

01

Linguagem de Programação

NewtonVirtual

Linguagem de Programação

1ª edição
2019

Autoria Ronan Loschi Rodrigues Ferreira
Parecerista Validador Emerson Paduan

*Todos os gráficos, tabelas e esquemas são creditados à autoria, salvo quando indicada a referência.

Informamos que é de inteira responsabilidade da autoria a emissão de conceitos. Nenhuma parte desta publicação poderá ser reproduzida por qualquer meio ou forma sem autorização. A violação dos direitos autorais é crime estabelecido pela Lei n.º 9.610/98 e punido pelo artigo 184 do Código Penal.

Sumário

Sumário

Unidade 1

1. Tratamento dedados homogêneos(vetor e matriz) em C.....8

Unidade 2

2. Ponteiros ou Apontadores.....21

Unidade 3

3. Alocação Dinâmica de Memória36

Unidade 4

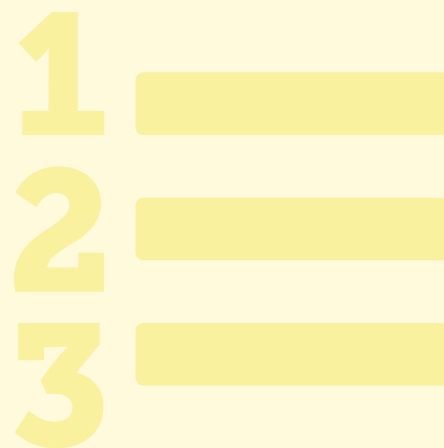
4. Registros ou Estruturas I.....54

Unidade 5

5. Registros ou Estruturas II68

Unidade 6

6. Programação Modular (Funções em C)83



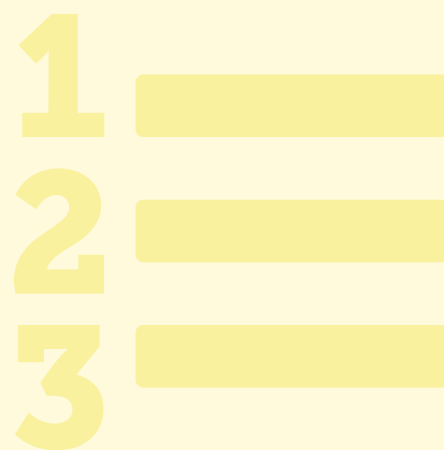
Sumário

Unidade 7

7. Recursividade.....97

Unidade 8

8. Manipulação de Arquivos 112





Palavras do professor

Seja bem-vindo(a) à disciplina de Linguagem de Programação! A proposta principal deste estudo é contextualizar o tema de maneira que você tenha condições de compreender do que se trata e de que modo utilizará a Linguagem de Programação em sua prática e escolha profissional.

Aqui, além das dicas e das boas práticas de programação, você encontrará os conceitos de linguagem de programação C, que servirão como base para a programação de computadores, sendo elas tratamento de dados homogêneos (vetor e matriz) em C, ponteiros ou apontadores, alocação dinâmica de memória, registros ou estruturas, programação modular e recursividade e manipulação de arquivos. Além disso, você será convidado a desenvolver um projeto de um *software* na linguagem C aplicado a problemas reais. Isso não é inspirador?

Ao final desta disciplina, você conseguirá compreender os conceitos de linguagem de programação, por meio da linguagem C e poderá, também, desenvolver os seus primeiros programas de computador, para aplicá-los a problemas reais. Desse modo, você poderá conhecer mais sobre a sua futura carreira profissional e decidir melhor sobre os caminhos que irá percorrer. Esses caminhos podem ser como empreendedor individual, empresário, concursado ou contratado tanto na área de programação de computadores quanto nas outras áreas da Tecnologia da Informação. Bons estudos e ótima sorte!





Objetivos da disciplina

- Descrever o conceito e a utilização de vetores e matrizes e suas aplicações.
- Esclarecer os conceitos de alocação dinâmica de memória e de ponteiro.
- Aplicar a construção de tipos de dados heterogêneos (registros).
- Explicar a programação modular.
- Empregar a recursividade por meio de exemplos.
- Praticar a manipulação de arquivos.

Unidade 1

1. Tratamento de dados homogêneos (vetor e matriz) em C



Para iniciar seus estudos

Nesta unidade, iniciaremos a nossa viagem pelo aprendizado sobre a linguagem de programação C. Durante essa viagem você conseguirá compreender o contexto da programação de computadores bem como escrever seus primeiros programas. Para isso, é importante organizar seus horários e o seu ambiente de estudos. Gostou? Então, vamos em frente!



Objetivos de Aprendizagem

- Enunciar os conceitos de estruturas de dados homogêneas unidimensionais (vetor) e multidimensionais (matriz).
- Manipular vetores e matrizes dos tipos *int*, *float*, *char*, *string*.
- Empregar a inicialização e o tratamento dos vetores e das matrizes.

Introdução da unidade

Nesta unidade, você estudará sobre os conceitos mais avançados para começar no desenvolvimento de programas de computador. Primeiramente, lhe serão apresentados conceitos sobre as estruturas de dados homogêneas unidimensionais (vetor) e multidimensionais (matriz) e sobre a manipulação de vetores e matrizes dos tipos *int*, *float*, *char*, *string*. E, ao final, você conhecerá como empregar, inicializar e tratar vetores e matrizes.

1.1 Conceito de estruturas de dados homogêneas unidimensionais (vetor) e multidimensionais (matriz)

Tanto os vetores como as matrizes são *arrays*, a diferença entre os dois é o número de dimensões, um *array* pode ser entendido como vários espaços de memória alocados de forma conjunta e em sequência. Eles podem ser acessados pelo mesmo identificador, quando vetor, sendo que cada campo é diferenciado somente pelo índice. Pode, também, ser diferenciado pelos índices, quando matriz. Esses índices representam qual posição eles ocupam dentro do arranjo. Conforme Deitel (2011):

Um *array* é um conjunto de espaços de memória que se relacionam pelo fato de que todos têm o mesmo nome e o mesmo tipo. Para se referir a um local ou elemento em particular no *array*, especificamos o nome do *array* e o número da posição do elemento em particular no *array* (DEITEL, 2011, p. 160).

1.1.1 Vetores

Um vetor pode ser entendido como uma variável que armazena mais de um valor. É importante considerar que o número de valores que a variável conseguirá armazenar deve ser definido no momento da sua declaração por ser ela uma estrutura estática.

Conforme definição de Lorenzi (2007):

O tipo vetor permite armazenar mais de um valor em uma mesma variável. O tamanho dessa variável é definido na sua declaração, e seu conteúdo é dividido em posições. Nessa estrutura todos os elementos são do mesmo tipo, e cada um pode receber um valor diferente. (LORENZI, 2007, p. 7).

Lorenzi (2007) ainda apresenta algumas características de um vetor, apresentando-o como uma estrutura estática, homogênea, com alocação sequencial dos *bytes* e com possibilidade de inserção e exclusão com realocação de elementos e sem liberação de memória quando algum elemento é excluído.

Para declarar um vetor em linguagem C a sintaxe é bem simples, bastando que se coloque o tipo dos dados a serem armazenados, o identificador do vetor a ser utilizado para referenciá-lo durante o código e, entre colchetes, o número de posições que o vetor deverá reservar na memória para uso. A seguir, é mostrada a sintaxe para a declaração de um vetor.

tipo<identificador>[<número de posições>];

Para exemplificar a sintaxe utilizada para a declaração de vetores, é mostrada, a seguir, a declaração de um vetor de inteiros com 10 posições.

```
int vet[12];
```

Os dados cadastrados no vetor devem ser acessados por meio do identificador utilizado na declaração, acompanhado do índice referente à posição que ele ocupa dentro da estrutura. Conforme Ascencio (2012, p. 155), o primeiro endereço alocado para um vetor é sempre tratado como o índice 0; Lorenzi (2007, p. 7) complementa que, “a partir do endereço do primeiro elemento é possível determinar a localização dos demais elementos do vetor”. A última posição será o tamanho do vetor diminuído de 1. A Figura 1, apresentada a seguir, representa os valores alocados sequencialmente na memória, parte em azul, e a forma de acesso a cada posição utilizando o identificador vet.

Figura 1 – Representação de um vetor

vet[0]	-45
vet[1]	6
vet[2]	0
vet[3]	72
vet[4]	1543
vet[5]	-89
vet[6]	0
vet[7]	62
vet[8]	-3
vet[9]	1
vet[10]	6453
vet[11]	78

Fonte: Adaptado de DEITEL; DEITEL, 2011, p. 161.



Saiba mais

Estruturas estáticas são aquelas que não podem ter o tamanho modificado em tempo de execução, pois o tamanho delas tem que ser definido na declaração. Já as estruturas homogêneas são aquelas que só armazenam um tipo de dados.

1.1.2 Matrizes

Para Lorenzi (2007, p. 7), “uma matriz é um arranjo bidimensional ou multidimensional de alocação estática e sequencial. Todos os valores da matriz são do mesmo tipo e tamanho, e a posição de cada elemento é dada pelos índices, um para cada dimensão.” A autora descreve, ainda, as principais características da estrutura,

classificando-a como homogênea, bi ou multidimensional, com alocação estática e posições contíguas de memória. A declaração de uma matriz bidimensional ocorre conforme o apresentado na sintaxe a seguir:

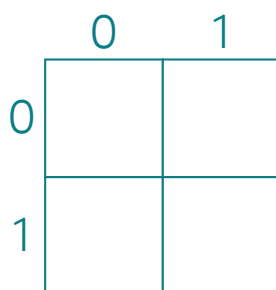
```
tipo<identificador> [<dimensão 1>][<dimensão 2>;
```

Para exemplificar a sintaxe utilizada na declaração de matrizes, é mostrada, a seguir, a declaração de uma matriz de inteiros com duas linhas (dimensão 1), por duas colunas (dimensão 2).

```
int matriz[2][2];
```

Assim como explicado para os vetores, os índices de uma matriz também se iniciam no 0 e cada dimensão vai até seu tamanho diminuído de 1. Na Figura 2, é demonstrada uma representação da matriz 2x2 declarada anteriormente. Observe que os índices vão de 0 a 1 em cada dimensão.

Figura 2 – Representação de uma matriz



Fonte: Adaptado de LORENZI, 2007, p. 9.



Fique atento!

Tanto em vetores quanto em matrizes a linguagem C não controla o fim da cadeia ao implementar essas estruturas. É importante que se tome muito cuidado para não acessar indevidamente outras áreas da memória que não as reservadas para a estrutura.

1.1.3 Declarar vetores e matrizes dos tipos: *int*, *float*, *char*, *string*

Conforme mostrado na apresentação e na conceituação de vetores e de matrizes, essas estruturas representam um arranjo de variáveis comuns que são armazenadas de forma sequencial na memória, portanto elas podem ser declaradas com os mesmos tipos já conhecidos para as variáveis comuns. Aqui, serão demonstradas as formas de declaração para os tipos mais comuns: *int*, *float*, *char* e *string*.

A declaração de um vetor de inteiros é bem simples de ser feita e já foi, inclusive, utilizada no exemplo anterior. Para exemplificar o uso desse vetor em um problema, imagine que é necessário que sejam digitadas as idades (em anos completos) de 50 pessoas. Considerando-se que a informação deverá ser armazenada como inteiro, teríamos a seguinte declaração para esse vetor:

```
int idades[50];
```

Em que foi criado um vetor de idades (identificador definido pelo programador), com 50 posições reservadas em sequência na memória para o tipo inteiro.

Um vetor do tipo real (*float*) segue a mesma forma de declaração utilizada para o vetor de inteiros, o que muda é o tipo. Como exemplo, imagine uma situação em que fosse necessário digitar a nota final dos 100 alunos de uma turma. Considerando-se que as notas finais representam a média das notas do semestre, essa informação sempre será declarada no tipo real. Para criação do vetor, basta colocar o tipo, o identificador e o número de posições desse vetor, conforme apresentado a seguir:

```
float notas[100];
```

Com essa declaração, foi criado um vetor de notas (identificador definido pelo programador), com 100 posições reservadas em sequência na memória para o tipo *float*.

Situações podem exigir que vetores de caracteres sejam criados para trabalhar os dados e, nesse caso, assim como nos outros já mostrados, a declaração segue a mesma sintaxe conforme será visto a seguir no vetor cargos, criado para receber o código do cargo de 50 empregados de uma empresa.

```
char cargos[50];
```

O vetor de *char*, conforme apresentado, é utilizado para armazenar vários caracteres que não necessariamente formam uma palavra, por isso um vetor de *char* não deve ser confundido com uma *string*. *String* não é um tipo primitivo na linguagem C, sendo tratado como um conceito que utiliza o tipo *char* para ser implementado. Ascencio (2012, p. 304) diz que a linguagem C não possui um tipo *string*, e que para declarar uma variável que receberá um texto é necessária a criação de um vetor de *char* onde cada posição é um caractere do texto. O que diferencia a *string* é a forma como ela é tratada no código.

Segundo explicado por Deitel (2011, p. 171), no final de uma *string* é acrescentado o caractere `\0` (nulo), isso mostra uma diferença importante para um vetor de caracteres quanto ao armazenamento que deve ser observado no momento da declaração da variável *string*, que deve ter uma posição livre no vetor para esse caractere, portanto é necessário que a *string* receba um nome com 20 caracteres e que o vetor seja declarado com 21 posições, conforme a representado a seguir:

```
char nome[21];
```

A leitura de valores *string* pode ser feita da mesma forma que a leitura de outros tipos, utilizando-se *scanf* com a *string* de controle `%s` que formata a entrada para o tipo texto. Para ler o nome declarado anteriormente, o comando de entrada ficaria assim:

```
scanf("%s",&nome);
```

Segundo Ascencio (2012, p. 306), o uso de *scanf* na leitura de *string* tem um problema, quando é digitado um texto composto por mais de uma palavra, ele só realiza a leitura até encontrar o primeiro espaço, desconsiderando todo o resto. Para resolver esse problema, deve ser utilizado o comando de entrada *gets()* que, para a *string* nome, ficaria assim:

```
gets(nome);
```

O código completo mostrado a seguir utiliza a função *gets()* para a leitura de *strings*.

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

int main(){

    char nome [51], endereco [101];
    setlocale(LC_ALL, "Portuguese");
    printf("Digite o Nome:");
    gets(nome);
    printf("Digite o Endereço Completo:");
    gets(endereco);
    printf("\nNome:%s\nEndereço: %s\n",nome, endereco);
}
```

Embora a leitura possa ser feita da mesma forma que é feita a leitura de variáveis primitivas da linguagem, a manipulação de uma *string* exige funções especiais para quase todas as operações, essas funções são disponibilizadas para o código por meio da inclusão do arquivo de cabeçalho *string.h*. O quadro 1 apresenta e descreve as funções mais utilizadas para a manipulação de *strings*.

Quadro 1 – Funções para manipulação de *strings*

Função de cadeia	Descrição
strcat(destino, origem)	Concatena a <i>string</i> origem no final da <i>string</i> destino.
strncat(destino, origem, n)	Concatena a <i>string</i> origem no final da <i>string</i> destino, usando no máximo n caracteres da origem.
strcmp(str1, str2)	Compara, pela ordem alfabética, as duas <i>strings</i> . Retorna zero se forem iguais, menor que 0 se $str1 < str2$, maior que 0, se $str1 > str2$.
strcmpi(str1, str2)	Compara as duas <i>strings</i> sem levar em conta maiúsculas e minúsculas.
strlen(str)	Calcula o comprimento da <i>string</i> sem o caractere nulo.
strlwr(str)	Converte cadeia para minúsculas.
strupr(str)	Converte cadeia para maiúsculas.
strcpy(destino, origem)	Copia cadeia origem para destino.

Fonte: Adaptado de ASCENCIO, 2012, p. 171.

A declaração de matrizes bidimensionais é feita com a definição do tamanho das duas dimensões. O trecho a seguir demonstra a declaração de três matrizes, uma do tipo inteiro, outra do tipo real e a última do tipo caractere. Todas as matrizes são três linhas por quatro colunas (3 x 4):

```
int idades[3][4];
float notas[3][4];
char cargos[3][4];
```

As demais regras já explicadas para vetores se aplicam às matrizes bidimensionais, o que as difere é o número de dimensões.

1.1.3.1 Vetores e matrizes do tipo texto (*string*)

Uma *string*, assim como já explicado, é um vetor de *char* com um tratamento especial no código que o torna uma *string*. Dessa forma, um vetor de *strings* será um vetor de vetores de *char*, o que na prática é uma matriz de *char*. No momento da declaração, deve-se observar que a primeira dimensão indica o tamanho do vetor e a segunda, o tamanho da *string*. O trecho de código a seguir demonstra a declaração de um vetor de *strings* com 10 posições, onde cada *string* poderá ter até 50 caracteres (mais o caractere de fim de *string* \0).

```
char nomes[10][51];
```

Para declarar uma matriz bidimensional de *strings* é necessário definir as duas dimensões e o tamanho da *string*, desse modo é declarada a matriz bidimensional da mesma forma que uma matriz de três dimensões. Na declaração mostrada a seguir, é criada uma matriz de três linhas por duas colunas, que comporta *strings* de até 50 caracteres.

```
char nomes[3][2][51];
```



Refleta

A estrutura, para percorrer uma matriz de *strings*, pode ser a mesma usada em uma matriz de outros tipos?

1.1.4 Inicializar e tratar vetores e matrizes

Um vetor pode receber valores de diversas formas, o que deve ser observado é se os dados a serem adicionados são compatíveis com o tipo declarado para o vetor e, ao atribuir o valor a uma posição, é obrigatório que se utilize o identificador do vetor juntamente ao índice da posição. A seguir, é apresentado um trecho de código que demonstra a inserção de um valor *float* diretamente na primeira posição do vetor *notas* declarado anteriormente.

```
notas[0]=10.5;
```

Quando é necessário o preenchimento de todo o vetor, é preciso buscar outra forma de inserir os valores na estrutura, pois não seria viável preencher um vetor inteiro colocando-se uma linha de código para cada posição, já que esses *arrays* podem ter desde poucas, até milhares ou milhões de posições conforme a necessidade. Como as posições do vetor são índices sequenciais é possível, e muito prático, o uso de estruturas de repetição para percorrer todo um vetor. No exemplo apresentado a seguir, um vetor de *float* com 100 posições é totalmente preenchido com valores digitados pelo usuário.

```
for(i=0; i<=99; i++){ // Repetição acontecerá com valores de 0 a 99

    printf("Digite a %iª idade:",i+1);
    scanf("%f",&notas[i]);
    // entrada de dados diretamente no vetor da mesma.
}
```



Saiba mais

A entrada de dados no vetor é feita da mesma forma que em uma variável comum, só é adicionado o índice onde o valor será inserido no vetor. No caso do preenchimento mostrado, será alocado o primeiro valor digitado na primeira posição ($i=0$), o segundo, na segunda posição ($i=1$) e, assim, até o último, quando i é igual a 99, nesse caso para um vetor de 100 posições.

Para exemplificar o uso de vetores, pense em um exercício em que seja necessário receber a idade de 50 pessoas e, depois de todas as idades digitadas, deve-se calcular e imprimir quantas dessas pessoas podem votar. Considere que pessoas com idade superior ou igual a 16 anos podem votar.

No código que resolve esse exercício, foi declarado um vetor de inteiros com o nome `idade` e 50 posições para receber as 50 idades. O tamanho do vetor foi declarado por meio da constante `TAMANHO_VETOR` e utilizado na declaração do `array`, assim como no limite das estruturas de repetição, nas quais o valor é usado diminuído de uma unidade. A primeira estrutura de repetição permite que o vetor seja inteiramente preenchido da forma já explicada. A segunda estrutura realiza a contagem das pessoas que já podem votar, utilizando a lógica do contador junto à estrutura de dados. Ao final, é impresso como resultado, o número de pessoas que podem votar.

```
#include<stdio.h>
#include<stdlib.h>
#include<locale.h>
#define tamanhoVetor 50

int main(){

    // Função responsável por mostrar corretamente caracteres
    // como acentos, cedilha entre outros.
    setlocale(LC_ALL, "Portuguese");
    int cont,i;
    int idades[tamanhoVetor];

    // Preenchimento do vetor
    for(i=0;i<=tamanhoVetor-1;i++){

        printf("Digite a %iª idade:",i+1);
        scanf("%i",&idades[i]);
    }
```

```
// Contagem das pessoas que podem votar (idade >=16)
cont=0;

for(i=0;i<=tamanhoVetor-1;i++){

    if(idades[i]>=16){

        cont++;

    }

}

// impressão do número de pessoas que podem votar
printf("%i pessoas desse grupo podem votar!",cont);
}
```



Saiba mais

No código que demonstra o preenchimento de um vetor de inteiros incluído no arquivo de cabeçalho locale.h e no código, foi utilizada uma função dessa biblioteca, a setlocale(LC_ALL, "Portuguese"), essa função permite que caracteres próprios da língua portuguesa sejam mostrados de forma correta pelos comandos de saída.

Assim como nos vetores, as matrizes também podem ser preenchidas diretamente em cada posição, a diferença é que em uma matriz bidimensional, ao invés de um índice por posição, são utilizados dois, um se refere à linha e outro, à coluna. O preenchimento da última posição de uma matriz 3x3 (duas linhas por duas colunas) é mostrado no código a seguir:

```
matriz[2][2]=15.2;
```

Assim como já explicado nos vetores, é pouco prático o preenchimento de cada posição da matriz, conforme mostrado anteriormente. O preenchimento total de uma matriz pode ser feito utilizando-se estruturas de repetição que, nesse caso, deverão ser duas, uma para controlar o índice referente às linhas e outra para o índice referente às colunas. O trecho de código mostrado a seguir apresenta a digitação de valores nos campos de uma matriz.

No código, o índice i controla a mudança de linhas da matriz, enquanto o índice j marca a coluna a ser preenchida. O preenchimento ocorre na estrutura de repetição mais interna e é sempre feito coluna a coluna, enquanto a repetição mais externa salta a linha para todas as colunas de todas as linhas a serem preenchidas.

```
for(i=0;i<=2;i++){

    for(j=0;j<=2;j++){

        printf("MATRIZ[%i][%i]:",i,j);
        scanf("%f",&matriz[i][j]);

    }

}
```


A seguir, é mostrado um exemplo de código completo, no qual a matriz é completamente preenchida e depois mostrada por inteiro. As estruturas de repetição no processo de mostrar são iguais às vistas no preenchimento, mas os valores dentro da matriz agora também são impressos.

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

int main(){

    setlocale(LC_ALL, "Portuguese");
    int j,i;
    float matriz[3][3];

    //Preenchimento da matriz
    for(i=0;i<=2;i++){

        for(j=0;j<=2;j++){

            printf("MATRIZ[%i][%i]:",i,j);
            scanf("%f",&matriz[i][j]);
        }
    }

    //Impressão da matriz inteira
    for(i=0;i<=2;i++){

        printf("\n");
        for(j=0;j<=2;j++){

            printf("MATRIZ[%i][%i]= %f  ",i,j,matriz[i][j]);
        }
    }
}
```

1.1.4.1 Manipular vetores e matrizes de *strings*

Quando é necessário percorrer o vetor de *string* para preencher ou imprimir os valores que estão nele, isso deve ser feito utilizando-se somente a primeira dimensão da declaração. A segunda dimensão é o tamanho do texto suportado e é preenchida automaticamente quando o tipo é definido como *string* na leitura com o uso do *gets()*. No código a seguir, são mostrados os processos de preenchimento e de impressão do vetor de *strings* nome, com 10 posições para *strings* de 50 caracteres.

```

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

int main(){

    int i;
    char nomes[10][51];
    setlocale(LC_ALL, "Portuguese");

    //Preenchimento do vetor de Strings
    for(i=0;i<=9;i++){

        printf("Digite o %iº nome completo:",i+1);
        gets(nomes[i]);
    }

    printf("\n");

    //Impressão do vetor inteiro
    for(i=0;i<=9;i++){

        printf("%s \n",nomes[i]);
    }
}

```

Para percorrer uma matriz bidimensional de *strings*, o processo é feito da mesma forma que ocorre em matrizes de outros tipos. Considerando-se que cada campo é endereçado por dois índices, são utilizadas duas estruturas de repetição aninhadas (uma dentro da outra), para que uma salte as linhas e a outra preencha as colunas daquela linha, a dimensão que representa o tamanho da *string* só é usada na declaração. O código a seguir mostra como é feito o acesso à matriz para inserção ou impressão de valores.

```

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

int main(){

    int i,j;
    char nomes[3][2][51];
    setlocale(LC_ALL, "Portuguese");

    //Preenchimento da matriz de Strings
    for(i=0;i<=2;i++){

        for(j=0;j<=1; j++){

            printf("nome [%i][%i]:",i,j);
            gets(nomes[i][j]);
        }
    }
}

```

```

printf("\n");

//Impressão da matriz de strings
for(i=0;i<=2;i++){

    printf("\n");
    for(j=0;j<=1; j++){

        printf("nome [%i][%i]= %s ",i,j,nomes[i][j]);

    }
}
}

```

Síntese da unidade

Você viu os conceitos e implementações para vetores, matrizes e *strings*. Viu, ainda, que um vetor é uma sequência de memória alocada de forma contígua, para que possa ser acessada por um só identificador, seguido pelo índice entre colchetes. Conheceu as características dessa estrutura que é estática, homogênea e unidimensional, devendo ser declarada com um tamanho e um tipo definidos. Foi apresentado a um código em que um vetor é percorrido para preenchimento e impressão com as explicações sobre seu funcionamento.

Você conheceu a estrutura do tipo matriz e viu que a única diferença entre ela e um vetor é o número de dimensões, dessa forma também pôde identificar as características do vetor como sendo uma estrutura estática, homogênea e multidimensional. A implementação de uma matriz bidimensional foi mostrada e explicada com destaque para o preenchimento das duas dimensões.

O uso e a manipulação de variáveis do tipo texto foram apresentados com destaque, uma vez que o tipo *string* não existe na linguagem C e se apresenta como um conceito que é implementado com um vetor de *char*, em que o número de posições representa o tamanho do texto a ser armazenado mais o nulo (\0), que indica o fim da *string*. A declaração e a manipulação de vetores e de matrizes de *strings* também foram destacadas, mostrando-se que a maior diferença está na declaração, enquanto o funcionamento do código permanece o mesmo para percorrer a estrutura. Foram apresentadas e explicadas as implementações para códigos com *string*, vetores de *strings* e matrizes de *string*.



Considerações finais

Parabéns, você chegou ao final da unidade 1 desta disciplina e deu o primeiro passo, e o mais importante, para iniciar a construção de programas de computador em linguagem C com o uso de estruturas de dados homogêneas (vetores e matrizes). Ao entender os conceitos e a implementação de vetores e de matrizes, é possível criar programas que utilizam a memória de forma mais organizada e eficiente, já que a linguagem C possibilita o uso de vários tipos de estruturas de dados mais avançadas, porém conhecer bem as primeiras estruturas apresentadas é o passo mais importante no entendimento desses conceitos. Agora, siga em frente!



