-la através da função free. Deve ser passado para a função o endereço que se deseja liberar, que foi devolvido quando a alocação da memória ocorreu. Assim, para liberar o vetor v, fazemos:

**free (v);**

Só podemos passar para a função free um endereço de memória que tenha sido alocado dinamicamente, ou seja, ele precisa ter sido inicializado com o retorno da função malloc anteriormente. Devemos lembrar ainda que não podemos acessar o espaço da memória depois de liberado.

Da mesma forma que em C++, a fim de evitar ponteiros pendentes, é preciso atribuir NULL ao ponteiro que foi desalocado pela função free.

**v = NULL;**

**Exemplo**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **#include <stdio.h>**  **#include <stdlib.h> /\* Para usar malloc() \*/ int main (void){  int \*v;  int a = 2;   v = (int \*) malloc(a\*sizeof(int));   if (!v){  printf ("\*\* Erro: Memoria Insuficiente \*\*");  exit;  }**  **free(v);**  **v = NULL;**  **return 0;**  **}** |
| --- | --- |

quebra de página

## Bibliografia

“C++ Dynamic Memory.” 2021. Tutorials Point. https://www.tutorialspoint.com/cplusplus/cpp\_dynamic\_memory.htm.

“C++ Language.” 2000. C++ Tutorials. http://www.cplusplus.com/doc/tutorial/.

“C Library.” 2000. C Library. http://www.cplusplus.com/reference/clibrary/.

CPDEE/UFMG. 1999. “Curso de C.” Curso de C. https://www.pucsp.br/~so-comp/cursoc/aulas/ca60.html.

“Dynamic Memory.” 2020. C++ Tutorials. https://www.cplusplus.com/doc/tutorial/dynamic/.

Feofillof, Paulo. 2018. “Alocação dinâmica de memória.” Projeto de algoritmos. https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html.

## Leitura complementar

<https://blog.pantuza.com/artigos/heap-vs-stack>

<https://courses.engr.illinois.edu/cs225/fa2020/resources/stack-heap/>

## Exercícios

A. Escreva um programa que possua uma função que insira *n* números inteiros em um vetor (alocado dinamicamente). Os números serão inseridos automaticamente, conforme o índice da posição (ou seja, para n=6, os elementos do vetor serão v = {0, 1, 2, 3, 4, 5}. Desenvolva outra função que apresente o conteúdo do vetor. Além disso, antes de finalizar o programa, deve-se liberar a área de memória alocada.

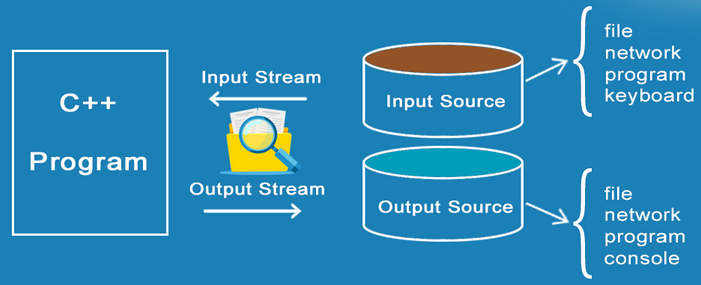
B. Altere o programa anterior e use aritmética de ponteiros para percorrer o vetor nas funções de inserção e impressão dos elementos.

C. Implemente um programa que solicita ao usuário a quantidade de alunos de uma turma e aloca um vetor de notas (números reais). O programa deve ter duas funções: uma para leitura de notas no vetor e outra para imprimir a média aritmética das notas. Deve-se liberar a área de memória alocada antes de finalizar o programa.

D. Escreva uma segunda versão do programa anterior usando aritmética de ponteiros para varrer os vetores nas funções.

E. Modifique o programa anterior, incluindo o nome, série e nota de cada aluno (defina uma *struct* com esses campos). Faça as alterações necessárias, a fim de que as informações sejam preenchidas e calculada a média aritmética da turma.

quebra de página

****

Manipulação de arquivos

## Armazenamento permanente

As variáveis são armazenadas em memória temporária, podemos acessar os dados enquanto o programa está ativo. Como visto anteriormente, quando a execução do programa é encerrada, a memória alocada para as variáveis é liberada. Então, se é desejável que os dados sejam mantidos por mais tempo, os dados devem ser armazenados em arquivos.