Unidade 2

2. Ponteiros ou Apontadores



Para iniciar seus estudos

Esta unidade tratará sobre ponteiros. Você aprenderá o conceito de ponteiros, verá como declarar e utilizar as variáveis ponteiro, conhecerá também os operadores específicos para trabalhar com endereços e a forma de manipular a memória utilizando essas variáveis especiais com exemplos de aplicação baseadas em matrizes e *strings*. Achou interessante? Então, vamos lá!



Objetivos de Aprendizagem

- Definir e compreender a funcionalidade de ponteiros em C.
- Identificar os riscos no uso de ponteiros.
- Empregar ponteiros comuns em C.
- Empregar ponteiros para ponteiros em C.

Introdução da unidade

Estruturas do tipo vetor e matriz alocam e manipulam a memória de uma forma mais avançada que as variáveis comuns, e, nesta unidade, conheceremos os ponteiros, que permitem acesso direto à memória e são a base para a maioria das estruturas de dados mais avançadas com alocação dinâmica de memória.

Em linguagem C, os ponteiros são muito importantes para o desenvolvimento de código com maior qualidade, uma vez que permitem a manipulação direta da memória e com isso dão um poder maior ao programador para a criação da lógica de programação. Ponteiros são tão necessários em C que são utilizados muitas vezes sem que se perceba; bons exemplos são a declaração de matrizes e a passagem de parâmetros por referência em uma função.

Assim, nesta unidade, veremos como criar ponteiros e manipular diretamente os endereços de memória, assim como alguns cuidados que devem ser tomados ao se acessar a memória diretamente pelos endereços.

2.1 Ponteiros (apontadores) em C

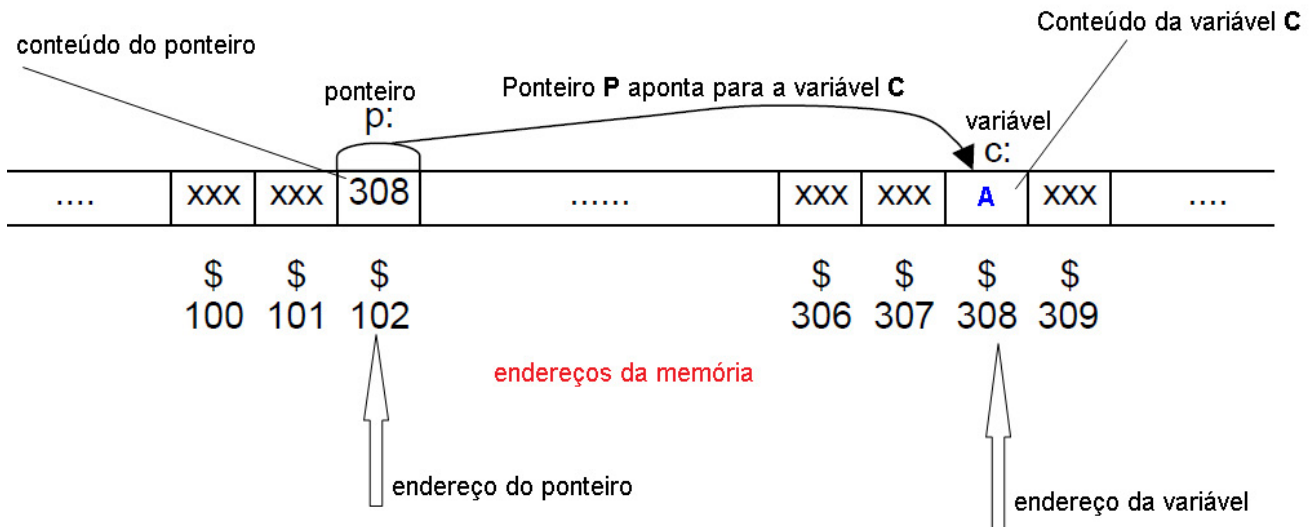
Soffner (2013) diz que ponteiros podem ser utilizados sempre que necessário manipular alguma área de memória diretamente ou em situações onde seja necessária a alocação dinâmica de memória ou a chamada por referência em funções.

Para Deitel (2011, p. 209), os ponteiros “[...] permitem que os programas simulem uma chamada por referência e criem e manipulem estruturas dinâmicas de dados, ou seja, estruturas de dados que podem crescer e encolher no tempo de execução [...]”.

2.1.1 Ponteiros como endereços

“O nome de uma variável indica o que está armazenado nela. O endereço de uma variável é um ponteiro. Seu valor indica em que parte da memória do computador a variável está alocada” (MIZRAHI, 2008, p. 257). Para entender melhor o conceito de ponteiro tomaremos como base a representação feita na Figura 3. Ela demonstra alocação de uma variável ponteiro apontando para uma variável comum na memória, onde o ponteiro está no endereço 102, e seu conteúdo aponta para o endereço 308, onde está a variável C. Veja:

Figura 3 – Alocação do ponteiro e da variável



Fonte: elaborada pelo autor.



Fique atento!

Ao utilizar ponteiros a memória é acessada de forma direta pelos seus endereços, e isso é muito útil, mas também muito perigoso, pois tudo que for alterado em um endereço de memória será alterado também em qualquer variável que utilize aquele endereço. No caso do exemplo, se o valor da posição \$308 for alterada, o valor de C também será.

2.1.2 Variáveis ponteiros

Variáveis ponteiros podem ser definidas como um tipo especial de variável que aponta para posições de memória. “Um ponteiro é uma variável que aponta sempre para outra variável de um determinado tipo. Para indicar que uma variável é do tipo ponteiro, coloca-se um asterisco antes dela” (DAMAS, 2006, p. 178).

A declaração da variável ponteiro é feita de forma similar à declaração de uma variável comum; a única diferença é o uso do asterisco que indica se tratar de um ponteiro. Veja: **tipo** *<identificador-da-variável>. A seguir, são mostrados exemplos da declaração de ponteiros de vários tipos:

```
char *p1, *ptnome
int *temp, *ptinício, *ptnota
```

Nesses exemplos, os ponteiros estão declarados, mas não foram ainda inicializados. Isso pode representar um risco à integridade dos dados e até mesmo ao funcionamento do sistema operacional, pois os ponteiros, quando criados, apontam para algum lugar que não pode ser definido. Portanto, ao se manipular o ponteiro nessas circunstâncias, ele pode alterar valores na memória que já estão sendo utilizados.

Cormen (2002, p. 15) diz que um ponteiro que não aponta para nada, deve ser apontado para NULL. Dessa forma, o ideal seria a declaração conforme descrita a seguir:

```
char *p1=NULL,*ptnome=NULL
int *temp=NULL, *ptinicio=NULL, *ptnota=NULL
```

NULL é um valor vazio que garante que o ponteiro não esteja apontando para nenhuma posição válida da memória. Iniciar ponteiros dessa forma é uma boa prática de programação, e isso evita o acesso não intencional às áreas da memória.

Para testar e entender melhor a diferença entre *declarar* e *inicializar* e *somente declarar*, o código a seguir faz a impressão antes e depois dos ponteiros terem sido inicializados:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char *p1,*ptnome;
    int *temp, *ptinicio, *ptnota;
    printf(“%p - %p - %p - %p - %p \n\n”, p1,ptnome,temp,ptinicio,ptnota);
    p1=NULL;
    ptnome=NULL;
    temp=NULL;
    ptinicio=NULL;
    ptnota=NULL;
    printf(“%p - %p - %p - %p - %p \n\n”, p1,ptnome,temp,ptinicio,ptnota);
}
```



Saiba mais

Para imprimir o endereço contido em um ponteiro, seja ele de qualquer tipo, é utilizada a *string* de controle %p. Isso fará com que o endereço seja impresso na notação hexadecimal.

2.1.3 Operadores ponteiros

Para a manipulação de ponteiros, são utilizados dois operadores, conforme mostrado no Quadro 2:

Quadro 2 – Operadores de ponteiros

Operador	Significado
Operador de endereço (&)	Endereço de uma variável
Operador indireto (*)	Conteúdo do endereço especificado

Fonte: adaptado de Deitel (2011, p. 2010).

Veja que o operador de endereço já é utilizado no comando de entrada *scanf* que você utiliza em seus códigos. O operador retorna o endereço da variável onde o valor digitado deve ser inserido na memória. A partir de agora, esses operadores serão utilizados também em outras aplicações. O trecho de código que será apresentado a seguir mostra a declaração de uma variável inteira (*count*) e de uma variável ponteiro para inteiros (*pt*). Note que, depois das declarações, o ponteiro *pt* recebeu o endereço da variável *count*.

A última linha mostra o operador indireto sendo utilizado para atribuir o valor 30 no endereço para onde o ponteiro aponta. Dessa forma, o valor 30 é inserido na variável *count*, alterando seu valor. Observe:

```
int count=10
int*pt=NULL
pt=&count
*pt=30
```

Para testar e entender melhor os operadores de ponteiros, a seguir, é mostrado o código comentado que manipula as variáveis e ponteiros, imprimindo seus valores. Observe:

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

int main()
{
    int count=10;
    int *pt=NULL;
    pt=&count;
    setlocale(LC_ALL, "Portuguese");
    //Ao imprimir os valores nesse momento percebe-se que o
    //valor contido na posição apontada pelo ponteiro e o
    //valor da variável são os mesmos.
    printf("O valor contido na variável count é %i\n", count);
    //imprime o endereço contido no ponteiro (conteúdo do
    //ponteiro)
    printf("O endereço contido no ponteiro é %p\n", pt);
    //imprime o conteúdo do endereço de memória apontado por
    //pt
    printf("O valor contido no endereço armazenado no ponteiro é %i\n", *pt);
    //quando é utilizado o operador indireto do ponteiro ele
    //acessa, a variável para a qual aponta de forma indireta,
    //quando recebe, o valor 30 esse valor é alterado na //variável count para onde ele
    aponta.
    *pt=30;
    printf("O valor contido na variável count agora é %i\n", count);
    printf("O valor contido no endereço armazenado no
    ponteiro agora é %i\n", *pt);
    //Ao imprimir os valores depois da atribuição com o
    //operador indireto, nos dois casos será impresso 30.
}
```

Veja na Figura 4 a execução do código exibindo onde são impressos os valores.

Figura 4 – Execução do código

```
0 valor contido na variável count é 10
0 endereço contido no ponteiro é 000000000062FE44
0 valor contido no endereço armazenado no ponteiro é 10
0 valor contido na variável count agora é 30
0 valor contido no endereço armazenado no ponteiro agora é 30
```

Fonte: elaborada pelo autor.



Fique atento!

O endereço apresentado nas telas de execução de código pode variar (e muito provavelmente irá variar). Ao executar o código ele mostrará o endereço da memória conforme a alocação feita, mas não necessariamente igual ao apresentado aqui, visto que, em computadores diferentes ou em momentos diferentes, a alocação será feita nas posições disponíveis da memória naquele momento.

2.1.4 Expressões com ponteiros

Variáveis ponteiros também podem ser utilizadas em expressões, assim como as variáveis comuns. Porém, devem ser utilizadas com bastante cuidado, pois os operadores de ponteiros fazem toda a diferença nessa hora. Caso o conteúdo do ponteiro esteja sendo manipulado diretamente (sem o uso do operador *), ele aceitará somente endereços de memória, e, dessa forma, não é possível atribuir o conteúdo de uma variável comum a uma variável ponteiro.

Quando é utilizado o operador indireto, a atribuição pode ser feita de acordo com o tipo do ponteiro, mas, nesse caso, o valor não será inserido na variável ponteiro, e sim na variável para a qual ele aponta.

O código comentado a seguir demonstra como podem ser feitas as atribuições com ponteiros.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *pt, valor1, valor2;
    valor1=15;
    // inicializando o ponteiro pt com o endereço de valor1
    pt=&valor1;
    // valor2 recebe o conteúdo de pt, ou seja 15
    valor2 = *pt;
    // o ponteiro pt modifica o valor da variável valor1,
    // para a qual ele aponta. Ou seja, valor1=12;
    *pt=12;
    // valor2 recebe o conteúdo da variável para a qual pt aponta, //ou seja, valor2=12.
```

```

    valor2 = *pt;
    printf("%i %i \n\n", valor1, valor2);
    system("PAUSE");
    return 0;
}

```

Também é possível comparar dois ponteiros em uma expressão relacional. Por exemplo, para dados dos ponteiros *p* e *q*, a seguinte declaração é perfeitamente válida. Veja:

```
if (p < q) printf("p aponta para um endereço menor que q\n");
```

2.1.5 Aritmética com ponteiros

Todas as operações feitas com variáveis comuns podem ser realizadas com acesso indireto a essas variáveis por meio de ponteiro. O código que demonstra a operação aritmética de soma sendo feita com o uso de acesso indireto ao endereço contido nos ponteiros de variáveis *valor1* e *valor2* são inicializadas com os valores 10 e 15 respectivamente. Os ponteiros *pt1* e *pt2* são apontados para o endereço dessas duas variáveis, na mesma ordem. Nessa operação, é realizada a soma do valor que está no endereço apontado por *pt1* com o valor que se encontra onde aponta o *pt2*. Ao imprimir a variável resultado que recebe a operação, o valor é 25, o que prova que indiretamente o conteúdo das variáveis *valor1* e *valor2* foram somados.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *pt1=NULL,*pt2=NULL, valor1,valor2, resultado;
    valor1=10;
    valor2=15;
    pt1=&valor1;
    pt2=&valor2;
    resultado=*pt1 + *pt2;
    printf("%i \n",resultado);
    system("PAUSE");
    return 0;
}

```

O endereço dos ponteiros também pode ser alterado em tempo de execução através de operações aritméticas. Quando se deseja incrementar o endereço do ponteiro basta, que se utilize o nome do ponteiro (sem o asterisco) seguido do `++`. Quando se deseja decrementar o endereço, indo para o anterior, basta seguir a mesma lógica com o `--`. Existe também a possibilidade de saltar vários endereços de uma vez com a atribuição do próprio ponteiro acrescido do número de posições a serem avançadas ou diminuído das posições a serem retrocedidas.

O quadro a seguir mostra como utilizar os operadores nos ponteiros para alterar os endereços ou para alterar o conteúdo da memória apontado.

Quadro 3 – Operações com ponteiros

Endereço	Conteúdo
pt1 ++; //próximo endereço de memória	(*pt1)++; //conteúdo apontado acrescido de 1
pt1 --; //endereço anterior na memória	(*pt1)--; //conteúdo apontado subtraído de 1
pt1=pt1+9; // 9 endereços à frente	(*pt1)-=2; // //conteúdo apontado subtraído de 2

Fonte: elaborado pelo autor.

O operador de endereço ++ que aponta o próximo endereço na memória será demonstrado na explicação sobre ponteiros e matrizes.

2.1.6 Ponteiros e matrizes

Para Pereira (2016), as matrizes são arranjos de variáveis alocadas de forma sequencial na memória. Quando se declara uma matriz, seja ela uni ou multidimensional, o que o compilador faz é reservar o número necessário para alocar os campos declarados sequencialmente.

No caso da matriz unidimensional, o identificador da matriz marca a posição inicial dentre as reservadas, e o índice indica qual das posições deve ser acessada dentre elas.

No código apresentado a seguir, é declarada uma matriz unidimensional de nome vet, ela é preenchida em uma estrutura de repetição com valores de 1 a 10, o ponteiro ptvetor recebe o endereço do início da matriz, e, na segunda estrutura de repetição, são impressos todos os valores que foram inseridos no vetor, mas isso é feito por meio do ponteiro, que é atualizado a cada vez para a próxima posição alocada. Veja:

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    int *ptvetor, i,vet[10];
    for(i=0;i<=9;i++)
    {
        vet[i]=i+1;
    }
    ptvetor=vet;
    for(i=0;i<=9;i++)
    {
        printf("%p - %i\n",ptvetor,*ptvetor);
        ptvetor++; // passa para o próximo endereço
    }
}
```

O resultado da execução do código pode ser visto na Figura 5, em que os valores impressos por meio do ponteiro são os mesmos armazenados no vetor. Os endereços de memória representam uma sequência de valores com a diferença de quatro posições, isto é, esse número de posições é necessário para armazenar cada inteiro.

Figura 5 – Execução do código de ponteiro e vetor

```

000000000062FE10 - 1
000000000062FE14 - 2
000000000062FE18 - 3
000000000062FE1C - 4
000000000062FE20 - 5
000000000062FE24 - 6
000000000062FE28 - 7
000000000062FE2C - 8
000000000062FE30 - 9
000000000062FE34 - 10

```

Fonte: Elaborada pelo autor.

Conforme abordado, as matrizes podem ser uni ou multidimensionais, mas é importante observar que essas posições são só uma forma lógica de organizar as informações. Na prática, o que acontece na memória é a alocação sequencial de posições para a matriz.

O próximo código mostra uma matriz 4x5 declarada e inicializada com valores de 1 a 5 em cada linha. O ponteiro ptmatriz recebeu o endereço da mat[0][0], que é a primeira posição da matriz, e utilizando somente o ponteiro foram impressos os valores que estavam armazenados na matriz. Observe:

```

#include<stdlib.h>
#include<stdio.h>
int main()
{
    int *ptmatriz, i, j, mat[4][5];
    for(i=0; i<=3; i++)
    {
        for(j=0; j<=4; j++)
        {
            mat[i][j]=j+1;
        }
    }
    ptmatriz = &mat[0][0];
    for(i=0; i<=19; i++)
    {
        printf("%p - %i\n", ptmatriz, *ptmatriz);
        ptmatriz++;
    }
}

```

Na Figura 6, é mostrado o resultado da execução do código. Nela aparecem os valores de 1 a 5 repetidos por quatro vezes, assim como o endereço de memória onde está cada valor.

Os endereços de memória estão dispostos de forma sequencial com a diferença de quatro campos entre um e outro, conforme já explicado, ou seja, são necessárias quatro posições de memória para armazenar um valor do tipo inteiro.

Figura 6 – Execução do código de ponteiro e matriz

```
000000000062FDF0 - 1
000000000062FDF4 - 2
000000000062FDF8 - 3
000000000062FDFC - 4
000000000062FE00 - 5
000000000062FE04 - 1
000000000062FE08 - 2
000000000062FE0C - 3
000000000062FE10 - 4
000000000062FE14 - 5
000000000062FE18 - 1
000000000062FE1C - 2
000000000062FE20 - 3
000000000062FE24 - 4
000000000062FE28 - 5
000000000062FE2C - 1
000000000062FE30 - 2
000000000062FE34 - 3
000000000062FE38 - 4
000000000062FE3C - 5
```

Fonte: Elaborado pelo autor.

2.1.7 Indexação de ponteiros

Os ponteiros são indexados de acordo com o tipo da variável para a qual apontam. No exemplo anterior, foi possível ver que um ponteiro para inteiros será indexado de quatro em quatro posições a partir da posição inicial para a qual ele foi apontado na memória. O código mostrado a seguir demonstrará como é indexado um ponteiro para os tipos *float*, *int*, *char* e *bool*.

No código, os ponteiros, um de cada tipo, são inicializados com variáveis de seus respectivos tipos. Depois, dentro de uma estrutura de repetição, esses ponteiros são incrementados e passam a apontar novas posições de memória. Em cada tipo (*int*, *float*, *double*, *char*, etc.), o número de posições de memória necessário para alocar uma variável é diferente.

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    int i, *ptinteiro, inteiro;
    float *ptreal, real;
    char *ptchar, caracter;
```

```

bool *ptbool, booleano;
ptinteiro=&inteiro;
ptreal=&real;
ptchar=&caracter;
ptbool=&booleano;
for(i=0;i<=5;i++)
{
    printf("|Inteiro - %p |Real- %p |Char- %p"
    "|Boolean- %p|\n", ptinteiro, ptreal, ptchar, ptbool);
    ptinteiro++;
    ptreal++;
    ptchar++;
    ptbool++;
}
}

```

Analisando o resultado da execução do código, podemos perceber que, para os tipos inteiro e real, o ponteiro é indexado de quatro em quatro posições de memória. Já no caso de ponteiros do tipo *char* e *bool*, o tamanho necessário para alocação é de uma posição de memória.

Figura 7 – Execução do código de ponteiro de tipos diferentes

Inteiro - 000000000062FE24	Real- 000000000062FE20	Char- 000000000062FE1F	Boolean- 000000000062FE1E
Inteiro - 000000000062FE28	Real- 000000000062FE24	Char- 000000000062FE20	Boolean- 000000000062FE1F
Inteiro - 000000000062FE2C	Real- 000000000062FE28	Char- 000000000062FE21	Boolean- 000000000062FE20
Inteiro - 000000000062FE30	Real- 000000000062FE2C	Char- 000000000062FE22	Boolean- 000000000062FE21
Inteiro - 000000000062FE34	Real- 000000000062FE30	Char- 000000000062FE23	Boolean- 000000000062FE22
Inteiro - 000000000062FE38	Real- 000000000062FE34	Char- 000000000062FE24	Boolean- 000000000062FE23

Fonte: Elaborado pelo autor.

Na Figura 7, é possível notar que, como esses ponteiros são alocados em sequência na memória, eles podem ser acessados por índices assim como em uma matriz, desde que criados como um arranjo de valores, conforme pode ser observado no código a seguir:

```

#include<stdlib.h>
#include<stdio.h>
#include<string.h>
int main()
{
    int *pt1, i;
    //alocação dinâmica de memória
    pt1 = (int*) malloc(sizeof(int) * 5);
    for(i=0; i<=4; i++)
    {
        printf("Digite o %iº valor inteiro:",i+1);
        scanf("%i",&pt1[i]);
    }
    for(i=0; i<=4; i++)
    {
        printf(" %iº valor = %i\n",i+1,pt1[i]);
    }
    free(pt1);
}

```



Saiba mais

A função **malloc** é utilizada para alocação dinâmica de memória (feita em tempo de execução), sua sintaxe é **newPtr=malloc(sizeof(struct node))**.

2.1.8 Ponteiros e *strings*

A *string* em C, assim como já abordado, é declarada como um vetor de caracteres. Quando se pretende utilizar um ponteiro para manipular a *string*, o que deve ser feito é criar o ponteiro, alocar a memória no tamanho necessário e utilizar o ponteiro da mesma forma que seria utilizado o vetor de *char*.

No código apresentado a seguir, foi declarado o ponteiro *pstring*, alocada a memória para uma palavra de até 30 caracteres, feita a leitura da *string* via teclado, utilizando a função *gets()*, e, na sequência, foi impresso o valor lido usando o ponteiro formatado como %s, para impressão de *strings*.

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
    char *pstring;
    pstring = malloc(sizeof(char)*31);
    setlocale(LC_ALL, "Portuguese");
    printf("Digite uma palavra de até 20 caracteres:");
    gets(pstring);
    printf("A palavra é %s",pstring);
    free(pstring);
}
```

2.1.9 Ponteiros para ponteiros

Ponteiro para ponteiro é uma variável que recebe o endereço de outra variável ponteiro, enquanto uma variável ponteiro comum aponta para o endereço de uma variável comum. Para declarar um ponteiro para ponteiro é importante que o identificador seja declarado antecedido de dois asteriscos, conforme visto na sintaxe abaixo:

tipo_da_variável **identificador_da_variável

Ao acessar de forma indireta os valores apontados, é importante considerar o seguinte:

- Quando se quer acessar de forma indireta o conteúdo da variável final, devem ser utilizados dois asteriscos (**) para acessar o conteúdo que está sendo apontado pelo ponteiro que é apontado por ele.

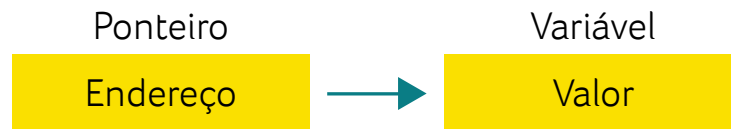
****nome_da_variável** é o conteúdo final da variável apontada

- Quando se utiliza um só operador indireto (*), o valor acessado é o endereço do ponteiro apontado.

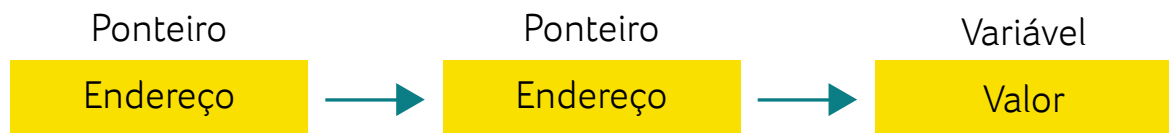
***nome_da_variável** é o conteúdo do ponteiro intermediário

A Figura 8 mostra um ponteiro para variável na primeira representação e, na segunda, um ponteiro para ponteiro.

Figura 8 – Representação de ponteiro para ponteiro



Indireção múltipla



Fonte: Adaptada de DEITEL; DEITEL, 2011.

Para demonstrar como funciona o ponteiro para ponteiro, será apresentado um código onde pt1 é um ponteiro comum e pt2 é um ponteiro para ponteiros.

```

#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
int main()
{
    int *pt1,**pt2, valor;
    setlocale(LC_ALL, "Portuguese");
    pt1=&valor;
    valor=100;
    pt2=&pt1;
    printf("Conteúdo do ponteiro pt2: %p\n",pt2);
    printf("Conteúdo do ponteiro apontado por (*pt2): ""%p\n",*pt2);
    printf("Conteúdo do endereço armazenado pelo"
    "ponteiro apontado por (**pt2): %i",**pt2);
}
  
```

Na implementação o ponteiro pt1 recebe o endereço da variável valor e, depois, de forma indireta, carrega o valor 100 na variável. O ponteiro pt2 recebe o endereço de pt1(pt2 = &pt1;) e passa a apontar para o endereço do ponteiro.

A impressão dos valores para demonstração é feita por pt2. Na primeira impressão, é impresso o conteúdo do ponteiro de forma direta; na segunda impressão, é impresso o conteúdo do ponteiro apontado (ou seja, o endereço armazenado em pt1) de forma indireta, e, na terceira, é impresso o valor armazenado no endereço apontado por pt1 de forma indireta (com dois asteriscos por ser um ponteiro para ponteiro).

Veja que, após ser executado o código, no resultado é possível visualizar que a primeira linha mostra o endereço armazenado em pt2; a segunda mostra, de forma indireta, o endereço apontado por pt1; e a última mostra o conteúdo da memória apontada por pt1, também de forma indireta.

Figura 9 – Execução do código de ponteiro para ponteiro

```
Conteúdo do ponteiro para pt2: 000000000062FE40
Conteúdo do ponteiro apontado por (*pt2): 000000000062FE3C
Conteúdo do endereço apontado pelo ponteiro apontado por (**pt1): 100
```

Fonte: Elaborado pelo autor.



Refleta

Caso na atribuição feita, onde `pt2=&pt1`, o operador de endereço (&) não fosse utilizado, conceitualmente pt2 estaria trabalhando como um ponteiro para ponteiro ou como um ponteiro simples?

Síntese da unidade

Nesta unidade, foi possível aprender que ponteiros são variáveis especiais que armazenam endereços de memória em vez de conteúdo final a ser guardado. Foi possível aprender também que, por isso, é importante se inicializar ponteiros com NULL, para evitar que acessem endereços indevidos.

Para a manipulação dos ponteiros, foram apresentados os operadores de endereço e as operações aritméticas com apontadores, com destaque para a manipulação dos endereços com incremento ou decremento de endereços.

O uso de ponteiro junto às matrizes também foi abordado com foco para a forma como matrizes ficam alocadas na memória; seus endereços foram impressos, o que mostrou que as matrizes multidimensionais são só uma organização lógica, uma vez que os dados ficam sempre armazenados de forma sequencial na memória.

Ponteiros foram utilizados com indexação, assim como matrizes, a partir de arranjos alocados na memória de forma dinâmica (em tempo de execução). A criação de uma matriz utilizando somente apontadores foi apresentada na manipulação de *strings* por ponteiros, onde foi feita a alocação dinâmica de memória utilizando a função **malloc**, possibilitando que o ponteiro fosse utilizado normalmente como uma *string*.

E, por fim, foram mostrados os ponteiros para ponteiros que, como o próprio nome já diz, apontam para o endereço de outros ponteiros, em vez da variável final.



Considerações finais

Os conhecimentos apresentados na unidade de aprendizado são muito importantes para o entendimento correto da linguagem C, pois o uso de ponteiros deixa o programador mais livre para manipular os dados da forma que necessitar, sem precisar se prender aos tipos e estruturas predefinidos pela linguagem. Para um entendimento completo dessa matéria, é fundamental que você acesse todos outros materiais da unidade.



3

Unidade 3

3. Alocação Dinâmica de Memória



Para iniciar seus estudos

Esta unidade tratará dos temas relacionados à alocação de espaços na memória do computador para que um futuro programa possa armazenar informações sem tamanho previamente definido. A partir disso, poderemos criar programas mais flexíveis. Vamos lá?



Objetivos de Aprendizagem

- Aplicar os conceitos de alocação dinâmica de memória em C.
- Entender a alocação de memória em tempo de execução.
- Aplicar as funções: *sizeof()*, *malloc()*, *free()* e *realloc()*.

Introdução da unidade

Nesta unidade, conheceremos as funções de alocação dinâmica da memória na Linguagem C. Abordaremos os conceitos sobre as funções de alocação dinâmica e, posteriormente, sobre a alocação e desalocação dessa memória. E, por fim, compreenderemos a forma correta de aplicar tais conhecimentos na criação de programas que racionalizarão o uso da memória do computador.

3.1 Alocação dinâmica

Shildt (1996) explica que os ponteiros fornecem suporte necessário para o poderoso sistema de alocação dinâmica de C. A **alocação dinâmica** é o meio pelo qual um programa pode obter memória enquanto está em execução.

Para Shildt (1996), nem variáveis globais nem locais podem ser acrescentadas durante o tempo de execução. Porém, haverá momentos em que um programa precisará usar quantidades de armazenamento variáveis. Por exemplo, um processador de texto ou um banco de dados aproveita toda a RAM de um sistema. Porém, como a quantidade de RAM varia entre computadores, esses programas não poderão usar variáveis normais. Em vez disso, esse e outros programas alocam memória conforme necessário, usando as funções de sistema de alocação dinâmica de C.

O uso de bibliotecas na Linguagem C evita erros de programação. Para fazer uso das funções de alocação dinâmica de memória, é preciso incluir uma biblioteca específica chamada *stdlib.h*.

3.1.1 Conceitos de alocação dinâmica

Toda a memória disponível para alocação no computador é conhecida como *heap*, ou seja, memória livre. Nessa memória, serão armazenadas variáveis e programas que possuem seus tamanhos em *bytes*. Porém, em algumas arquiteturas computacionais, esses valores tendem a sofrer variação de tamanho. Além dessa particularidade, quando se utilizam estruturas de dados, o cálculo desses *bytes* torna-se muito mais complexo. Nesse caso, não é possível saber de antemão quantas entradas de dados serão fornecidas e não será possível fazer cálculos da quantidade de *bytes* necessários para armazenar dados de uma determinada aplicação. Assim sendo, para resolver esse problema com a Linguagem C, usaremos a função *sizeof()*, que retornará o tamanho em *bytes* do espaço necessário ao armazenamento na memória, que, em conjunto com as funções *malloc()*, *realloc()* e *free()*, possibilitará a alocação adequada ao tipo de dado que precisa ser armazenado na memória. A seguir será explanada cada uma delas.



Fique atento!

Para desenvolver programas que fazem uso de funções de alocação dinâmica, é imprescindível que a biblioteca *stdlib.h* seja incluída no cabeçalho das aplicações:

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

3.1.2 Alocação de espaços na memória durante a execução de um programa

A memória do computador é gerenciada pelo sistema operacional que realiza o carregamento ou execução de aplicativos. Para realizar essa tarefa, o sistema operacional reserva uma região na memória na qual armazenará o aplicativo. Para isso, utiliza os espaços vazios (*heap*), nesse caso, reservados para essa operação.

Após carregar o aplicativo na memória, o sistema operacional recebe instruções de execução do aplicativo para que reserve espaços na memória, com o objetivo de manipular informações de entrada. Quando esse espaço reservado não é suficiente para a operação a ser realizada pelo aplicativo, podemos solicitar mais memória ao sistema operacional usando as funções de alocação dinâmica.

3.1.2.1 Funções: *sizeof()*, *malloc()*, *free()*, *realloc()*

Função *sizeof()*

Autores, como Damas (2007) e Shildt (1996), tratam essa função como um operador. Apesar disso, entendemos que a *sizeof()* apresenta o comportamento de uma função, por isso, assim a apresentaremos.

Damas (2007) explica que o tamanho em *bytes* de um inteiro varia de arquitetura para arquitetura, sendo os valores habituais de dois a quatro *bytes*.

Ainda, Damas (2007) afirma que é importante saber o tamanho de um inteiro quando se desenvolve uma aplicação. Caso contrário, corre-se o risco de tentar armazenar um valor numa variável inteira com um número de *bytes* insuficiente.

Diante dessas informações sobre diferentes arquiteturas, utilizaremos a função *sizeof()* para passar o tamanho exato em *bytes* dos inteiros para as funções de alocação dinâmica.

Sintaxe:

```
Sizeof < expressão >
      ou
sizeof (< tipo >)
```

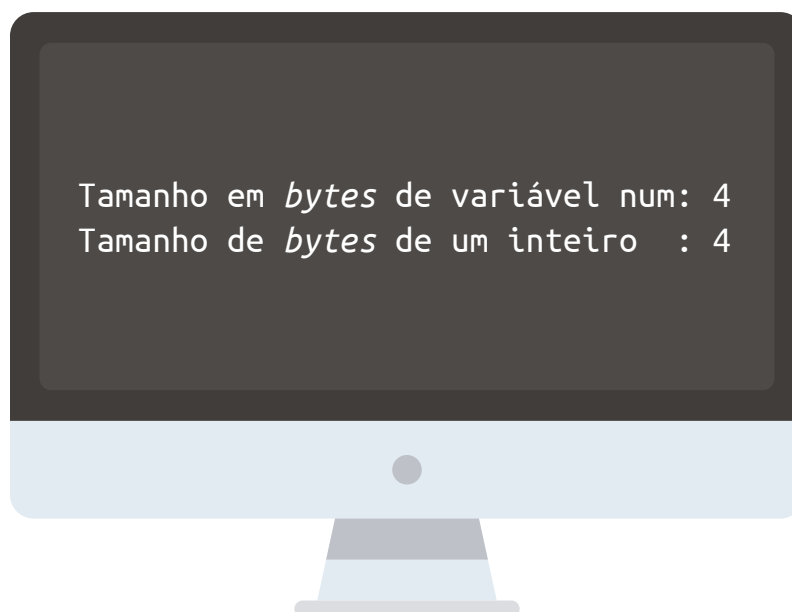
O código a seguir imprime na tela o tamanho de um inteiro em *bytes*, dependendo da arquitetura, este valor poderá variar.

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

int main(){

    int num;
    printf("\nTamanho em bytes da variavel num: %i\n",sizeof num);
    printf("Tamanho em bytes de um inteiro: %i\n",sizeof(int));
}
```

Figura 10 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Função *malloc()*

A função *malloc()* (abreviatura de *memoryallocation*) recebe como argumento um número inteiro positivo que representa a quantidade de *bytes* de memória a ser alocada. O número de *bytes* é especificado no argumento da função. A função solicita ao sistema operacional que reserve espaço na memória e este retorna um ponteiro *void** (de tipo genérico) com o endereço do primeiro *byte* do novo bloco de memória que foi alocado. Se não houver memória suficiente para satisfazer a exigência, a função *malloc()* retornará um ponteiro com o valor *NULL*. Os *bytes* alocados são inicializados com lixo (MIZRAHI, 2009).

O retorno do valor *NULL* permite fazer um tratamento de erro ou desenvolver novas linhas de programação para nos desviar do problema de falta de memória, o que causaria erro na execução do programa.

Em relação ao lixo a que se refere a autora, pode-se afirmar que são instruções ou dados de outras aplicações que foram abandonados pelos programas que fizeram uso dessa região de memória. Assim sendo, essas informações ficam na memória.

A seguir, será apresentado o protótipo da função *malloc()*:

```
void *malloc(size_t n_Bytes)
```

Size_t: é definido na biblioteca *stdlib.h*. Ele corresponde a unsigned *int*, ou seja, inteiro positivo ou inteiro sem sinal, sendo que este só poderá armazenar endereços de memória.

A seguir veja o exemplo da alocação de 1 *byte*:

```
#include <stdio.h>
#include <stdlib.h> //Biblioteca para usar as funções de alocação.

int main(void){

    char *num;
```

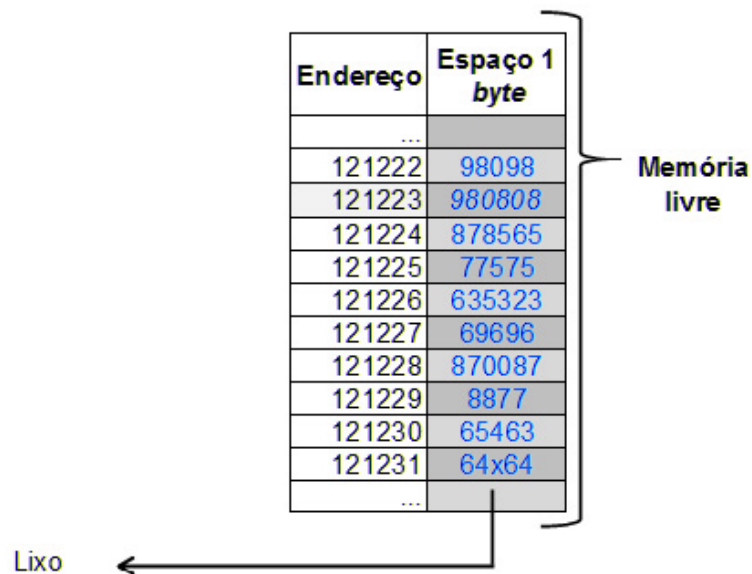
```

num = malloc (1); //1 byte alocado
*num = 'H';
printf ("Resultado: %c", num); // saída dos dados armazenados no endereço ponteiro num.
}

```

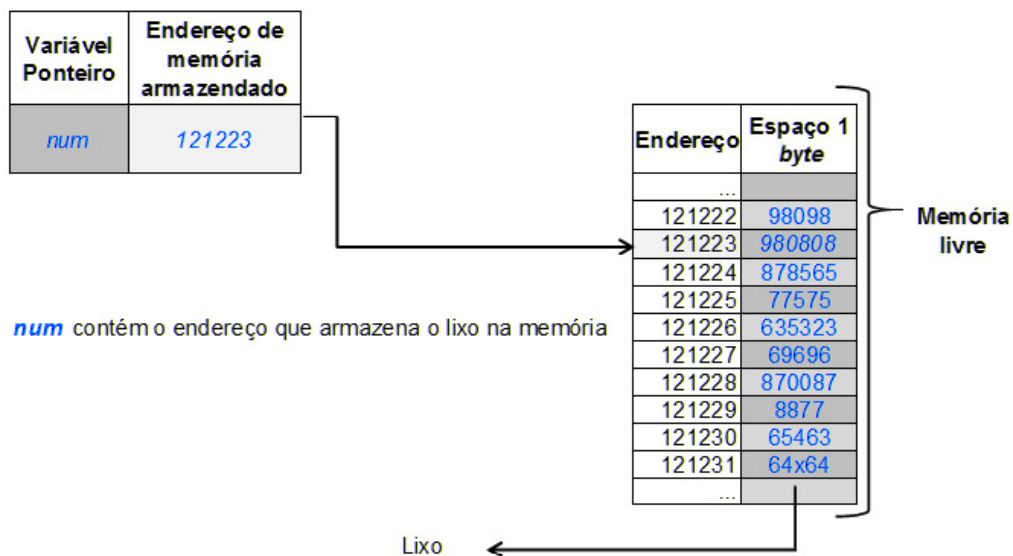
As figuras a seguir demonstram visualmente como ocorre a alocação de 1 *byte* na memória do computador. Nas figuras, tomamos a coluna de endereços de memória com endereçamento e informações fictícios para demonstrar o funcionamento desse processo.

Figura 11 – Região da fictícia de áreas livres na memória



Fonte: Elaborada pelo autor.

Figura 12 – Região da memória com o ponteiro *num* reservando uma região de 1 *byte*:



Fonte: Elaborada pelo autor.



Saiba mais

É essencial dominar os conceitos e práticas de ponteiros para compreender melhor o funcionamento das funções de alocação dinâmica, que, essencialmente, usam ponteiros.

A respeito das diferentes arquiteturas computacionais, sabe-se que passar valores diretamente para a função *malloc()* poderá causar erro no tamanho a ser reservado para tipos inteiros. Para evitar esse tipo de erro, o melhor a ser aplicado é o uso da função *sizeof()*, que retorna o tamanho exato dos *bytes* a serem reservados para que o tipo de dado seja armazenado sem erro.

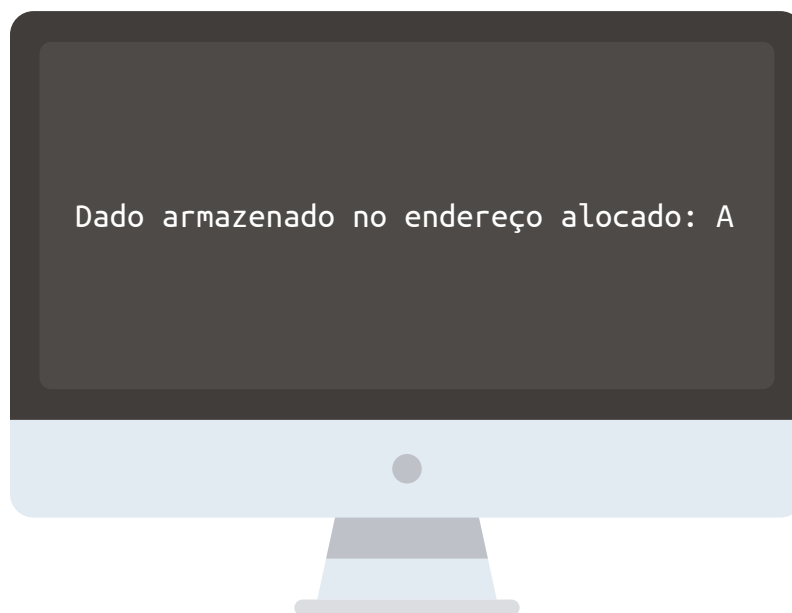
O exemplo a seguir faz o uso da função *sizeof()* aplicada ao código:

```
#include <stdio.h>
#include <stdlib.h> //Biblioteca de funções de alocação.

int main(void){

    char *ptr;
    ptr = malloc(sizeof(char));
    *ptr = 'A';
    printf("Dado armazenado no endereço alocado:%c",*ptr);
}
```

Figura 13 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

No código apresentado, veja que a linha sublinhada poderia ser facilmente substituída por uma operação de multiplicação, como pode ser observado a seguir:

```
char *ptr = malloc( 5 * sizeof(char));
```

Aplicando esses conceitos do uso da função `sizeof()`, alocaremos uma região na memória para armazenar uma estrutura de dados com múltiplos valores. Veja:

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

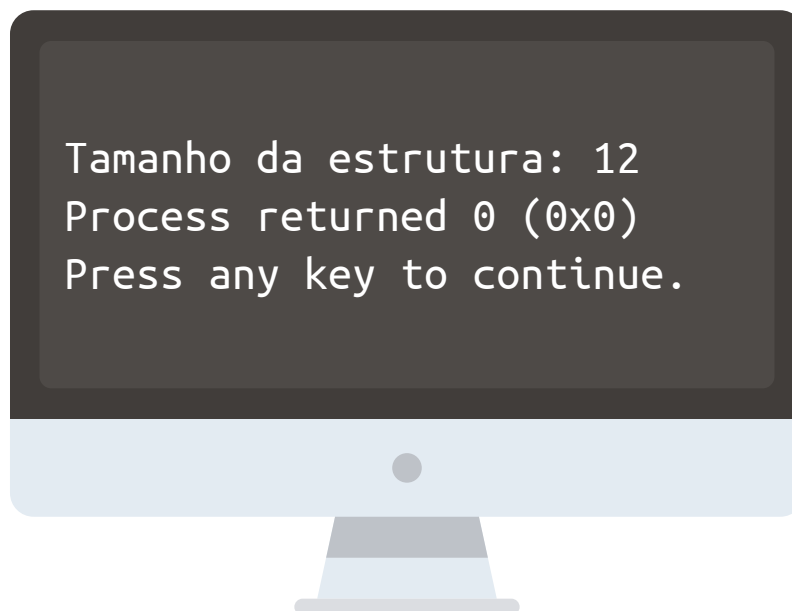
struct x{

    int a;
    int b;
    int c;
};

int main(void){

    struct x estrutura;
    int *y = malloc ( sizeof(estrutura));
    printf("%li", sizeof(estrutura));
}
```

Figura 14 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

O comando **printf** no final do código exibirá a quantidade de *bytes* armazenados para a estrutura na memória.



Saiba mais

Observe que, em alguns compiladores, é preciso forçar a alocação usando cast para evitar erro de compilação:

```
int *y = (int*) malloc ( sizeof(estrutura));
```

O uso de **cast** força que o retorno de um ponteiro void* genérico seja convertido num tipo inteiro.

Função *free()*

Uma vez alocada a memória dinamicamente, ela continuará ocupada até que seja desalocada explicitamente pela função *free()*. Em outras palavras, uma variável criada com a função *malloc()* existirá e poderá ser acessada em qualquer parte do programa enquanto não for liberada por meio da função *free()*, bem como seu espaço de memória devolvido ao sistema operacional (MIZRAHI, 2009).

Segundo Schildt (1996), a função *free()* é o oposto de *malloc()*, visto que ela devolve memória previamente alocada ao sistema. Uma vez que a memória tenha sido liberada, ela pode ser reutilizada por uma chamada subsequente de *malloc()*.

A seguir, será apresentado o protótipo da função *free()*:

```
Void free (void *p);
```

No programa exemplificado a seguir, **pi* é um ponteiro para a memória alocada anteriormente por *malloc()*.

Exemplo:

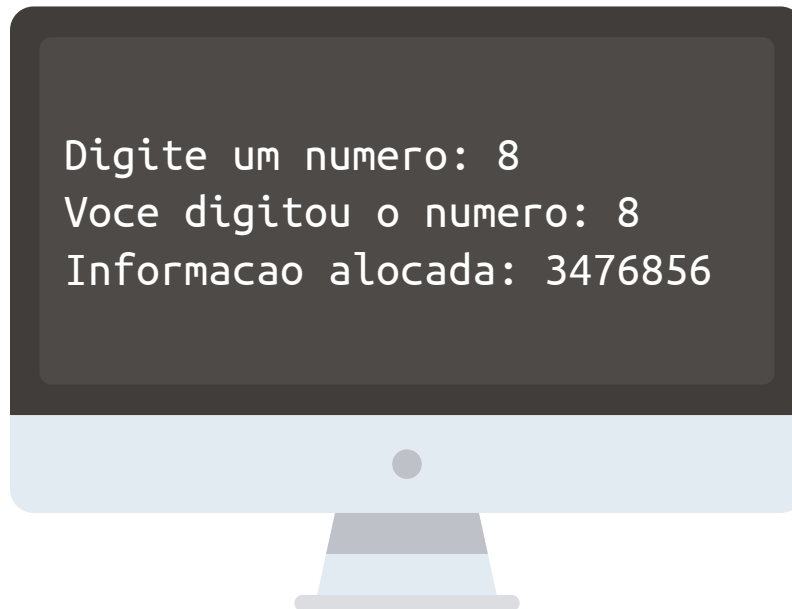
```
# include <stdio.h>
# include <stdlib.h>

int main (){

    int *pi;
    pi = (int *) malloc(sizeof(int));
    printf("\n Digite um numero : ");
    scanf("%d", pi);
    printf("\n Voce digitou o numero: %d\n", *pi);
    free(pi);
    printf("\n Informacao alocada: %d\n\n", *pi);
    system ("pause");
    return 0;
}
```

Após a função *free()* executar, mandamos imprimir na tela a informação armazenada no endereço do ponteiro **pi*. Observe a seguir a tela de execução do programa. A informação contida no endereço apostado pelo ponteiro não é mais o valor informado na entrada de dados.

Figura 15 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Função *realloc()*

Durante a execução do programa, pode-se necessitar alterar o tamanho do blocos de *bytes* alocados por *malloc()*. Isso poderá ocorrer, principalmente, por mudanças na arquitetura computacional. Para esses casos, precisaremos utilizar a função *realloc()* para modificar o tamanho da memória reservada anteriormente.

Veja o protótipo da função *realloc()*:

```
void *realloc (void *ptr, unsigned int num);
           ou
void *realloc (void *ptr, sizeof(int));
```

Quando não houver espaço sequencial ao bloco de memória já alocado, para aumentar, um novo bloco de memória será alocado, e todo o conteúdo armazenado no bloco anterior será copiado para o novo bloco.

Exemplo: para demonstrar o uso da função *realloc()*, faremos um pequeno programa que armazenará inteiros em um vetor de 10 elementos e o aumentaremos para 15 elementos:

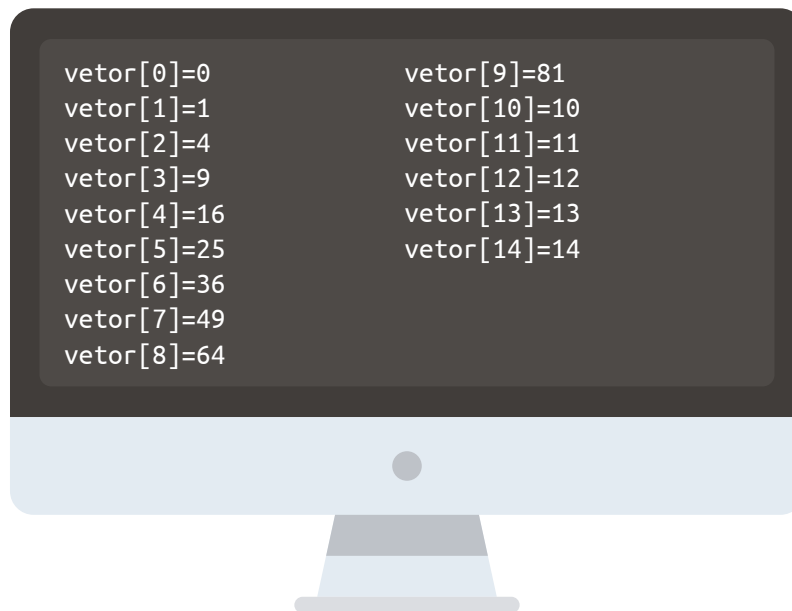
```
#include <stdlib.h>
#include<stdio.h>

int main (){

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

```
for (int n = 0; n < 10; n++)  
    vetor[n]=n*n;  
  
vetor = realloc (vetor, 15 * sizeof (int));  
  
for (int n = 10; n < 15; n++)  
    vetor[n]=n;  
  
printf (“\n”);  
  
for (int n = 0; n < 15; n++)  
    printf (“\n vetor[%i]=%i”,n,vetor[n]);  
}
```

Figura 16 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

No programa exemplo, a função *realloc()* modificou o tamanho do vetor para 15 elementos e novos elementos foram adicionados a esses cinco novos. Tentar armazenar dados em regiões não alocadas causaria erro de execução no programa.



Fique atento!

Tentar armazenar dados incompatíveis em tamanho dentro de variáveis causará erro em tempo de execução. O uso da função *realloc()* adaptará a variável, evitando esse tipo de erro.

Para melhor exemplificar o uso das funções de alocação dinâmica, serão apresentados problemas a seguir exemplificando o uso das funções: *sizeof()*, *malloc()*, *free()* e *realloc()*.

A aplicação a seguir demonstrará a conversão em *bytes* de um inteiro usando a função *sizeof*.

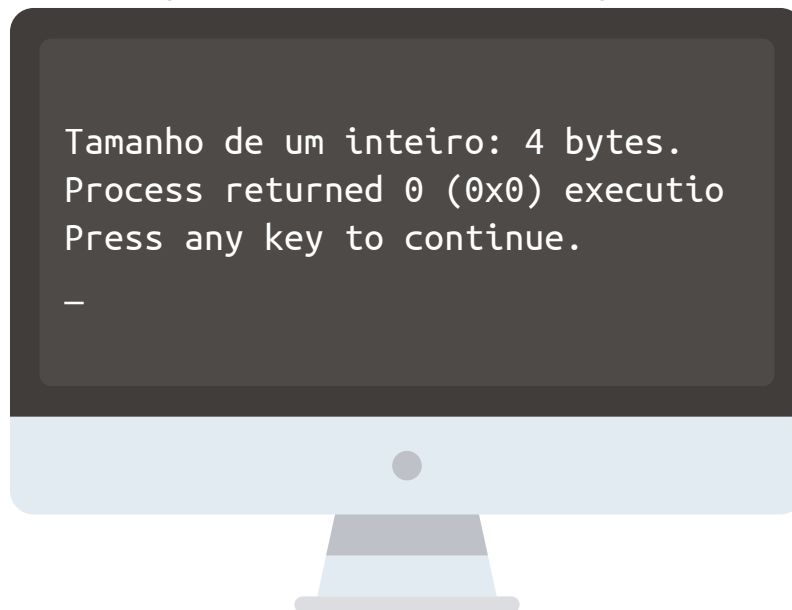
Problema 1 - Escreva um programa que converta em valor inteiro o tamanho de uma variável e imprima esse valor em *bytes* na tela com computador:

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

int main(void){

    int pt;
    pt = sizeof(int);
    printf("\n Tamaho de um inteiro:%i bytes.\n",pt);
    return 0;
}
```

Figura 17 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Fazendo uso da função *malloc()*, passaremos o tamanho em *bytes* da região a ser alocada usando a função *sizeof()*.

Agora será usada uma função para passar o valor em *bytes* para as funções de alocação dinâmica.

Problema 2 - Escreva um programa que aloque dinamicamente na memória o tamanho exato para armazenar um valor inteiro inserido por meio de uma entrada de dados. Imprima o valor armazenado e o tamanho em *bytes* da região alocada para alocar o valor, usando a função *malloc()*:

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

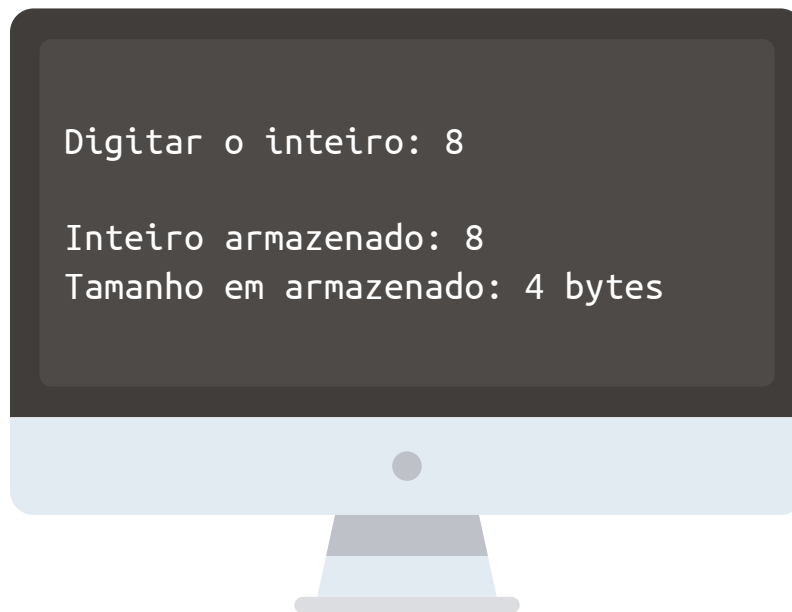
int main (){
```

```

int *p;
p = (int *) malloc( sizeof (int));
printf (“\n Digitar o inteiro:”);
scanf(“%i”,&*p);
printf(“\n Inteiro armazenado: %i\n”,*p);
printf(“ Tamanho da região: %li bytes\n\n”,sizeof(*p));
return 0;
}

```

Figura 18 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Usando o exemplo anterior, faremos agora a junção entre as funções *malloc()* e *free()*. Aplicando *malloc()*, reservaremos uma região de memória para armazenar uma informação e, a seguir, usaremos *free()* para liberar a região.