Outro aspecto é a quantidade de informações; embora a capacidade de memória seja considerável, algumas aplicações podem exigir que uma quantidade maior de dados seja manipulada, requerendo outro recurso de armazenamento além da memória principal (Forbellone and Eberspächer 2005).

Há dois tipos de tratamento que podem ser dados a arquivos sequenciais em C/C++: texto ou binário (“Binary file” 2021). Nos arquivos texto, os caracteres são armazenados (inclusive as quebras de linha) e o arquivo pode ser aberto e compreendido facilmente por um programador ou usuário. A leitura e escrita nesses arquivos tem tamanho definido (conforme o tamanho das variáveis). O formato texto é utilizado em linguagens de marcação (HTML), mensagens (XML, JSON) e dados (CSV, TXT).

Nos arquivos binários, o conteúdo é armazenado como sequência de bytes, e apenas a aplicação consegue decodificar seu conteúdo. As operações de leitura e escrita não tem nenhum formato definido, a sequência de bytes deve ser tratada pelo programa. É o formato utilizado em imagens, áudio, vídeo, serviços e arquivos compactados[[4]](#footnote-3).

## Fluxos de dados em C/C++

Na linguagem C, não há funcionalidades específicas de entrada e saída da própria linguagem. Os compiladores dependem de bibliotecas externas para executar as operações de entrada e saída, tais como printf() e scanf(), fornecidas pela biblioteca stdio. C++ segue a abordam do padrão ANSI C e formaliza os pacotes de entrada e saída em bibliotecas como iostream e fstream (Saqib 2021).

Em C++, os arquivos são tratados como fluxos de dados de entrada e saída dos programas. Fluxos são um tipo de dados básico para manipular todas as operações de entrada e saída. Há diferentes tipos de fluxos de dados para entrada e saída, conforme o tipo de dados a que estão associados. Após essa associação (que equivale a declaração de uma variável), são usados os operadores padrão de redirecionamento de entrada e saída para esses fluxos (Saqib 2021).

A associação de um arquivo a um fluxo tratado pelo programa requer a abertura do arquivo e, quando não for mais utilizado, seu fechamento. Para isso, C e C++ usam bibliotecas distintas, como especificado a seguir.

## Manipulação de arquivos em C++

Em C++, a biblioteca fstream disponibiliza três tipos de fluxos (*stream*) que podem ser associados a arquivos (“C++ Files and Streams” 2021):

* **ofstream**: representa o fluxo de saída e é usado para criar arquivos e escrever dados nele;
* **ifstream**: representa o fluxo de entrada e é usado para ler informações de um arquivo;
* **fstream**: representa um fluxo geral, tendo as capacidades dos dois tipos anteriores (ofstream e ifstream), podendo criar arquivos, escrever e ler dados.

Assim, a instrução

**fstream arq (“arquivo.txt”);**

associa o identificador arq ao fluxo de entrada e saída para o arquivo arquivo.txt[[5]](#footnote-4). O mesmo pode ser feito com os outros tipos, sendo que a abertura do arquivo deve ser verificada, antes de executar qualquer operação. São sintaxes válidas para a abertura de arquivos:

**ifstream arq;**

**arq.open (“arquivo.txt”);**

**ofstream arq\_out (“arquivo.txt”);**

A partir disso, a leitura e escrita no arquivo podem ser feitas, utilizando os redirecionadores de fluxo << para a escrita dos dados e >> para a leitura dos dados. Por exemplo, a instrução

**arq << "Testando escrita de dados" << endl;**

irá escrever o texto entre aspas no arquivo, seguido pela quebra de linha, enquanto a instrução

**arq >> linha;**

irá ler dados do arquivo e atribuir o conteúdo à variável linha.

Concluídas as operações, o arquivo deve ser fechado, por meio da instrução close:

**arq.close();**

**Exemplo**

Neste exemplo, foram implementadas funções distintas para a escrita e leitura do arquivo. A função escreve\_arq() abre o arquivo “dados.txt” usando o tipo ofstream (linha 7); se o arquivo não existir, será criado. Nas linhas 8 a 11, está o bloco de verificação, para que a função seja encerrada caso a abertura do arquivo falhe. Na linha 12, o stream recebe, por meio do operador de redirecionamento, a string "Testando escrita de dados", seguida por um endl. A esse conteúdo, é adicionado o conteúdo da string linha (linha 16) e o arquivo é fechado (linha 17).

A função mostra\_arq() abre o arquivo “dados.txt” usando o tipo ifstream (linha 23) e é verificado se a operação foi bem sucedida (linhas 24 a 27). No bloco que inicia na linha 29, todo o conteúdo do arquivo (verificado pela condição !dados.eof()[[6]](#footnote-5)) será mostrado linha a linha. A função getline busca do *buffer* (dados) uma linha e atribui à variável linha, que é mostrada na linha 31.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | **#include <iostream> #include <fstream>  using namespace std;  int escreve\_arq(){  ofstream dados ("dados.txt"); // abre o arquivo para escrita  if (!dados){  cout << "Arquivo dados.txt nao pode ser aberto" << endl;  return -1;  }  dados << "Testando escrita de dados" << endl;  cout << "Insira linha: ";  string linha;  getline(cin, linha);  dados << linha << endl;  dados.close();  return 0;  }  int mostra\_arq(){  cout << "Abrindo o arquivo para leitura" << endl;  ifstream dados ("dados.txt"); // abre o arquivo para leitura  if (!dados){  cout << "Arquivo dados.txt nao pode ser aberto" << endl;  return -1;  }  string linha;  while (!dados.eof()){  getline(dados, linha);  cout << linha << endl;  }  dados.close();  return 0; }  int main(){  escreve\_arq();  mostra\_arq(); }** |
| --- | --- |

### 

### 

### Modos de abertura

É possível limitar o tipo de operação que pode ser efetuada no arquivo, conforme o modo de abertura (“Input/output file stream class” 2000):

| **Modo** | **Sintaxe** | **Descrição** |
| --- | --- | --- |
| input | ios::in | Arquivo aberto para leitura, operações de entrada são suportadas. |
| output | ios::out | Arquivo aberto para escrita, operações de saída são suportadas. |
| binary | ios::bin | Operações são executadas em um arquivo binário, e não em modo texto. |
| at end | ios::ate | O arquivo é aberto, sendo que a posição para saída inicia no fim do arquivo. |
| append | ios::app | Todas as operações de saída serão realizadas no fim do arquivo, sendo adicionadas ao conteúdo existente. |
| truncate | ios::trunc | Todo o conteúdo existente no arquivo antes da abertura será descartado. |

Esses modos podem ser combinados, desde que não sejam conflitantes[[7]](#footnote-6), utilizando o operador **ou** (pipe)(|).

**Exemplo**

No exemplo a seguir, o arquivo “dados.txt” está sendo aberto para leitura e escrita, usando a instrução fstream (linha 9), usando os modos in e out. Na linha 14, é inserido o conteúdo da *string* entre aspas, seguido de uma quebra de linha (endl), usando o operador de redirecionamento <<. O mesmo acontece com o conteúdo da variável n (linha 15). Quando essa instrução é executada, o identificador associado ao arquivo está apontando para a próxima posição onde um novo dado pode ser inserido. No entanto, a operação desejada é a impressão do conteúdo do arquivo; para isso, o identificador deve ser posicionado no início do arquivo. É isso que a instrução seekg, com o parâmetro zero, faz (linha 15). Então, o conteúdo do arquivo até a quebra de linha é atribuído à variável linha, usando a instrução getline (linha 18). Na linha 19, um valor inteiro é obtido do arquivo e atribuído à variável n, usando o operador de redirecionamento >>. O conteúdo das variáveis linha e n são impressos, permitindo que o conteúdo do arquivo seja visualizado. Como as operações no arquivo se encerraram, o arquivo é fechado, usando a instrução close (linha 22).