Problema 3 - Crie um programa que faça alocação dinâmica e um tipo inteiro e armazene uma informação nessa região. Usando a função *free()*, libere o espaço da memória. Imprima os resultados após a entrada de dados e após o uso da função *free()*:

```

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

int main (){

    int *p;
    p = (int *) malloc( sizeof (int));
    printf (“\n Digitar o inteiro:”);
    scanf(“%i”,&*p);
    printf(“\n Inteiro armazenado: %i\n”,*p);

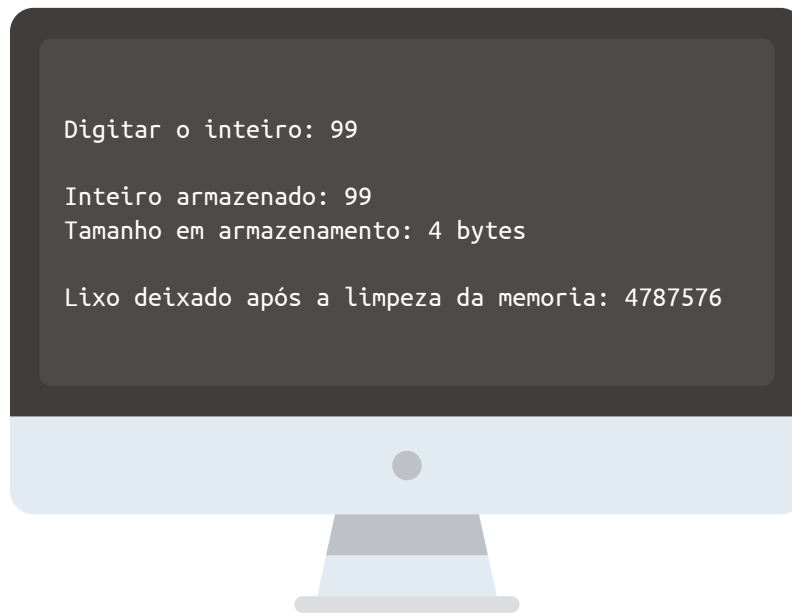
```

```

printf(" Tamanho da região: %li bytes\n\n",sizeof(*p));
free(p);
printf("\n Lixo deixado apos a limpeza da memoria: %i\n",*p);
return 0;
}

```

Figura 19 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Problema 4 - Aloque um vetor para 10 inteiros de forma dinâmica e inicialize os vetores com algum valor e imprima na tela:

```

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

int main(){

    int *v,n =10;
    v = (int *) malloc( sizeof (int)*n);
    int i;

    for (i=0;i<n;i++)
        v[i] = i*i;

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

```

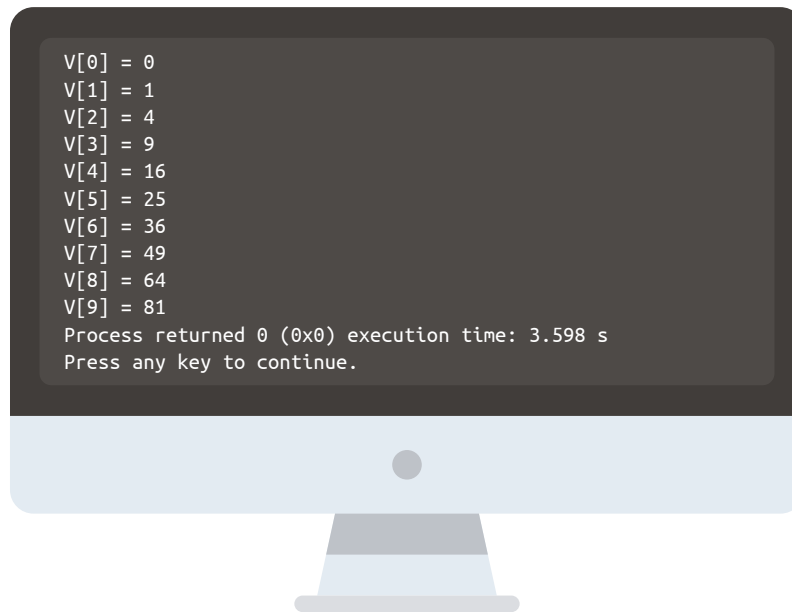


```

free (v);
return 0;
}

```

Figura 20 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

O problema a seguir aloca cinco regiões na memória para armazenar inteiros. Observe que o vetor é criado de forma dinâmica quando a função *malloc()* multiplica a região a ser alocada por *n*. Usaremos a função *free()* para devolver ao sistema operacional o controle da região alocada.

Problema 5 - Aloque um vetor para cinco inteiros de forma dinâmica e imprima na tela os valores contidos nos blocos alocados pela função *malloc()*:

```

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

int main(){

    int *v,n =5;
    v = (int *) malloc( sizeof (int)*n);

    for (int i=0;i<n;i++)
        v[i] = i*i;

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

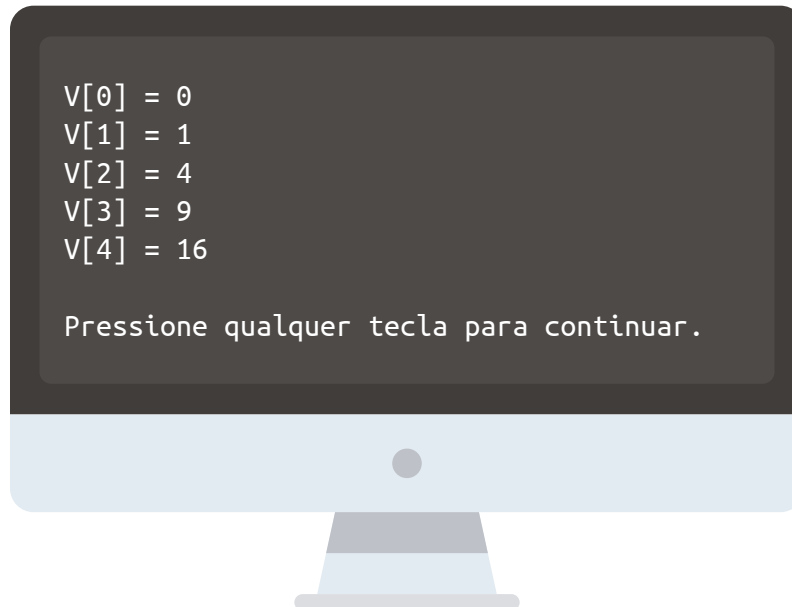
```

```

printf ("\n");
system("pause");
free (v);
return 0;
}

```

Figura 21 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Abordaremos no problema 6 o uso da função *realloc()*. Sua função é realocar regiões já alocadas pela função *malloc()*, aumentando ou diminuindo o espaço utilizado pela função.

Problema 6 - Aloque dinamicamente uma variável de um tamanho e a realoque com tamanho maior. Ex.: aloque um vetor de cinco elementos e realoque o mesmo com sete elementos.

```

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

int main(){

    int *v,i;
    v = (int*) malloc (5 * sizeof (int));

    for (inti = 0; i< 5; i++)
        v[i] = i+1;

    printf ("-----Malloc-----\n");

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

    system("pause");
}

```

```

printf ("-----Realloc-----\n");
v = (int*)realloc (v, 7 * sizeof (int));

for (int i = 5; i< 7; i++)
v[i] = i+1;

for (int i = 0; i< 7; i++)
printf ("v[%d] = %d\n",i, v[i]);

free (v);
return 0;
}

```

Figura 22 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Para o problema 7, voltaremos à função *malloc()* para demonstrar o tratamento de erro na tentativa de alocar memória excessivamente grande.

Problema 7 - Aloque dinamicamente um bloco de memória com o objetivo de gerar um retorno *NULL* do ponteiro por falta de espaço:

```

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

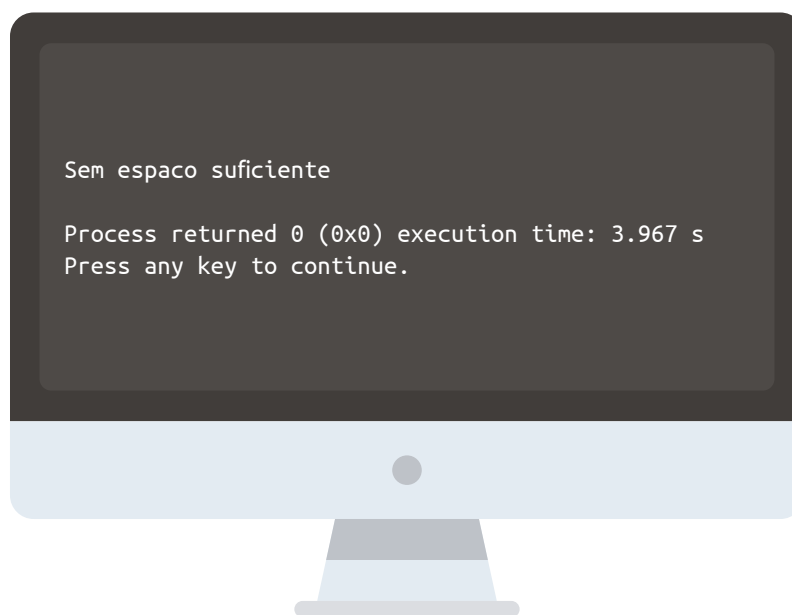
int main () {

    double *v;
    inti;
    v = (double*) malloc (5000000000000000 * sizeof (double));

```

```
if(v == NULL){  
  
    printf("Sem espaço suficiente\n");  
}  
else{  
    for (inti = 0; i< 5000000000000000; i++){  
  
        v[i] = i+1;  
        printf ("v[%f] = %f\n",i, v[i]);  
    };  
}  
  
free (v);  
return 0;  
}
```

Figura 23 – Resultado da execução do programa



Fonte: Elaborada pelo autor.

Síntese da unidade

Nesta unidade, conhecemos como alocar blocos de memória em regiões livres e o gerenciamento dessas alocações por meio das funções *malloc()*, *free()* e *realloc()*. A partir dessas informações, você poderá controlar melhor o uso dos recursos computacionais e das aplicações alocando ou desalocando blocos de memória. Ainda saberá quando algum bloco não pode ser alocado ao receber o *NULL* como retorno do ponteiro que recebe o endereço alocado.



Considerações finais

Aprendemos aqui que o uso racionalizado da memória tornará nosso programa adaptável às arquiteturas computacionais existentes. Vimos como alocar memória sem precisar passar um valor exato para a função, descobrimos que é possível liberar a região de memória já utilizada para dar mais espaço a programas e também que é possível modificar o tamanho de uma variável para que o programa se adapte quando necessário. Agora você será capaz de aplicar esses conceitos ao desenvolvimento de novas aplicações para aumentar o desempenho e reduzir erros de memória.



4

Unidade 4

4. Registros ou Estruturas I



Para iniciar seus estudos

Olá! Nesta unidade, você compreenderá uma nova forma de se manipular dados na programação, as estruturas de dados heterogêneas. O foco aqui é que você entenda como são declarados e manipulados os registros, conhecidos como estruturas em Linguagem C, entendendo o conceito e analisando aplicações práticas. E aí, vamos estudar?



Objetivos de Aprendizagem

- Definir uma estrutura de dados heterogênea.
- Entender onde se aplicam as estruturas heterogêneas.
- Declarar *structs* (registros) em Linguagem C.
- Utilizar *structs* (registros) em Linguagem C.

Introdução da unidade

Os vetores e matrizes, estruturas de dados vistas no início desta disciplina, são muito interessantes e têm algumas aplicações bem úteis. Mas elas têm uma característica que as limitam para aplicações mais robustas, são estruturas de dados homogêneas. Por isso, só trabalham um tipo de dados.

Uma estrutura de dados tem várias características a serem analisadas, e uma delas é quanto aos tipos de dados que ela consegue armazenar. Nesse caso, a estrutura pode ser homogênea ou heterogênea, sendo a segunda o tipo mais aplicável, por permitir agrupar dados de tipos diferentes que pertencem a um mesmo contexto.

Na Linguagem C, a declaração e uso de estruturas de dados heterogêneas é feita com o uso de registros ou estruturas (*structs*). Nesta unidade, você entenderá o conceito, assim como as formas de utilização para *structs*, com sintaxe, explicações e códigos para melhor exemplificação.

4.1 Estruturas de dados heterogêneas

Uma estrutura de dados heterogênea é aquela que consegue armazenar mais de um tipo. Isso possibilita uma modelagem mais real dos problemas, pois, na maioria das vezes, os dados que fazem parte do mesmo contexto não são do mesmo tipo. Como exemplo, suponha um programa que trabalhe as informações principais de uma pessoa. Geralmente, esse problema tratará o nome, o endereço e o telefone, que são *string*, a idade e o CPF, que são inteiros, a altura e o salário, que são reais. Na Figura 24, é mostrado um formulário de inscrição muito comum em várias situações que precisam ser tratadas pelos programas de computador:

Figura 24 – Formulário de inscrição



Fonte: SHUTTERSTOCK, 2018.

Já no Quadro 4 é mostrado um esquema de quais dados deveriam constar em uma estrutura para armazenar esses dados:

Quadro 4 – Dados em estrutura heterogênea

Campo	Tipo
Nome	<i>String</i>
Endereço	<i>String</i>
Telefone	<i>String</i>
Idade	<i>int</i>
CPF	<i>String</i>
Altura	<i>float</i>
Salário	<i>float</i>

Fonte: Elaborado pelo Autor.

Dados heterogêneos, como os de um formulário, precisam de estruturas de dados que agrupem seus campos. Em Linguagem C, essas estruturas heterogêneas são tratadas como *structs*.

4.1.1 Registros ou estruturas (*structs*)

As linguagens de programação disponibilizam diversas formas de se trabalhar com os dados, desde as variáveis primitivas, que tratam os tipos mais comuns, como inteiro, real e *char*, até os *arrays* (matrizes), que agrupam várias dessas informações em um único identificador e permitem que esses valores sejam cadastrados e recuperados posteriormente durante a execução do programa.

O problema de todos os tipos que foram citados aqui é que eles têm limitações quanto ao tipo de dados. No caso das variáveis simples, elas armazenam um único valor cada uma, e isso dificulta o agrupamento das informações quando necessário. Já nos casos dos *arrays* comuns, o agrupamento das informações é feito, mas só é possível se trabalhar com valores do mesmo tipo.

Os registros ou estruturas (são conhecidos das duas formas) são estruturas que permitem a criação de vários campos dentro de uma única variável. Esses campos podem ser de vários tipos, desde os primitivos até os tipos construídos por estruturas. Para Forbelone:

Um registro é composto por campos que são partes que especificam cada uma das informações que o compõem. Uma variável do tipo registro é uma variável composta, pois engloba um conjunto de dados, e é heterogênea, pois cada campo pode ser de um tipo primitivo diferente (FORBELONE, 2005, p. 85).



Fique atento!

Os registros são considerados variáveis compostas, pois têm mais de um campo para armazenamento de dados, assim como os vetores e matrizes. Porém, nos vetores, os dados armazenados são de vários indivíduos. Cada campo armazena a informação de um elemento, enquanto na *struct* os valores armazenados são todos referentes a um elemento. Outra diferença é o fato de que matrizes são homogêneas, enquanto *structs* são heterogêneas.

Structs possibilitam uma organização mais eficiente dos dados a serem trabalhados em uma aplicação, desde que sejam projetadas corretamente.

4.1.1.1 Declaração de uma estrutura (*struct*)

Na Linguagem C, os registros são tratados como estruturas. É utilizada a palavra reservada *struct*, que cria um novo tipo para a declaração de variáveis. As variáveis declaradas com esse tipo herdarão todos os campos da estrutura declarada. Ascencio diz que “os registros em C/C++, chamados de estruturas, são definidos por meio da utilização da palavra reservada *struct* (...). A partir da estrutura definida, o programa poderá considerar que existe um novo tipo de dado a ser utilizado (...)” (ASCENCIO, 2012, p. 338).

A sintaxe a seguir mostra como deve ser declarada uma estrutura. Após a palavra reservada *struct*, coloca-se o identificador da estrutura. É por meio dele que a estrutura será utilizada como um tipo para a declaração de variáveis. O conteúdo da estrutura está definido entre chaves {...}. Uma *struct* pode ser declarada com um ou vários campos, que podem ser do mesmo tipo ou de tipos diferentes:

```
struct identificador{

    tipo1 campo1;
    tipo2 campo2;
    ...
    tipo campoN;
};
```

Para demonstrar como é criada uma *struct* no código, a seguir é apresentado um exemplo que mostra a criação da estrutura FUNCIONARIO, que deve armazenar as informações básicas do trabalhador de uma empresa. São elas a matrícula e nome, do tipo *string*, salário, do tipo *float*, e idade, do tipo inteiro:

```
struct FUNCIONARIO{

    char matricula[11], nome[21];
    float salario;
    int idade;
};
```

A estrutura declarada anteriormente pode ser demonstrada conforme o Quadro 5, apresentado a seguir:

Quadro 5 – Representação de uma estrutura de dados heterogênea (*struct*)

FUNCIONÁRIO	
Campo	Tipo
matricula	String
nome	String
salario	float
idade	Int

Fonte: Elaborado pelo autor.

Ao se declarar uma *struct*, ela se comporta como um tipo criado pelo programador. Por isso, para que ela seja utilizada, é necessária a declaração de uma variável desse novo tipo.

4.1.1.2 Uso de uma Estrutura

Uma estrutura não é uma variável, portanto ela não pode ser utilizada diretamente. Na verdade, a estrutura se comporta como um tipo. Por isto, é chamada de tipo abstrato de dados, por ser criada pelo programador, diferente dos tipos primitivos, que são declarados pela linguagem (*int*, *float*, *char*, entre outros). Considerando a *struct* declarada a seguir, de nome REGISTRO, e com os campos número, letra e real, após declarada a estrutura, é criada a variável que a manipulará, da mesma forma que se cria uma variável de um tipo primitivo. Porém, nesse caso, não é utilizado *int*, *float* ou *char*, mas sim o identificador do registro criado:

```
struct REGISTRO{
    int numero;
    char letra;
    float real;
};
...
REGISTRO variavel;
```

O registro declarado é representado na figura 4 a seguir. Nela, são mostrados os campos *numero(int)*, *letra(char)* e *real(float)*:

Quadro 6 – Representação de uma estrutura de dados heterogênea (*struct*)

REGISTRO	
Campo	Tipo
numero	<i>int</i>
letra	<i>char</i>
real	<i>float</i>

Fonte: Elaborado pelo autor, 2018.



Saiba mais

Uma forma muito interessante de uso das estruturas ou registros é a criação de estruturas de dados dinâmicas, como listas, filas, pilhas e árvores. Quando isso acontece, ela passa a se comportar como uma parte da estrutura de dados que tem várias *structs*, e as estruturas se ligam graças a campos ponteiros.

A nova variável declarada com o tipo REGISTRO tem acesso a todos os campos declarados na *struct*. Por isso, sua forma de acesso será diferente de uma variável comum.

4.1.1.3 Inicialização dos campos de uma estrutura

Para acessar os campos de uma estrutura, é necessário que ela tenha sido utilizada para declarar uma variável. O acesso é feito por meio do identificador da variável seguido de um ponto e do nome do campo a ser acessado, conforme apresentado de forma geral no modelo proposto por Ascencio (2012), a seguir:

`identificador_da_variavel.identificador_do_campo`

Deitel (2011) afirma que, para o acesso aos dados, podem ser utilizados dois operadores, o operador de membro da estrutura (.), também chamado de operador de ponto, assim como o operador de ponteiro de estrutura (->), também chamado de operador de seta.

Nesse primeiro momento do aprendizado de *struct*, será utilizado somente o operador de ponto, pois não será empregado ponteiro com estruturas nesta unidade. Para exemplificar a forma de inicialização dos campos de uma estrutura, tomando por base a variável declarada anteriormente do tipo REGISTRO, é mostrado o código a seguir:

```
variavel.numero=5;
variavel.letra='A';
variavel.real=5.3;
```

No exemplo completo abaixo, os valores 5, 'A' e 5.3 são inseridos, respectivamente, nos campos numero, letra e real da variável:

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

struct REGISTRO{

    int  numero;
    char letra;
    float real;
};

int main(){

    struct REGISTRO variavel;

    setlocale(LC_ALL, "Portuguese");
    variavel.numero=5;
    variavel.letra='A';
    variavel.real=5.3;
    printf("\nnúmero = %i",variavel.numero);
    printf("\nletra= %c",variavel.letra);
    printf("\nreal= %f \n",variavel.real);
}
```

Os campos de uma estrutura também podem receber valor diretamente do teclado. Nesse caso, é importante observar o tipo do campo que está sendo acessado para receber o valor com o formato correto e evitar erros. A leitura pode ser feita conforme o exemplo demonstrado a seguir:

```
scanf("%i",&variavel.numero);
scanf("%c",&variavel.letra);
scanf("%f",&variavel.real);
```

A inicialização de uma *struct* também pode ser feita diretamente com o identificador da variável que a manipula, sem utilizar o nome do campo. Nesse caso, ela é toda inicializada de uma vez, com valores passados entre chaves:

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

struct REGISTRO{

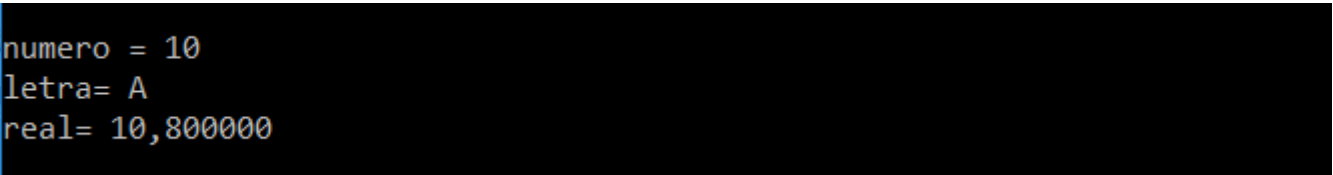
    int numero;
    char letra;
    float real;
};

int main(){

    struct REGISTRO variavel;
    variavel={10,'A',10.8};
    setlocale(LC_ALL, "Portuguese");
    printf("\numero = %i",variavel.numero);
    printf("\nletra= %c",variavel.letra);
    printf("\nreal= %f \n",variavel.real);
}
```

Na Figura 25, é mostrado o resultado da execução do código de preenchimento direto da estrutura. Os valores impressos mostram que o preenchimento foi feito corretamente:

Figura 25 – Execução do código de preenchimento direto



```
numero = 10
letra= A
real= 10,800000
```

Fonte: Elaborada pelo autor.

Uma estrutura só pode ser acessada utilizando somente o identificador da variável quando se deseja passar uma lista com todos os valores dos campos. Os valores devem estar na ordem correta de acordo com os tipos dos campos. Também devem ser colocados entre chaves, conforme o exemplo a seguir. Para a leitura de valores via teclado, é necessário fazer o acesso a cada campo para a sua manipulação:

```
variavel={5,'A',5.3};
```

Tanto para o acesso individual quanto para o carregamento de todos os campos, é necessária atenção ao tipo de cada campo declarado no registro.

4.1.1.4 Acesso aos dados cadastrados em uma *struct*

Assim como é feito para inserir valores nos campos de uma estrutura, é feito para ler os valores cadastrados em uma variável declarada como estrutura. O código a seguir exibe a declaração da estrutura REGISTRO, conforme já mostrado. Na função principal, é declarada a variável do tipo REGISTRO. Na sequência, é feita a entrada de dados diretamente nos campos da estrutura. Por fim, os dados são impressos em cada campo, por meio da função *printf()* utilizando o identificador da variável, o operador ponto e o nome do campo. O trecho de código a seguir mostra como é feita a impressão dos valores que estão na *struct*:

```
printf("\nnúmero = %i",variavel.numero);
printf("\nletra= %c",,variavel.letra);
printf("\nreal= %f \n",variavel.real);
```



Fique atento!

A formatação de tipo que é feita para a impressão no *printf()* utilizará o tipo do campo a ser impresso. No exemplo, **variavel.numero** foi formatada como inteiro ("%i"), a impressão de **variavel.letra**, formatada como *char* ("%c"), e **variavel.real**, formatada como *float* (%f).