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **#include <iostream> #include <fstream>  using namespace std;  int main(){  string linha;  int n = 1001;  fstream dados ("dados.txt", ios::in|ios::out);  if (!dados){  cout << "Arquivo dados.txt nao pode ser aberto" << endl;  return -1;  }  dados << "Testando escrita de dados" << endl;  dados << n;  n = 0;  dados.seekg(0); //Posicionando no inicio do arquivo  getline(dados, linha);  dados >> n;  cout << "Leitura: " << linha << endl;  cout << "n = " << n << endl;  dados.close();  return 0; }** |
| --- | --- |

### Posicionando ponteiros em arquivos

Outra função necessária para a manipulação de arquivos é posicionar o identificador do fluxo. Considere, por exemplo, o caso de uma busca. Se o fluxo for recentemente aberto (desde que o modo de abertura não seja at end ou append), o identificador estará apontando para o início dos dados. No entanto, após a primeira busca e alguma instrução de leitura, para que uma nova busca seja realizada (lembrando que estamos manipulando arquivos sequenciais), é necessário voltar ao início dos dados:

**arq.seekg(0);**

As instruções **seekg** (“*seek get*”, para fluxo ifstream) e **seekp** (“*seek put*”, para fluxo ofstream) são responsáveis por reposicionar o identificador, conforme o parâmetro fornecido (“C++ Files and Streams” 2021). Em geral, o parâmetro é um inteiro. O segundo parâmetro (opcional) indica a direção de busca. A direção padrão é a partir do início (ios::beg), mas pode ser a partir da posição corrente (ios::cur) ou a partir do fim do fluxo (ios::end).

O identificador (ponteiro) de um arquivo é um valor inteiro, que especifica a localização no arquivo como um número de bytes a partir do início. Seguem alguns exemplos (“C++ Files and Streams” 2021):

* Posiciona no nº byte do arquivo (assume ios::beg):

**arq.seekg(n);**

* Posiciona em n bytes a partir da posição atual:

**arq.seekg( n, ios::cur );**

* Posiciona em n bytes a partir do fim do arquivo:

**arq.seekg( n, ios::end );**

* Posiciona no fim do arquivo:

**arq.seekg( 0, ios::end );**

**Exemplo**

Neste exemplo, uma lista de livros está sendo manipulada. Cada livro é descrito por seu título e código (conforme a *struct* nas linhas 7 a 10). Há três funcionalidades, disponibilizadas no main() e implementadas em funções isoladas: inserção, listagem e busca de registros.

Esse código contém alocação dinâmica, passagem de *stream* por parâmetro, função com retorno de registro. Observe atentamente a implementação desses conceitos.

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113 | **#include <iostream> #include <iomanip> #include <fstream>  using namespace std;  typedef struct {  string titulo;  int codigo; } Livro;  int mostra\_todos(){  Livro livro;  ifstream arq ("livros.txt");  if (!arq){  cout << "Nao foi possivel abrir o arquivo livros.txt" << endl;  return -1;  }  cout << "Codigo Titulo " << endl;  while (arq.good()){ //verifica eof ou alguma outra incorretude no arquivo  getline(arq, livro.titulo);  arq >> livro.codigo;  cout << setw(6) << livro.codigo << " ";  cout << livro.titulo << endl;  }  arq.close();  return 0; }  int insere\_livros(){  Livro livro;  ofstream arq ("livros.txt", ios::app);  if (!arq){  cout << "Nao foi possivel abrir o arquivo livros.txt" << endl;  return -1;  }  do{  cout << "Codigo (0 (zero) para finalizar): ";  cin >> livro.codigo;  if (livro.codigo != 0){  cout << "Titulo: ";  cin.get();  getline(cin, livro.titulo);  arq << livro.titulo << endl;  arq << livro.codigo;// << endl;  }  } while(livro.codigo != 0);  arq.close();  return 0; }  Livro\* busca\_livro(int cod, ifstream& arq){ //& especifica a passagem por referência  Livro \*livro;  livro = new (nothrow) Livro;  arq.seekg(0);  do{  getline(arq, livro->titulo);  arq >> livro->codigo;  } while (livro->codigo != cod && !arq.eof());  if (livro->codigo == cod)  return livro;  else  return NULL; }  int busca(){  int cod;  Livro \*livro;  ifstream arq ("livros.txt");  if (!arq){  cout << "Nao foi possivel abrir o arquivo livros.txt" << endl;  return -1;  }  do{  cout << "Qual livro deseja procurar (0 (zero) para finalizar)? ";  cin >> cod;  if (cod != 0){  livro = busca\_livro(cod, arq);  if (livro)  cout << "Titulo: " << livro->titulo << endl;  else  cout << "Livro nao cadastrado" << endl;  }  } while(cod != 0);  arq.close();  delete livro;  livro = NULL;  return 0; }  int main(){  char opcao;  do{  cout << "(I)nsere (L)ista (B)usca (F)inaliza" << endl;  cin >> opcao;  opcao = toupper(opcao);  cin.get();  switch (opcao){  case 'I':  insere\_livros() == -1? cout << "Nao foi possivel abrir o arquivo para**  **insercao de dados" : cout << endl;  break;  case 'L':  mostra\_todos() == -1? cout << "Nao foi possivel abrir o arquivo para**  **leitura dos dados" : cout << endl;  break;  case 'B':  busca();  break;  }  } while (opcao != 'F');  return 0; }** |
| --- | --- |

## 

## Manipulação de arquivos em C

### Operações comuns

**fopen()** – abre arquivo

**fclose()** – fecha arquivo

**putc()** – escreve um caractere no arquivo

**getc()** – lê um caractere no arquivo

**fprintf()** – é para um arquivo o que printf() é para console

**feof()** – devolve verdadeiro se o fim do arquivo for alcançado

**ferror()** – devolve verdadeiro se ocorrer um erro

**remove()** – apaga um arquivo

**rewind()** – recoloca o indicador de posição no início do arquivo

**fflush()** – descarrega um arquivo

**fseek()** – posiciona o arquivo em um byte específico

### Modos de abertura

**r** – abre um arquivo-texto para leitura

**w** – cria um arquivo-texto para escrita

**a** – anexa um arquivo –texto

**rb** – abre um arquivo binário para leitura

**wb** – cria um arquivo binário para escrita

**r+** - abre um arquivo-texto para leitura e escrita

**w+** - cria um arquivo-texto para leitura e escrita

**a+** - anexa ou cria um arquivo-texto para leitura e escrita

**r+b** – abre um arquivo binário para leitura e escrita

**w+b** – cria um arquivo binário para leitura e escrita

**a+b** – anexa ou cria um arquivo binário para leitura e escrita

### 

### Abrir um arquivo

**FILE \*fp;**

**if((fp=fopen(“test”, “w”)) == NULL)**

**{**

**printf(“O arquivo não pode ser aberto”);**

**exit(1);**

**}**

### Fechar uma arquivo

**int fclose(FILE \*fp);**

Em que fp é o ponteiro de arquivo devolvido pela chamada fclose();

**Exemplo 1**

| 1  2  3  4  5  6  7  8  9  10  11 | **# include <stdio.h> int main() {  file \*arq; //ponteiro para a estrutura file  arq = fopen ("aula1.txt", "r+"); // abre o arquivo aula1.txt com modo de leitura e escrita  fprintf(arq, "ola .\n"); // escreve a mensagem no arquivo aberto  fclose(arq); /\* fecha o arquivo  return 0; }** |
| --- | --- |