O primeiro código completo envolvendo registros, mostrado a seguir, é bastante simples e apresenta a entrada de dados via teclado utilizando o *scanf* e, em seguida, a impressão dos valores no registro, por meio do *printf*:

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

struct REGISTRO{

    int numero;
    char letra;
    float real;
};

int main(){

    struct REGISTRO variavel;
    setlocale(LC_ALL, "Portuguese");
    printf("Digite um número inteiro:");
    scanf("%i",&variavel.numero);
    printf("Digite uma letra:");
    fflush(stdin);
    scanf("%c",&variavel.letra);
    printf("Digite um número:");
```

```
scanf("%f",&variavel.real);

//Impressão dos campos de uma estrutura
printf("\nnumero = %i",variavel.numero);
printf("\nletra= %c",variavel.letra);
printf("\nreal= %f \n",variavel.real);
}
```

Assim como ocorre na inserção de valores em um registro, é necessária atenção ao tipo dos campos, para que a formatação de impressão seja feita da forma correta.

4.1.2 Aplicação para estruturas de dados heterogêneas

Para entender a utilidade de uma *struct*, é importante imaginar situações práticas em que ela pode ser aplicada. Como primeiro exemplo, imagine uma empresa de entregas que precisa das informações sobre a forma de pagamento (*int*), nome, telefone e endereço (*string*) e valor e taxa de entrega (*float*). O programa da empresa precisa receber os dados e calcular o frete. Pela regra da empresa, pagamentos em dinheiro com valor maior ou igual a R\$ 30,00 têm frete grátis, as demais entregas terão uma taxa de R\$ 3,50.

Nas primeira parte do código, é mostrada a declaração da *struct* PEDIDOS contendo os campos formaDePagamento(*int*), telefone, nome e endereço (*string*), valor e taxaEntrega (*float*) conforme o necessário para a resolução do problema proposto.

Foi declarada a variável entrega do registro PEDIDOS. Ela que será utilizada para a manipulação da *struct*:

```
#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
struct PEDIDOS{

    int formaDePagamento;
    char telefone[15],nome[21],endereço[41];
    float valor,taxaEntrega;
};
```

```
int main(){

    struct PEDIDOS entrega;
    /*
```

As informações foram recebidas do teclado utilizando o identificador **entrega**, seguido do operador ponto (.) e do campo a ser preenchido em cada entrada de dados:

```
*/

    setlocale(LC_ALL, "Portuguese");
    printf("Digite o código da forma de pagamento:"
           "1 - Dinheiro, 2 - Cartão:\n");
    scanf("%i",&entrega.formaDePagamento);
    printf("Digite o Nome do Cliente:");
```

```

fflush(stdin);
gets(entrega.nome);
printf("Digite o Endereço do Cliente:");
fflush(stdin);
gets(entrega.endereco);
printf("Digite o telefone:");
fflush(stdin);
gets(entrega.telefone);
printf("Digite o valor do pedido:");
scanf("%f",&entrega.valor);

```

```
/*
```

A seguir, é mostrada a condicional que calcula o frete, caso o valor da compra for maior que R\$ 30 e a opção for igual a 1 – Dinheiro, o campo taxaEntrega da variável entrega recebe 0, caso contrário, ele recebe 3.5. Feito isso, são impressos todos os valores da estrutura, exceto o campo com o código de pagamento:

```
*/
```

```

if(entrega.valor >= 30 && entrega.formaDePagamento == 1){

    entrega.taxaEntrega=0;
}
else
entrega.taxaEntrega=3.5;

printf("\nCliente: %s", entrega.nome);
printf("\nEndereço: %s", entrega.endereco);
printf("\nTelefone: %s\n", entrega.telefone);
printf("\nValor do pedido: %f", entrega.valor);
printf("\nTaxa de Entrega: %f", entrega.taxaEntrega);
}

```

Como outro exemplo de aplicação para as estruturas em C, imagine agora que uma escola deseja cadastrar as informações de um aluno. Considere que as informações importantes para cada aluno são o identificador (inteiro), o nome e a disciplina (*string*), cinco notas e a média final (real), conforme mostrado no formulário utilizado para controle manual, na Figura 26, a seguir:

Figura 26 – Ficha do aluno

Identificador: _____						
Nome: _____						
Disciplina: _____						
	Avaliação 1 (0 - 10)	Avaliação 2 (0 - 10)	Avaliação 3 (0 - 10)	Avaliação 4 (0 - 10)	Avaliação 5 (0 - 10)	Média Final
Notas						
Situação do Aluno:				CRITÉRIOS PARA APROVAÇÃO Aprovado: Média final >=6 Reprovado: Média final <6		

Fonte: Elaborada pelo autor.

Ao final, o programa deve mostrar se o aluno foi aprovado, caso a média final for maior ou igual a 6 (nesse caso deve ser impresso o nome do aluno, da disciplina, a média), caso o aluno tenha sido reprovado, a mensagem deve mostrar seu nome, o nome da disciplina e a média.

No Quadro 7 mostra como ficará a *struct* declarada para a resolução do problema proposto no exemplo. Nela, são vistos todos os campos e seus respectivos tipos:

Quadro 7 – Representação de uma estrutura de dados heterogênea (*struct*)

ALUNO	
Campo	Tipo
Identificador	<i>int</i>
Nome	<i>char[31]</i>
Disciplina	<i>char[31]</i>
nota1	<i>float</i>
nota2	<i>float</i>
nota3	<i>float</i>
nota4	<i>float</i>
nota5	<i>float</i>
mediaFinal	<i>float</i>

Fonte: Elaborada pelo autor.

O código proposto para a resolução do problema está a seguir. Ele foi dividido em três partes, com explicação de cada uma delas para melhor compreensão de seu funcionamento.

A primeira parte do código mostra a declaração da *struct* ALUNO. Nela, foram declarados os campos identificador (*int*), nome e disciplina (*string*), mediaFinal, nota1, nota2, nota3, nota4, nota5 (*float*). Essas são as informações da ficha do aluno mostrada anteriormente:

```
#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
struct ALUNO{

    int identificador;
    char nome[31], disciplina[31];
    float mediaFinal,nota1,nota2,nota3,nota4,nota5;
};

int main(){

    struct ALUNO notasAluno;
    /*
```

A segunda parte mostra a entrada de dados sendo feita para os campos da *struct*. Observe que em todas as leituras feitas é utilizado o nome da variável notasAluno seguido do operador ponto (.) e do campo a ser preenchido:


```

*/
setlocale(LC_ALL, "Portuguese");
printf("Digite o identificador do Aluno:");
scanf("%i",&notasAluno.identificador);
fflush(stdin);
printf("Digite o nome do Aluno:");
gets(notasAluno.nome);
fflush(stdin);
printf("Digite o nome da Disciplina:");
gets(notasAluno.disciplina);
printf("Digite a 1ª nota:");
scanf("%f",&notasAluno.nota1);
printf("Digite a 2ª nota:");
scanf("%f",&notasAluno.nota2);
printf("Digite a 3ª nota:");
scanf("%f",&notasAluno.nota3);
printf("Digite a 4ª nota:");
scanf("%f",&notasAluno.nota4);
printf("Digite a 5ª nota:");
scanf("%f",&notasAluno.nota5);
/*

```

Na terceira parte do código, foi feito o cálculo da `mediaFinal` do aluno com a soma de todas as notas presentes no vetor e dividido o resultado por 5. Com a `mediaFinal` calculada, foi feita a condicional para informar se o aluno foi aprovado ou reprovado. Caso o campo `mediaFinal` da variável `notasAluno` for maior ou igual a 6, é impressa uma mensagem informando que ele está aprovado na disciplina com a média que conseguiu. No caso contrário (else), a mensagem informa que ele está reprovado na disciplina e imprime a média calculada:

```

*/
notasAluno.mediaFinal=(float)(notasAluno.nota1+
notasAluno.nota2+notasAluno.nota3+notasAluno.nota4+
notasAluno.nota5)/5;

if(notasAluno.mediaFinal>=6){

    printf("\nO aluno %s foi aprovado na disciplina de %s com a"
    " média %.2f",notasAluno.nome, notasAluno.disciplina,
    notasAluno.mediaFinal);
}
else
printf("\nO aluno %s foi reprovado na disciplina de %s com a"
" média %.2f", notasAluno.nome, notasAluno.disciplina, notasAluno.mediaFinal);
}

```



Refleta

Os 5 campos para nota (nota1, nota2, nota3, nota4 e nota5) utilizados nessa *struct* poderiam ter sido declaradas como um vetor de *float* com cinco posições?

A aplicação apresentada mostra um exemplo para o uso de estruturas heterogêneas, que, devido à maior flexibilidade na aplicação em relação às homogêneas, podem ser utilizadas na resolução de diversos outros problemas.

Síntese da unidade

Nesta unidade, você viu o conceito de estrutura de dados heterogêneas, que tem a capacidade de armazenar valores de tipos diferentes, o que é especialmente útil para a organização e agrupamento de informações dentro de um programa.

Viu também o conceito de registros ou estruturas, que são estruturas da programação que permitem a criação de variáveis compostas que armazenam valores de diversos tipos, sendo capazes de demonstrar de forma melhor as informações do mundo real dentro da aplicação que está sendo desenvolvida.

Na Linguagem C, um registro é chamado de estrutura e declarado com o uso da palavra reservada *struct*. Essa informação foi vista e detalhada no material, assim como o fato de que uma *struct* não é uma variável, e sim um tipo abstrato de dados que possibilita a criação de variáveis a partir do seu identificador. As variáveis criadas assumem os campos declarados na estrutura, o que as torna uma variável composta, por armazenar mais de um valor.

Foram explicadas as formas de inserção de valores nos campos de uma estrutura, assim como sua manipulação e retirada de dados, sempre com o uso do operador ponto (.) e com o identificador da variável criada para isso.

Por fim, observou uma aplicação prática para as estruturas, com o controle de notas de um aluno, em que foi criado cadastro para um aluno utilizando estrutura definida pelo programador.



Considerações finais

Você chegou ao final da unidade 4 desta disciplina. Até aqui, você já conheceu os vetores e matrizes, ponteiros e a alocação dinâmica de memória, todos com conceituação e demonstração de aplicações práticas. Nesta unidade, você conheceu o conceito de registros ou estruturas e suas aplicações práticas na Linguagem C. Para o entendimento completo de cada unidade, é fundamental que você acesse todos os objetos disponíveis para ela. Não deixe de assistir à videoaula e utilizar os materiais de resumo e aprofundamento de conteúdo, bem como resolver as questões propostas e o desafio, com o máximo de atenção. Agora siga em frente!



5

Unidade 5

5. Registros ou Estruturas II



Para iniciar seus estudos

Nesta unidade, você conseguirá compreender o uso avançado de *structs* em C, com ponteiros, matrizes e passagem de estruturas para funções. A unidade aborda aspectos importantes da implementação de registros em C para que seu uso se torne mais prático e funcional. Vamos estudar?



Objetivos de Aprendizagem

Ao final desta disciplina, esperamos que você:

- Declare e utilize um ponteiro para *struct*.
- Declare e utilize *structs* com campos compostos (matrizes).
- Declare e utilize uma matriz de *structs*.

Introdução da unidade

Na última unidade, foram apresentadas as estruturas de dados heterogêneas *structs* e a forma como ela deve ser declarada, inicializada e utilizada. Foram apresentados exemplos de estruturas básicas sem a utilização de campos compostos. Nesta unidade, você conhecerá as formas mais sofisticadas de se utilizar as *structs*, e será apresentado seu uso com campos compostos (matriz dentro da *struct*). Quanto à declaração e uso, serão apresentadas outras formas de se declarar e utilizar as estruturas, primeiro com a declaração de um ponteiro e alocação dinâmica de memória para a *struct*, e, por fim, com a declaração de uma matriz do tipo de uma estrutura declarada. Toda a explicação será apresentada com conceitos, explicações e exemplos práticos para uma melhor assimilação do conhecimento.

5.1 Declaração de Structs

“Por meio da palavra-chave *struct* definimos um novo tipo de dado. Definir um novo tipo de dado significa informar ao compilador seu nome, tamanho em *bytes* e forma como deve ser armazenado e recuperado da memória” (MIZRAHI, 2008, p. 223). A declaração de *structs*, já vista na unidade anterior, será lembrada aqui, onde serão vistas também algumas novas possibilidades. A sintaxe para se declarar uma *struct* é a seguinte:

```
structidentificador
{
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipocampoN;
};
```

Algumas variações não exploradas na Unidade 4 serão apresentadas, nos exemplos de declaração de *structs*; a primeira delas é quando se deseja declarar uma *struct* que tenha matrizes ou vetores entre seus campos. Conforme mostrado a seguir, a *struct* VIAGEM tem campos para número de viajantes e gasto total, além de um vetor gastos pontuais, que armazenará 4 gastos pontuais durante a viagem. O campo vetor de uma estrutura é declarado da mesma forma que um vetor seria declarado fora do registro.

```
struct VIAGEM
{
    intnumeroDeViajantes;
    floatgastoTotal,gastosPontuais[4];
};
```

Quadro 8 – Representação da estrutura de dados heterogênea (*struct*) com vetor

VIAGEM	
Campo	Tipo
numeroDeViajantes	<i>int</i>
GastoTotal	<i>float</i>
gastosPontuais[0]	<i>float</i>
gastosPontuais [1]	<i>float</i>
gastosPontuais [2]	<i>float</i>
gastosPontuais [3]	<i>float</i>

Fonte: Elaborado pelo autor.

A variável que manipulará a estrutura é declarada da mesma forma que a declaração para as *structs* sem campos vetores, conforme apresentado a seguir.

```
structVIAGEM viagem1
```

Tanto uma variável *struct* com vetores quanto uma sem vetores são declaradas da mesma forma, uma vez que a diferença está nos campos da *struct*, e não em sua declaração.

5.1.1 Referenciando campos da Structs

Para Ascencio (2012, p. 340), “O uso de uma *struct* é possível por meio do acesso individual a seus membros, quer seja para gravar quer seja para recuperar um dado”. A referência a campos de uma *struct* também já foi discutida na unidade anterior, porém, quando se tem uma *struct* em que um dos campos é um vetor ou matriz, a forma de referenciar esse campo é diferente dos demais campos, além de utilizar o identificador da variável declarada como registro, conforme sintaxe apresentada abaixo.

```
identificador_da_variavel.identificador_do_campo
```

Ascencio (2012, p. 341) diz que, quando o campo é um vetor, ele ainda traz a característica do vetor, que é o acesso feito por índices. Nesse caso, a sintaxe pode ser apresentada da seguinte forma:

```
identificador_da_variavel.identificador_do_campo[índice_do_vetor]
```

5.1.2 Atribuição de *structs*

Para Deitel (2011, p. 321), a atribuição de valores a uma *struct* pode ser feita com a atribuição direta, já explicada na unidade anterior, acesso a cada campo, também já apresentado, e atribuição de uma variável registro a outra do mesmo tipo, que será demonstrada aqui com o preenchimento de uma *struct* com vetor e a cópia dele para outra variável do mesmo tipo.

Quando se deseja preencher um vetor sequencialmente, isso é feito com uma estrutura de repetição, e essa regra também vale para quando o vetor está dentro de uma *struct*. O código apresentado a seguir mostra como isso é feito quando o vetor é um campo de uma estrutura e depois copia o conteúdo da variável *struct* para outra variável igual e imprime os valores por meio da segunda.

```

struct VIAGEM
{
    int numeroDeViajantes;
    float gastoTotal, gastosPontuais[20];
};
...
struct VIAGEM viagem1, viagem2;
...
for(i=0; i<=4; i++){
    printf("Digite o valor do %iº gasto pontual:", i+1);
    scanf("%f", &viagem1.gastosPontuais[i]);
}
viagem2=viagem1;
...
for(i=0; i<=4; i++){
    printf("Gasto %i = %f", i, viagem2.gastosPontuais[i]);
}

```

Atribuir uma *struct* a outra *struct* é muito fácil, pois é feita da mesma forma que uma atribuição entre variáveis simples. E, dessa forma, os conteúdos de todos os campos de uma variável *struct* são copiados para a outra.

5.1.3 Passando estruturas para funções

A passagem de estrutura para uma função pode ser feita de forma semelhante à passagem de uma variável comum, e ela será tratada como um parâmetro com seu tipo; a diferença é que o tipo não será um dos tipos primitivos da linguagem, mas sim um tipo construído por uma *struct*. Para exemplificar a forma como é feita a passagem do parâmetro, o código a seguir faz o preenchimento da estrutura no programa principal e a impressão de seus valores na função. A *struct* declarada como ALUNO é utilizada como o tipo do parâmetro aluno2, e, também como tipo da variável aluno1 na função principal, a variável aluno é passada normalmente como argumento para a função no momento de sua chamada.

```

#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
struct ALUNO
{
    int identificador;
    char nome[31];
};
void imprime(struct ALUNO aluno2)
{
    printf("Identificador: %i\n", aluno2.identificador);
    printf("Nome: %s\n", aluno2.nome);
}
int main()
{
    struct ALUNO aluno1;
    printf("Digite o identificador do Aluno:");
    scanf("%i", &aluno1.identificador);
}

```

```

    printf("Digite o nome do Aluno:");
    fflush(stdin);
    gets(aluno1.nome);
    imprime(aluno1);
}

```

É importante entender a forma correta de se passar *structs* para funções, pois, quando se cria um programa maior, para resolver algum problema real, o uso de funções é inevitável.

5.1.4 Ponteiros para estruturas

Ponteiros para estruturas são declarados da mesma forma que ponteiros para tipos comuns, porém, no momento de se referenciar os campos da estrutura, para Deitel (2011, p. 322), isso deve ser feito com o operador seta (->), mas o acesso é feito normalmente, e o funcionamento segue as mesmas regras já discutidas para registros. Na declaração e atribuição de valores em uma *struct*, mostrada a seguir, é considerada a mesma estrutura ALUNO. Declaração com alocação de memória para o ponteiro:

Declaração com alocação de memória para o ponteiro:

```
struct ALUNO * aluno1 = (struct ALUNO*) malloc(sizeof(struct ALUNO))
```

Atribuição de valores aos campos utilizando ponteiro para estrutura:

```
aluno1->identificador=1234;
strcpy(aluno1->nome,"Maria");
```

Ponteiros são muito importantes na linguagem C, pois possibilitam ao programador manipular a memória de forma mais direta; porém devem ser utilizados com cuidado: podem ocasionar erros graves no programa por acesso indevido à memória. Tenha muito cuidado ao utilizar ponteiros em seus programas.

5.1.5 Matrizes dentro de estruturas

Quando estudamos uma aplicação para estruturas, vimos o exemplo que mostrava a forma de tratar a ficha de acompanhamento do aluno, mostrada na Figura 27, a seguir, dentro do código de programação. Na ocasião, a resolução foi feita com uma *struct* que continha 5 campos de notas. Neste exemplo, será feito um programa que receba as informações da ficha e ao final mostre se o aluno foi aprovado ou reprovado, de acordo com os mesmos critérios escritos na ficha. O programa faz também a listagem das notas quando o aluno for aprovado e utiliza um vetor para o armazenamento e manipulação das 5 notas.

Figura 27 – Ficha do aluno

Identificador: _____						
Nome: _____						
Disciplina: _____						
	Avaliação 1 (0 - 10)	Avaliação 2 (0 - 10)	Avaliação 3 (0 - 10)	Avaliação 4 (0 - 10)	Avaliação 5 (0 - 10)	Média Final
Notas						
Situação do Aluno:				CRITÉRIOS PARA APROVAÇÃO Aprovado: Média final ≥ 6 Reprovado: Média final < 6		

Fonte: Elaborado pelo autor.

Para começar a entender a resolução desse problema, você deve observar o Quadro 9, que representa a estrutura declarada para a resolução do problema proposto – nela está demonstrada a forma desmembrada do vetor de notas, embora ele pudesse estar mostrado em uma única linha como notas[5]: optou-se por mostrar, assim, para que se entenda melhor que essa parte deve ter seus campos preenchidos.

Quadro 9 – Representação de uma estrutura de dados heterogênea (*struct*)

ALUNO	
Campo	Tipo
identificador	<i>int</i>
Nome	<i>String</i>
Disciplina	<i>String</i>
notas[0]	<i>float</i>
notas[1]	<i>float</i>
notas[2]	<i>float</i>
notas[3]	<i>float</i>
notas[4]	<i>float</i>
mediaFinal	<i>float</i>

Fonte: Elaborado pelo autor.

O código a seguir foi separado em 4 partes, e entre essas partes foram inseridas explicações sobre cada trecho da lógica. A primeira parte do código mostra a declaração da estrutura Aluno, onde foram criados os campos, identificador, nome, disciplina, mediaFinal e um vetor para as notas; nessa parte, ainda é mostrado o início da função principal, onde é declarada a variável notasAluno do tipo ALUNO, que foi criada como uma *struct*.

```
#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
struct ALUNO
{
    int identificador;
    char nome[31], disciplina[31];
    float mediaFinal, notas[5];
};
int main()
{
    struct ALUNO notasAluno;
/*
```

No segundo trecho do código, são mostradas as variáveis *soma* e *i*, que serão utilizadas para o cálculo da média e para o preenchimento do vetor das notas respectivamente. O preenchimento dos campos do registro é feito utilizando o identificador *notasAluno* seguido do ponto e do campo a ser preenchido. As notas que foram criadas como um vetor de 5 posições foram preenchidas utilizando uma estrutura de repetição para preencher todas as 5 notas em sequência. Nesse caso, o vetor é impresso utilizando o identificador da variável *notasAluno*, o operador ponto, e o vetor *notas* utilizando o índice *i* da repetição que varia de 0 a 4, limites do vetor *notas*.

```
*/
    float soma=0;
    int i;
    setlocale(LC_ALL, "Portuguese");
    printf("Digite o identificador do Aluno:");
    scanf("%i",&notasAluno.identificador);
    fflush(stdin);
    printf("Digite o nome do Aluno:");
    gets(notasAluno.nome);
    fflush(stdin);
    printf("Digite o nome da Disciplina:");
    gets(notasAluno.disciplina);
    //Preenchimento do vetor notas da struct
    for(i=0;i<=4;i++)
    {
        printf("Digite a %iª nota:",i+1);
        scanf("%f",&notasAluno.notas[i]);
    }
/*
```

A terceira parte do código mostra o acesso à estrutura para cálculo da média, e nela o vetor *notas* da *struct* é percorrido, e o valor de cada posição dele é acumulado pela variável *soma*, que, na sequência, é utilizada para o cálculo da média final – armazenado no campo correspondente da estrutura.

```

*/
//Percorre a vetor notas da struct para calcular a média
for(i=0;i<=4;i++)
{
    soma+=notasAluno.notas[i];
}
notasAluno.mediaFinal=(soma/5;
/*

```

Na quarta e última parte desse código, é dado o resultado do aluno de acordo com a média final; caso o aluno tenha média maior ou igual a 6, ele é considerado aprovado. Para mostrar o aluno como aprovado, é feita a impressão de seu nome, da disciplina e da média, além das 5 notas cadastradas no vetor. Caso o aluno tenha média menor que 6, será impressa uma mensagem com o nome da disciplina e a média final, informando que o aluno foi reprovado.

```

*/
    if(notasAluno.mediaFinal>=6)
    {
printf("\nO aluno %s foi aprovado na disciplina de" "%scom a média %.2f",notasAluno.nome,
notasAluno.disciplina, notasAluno.mediaFinal);
        printf("\n\nDetalhamento das notas do aluno:");
        printf("\n\n_____ \n");
        //Impressão do vetor de notas
        for(i=0;i<=4;i++)
        {
            printf("Nota da %iª avaliação: %.2f\n",
                    i+1,notasAluno.notas[i]);
        }
        printf("_____ \n");
    }else
        printf("\no aluno %s foi reprovado na disciplina de %scom amédia %.2f", notasA-
luno.nome, notasAluno.disciplina,      notasAluno.mediaFinal);
}

```



Fique atento!

Para percorrer um vetor dentro da *struct*, o programa deve acessar campo por campo do vetor, como ele está associado à variável (*notasAluno*) declarada como *ALUNO* (tipo construído por meio da *struct*). Dentro da repetição está o código *notasAluno.notas[i]*, onde *i* é a variável de controle da estrutura de repetição que está configurada para ir de 0 a 4, os limites do vetor.

O código apresentado como aplicação para *structs*, no exemplo anterior, resolve a situação proposta, mas, caso o usuário erre alguma nota, ele precisa voltar e refazer a digitação de todos os dados. Para resolver o problema,

levantado o código, pode ser melhorado, permitindo o acesso a cada nota de forma individual. O código mostrado a seguir traz essa melhoria, onde o cadastro de notas é feito diretamente na posição que é escolhida pelo usuário no momento da digitação, e isso permite que seja feita a correção, caso alguma nota seja digitada de forma errada. Nesse caso, foi apresentado o código somente da parte onde ocorreu a mudança. A seguir, está a explicação das mudanças feitas.

As notas são inseridas dentro de um *flag* (estrutura de repetição sem variável de controle), isso para que possam ser cadastradas todas as notas, mesmo que não seja feito de forma sequencial. Para o cadastro, são dadas duas opções, digitar ou editar notas e fechar relatório. Caso seja escolhida a opção 1, será solicitado ao usuário que digite a avaliação entre 1 e 5 para a inserção da nota, e, depois (caso a nota esteja entre 1 e 5), será solicitada a digitação da nota – armazenada no vetor de notas do registro. Caso a opção 2 seja escolhida, o *flag* será encerrado, será feito o cálculo da média final, e a situação dos alunos será mostrada.

```
op=1;
while(op!=2)
{
    printf("\n\n 0 que deseja fazer:\n"
        " 1 - Digitar ou editar nota de avaliação:"
        "\n 2 - Fechar Resultado:\n");
    scanf("%i",&op);
    if(op==1)
    {
        printf("Digite o número da avaliação que deseja"
            " cadastrar ou editar (de 1 a 5):");
        scanf("%i",&i);
        if(i>=1 && i<=5)
        {
            printf("Digite a nota da %iª " "avaliação:",i);
            scanf("%f",&notasAluno.notas[i-1]);
        } else
        {
            printf("Você deve digitar o número da "
                "avaliação entre 1 e 5!\n");
        }
    }
}
```



Refleta

Os registros ou estruturas são muito interessantes para a organização e manipulação de dados em um programa, mas os registros sozinhos conseguem tratar apenas a informação de um indivíduo. Como poderiam ser agrupados os dados de mais de um indivíduo utilizando uma *struct*?

Outra forma de utilizar *structs* e matrizes é criando os vetores e matrizes de *structs*. Nesse caso, diferentemente de uma matriz de tipos primitivos, estudada na Unidade 1 da disciplina como uma estrutura de dados homogênea, quando uma matriz é declarada para manipular uma estrutura heterogênea criada pela *struct*, ela se torna uma estrutura de dados heterogênea com as outras características de uma matriz. Uma matriz 3 x 5 de *structs* é demonstrada na Figura 3 a seguir; nela é possível ver os dados de vários tipos armazenados em cada célula da matriz.

Quadro 10 – Matriz bidimensional de *structs*

String placa	String placa	String placa	String placa	String placa
Real valorHora	Real valorHora	Real valorHora	Real valorHora	Real valorHora
Time entrada	Time entrada	Time entrada	Time entrada	Time entrada
String placa	String placa	String placa	String placa	String placa
Real valorHora	Real valorHora	Real valorHora	Real valorHora	Real valorHora
Time entrada	Time entrada	Time entrada	Time entrada	Time entrada
String placa	String placa	String placa	String placa	String placa
Real valorHora	Real valorHora	Real valorHora	Real valorHora	Real valorHora
Time entrada	Time entrada	Time entrada	Time entrada	Time entrada

Fonte: elaborado pelo autor.

Para entender melhor como funciona uma matriz de estruturas, imagine o estacionamento mostrado na Figura 28.

Figura 28 – Estacionamento



Fonte: SHUTTERSTOCK, 2018.

Nele, cada sequência de vagas é tratada como setor, cada setor é uma linha da matriz, e cada coluna dentro de uma linha é uma vaga de estacionamento. Para esse estacionamento, foram considerados 8 setores com 10 vagas cada; por isso, foi declarada uma matriz 8 x 10. O programa a seguir funciona recebendo o valor da hora; em seguida, ele começa o controle dos carros estacionados nas vagas; quando o usuário escolhe estacionar o carro, ele digita o número do setor e o número da vaga; com essa informação o programa confere se a vaga está vazia – caso esteja, ele solicita que o usuário digite a placa do veículo, e o cadastra na vaga com horário de entrada cadastrado automaticamente e preço da hora carregada diretamente da variável digitada anteriormente. Quando ele resolve retirar o veículo, é feita uma pesquisa pela placa do veículo, e, quando encontrada, é feito o cálculo do valor a pagar, bem como a retirada da vaga. Para melhor entendimento do código, ele está separado em várias partes com explicações entre elas.



Saiba mais

Para o controle do tempo, foi inserida a biblioteca `time.h`. Para Ascencio (2012, p. 70), a função `time(NULL)` retorna a hora do sistema. Segundo a documentação da biblioteca, a função `time()` utilizada no código, quando chamada com o parâmetro `NULL` (`time (NULL)`) retorna o tempo em segundos desde 00:00 horas, 1º de janeiro de 1970.

A primeira parte do código, mostrada a seguir, apresenta a declaração da estrutura `VAGA`, que é montada para receber as informações necessárias para o controle de estacionamento (placa e horário de entrada). No início da função principal, é declarada a matriz 8 x 10 para representar as vagas do estacionamento.

```
#include<stdlib.h>
#include<stdio.h>
#include<locale.h>
#include<time.h>
#include<string.h>
struct VAGA
{
    char placa[9];
    time_t entrada;
};
int main()
{
    structVAGA todasVagas[8][10];
    /*
```

As demais variáveis declaradas na função principal são o setor (`s`) e a vaga (`v`), o tempo (variável que receberá o cálculo do tempo em que a vaga ficou ocupada), `op` (receberá a opção, estacionar, retirar o carro ou sair), `i` e `j` (índices para percorrer a matriz), `pi` e `pj` (variáveis que receberão a posição onde o valor foi encontrado na busca pelo placa do carro), `valorPagar` (receberá o valor a pagar), `valorH` (receberá o valor a ser cobrado por hora), e a variável `busca` (uma *string* de 9 caracteres que receberá a placa a ser buscada no momento da retirada do carro).

A primeira coisa que é feita nesse código é a entrada do valor da hora, e esse valor é armazenado na variável `valorH`, e será utilizada para cadastrar o campo `valorHora` na estrutura.

```
*/
    setlocale(LC_ALL, "Portuguese");
    int s, v, tempo=0;
    int op=3, i, j, pi, pj;
    float valorPagar, valorH;
    char busca[9];
    for(i=0; i<8; i++)
        for(j=0; j<10; j++)
            strcpy(todasVagas[i][j].placa, "-");
    printf("Digite o valor da hora:");
    scanf("%f", &valorH);
/*
```

A manipulação das vagas do estacionamento começa nesse ponto, tudo acontece dentro de um *flag* (`while(op!=0)`), a tela é sempre limpa (`system("cls")`) entre uma interação e outra; em seguida, é fornecido um menu ao usuário, nele são fornecidas as opções 1–Estacionar na Vaga, 2–Retirar da Vaga ou 0–Sair do Sistema. O valor digitado será armazenado na variável `op`.

```
*/
    while(op!=0)
    {
        system("cls");
        printf("----Menu de Funções:----\n"
            "1 - Estacionar na Vaga:\n"
            "2 - Retira da Vaga:\n"
            "0 - Sair do Sistema:");
        scanf("%i", &op);
    }
/*
```

A primeira opção tratada é a opção 1 – Estacionar na Vaga, nesse caso o usuário digitará o setor (de 0 a 7) e a vaga (de 0 a 9) onde deseja estacionar; o programa testará se a vaga digitada existe, e se a vaga está livre (placa vazia "-"). Caso a vaga seja válida e esteja vazia, será solicitada ao usuário a digitação da placa, e feito o cadastro na *struct* com o tempo no horário de entrada.

Quando a vaga está ocupada, o programa mostra a mensagem informando. E, caso a vaga seja inválida, ele mostra uma mensagem informando também.

```
*/
        if(op==1)
        {
            printf("Digite o setor (de 0 a 7):\n");
            scanf("%i", &s);
            printf("Digite o número da vaga (de 0 a 9):\n");
            scanf("%i", &v);
            if((s>=0 && s<=7)&&(v>=0 && v<=9)&&(strcmp(todasVagas[s][v].
                placa, "-")==0))
            {
                printf("\nDigite a placa do veículo:");
                fflush(stdin);
            }
        }
    }
/*
```

```

gets(todasVagas[s][v].placa);

todasVagas[s][v].entrada=time(NULL);
printf("Placa - %s - na vaga %i - %i\n",
      todasVagas[s][v].placa,s,v );
    }else
    if(strcmp(todasVagas[s][v].placa,"-")!=0)
        printf("A vaga está ocupada!\n");
    else
printf("A vaga não existe, confira e digite novamente.\n");
/*

```

Caso o usuário tenha digitado a opção 2-Retirar da vaga, o processo que será feito é o de retirada, o qual começa com a inserção da placa a ser buscada pelo usuário, seguido pela realização de uma pesquisa sequencial em busca da placa na matriz de *structs*. A pesquisa é feita comparando o campo placa de cada posição da matriz com a placa digitada para busca. Caso encontre a placa, as posições *i* e *j* são armazenadas em *pi* e *pj* respectivamente, e a repetição é encerrada.

```

*/
    }elseif(op==2)
    {
        printf("Digite a placa do carro que está saindo:");
        scanf("%s",usca);
        pi=-1;
        for(i=0;i<=7;i++){
            for(j=0;j<=9;j++){
                {
                    if(strcmp(todasVagas[i][j].placa,busca)==0)
                    {
                        pi=i;
                        pj=j;
                        break;
                    }
                }
            }
        }
    }
/*

```

O segundo passo é informar que a placa não foi encontrada, caso ela não esteja cadastrada; para isso o *pi* terá o valor -1, pois recebeu isso antes do começo da busca. Como ele só é alterado novamente quando encontra o valor buscado, terá esse valor, caso não encontre.

Caso a placa seja encontrada, será feito o cálculo do tempo em que o veículo ficou na vaga e do valor a ser pago; esses valores serão impressões, e a vaga receberá a placa vazia ("-"), indicando que a vaga está livre novamente.

```

*/
        if(pi==-1)
        {
printf("A placa não consta no cadastro!");
        }
    else{

```



```

tempo=time(NULL)-todasVagas[pi][pj].entrada;
valorPagar=(float)tempo/3600*valorH;
printf("Tempo (minutos): %.2f\n"
        "Valor: R$ %.2f\n",
(float)tempo/60, valorPagar);
        strcpy(todasVagas[pi][pj].placa,"-");
    }
}
    system("Pause");
}

```

A aplicação apresentada mostra o que é possível fazer com uma estrutura de dados heterogênea, além de apresentar uma forma mais poderosa de se trabalhar com matrizes, as matrizes de *structs* e um algoritmo de busca sequencial na matriz para a procura pela placa.

Síntese da unidade

Nesta unidade, você viu o conceito de estrutura de dados heterogêneos com aplicações avançadas, como o uso com ponteiros e matrizes. Para uso com ponteiros, foi demonstrado como declarar e como utilizar o operador seta (->), necessário para acesso aos campos da *struct* quando se utilizam ponteiros para a estrutura.

O uso de matrizes com as estruturas foi demonstrado de duas formas, a primeira abordou o uso de vetores dentro da *struct*, mostrando como declarar e como acessar os dados contidos nesses campos compostos. A segunda forma de uso de matrizes com registro foi o uso de uma matriz de registros, quando uma matriz é declarada como o tipo definido pela *struct* e se torna uma matriz heterogênea, juntando características de matrizes e de *structs*.

Foram apresentados dois exemplos de aplicação para as *structs* com matrizes, na primeira foi mostrado o controle de notas de um aluno, em um melhoramento de um exemplo apresentado na unidade 4, onde agora as notas foram armazenadas em um campo composto, uma matriz unidimensional (vetor). No segundo exemplo, foi declarada uma matriz de *structs* bidimensional para o controle das vagas e da cobrança em um estacionamento.



Considerações finais

Parabéns, você chegou ao final da Unidade 5 desta disciplina! Até aqui, você já conheceu os vetores e matrizes, ponteiros e a alocação dinâmica de memória, todos com conceituação e demonstração de aplicações práticas. Nas Unidades 4 e 5, o foco do estudo foram as *structs*, sendo vistos os conceitos e aplicações mais básicos na Unidade 4, enquanto o uso e aplicações mais avançadas foram apresentados nesta unidade, propiciando um entendimento das diversas formas como uma estrutura pode ser declarada e utilizada em C. Para o entendimento completo de cada unidade, é fundamental que você acesse todos os objetos disponíveis para ela, não deixe de assistir à videoaula, e utilizar os materiais de resumo e aprofundamento de conteúdo, bem como resolva as questões propostas e o desafio com o máximo de atenção. Agora siga em frente!



6

Unidade 6

6. Programação Modular (Funções em C)



Para iniciar seus estudos

Seja bem-vindo à modulação de programas. A partir desta etapa, você compreenderá que um bom programa deve ser dividido em partes, para que seja melhor estruturado, e ser reutilizado por outros programas.



Objetivos de Aprendizagem

- Construir programas mais ágeis e fáceis de identificar o erro.
- Fazer manutenção no código.
- Entender, através do conhecimento de funções, as diversas maneiras de utilizar a modularização de um programa e reutilização de estruturas previamente desenvolvidas e em outros programas.
- Executar programas de dentro de outros programas.

Introdução da unidade

Aprenderemos aqui a manipular funções para que os programas desenvolvidos pelo aluno sejam melhor estruturados. Nosso grande objetivo é a modularização de um programa usando o recurso de funções e reduzir drasticamente o tempo de desenvolvimento evitando repetições tornando o programa mais eficiente e melhor ajustável as diversas condições de desenvolvimento. A expressão “dividir para conquistar” se aplica muito bem ao conceito de desenvolvimento modular de um programa. As páginas seguintes mostrarão o valor desta ferramenta.

6.1 O que é uma função?

Função pode ser definida até por seu próprio nome, a função de alguma coisa. Então, elabora-se um conjunto de instruções para realizar uma determinada função ou tarefa. Pode-se dizer que é a solução de um determinado problema através de uma sequência de passos.

Uma função é um conjunto de instruções desenhadas para cumprir uma tarefa particular e agrupadas numa unidade com um nome para referenciá-la. Funções dividem grandes tarefas de computação em tarefas menores e permitem às pessoas trabalharem sobre o que outras já fizeram, em vez de partir do nada. Uma das principais razões para escrever funções é permitir que todos os outros programadores C a utilizem em seus programas (VIVIANE, 2009, p.110).

Entretanto, mesmo a essa altura do nosso conhecimento em linguagem C, já utilizamos diversas funções previamente desenvolvidas e aplicamos em nossos programas iniciais as armazenadas em bibliotecas da linguagem C. Entre elas estão: *printf*, *scanf*, etc.

Para identificarmos a necessidade de uma função, é algo bem simples. Basta observarmos se há a necessidade de fazer a mesma sequência de comandos, sempre que precisarmos resolver o mesmo problema repetidas vezes. Então, essas sequências são fortes candidatas a se tornarem uma função.

Sintaxe de uma função:

```
tipo nome_função(tipo1 param1, tipo2 param2, ..., tipoN paramN)
{
    corpo da função
}
```

6.1.1 Características de funções

- Seu nome deve ser único para ser chamada em qualquer parte do programa.
- Pode ser chamada de dentro de outras funções.
- Deve realizar uma única tarefa bem definida.
- Não interessa como a função funciona, apenas que o resultado seja sem efeitos colaterais.
- A codificação de uma função deve ter total independência do programa, para que possa ser reutilizada.

- Poderá receber parâmetros que modifiquem seu comportamento ou forcem sua adaptação onde for chamada.
- Poderá ou não retornar um valor, como resultado do seu trabalho, a entidade que a invocar.
- O nome de uma função deve ser criado seguindo as mesmas regras de definição dos nomes de variáveis: deve ser único; deve ser diferente entre as funções e diferente das variáveis.
- O nome ainda deve sintetizar o que a função faz, de forma clara e de fácil leitura e interpretação.
- Os parâmetros de uma função são separados por vírgula, e é absolutamente necessário que, para cada um deles, seja indicado o seu tipo.

6.1.2 Funcionamento de funções

- Uma função é executada quando seu nome é invocado de alguma parte do programa principal ou até mesmo de outras funções.
- Ao invocar, ou chamar uma função, a parte do programa que fez a chamada fica “suspensa” temporariamente até que a função termine suas instruções. Ao terminar suas instruções a função devolve o controle ao local onde foi invocada.
- Quando um programa faz a chamada de uma função, este deverá passar argumentos para a função que inicializará as variáveis locais com valores. Essas variáveis recebem o nome de parâmetros.
- Quando a função termina de executar suas instruções ela poderá ou não devolver um valor ao programa que a invocou.

6.1.3 Chamando uma função

Este é o meio pelo qual pedimos ao programa principal que interrompa sua execução para executar as instruções de uma função. Já utilizamos funções contidas nas bibliotecas básicas da linguagem C e exemplificaremos abaixo as funções de entrada e saída:

```
#include <stdio.h>
int main (){
    int num;
    printf("Informe um número inteiro: ");
    scanf("%i",&num);
    printf("O quadrado deste valor eh: %i", num*num);
    return 0;
}
```

6.1.4 Função simples

Uma função nada mais é do que um subprograma do programa principal. A função `main()` é a parte principal do nosso programa ou é a função principal. Ela sempre é a primeira a ser executada. Veremos agora uma função que retorna a área do retângulo após receber os argumentos necessários ao seu cálculo.

```
#include <stdio.h>
float arearetangulo(float b, float a); /*Protótipo */

int main (){
    float base, altura, area;
    printf("Base: ");
    scanf("%f",&base);
    printf("\nAltura: ");
    scanf("%f",&altura);
    area = arearetangulo(base,altura);
    printf(" A área do retângulo é: %f", area);
    return 0;
}

float arearetangulo(float b, float a){
    float arearet;
    arearet=b * a;
    return arearet;
}
```



Fique atento!

O nome das variáveis (parâmetros presentes no cabeçalho de uma função é totalmente independente do nome das variáveis que lhe serão enviadas pelo programa que a invoca.

6.1.5 Protótipo de uma função

Uma função não pode ser chamada sem ter sido declarada. A declaração é chamada de protótipo da função. É uma instrução geralmente colocada no início do programa que estabelece o tipo de função e os argumentos que ela recebe. O protótipo da função permite que o compilador verifique a sintaxe de chamada à função (VIVIANE, 2009, p. 112).

Usando o exemplo anterior, podemos observar que o protótipo deve ser declarado antes da primeira declaração de função. No caso abaixo, antes mesmo da função `main()`.

O protótipo de uma função corresponde ao seu cabeçalho seguido de um ponto e vírgula(;). Este identifica toda a estrutura da função (nome, parâmetros e tipo de retorno) (DAMAS, 2007, p. 118).

```
#include <stdio.h>

float arearetangulo(float b, float a); /*Protótipo */

int main (){
.
.
.
}
```

6.1.5.1 Protótipo externo e local

Poderemos declarar funções em C de duas maneiras, a mais utilizada é a protótipo externo, que é declarada antes da função main(). Dessa forma, a função se torna visível em todo o programa. A outra forma é chamada de protótipo local, onde a função é declarada no corpo da função onde ela for chamada.

```
#include <stdio.h>
float arearetangulo(float b, float a); /* Protótipo externo */

float main (){
...}

#include <stdio.h>

float main(){

    float arearetangulo(float b, float a); /* Protótipo local */

...}
```



Fique atento!

Uma boa prática de programação é declarar o protótipo de todas as funções que serão usadas em seu programa antes de a primeira função ser criada.

6.1.6 O tipo de uma função

Uma função deverá ser do tipo de dado que ela retorna através do comando *return*, e não pelo tipo de argumento que ela recebe. No caso de a função não retornar nada, ela será do tipo *void* (VIVIANE, 2009, p. 113).

6.1.7 Comando *return*

O comando *return* é executado no final da função e devolve o controle de execução ao ponto onde a função foi chamada. Veja a seguir as formas de execução deste comando:

```
return;
return expressão;
return (expressão);
```

Em funções que não retornam valor, do tipo *void*, o comando *return* não precisa ser usado, pois a função termina ao encontrar a chave (}) que fecha a função. Entretanto, se for declarado, deverá aparecer sem a expressão, para que apenas finalize a função.



Saiba mais

O comando *return* possui a limitação de retornar apenas um valor.

6.1.8 Parâmetros de uma função

As variáveis que receberão as informações enviadas a uma função são chamadas de parâmetros. A função deve declarar essas variáveis entre parênteses, no cabeçalho de sua definição ou antes das chaves que marcam o início do corpo da função. Os parâmetros podem ser utilizados livremente no corpo da função (VIVIANE, 2009, p.115).

“b” e “a”, são os parâmetros da função.

```
float arearetangulo(float b, float a);
```

6.1.9 Passagem de argumento por valor

No exemplo abaixo, é criada uma variável “x” que receberá o valor passado pela entrada de dados da função “*scanf*”. Em seguida, a função *abs()* passa o valor armazenado na variável para o argumento “a” da função *abs(int a)*, que retornará o valor absoluto para o programa principal (*main*).

```
#include <stdio.h>
int abs(int a );
int main(){
    int x;
    printf("Valor :\n");
    scanf("%i",&x);
    printf("\n0 valores absoluto eh: %i",abs(x));
    return 0;
}
int abs(int a ){
    return (a<0)? a : -a;
}
```


6.1.10 Passagem de argumento por referência

Argumentos são variáveis que contém informações para serem passadas às funções. Passar argumentos por referência é passar endereços de memória que contenha alguma informação, para que a função possa trabalhar essas informações através de seus endereços e até mesmo fazer modificações nos dados contidos nestes endereços se esta for a tarefa proposta à função.

O exemplo a seguir demonstra essa passagem de argumento por referência. Veja que a variável “valor” recebe um valor inteiro usando a função criada, “ao quadrado”. O endereço do argumento armazenado na memória é direcionado para a variável ponteiro “num” da função, que passa o valor contido no endereço do argumento para a variável “n”, que tem seu valor dobrado e retornado para o endereço de “num”, retornando esse valor para o endereço do argumento “valor”. O resultado desta alteração é impresso na tela através da linha “*printf* (“Quadrado: %d”,valor)”. Observe que ao comando *return* não precisou ser utilizado, pois a variável ponteiro escreveu o valor alterado diretamente no endereço da variável “valor”.

```
void aoquadrato(int *num);
void main()
{
    int valor;
    printf("Digite um valor: ");
    scanf("%d", &valor);
    aoquadrato(&valor);
    printf("Quadrado: %d",valor);
}
void aoquadrato(int *num)
{
    int n;
    n=*num;
    *num=n*n;
    return n;
}
```

Uma função deve ser declarada conforme o tipo de dados que ela irá retornar e não com base nos argumentos que serão passados a ela. Caso não seja declarado o tipo da função, a linguagem C interpretará que a função é do tipo *int*.

6.1.11 Variável local e global

As variáveis declaradas dentro do escopo de uma função se tornam inacessíveis em qualquer região do programa e são conhecidas como variáveis locais, porém as variáveis declaradas antes da função *main()*, ou seja, do programa principal, serão acessíveis a todas as regiões do programa, inclusive no interior de qualquer função. São chamadas de variáveis globais.

O programa seguinte cria uma variável chamada “variável global” e armazena nela o valor 1(um). Observe que não houve passagem de valor para o argumento da função e mesmo assim foi facilmente acessada em qualquer região do programa. Veja a saída deste programa na tela a seguir.

```
#include <stdio.h>

int variaveis(void );
int variavelglobal =1;

int main(){
    printf("Variavel global:%i\n",variavelglobal);
    variaveis();
    return 0;
}

int variaveis(void ){
    printf("Variavel global dentro da funcao:%i\n",variavelglobal);
    return 0 ;
}
```

Figura 29 – Variável global e local



Fonte: Elaborada pelo autor.

Se tentarmos imprimir o argumento de uma variável local inicializada dentro de uma função, não obteremos o resultado demonstrado no código a seguir:

```
#include <stdio.h>
void variaveis(void );

void main(){
    variaveis();
    // printf("Variavel local da funcao:%i\n",variavellocal);
}
```

```
//removendo as barras de comentário acima teremos erro
return 0;
}
void variaveis(void ){
    int variavelocal = 3;
    printf("Variavel local da funcao:%i\n",variavelocal);
    return 0 ;
}
```

A “variável local” só poderá ser acessada dentro da função onde foi declarada.

6.1.12 Recursividade

Em C, funções podem chamar a si mesmas. Quando isso ocorre, a função é chamada de recursiva. A função é recursiva se o comando no corpo da função chama a própria função. Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de definição circular. Um exemplo simples de função recursiva é a função fatorial(), como no exemplo a seguir, que calcula o fatorial de um inteiro. O fatorial de um número n é o produto de todos os números inteiros entre 1 e n . Por exemplo, fatorial de 3 é $1 \times 2 \times 3$, ou seja, 6 (DAMAS, 2007).

Exemplo:

```
#include <stdio.h>
int fatorial(int x);
int main(void){
    int numero;
    printf("Calcule o fatorial de:");
    scanf("%i",&numero);
    printf("\n0 fatorial eh: %i\n",fatorial(numero));
    return 0;
}
int fatorial(int x){
    int resultado;
    if (x == 0){
        resultado=1;
    }else {
        resultado = x * fatorial(x - 1);
    }
    return resultado;
}
```

Calculando o fatorial 3 pelo programa anterior, funcionaria assim:

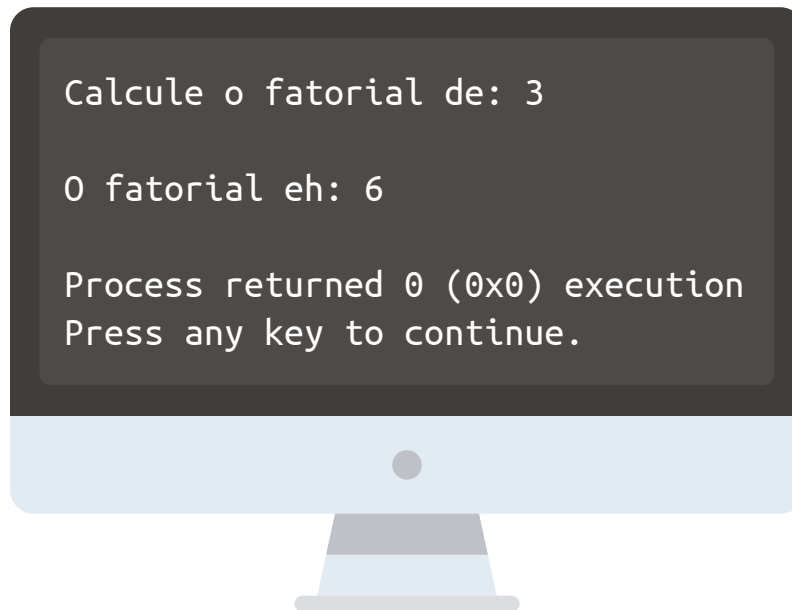
O valor 3 será passado para a função que testará se o valor é igual a zero. Se não, atribuirá um cálculo à variável resultado nos passos a seguir:

Figura 30 – Cálculo do fatorial 1



Fonte: Elaborada pelo autor.

Figura 31 – Cálculo do fatorial 2



Fonte: Elaborada pelo autor.

Problema1: Desenvolva um programa que receba o raio de uma circunferência, através de uma entrada de dados, e que retorne através de uma função que área de um círculo. Sabendo que a área do círculo é: $\pi * R^2$.

```
#include <stdio.h>
float areacirc(float r);
int main(){
    float raio;
    printf("Valor do raio: ");
    scanf("%f",&raio);
    printf("\n\nA area do circulo eh: %f ",areacirc(raio));
    return 0;
}
float areacirc(float r){
    float area= 3.14*(r*r);
    return area;
}
```

Problema2: Crie um programa que receba a temperatura em fahrenheit e retorne o valor em Celsius.

```
#include <stdio.h>
float temp(float f);
int main(){
```

```

    float t;
    printf("Temperatura Fahrenheit:");
    scanf("%f",&t);
    printf("\nTemperatura em celcius: %0.2f",temp(t));
    return 0;
}
float temp(float f){
    f=(f - 32) * 5/9;
    return f;
}

```

Problema3: Elabore um programa simples que passe um valor a uma função, e esta passe o valor para outra e retorne o resultado no programa principal. Será a soma de dois valores absolutos.

```

#include <stdio.h>
float somadeDigitos(float num1,float num2);
float valorAbsoluto(float x);

int main(void){
    float a,b,soma;
    printf("Primeiro Valor: ");
    scanf("%f",&a);
    printf("Segundo Valor: ");
    scanf("%f",&b);
    soma = somadeDigitos(a,b);
    printf("A soma dos valores absolutos: %f ",soma);
    return 0;
}
float somadeDigitos(float num1,float num2){
    if (num1<0){
        num1 = valorAbsoluto(num1);
    }
    if (num2<0){
        num2 = valorAbsoluto(num2);
    }
    return num1+num2;
}
float valorAbsoluto(float x){
    return x * -1;
}

```

Problema4: Crie um programa que receba dois valores e que retorne através de duas funções a soma e a multiplicação de ambos em duas funções distintas.

```

#include <stdio.h>
int somar(int num1,int num2);
float multiplicar(int val1,int val2);

int main(void){

```

```

    float mult;
    int soma,v1,v2;

    printf("Primeiro valor:");
    scanf("%i",&v1);
    printf("Segundo valor:");
    scanf("%i",&v2);
    soma=somar(v1,v2);
    mult=multiplicar(v1,v2);
    printf("Soma: %i", soma);
    printf("\nMultiplicacao: %f",mult);
    return 0;
}
int somar(int num1,int num2){
    return num1+num2;
}
float multiplicar(int val1,int val2){
    return val1*val2;
}

```

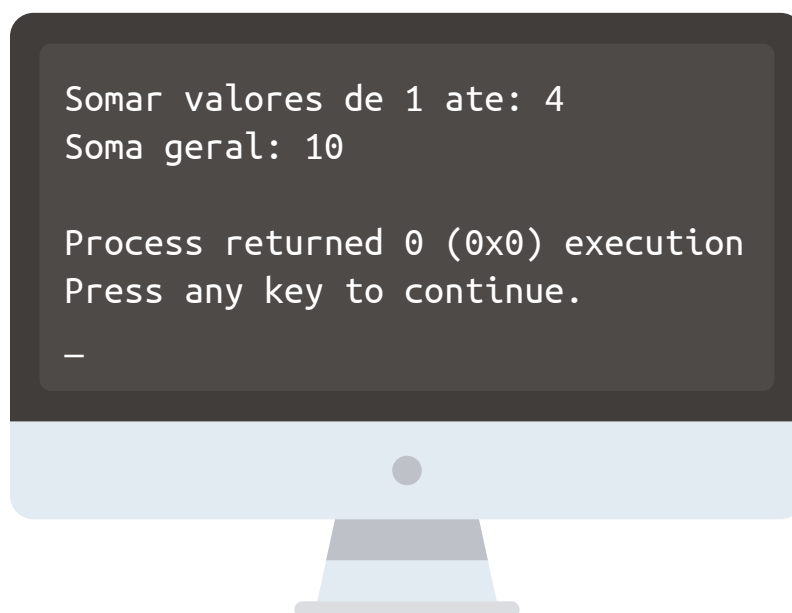
Problema5 - Crie uma função recursiva que some os “n” números a partir do valor 1(um) inicial.