**Exemplo 2**

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | **#include <stdio.h> #include <stdlib.h>  typedef struct {  char titulo[20];  int codLivro; } regLivro;  void mostra\_arq(){  FILE \*arq ;  regLivro livro;  if ((arq = fopen("livros.bin", "rb" )) == NULL ) {  printf ("\nNao foi possivel abrir o arquivo \n");  exit (1);  }  rewind(arq);  while (fread(&livro, sizeof(livro), 1, arq) == 1) {  printf("\nTitulo: %s", livro.titulo);  printf("\nCodigo: %d", livro.codLivro);  printf("\n");  }  fclose (arq); }  void insere\_arq(){  FILE \*arq;  regLivro livro;  char opcao;  if ((arq = fopen("livros.bin", "ab" )) == NULL ) {  printf ("\nNao foi possivel abrir o arquivo \n");  exit (1);  }  do {  printf("\nTitulo: ");  fflush(stdin);  scanf("%19[^\n]%\*c", livro.titulo);  printf("\nCodigo: ");  fflush(stdin);  scanf("%d", &livro.codLivro);  fwrite(&livro, sizeof(livro), 1, arq);  printf("\nAdiciona outro livro (S/N)? ");  fflush(stdin);  scanf("%c", &opcao);  } while(toupper(opcao) == 'S');  fclose(arq); }  int main () {  insere\_arq();  mostra\_arq();  return 0; }** |
| --- | --- |

quebra de página

## Bibliografia

“Binary file.” 2021. Wikipedia. https://en.wikipedia.org/wiki/Binary\_file.

“C++ Files and Streams.” 2021. Tutorials Point. https://www.tutorialspoint.com/cplusplus/cpp\_files\_streams.htm.

“C++ Language.” 2000. C++ Tutorials. http://www.cplusplus.com/doc/tutorial/.

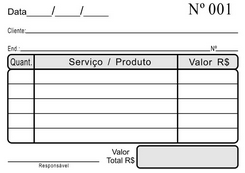
Forbellone, André Luiz V., and Henri F. Eberspächer. 2005. Lógica de Programação: A construção de algoritmos e estruturas de dados. 3ª ed. São Paulo: Pearson.

“Input/output file stream class.” 2000. C++ Reference. http://www.cplusplus.com/reference/fstream/fstream/open/.

Saqib, M. 2021. “File Handling in C++.” MYCPlus. https://www.mycplus.com/tutorials/cplusplus-programming-tutorials/file-handling-in-cpp/.

## Exercícios

A. Considere um pedido, no seguinte formato:



Implemente um programa com funções para ler e imprimir os campos e salvar os registros em um arquivo físico.

Utilize *structs* para a data (dia, mês e ano) e Itens do pedido (qtde, serviço/produto e valor). Os itens do pedido devem ser incluídos até que a quantidade lida seja 0. Lembre-se de totalizar o valor do pedido.

Defina os registros usando typedef e passagem de registros por parâmetro por referência.

quebra de página

1. A variável total havia sido inicializada com 0 e foi adicionada do valor apontado por p, ou seja, o conteúdo de n. Portanto, o valor de total é 100. [↑](#footnote-ref-0)
2. Por exemplo, na linha 4 do exercício A1, será mostrado o valor da variável que está sendo apontada por p\_contador (o valor de i), ou seja, 5. [↑](#footnote-ref-1)
3. Saiba mais: <https://stackoverflow.com/questions/54357190/c-dangling-pointers-and-memory-leaks> [↑](#footnote-ref-2)
4. Saiba mais: <https://blog.pantuza.com/artigos/diferencas-entre-arquivos-texto-e-binario> [↑](#footnote-ref-3)
5. Junto com o nome do arquivo, entre aspas, pode ser especificado todo o *path*, por exemplo: "C:/Users/Documents/dados.txt".  
   É necessário garantir que o diretório possua permissão para escrita e leitura. [↑](#footnote-ref-4)
6. O operador eof() indica fim de arquivo (acrônimo para *End Of File*). [↑](#footnote-ref-5)
7. Se são especificados os modos trunc e app, a instrução de abertura falha. O mesmo ocorre se o modo trunc é especificado, mas não o modo out. [↑](#footnote-ref-6)