```

#include <stdio.h>
int soma(int n);
int main(void){
    int num,valores;
    printf("Somar de um ate:");
    scanf("%i",&num);
    valores = soma(num);
    printf("soma geral: %i",valores);
    return 0;
}
int soma(int n){
    if (n==0){
        return 0;
    }
    else {
        return n+soma(n-1);
    }
}

```

Figura 32 – Calculo do fatorial 3



Fonte: Elaborada pelo autor.

Síntese da unidade

Nesta unidade, vimos os conceitos de função que dá ao desenvolvedor a possibilidade de subdividir seu programa em seções, possibilitando criar centenas de funções que podem ser reutilizadas ao longo da programação. Isso faz com que o programa se torne mais eficiente, padronizado e de fácil manutenção. Com o uso dos conhecimentos desta unidade, o aluno desenvolverá aplicações mais eficientes e com menor necessidade de manutenção.



Considerações finais

Utilize os conceitos de função para solucionar os novos programas que desenvolver. Sempre que for necessário executar uma tarefa específica, priorize o desenvolvimento de novas funções para aprimorar seus conhecimentos.



Unidade 7

7. Recursividade



Para iniciar seus estudos

Nesta unidade, abordaremos um dos recursos mais interessantes na área de programação a recursividade. Esta técnica traz uma nova visão no que se refere à solução de problemas. Sua principal característica é dividir para conquistar. Vamos agora mergulhar nossas mentes no melhor entendimento deste recurso.



Objetivos de Aprendizagem

- Identificar o pensamento recursivo através de algoritmos.
- Esclarecer os parâmetros para iniciar e finalizar uma função recursiva.
- Aplicar algoritmos de recursão e passos de criação.
- Analisar um algoritmo genérico.
- Identificar quando não se deve utilizar recursividade.

Introdução da unidade

Uma das características das funções é permitir que outras funções sejam chamadas de dentro delas para executar determinada tarefa própria da respectiva função. Abordaremos agora uma técnica de programação que, aplicada às funções, faz com que a própria função chame a si mesma criando a recursividade. Utilizando este recurso, poderemos criar aplicações para trazer soluções a problemas mais complexos para trazer soluções em todas as áreas de desenvolvimento de aplicações para as áreas de inteligência artificial (IA), resolução de teoremas matemáticos, análise e reconhecimento de padrões entre outras (KOFFMAN, 2008. Adaptado.).

7.1 Pensamento recursivo

A recursão é uma técnica que busca a solução de problemas que permite a subdivisão do problema em subproblemas, gerando soluções mais simples para os que teriam uma alta complexidade se não fosse utilizada esta técnica. Como exemplo de um problema de n itens, buscaríamos uma solução com $n - 1$ item, e, dividindo novamente o problema, teríamos $n - 2$ itens e, novamente, $n - 3$ e assim por diante. Devemos pensar que se a solução com apenas um item é simples, poderemos criar uma solução de todo o problema através dos problemas mais simples.

7.1.1 Algoritmo recursivo para um conjunto de imagens

Para exemplificar o funcionamento do pensamento recursivo, vamos considerar um conjunto de imagens como mostra a FIG. 29, a seguir. Criaremos uma solução recursiva em formato de algoritmo para demonstrar como poderíamos “processar” essas imagens de alguma forma (como se precisássemos contar as imagens ou outra tarefa determinada), haveria muita dificuldade em realizar tal tarefa, pois não é informado total de objetos contidos. Seria então possível desenvolver uma recursão para resolver o problema, veja como poderia ser pensada esta solução nos passos a seguir (KOFFMAN, 2008. Adaptado.).

Figura 33 – Conjunto de Imagens



Fonte: SHUTTERSTOCK, 2018.

Algoritmo recursivo para processar as imagens:

1. **Se (if)** existir apenas uma imagem no conjunto.
2. **Então** faça o que é solicitado com a imagem.
- Se não (else):**
3. Faça o que é solicitado com a imagem mais externa do conjunto.
4. Processe o conjunto de imagens aninhadas dentro da imagem mais externa da mesma forma.

Vamos à explicação dos passos do algoritmo:

Passo 1. Se existe apenas uma imagem execute o passo 2. Se existir mais de uma imagem, então execute o passo 3 para processar a imagem mais externa e execute o passo 4, que é a operação recursiva que processará o conjunto de imagens dentro da imagem mais externa. A cada conjunto de imagens terá uma imagem a menos, tornando-se uma versão menor do problema principal.

7.1.2 Algoritmo recursivo para busca em vetor

Seguindo esta lógica do pensamento recursivo, analisaremos a busca por um valor em um vetor de n elementos. Suponhamos um vetor que contenha elementos em ordem crescente. Usando o pensamento recursivo como anteriormente, substituiremos o problema fazendo uma pesquisa no vetor de n elementos por uma pesquisa no vetor de $n / 2$ elementos. Para fazer isso, analisaremos o elemento do meio desse vetor n . Se forem iguais, resolvemos o problema. Caso o elemento seja menor, pesquisaremos nos elementos que estão antes do elemento do meio. Caso contrário, pesquisaremos nos elementos que são posteriores ao elemento do meio. Dessa forma, realizaremos a busca em um vetor de $n / 2$ elementos como descreveremos no algoritmo a seguir:

1. **Se (if)** o vetor estiver vazio.
2. Retorne -1 como o resultado da busca para indicar que está vazio.
- Se não Se (else if)** o elemento do meio for igual ao alvo.
3. Retorne ao elemento do meio como resultado da busca.
- Senão Se (else if)** o alvo for menor do que o elemento do meio.
4. Pesquise recursivamente os elementos do vetor que estão antes do elemento do meio e retorne o resultado.
- Senão (else)**
5. Pesquise recursivamente os elementos do vetor que estão após o elemento do meio e retorne o resultado.

Vamos à explicação dos passos do algoritmo:

Passo 1. Teste para ver se o vetor está vazio. Se estiver, salte para o passo 2 e retorne o valor para indicar isso e termine a busca. Se o passo 1 não for verdadeiro, seguirá para o passo 3 e, se o valor pesquisado for igual ao elemento do meio, será retornado o que achou o elemento. Caso contrário, será executado o passo 4 e 5, pesquisando sempre vetores menores a cada chamada, aproximadamente a metade do tamanho anterior. Assim, cada região menor será diferente e cada elemento do meio também.

7.2 Algoritmo recursivo genérico

7.2.1 Algoritmo genérico

Apresentaremos agora a forma mais genérica da abordagem de algoritmos recursivos:

1. **Se (if)** o problema puder ser resolvido diretamente para o valor atual de n.
2. Resolva-o.
- Senão (else)**
3. Aplique ações recursivas a um ou mais problemas, envolvendo valores menores de n.
4. Combine as soluções dos problemas menores para obter a solução do problema original.

Como regra, determinaremos que o passo 1 faça um teste para finalizar o algoritmo, retornando o resultado no passo 2. A recursão acontecerá no passo 3. Assim, sempre que dividimos o problema, reiniciaremos no passo 1 e verificaremos se esse é o caso inicial ou o caso recursivo.

Vamos agora fazer diversos exercícios para aplicar o pensamento recursivo aos problemas a seguir. Para compreender melhor o funcionamento de uma recursividade, usaremos recursos visuais para demonstrar como tudo funciona visualmente dentro da memória.

7.2.2 Problemas recursivos resolvidos

Problema01: Crie uma função recursiva para somar os números positivos anteriores a partir de um valor informado, através de uma entrada de dados no programa principal, e que retorne a soma ao final da execução programa.

Solução: A seguir, criamos a função baseada nas informações do problema que pede que o valor seja inteiro positivo. Esse valor será passado para a função que fará o teste. Devemos imaginar que o menor valor só poderia ser um. Caso seja menor que um, será retornado sempre 1. Então esse será o limite, ou seja, a função irá parar quando atingir esse valor. Se o valor for maior que 1, então será executada a chamada recursiva que irá somar o valor informado com a chamada recursiva que dará início ao processo novamente até atingir o valor mínimo 1.

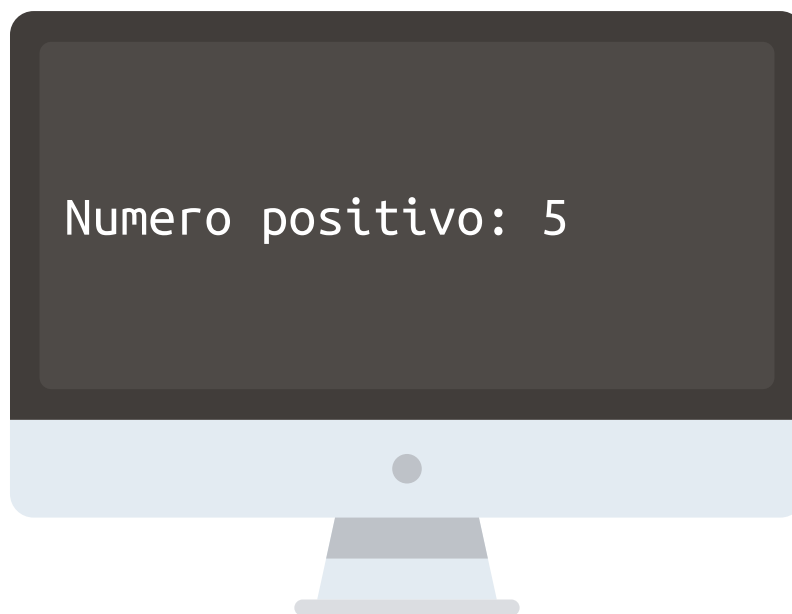
Função do problema1:

```
int somar (int n){
    if (n <=1){
        return 1;
    }
    else {
        return n+somar (n-1);
    }
}
```

Ao passarmos um valor para a função, ela será alocada para memória. Se ela entrar na primeira condição de parada, o programa será finalizado e será retornado o valor 1 ao programa principal, que será listado após as explicações a seguir.

Apresentaremos a seguir os passos da execução da função recursiva através de imagens. Na FIG. 34, faremos a entrada de dados com o valor 5 e apresentaremos os passos do funcionamento da recursividade através de imagens.

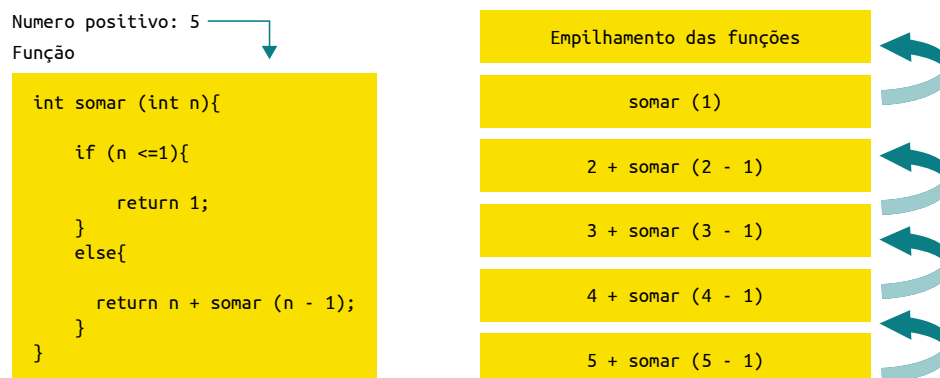
Figura 34 – Entrada de dados do problema01



Fonte: Elaborada pelo autor.

Após enviar o valor, o programa irá passá-lo para a função que, a cada chamada, irá criar empilhamento de processo na memória.

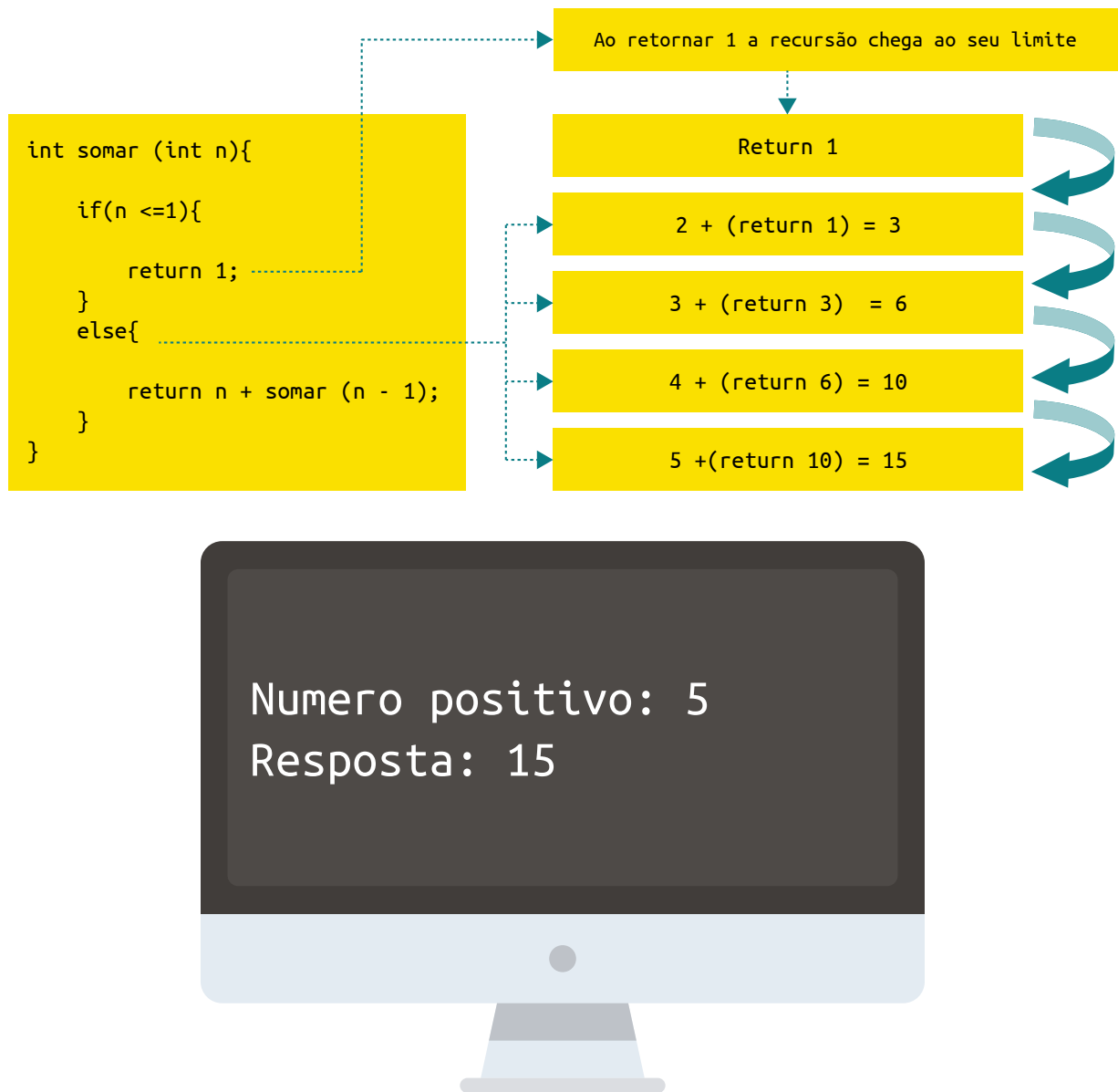
Figura 35 – Passagem de valor para a função



Fonte: Elaborada pelo autor.

Vimos funções serem empilhadas da primeira chamada que recebeu o valor 5 e seguirá até que atinja o limite de 1. Quando isso ocorrer, o empilhamento será encerrado e dará início a outro processo o desempilhamento.

Figura 36 – Desempilhamento da função



Fonte: Elaborada pelo autor.



Saiba mais

Aplicações: Muitas vezes, uma situação recursiva de um problema não poderá garantir a melhor opção para o uso da recursividade. Um bom exemplo é o uso na sequência Fibonacci. A recursão, nesse caso, não seria a melhor opção.

Solução completa do problema 1:

```
#include <stdio.h>

int somar (int n);

int main(){
    int num;
    printf("\n Numero positivo:");
    scanf(" %i", &num);
    printf("\n Resposta: %i", somar (num));
    return 0;
}

int somar (int n){
    if (n <= 1){
        return 1;
    }
    else
    {
        return n + somar(n -1);
    }
}
```

**Saiba mais**

A aplicação da técnica de recursividade pode ser melhor aproveitada em algumas situações: manipulação de árvores; analisadores léxicos recursivos de compiladores; problemas que envolvem tentativa e erro.

Problema 2: Crie uma função que retorne o fatorial de um valor inteiro positivo. Desenvolva um programa para testar a função.

Solução: Como sabemos, o fatorial de um número é a multiplicação deste por todos os seus antecessores até chegar no número 1(um). Observe que o 0(zero) é excluído. Podemos observar, por essa informação, que o zero é o limite para nossa função; logo, a função deverá se subdividir, criando um empilhamento de funções na memória até atingir o valor limite que é zero. Usaremos novamente exemplos visuais para demonstrar todo o processo.

A função ficaria então assim:

```
int fat (int n){
    if (n==0) {
        return 1;
    }
    else{
        return n * fat(n - 1);
    }
}
```

Seguindo nossa análise pelo algoritmo genérico, poderemos observar que o limite da função é determinado pelo valor zero, e a condição da diferença cria a recursividade. Então, desvendando os passos da função descrita, um valor inteiro será passado para a função que testará se o valor é igual a zero. Se for igual, então a função retornará o valor 1. Caso contrário, entrará em questão a chamada recursiva que, a cada passo, elimina um elemento, criando o empilhamento, até que o elemento da condição seja nosso limite, que é 0(zero).

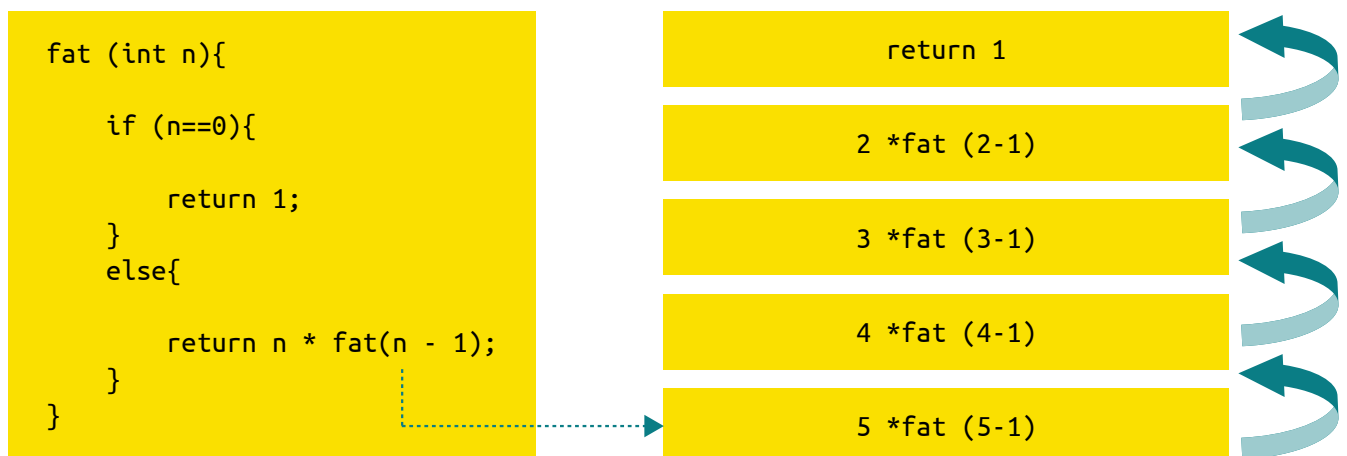
Vejamos agora nosso empilhamento com ficaria se passássemos o valor 5 para o programa.

Figura 37 – Tela de entrada de dados



Fonte: Elaborada pelo autor.

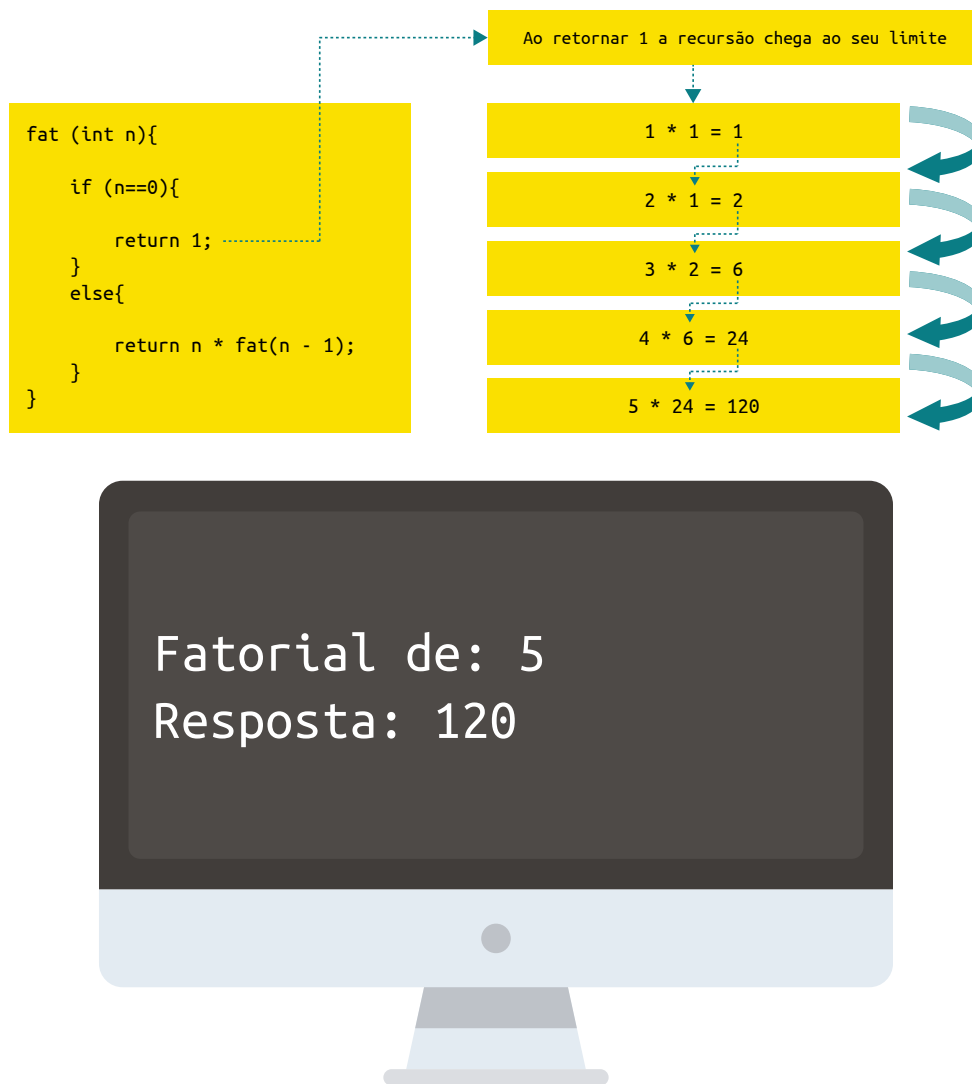
Figura 38 – Empilhamento da função recursiva



Fonte: Elaborada pelo autor

A função recebeu o valor 5, que é testado pela condicional *if*, acionou a recursão e repassou o controle para recursão, pois o valor é maior que zero. Em seguida, começam a ser empilhadas as funções até atingir o limite restringido na condicional que é 0(zero). Nesse instante, a condicional retornará o valor 1 que se multiplicará com *n* e, assim, desempilhará até que toda a pilha seja desmontada, atingindo o início onde foi criada na entrada com o valor 5.

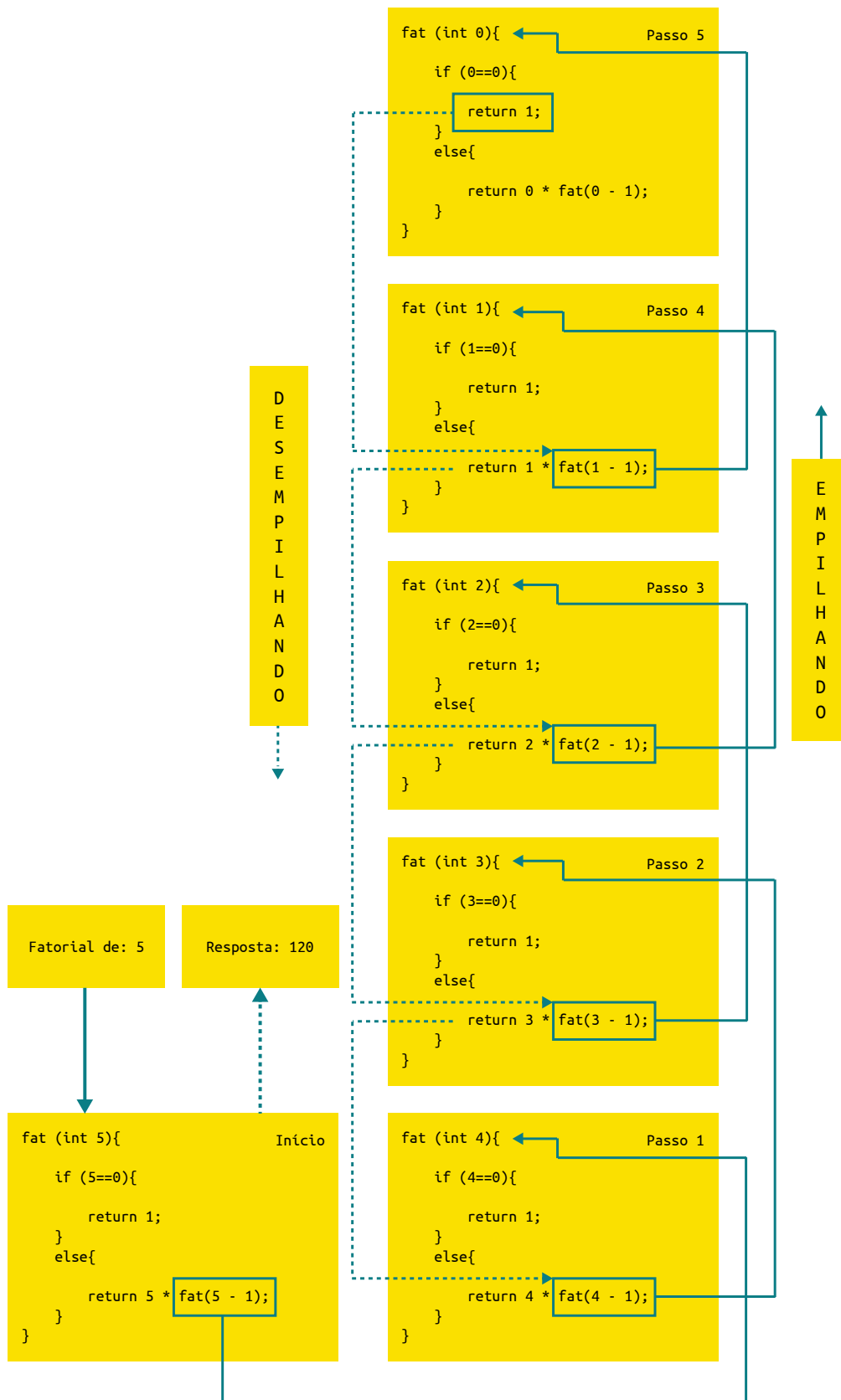
Figura 39 – Desempilhamento da função recursiva



Fonte: Elaborada pelo autor.

Poderíamos fazer passo a passo o empilhamento usando a própria função para mostrar ainda mais como cada passo funciona. A seguir, faremos a alocação da função em pilhas na memória e detalharemos os passos fazendo a substituição das variáveis e dos retornos. A função envia para o passo seguinte.

Figura 40 – Empilhamento da função recursiva



Fonte: Elaborada pelo autor.

Para alguns, essa forma pode ser mais clara de entender o processo de empilhamento. Observando cada passo do empilhamento, é possível notar que, a partir do primeiro passo, a recursividade ocorre a partir do passo 1, o mesmo número de vezes que o valor de entrada, 5.

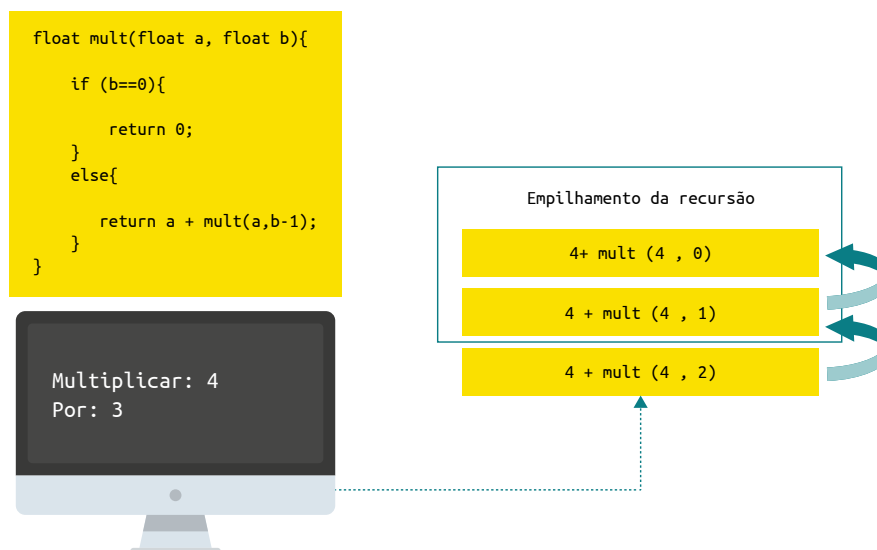
O código abaixo tem a solução do problema 2.

```
#include <stdio.h>
int fat(int n);
int main()
{
    int num=0;
    printf("\n Fatorial de:");
    scanf(" %i",&num);
    printf("\n Resposta: %i\n\n\n\n",fat(num));
    return 0;
}
int fat(int n){
if(n==0){
return 1;
}
else
{
    return n*fat(n-1);
}
}
```

Problema 3: Elabore uma função recursiva que resolva o cálculo de multiplicação usando a operação de soma. Crie o programa para testar a função.

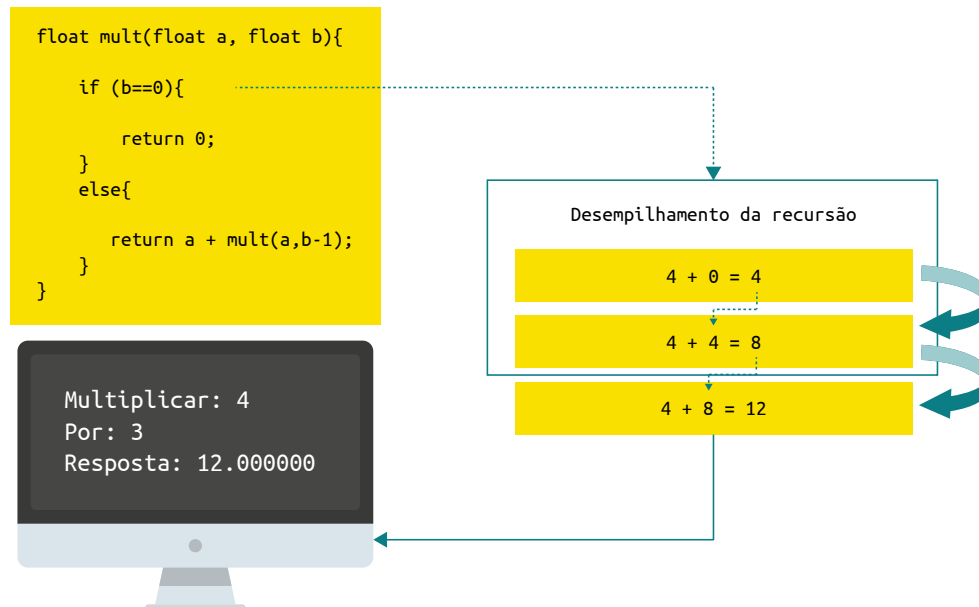
Solução: este problema baseia-se em substituir a operação de multiplicação por operações para somar cada elemento a um número determinado de vezes que será determinado pela entrada de dados. Matematicamente falando, seria, por exemplo, multiplicar $4 \times 5 = 20$. Para a solução, seria calculado da seguinte maneira $4+4+4+4+4 = 20$. Logo, a função a seguir resolve o problema.

Figura 41 – Empilhamento da função recursiva



Fonte: Elaborada pelo autor.

Figura 42 – Desempilhamento da função recursiva



Fonte: Elaborada pelo autor.

O código abaixo tem a solução do problema 3.

```

#include <stdio.h>
float mult(float a, float b);
int main(){
    int x,y;
    printf("\n Multiplicar:");
    scanf(" %i",&x);
    printf("\n Por:");
    scanf(" %i",&y);
    printf("\n Resposta: %f ",mult(x,y));
    return 0;
}
float mult(float a, float b){
    if (b==0){
        return 0;
    }
    else
    {
        return a + mult(a,b-1);
    }
}

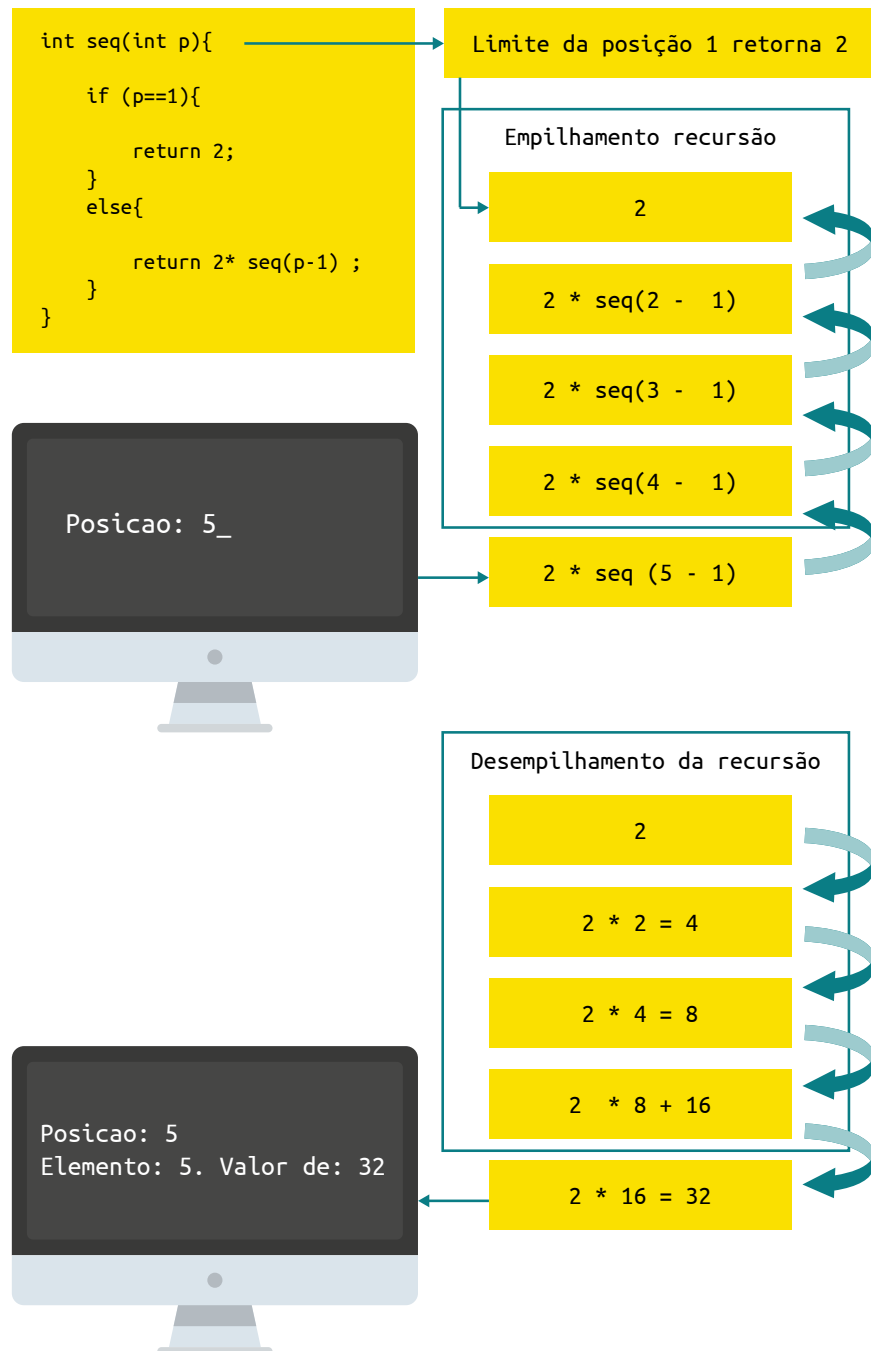
```

Problema 4: Observando a sequência de números inteiros, verificamos que há uma similaridade. Elabore uma função recursiva que receba como referência uma posição indeterminada de algum valor dessa sequência e que retorne o valor correspondente. Apresente o programa completo.

$S = \{ 2, 4, 8, 16, 32 \dots \}$

Solução: Podemos observar que a sequência obedece a uma regra de crescimento a cada passo. Quando salta de uma posição para outra, ela se multiplica por 2. Para encontrarmos o elemento, nos será informada a posição deste na sequência; entretanto, não temos essa sequência completa. Pegaremos a posição informada e a reduziremos até que chegue à posição inicial, que é 1. Da mesma forma que poderíamos criar uma busca evoluindo da posição 1 até a informada, também poderemos fazer usando a recursão partindo do ponto final, da informação da posição do elemento. Então, a função ficaria assim:

Figura 43 – Empilhamento da função recursiva



Fonte: Elaborada pelo autor.

Solução completa do problema 4.

```
#include <stdio.h>
#include <stdlib.h>
int seq (int p);
int main (){
    int num;
    printf("Posicao:");
    scanf("%i",&num);
    printf("Elemento: %i . Valor de: %i",num,seq(num));
}
int seq (int p){
    if (p==1) {
        return 2;
    }
    else {
        return 2* seq(p-1) ;
    }
}
```

Síntese da unidade

Abordamos os conceitos de recursividade. Aprendemos que esta técnica aborda sequências que têm comportamento recursivo, limitando-se por um ponto de parada e um ponto de partida. Entendemos que nem todas as soluções de problemas podem ser resolvidas através da recursão, pois isso causaria um excesso de alocações e desalocação, comprometendo os recursos computacionais. O uso de recursividade deve sempre ser analisado juntamente a soluções sem recursão, mesmo que sejam mais complexas.



Considerações finais

Finalizamos mais uma etapa inspiradora em linguagem C. Esta unidade trouxe muito saber e muito a ser analisado para se desenvolver, com mais cuidado, aplicações para que, em linguagem C, sejam feitas com muita meticulosidade. Portanto, siga em frente e descubra mais, muito mais sobre esta linguagem incrível. Parabéns!



8

Unidade 8

8. Manipulação de Arquivos



Para iniciar seus estudos

Nessa unidade, você conseguirá compreender o conceito de arquivos em C com teoria e implementação de exemplos para demonstração. Entenderá também a necessidade de uso dos arquivos para a persistência de dados e o quão importante isso é para um bom programador. Então, vamos estudar?



Objetivos de Aprendizagem

- Descrever as principais funções utilizadas em arquivos.
- Aplicar o conhecimento sobre arquivos em seus códigos.
- Criar programas que armazenem dados em arquivos.

Introdução da unidade

Durante esta disciplina, serão discutidas várias formas de manipular os dados na programação. Você conhecerá as variáveis primitivas, os tipos construídos (*structs*), as variáveis compostas (vetores e matrizes), entre outras informações necessárias para a criação de bons códigos de programação.

Você perceberá que, em momento algum, seus programas manipularão dados salvos na memória secundária do computador.

Nesta unidade, também será apresentada e demonstrada a manipulação de arquivos, conhecimento este que lhe permitirá criar programas que salvem dados no computador de forma que esses dados não se percam quando o programa for fechado, ou o computador, desligado.

E mais: serão apresentados os principais conceitos relativos à manipulação de arquivos, ponteiros para arquivos, abertura, escrita, leitura, fechamento e exclusão de arquivos.

Por fim, será apresentada uma implementação com leitura e escrita em arquivos.

8.1 Conceito de manipulação de arquivos em C

A manipulação de arquivos em um código de programação possibilita guardar dados que poderão ser acessados posteriormente na programação, mesmo que o programa seja fechado, e o computador, desligado. Isso é chamado de persistência de dados, sendo este um conceito muito importante para a implementação de programas que necessitam armazenar informações de forma permanente no computador.

No texto a seguir, Deitel destaca a importância do uso de arquivos na programação:

O armazenamento de dados em variáveis e *arrays* é temporário – esses dados são perdidos quando um programa é encerrado. Arquivos são usados na conservação permanente de dados. Os computadores armazenam arquivos em dispositivos secundários de armazenamento, especialmente dispositivos de armazenamento de disco (DEITEL, 2011, p. 350).

Quando um programa cria e manipula arquivos, é gerado um arquivo com o nome e o tipo do arquivo. O nome é criado pelo programador de acordo com a necessidade. Já o tipo deve ser alguma espécie de arquivo válido no sistema operacional (txt para arquivos de texto, xls para planilhas eletrônicas, dat para armazenamento de dados em geral, entre outros tipos válidos para um arquivo).

Não é viável para um programa utilizar uma memória secundária do computador diretamente durante o processamento. Isso seria um gargalo, pois o acesso a essas formas de armazenamento é bem mais lento que o acesso à memória principal. Para evitar esse acesso, que poderia tornar o programa inviável para o uso devido à lentidão, é criado um buffer, que, conforme Mizrahi (2008, p. 371), proporciona várias vantagens ao programa, como poder executar várias operações nos dados do buffer sem ter que acessar o arquivo a todo o momento.



Fique atento!

Todos os arquivos criados nos exemplos desse Material Didático serão gerados no mesmo diretório onde o código está salvo.



Refleta

Um programa que não armazena os dados fica limitado a aplicações mais simples, sendo os programas mais utilizáveis aqueles que fazem a persistência de dados.

8.1.1 Ponteiros para arquivos

Quando se cria um arquivo em linguagem C, a posição que ele ocupa na memória é acessada por meio de uma variável ponteiro. “Abrir um arquivo retorna um ponteiro para uma estrutura FILE (definida em <stdio.h>), que contém informações usadas para processar o arquivo” (DEITEL, 2011, p. 351).

A diferença da utilização de arquivos para o uso de *arrays* é que estes ficam na memória principal do computador e, como essa memória é volátil, o que é salvo nela não fica disponível após o fim da execução do programa ou desligamento do computador.

No caso do uso de arquivos, o arquivo é salvo na memória secundária, onde fica guardado para ser acessado depois, podendo inclusive ser acessado fora do programa onde foi criado, diretamente no arquivo.

8.1.2 Abertura de arquivos

Na linguagem C, existem várias formas de se fazer o armazenamento de dados na memória secundária, sendo uma dessas formas a criação e edição de arquivos. Mizrahi recomenda que, “para abrir um arquivo em alto nível devemos usar a função `fopen()`” (MIZRAHI, 2008, p. 342).

A autora ainda apresenta a seguinte sintaxe para a função `fopen()`:

```
FILE fopen(const char *nome_arquivo, const char *modo_de_abertura)
```

O nome do arquivo deve ser uma constante literal, composta pelo nome com que o arquivo será criado seguido de um ponto e do tipo de arquivo. Exemplo para um arquivo do tipo texto (txt):

“arquivo1.txt”

Já o argumento `modo_de_abertura` é onde será colocado algum dos modos mostrados e descritos no quadro a seguir:

Quadro 11 – Modos de Abertura de um Arquivo

"r"	Abre arquivo de texto apenas para leitura.
"w"	Cria um arquivo de texto apenas para escrita.
"a"	Anexa novos dados a um arquivo de texto.
"rb"	Abre um arquivo binário apenas para leitura.
"wb"	Cria um arquivo binário apenas para escrita.
"ab"	Anexa novos dados a um arquivo binário.
"r+"	Abre arquivo de texto para leitura e escrita.
"w+"	Cria um arquivo de texto para leitura e escrita.
"a+"	Anexa novos dados ou cria um arquivo de texto leitura e escrita.
"rb+"	Abre um arquivo binário para leitura e escrita.
"wb+"	Cria um arquivo binário apenas para escrita e leitura.
"ab+"	Anexa novos dados a um arquivo binário para operações de leitura e de escrita.

Fonte: Adaptado de Ascencio (2012, p.423).

Por fim, tem-se que a abertura de um arquivo para escrita utilizando a função `fopen()` fica conforme mostrado:

```
p=fopen("arquivo.txt","w");
```

Onde "arquivo.txt" representa o nome e o tipo do arquivo respectivamente; "w" indica que o arquivo será aberto somente para escrita; e p é um ponteiro para FILE.



Fique atento!

Algumas situações podem impedir que se tenha sucesso na abertura de arquivos. Mizrahi cita duas das mais comuns: "Se você tiver pedido para abrir um arquivo para a gravação, é provável que não abra por não haver mais espaço no disco. E se foi para leitura, talvez o arquivo não abra por ainda não ter sido criado" (MIZRAHI, 2008, p. 342).

O código que será apresentado a seguir foi proposto por Ascencio (2012, p. 425) para exemplificar a abertura de um arquivo no código em C. Nele, pode-se notar que foi criado o ponteiro *p do tipo FILE, e esse ponteiro recebe o endereço do arquivo aberto para escrita.

O código mostra uma estrutura condicional `if(p==NULL)`, que será verdadeira caso tenha ocorrido algum erro na abertura de "arquivo.txt". Caso o arquivo tenha sido aberto corretamente (`else`), é mostrada uma mensagem informando "Sucesso na abertura", sendo realizado o fechamento do mesmo. Caso a resposta da função `fclose(p)` for 0, é impresso o sucesso no fechamento; caso contrário, é mostrado que ocorreu um erro.

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

```
int main()
```

```
{
```

```
    FILE *p;
```

```

int resposta;
p=fopen("arquivo.txt","w");
if(p==NULL)
    printf("\nErro na abertura");
else
{
    printf("\nSucesso na abertura");
    resposta=fclose(p);
    if(resposta==0)
        printf("\nSucesso no fechamento");
    else printf("\nErro no fechamento");
}
return 0;
}

```



Fique atento!

Segundo Deitel (2011, p. 353), um erro comum na manipulação de arquivos acontece quando se abre um arquivo existente para gravação ("w") e quando não se tem a intenção de descartar o conteúdo e, mesmo de forma não intencional, o conteúdo do arquivo é descartado sem aviso.

Deitel ainda destaca outro erro, que é quando se tenta referenciar um arquivo que não foi aberto no programa.

8.1.3 Escrita em arquivos

Mizrahi (2008, p. 344) explica que, quando você envia um caractere para um arquivo via `fputc()`, ele é colocado no *buffer* e só é gravado no arquivo quando o mesmo é fechado.

Ascencio (2012, p. 425) propõe no código a seguir uma implementação com o uso da função `fputc()` para a inserção de caracteres em um arquivo. O código é muito parecido com o apresentado anteriormente sobre a abertura de arquivos em C. Nele, pode-se notar que foi criado o ponteiro `*p` do tipo `FILE`, que recebe o endereço do arquivo aberto para escrita.

O código mostra ainda uma estrutura condicional `if(p==NULL)`, que será verdadeira caso tenha ocorrido algum erro na abertura de "caract.dat". Caso o arquivo tenha sido aberto corretamente (`else`), é solicitado ao usuário que digite um caractere para ser armazenado. Caso esse caractere seja diferente de 'f' – devido ao `while(carac!='f')`

–, é chamada a função `fputc(carac, p)` com o caractere a ser armazenado e o ponteiro que indica onde está o arquivo. Caso tenha acontecido algum erro na gravação do caractere (`if(ferror(p))`), uma mensagem é impressa informando sobre a situação. Caso contrário, é impressa uma mensagem informando que a gravação foi realizada com sucesso.

Em seguida, o programa possibilita que o usuário digite novamente um caractere e, como o código acontece dentro de uma repetição, só encerrará o cadastro de valores quando o usuário digitar o caractere 'f', conforme condição do `while`.

Tudo o que aconteceu até aqui foi feito no *buffer* e, assim que a repetição é encerrada, realizam-se o fechamento do buffer e a gravação dos valores no arquivo.

Para Ascencio (2012, p. 424), “a função `ferror()` detecta se ocorreu algum erro durante uma operação com arquivos(...)”. A autora informa que, se a função retornar algum valor diferente de 0 (zero), ocorreu um erro durante a última operação realizada com o arquivo, sendo apresentado o seguinte protótipo para a função:

```
int ferror(FILE *arq);

#include<stdio.h>

int main()
{
    FILE *p;
    char carac;
    p=fopen("caract.dat","a");
    if(p==NULL)
        printf("\nErro na abertura");
    else
    {
        printf("\nDigite um caractere:");
        scanf("%c",&carac);
        while(carac!='f')
        {
            fputc(carac, p);
            if(ferror(p))
                printf("\nErro na gravação do caractere");
            else printf("\nGravação realizada com sucesso");
            printf("\nDigite outro caractere:");
            scanf("%c",&carac);
        }
    }
}
```

```

    }
}

fclose(p);

return 0;

}

```

Os arquivos podem ser escritos tanto com um caractere por vez quanto utilizando cadeias de caracteres (*strings*). Quando se desejam escrever cadeias de caracteres em arquivos, deve ser utilizada a função `fputs()`, que recebe como argumentos (i) a constante literal ou variável que se deseja escrever no arquivo e (ii) o ponteiro que está com o endereço do arquivo.

Para Mizrahi (2008, p. 347), “a função `fputs()` recebe dois argumentos: o ponteiro *char*, para a cadeia de caracteres a ser gravada, e o ponteiro para a estrutura `FILE` do arquivo”.

A autora ainda apresenta a função conforme a seguir:

```
int fputs(const char *string, FILE *ponteiro_arquivo);
```

Onde **string* é o valor a ser armazenado no arquivo, e **ponteiro_arquivo* é o ponteiro do tipo `FILE` que tem o endereço do arquivo obtido pela função `fopen()`.

No código a seguir, proposto por Ascencio (2012, p. 427), é criada a variável ponteiro **p* para o tipo `FILE`, além de uma *string* de nome cadeia. O ponteiro **p* recebe o endereço do arquivo aberto pela função `fopen()` e é conferido pelo programa se ocorreu algum erro na abertura do arquivo (`if(p==NULL)`); caso o arquivo tenha sido aberto corretamente, o usuário poderá digitar uma *string* para ser armazenada. Enquanto essa *string* for diferente de “fim”, o programa continuará em execução, recebendo novas cadeias de caracteres e inserindo no arquivo por meio da função `fputs()`. Quando a palavra “fim” for digitada, o programa sairá da repetição, e o buffer do arquivo será fechado enquanto seus valores serão gravados no arquivo `cadeias.txt`.

```

#include<stdio.h>

#include<string.h>

int main()
{
    FILE *p;

    char cadeia[30];

    p=fopen("cadeias.txt","a");

    if(p==NULL)
        printf("\nErro na abertura");

    else
    {

```

```

printf("\nDigite uma cadeia de caracteres: ");
gets(cadeia);
while(strcmp(cadeia,"fim")!=0)
{
    fputs(cadeia,p);
    if(ferror(p))
        printf("\nErro na gravação da cadeia");
    else
        printf("\nGravação realizada com sucesso");
    printf("\nDigite outra cadeia: ");
    gets(cadeia);
}
}
fclose(p);
return 0;
}

```



Saiba mais

A função `strcmp()` compara duas cadeias de caracteres (*strings*) sem considerar a diferença entre letras maiúsculas e minúsculas, retornando 0 caso sejam iguais. A função é definida em *string.h*

8.1.4 Leitura de arquivos

Quando se quer ler um arquivo, é necessário saber quando o mesmo chegou ao fim para se encerrar a leitura. A função `feof()` indica o fim das informações no arquivo.

Conforme Mizrahi (2008, p. 344), a função `feof()` retorna verdadeiro se o final do arquivo for atingido; caso contrário, retorna como falso.

O código para leitura de um arquivo, proposto por Ascencio (2012, p. 426), mostra a declaração do ponteiro `*p` e a abertura do arquivo "caract.dat" no modo somente leitura ("r").

Após conferir se o arquivo foi aberto corretamente, é feita a leitura dos caracteres cadastrados no arquivo aberto. A leitura acontecerá enquanto existirem caracteres a serem lidos, conforme a estrutura de repetição `do.while`

com a condição de saída `while(!feof(p))`, que retornará verdadeiro quando o arquivo não tiver mais valores a serem impressos.

A variável `carac` (que foi declarada do tipo `char`) recebe o retorno da função `fgetc(p)`, que, segundo Ascencio (2012, p. 426), retorna o caractere cadastrado no arquivo apontado por `p`. Em seguida, é conferido se ocorreu algum erro na última operação (`if(ferror(p))`); caso não tenha acontecido, é impresso o valor do caractere lido do arquivo. Depois que o arquivo for todo impresso, ele é fechado, e a execução do programa é terminada.

```
#include<stdio.h>

int main()
{
    FILE *p;
    char carac;
    p=fopen("caract.dat","r");
    if(p==NULL)
        printf("\nErro na abertura");
    else
    {
        do{
            carac=fgetc(p);
            if(ferror(p))
                printf("\nErro na leitura do caractere");
            else
                {if(!feof(p))
                    {
                        printf("\nLeitura realizada com sucesso");
                        printf("Caractere lido: %c",carac);
                    }
                }
        }while(!feof(p));
    }
    fclose(p);
    return 0;
}
```


Para realizar a leitura de um arquivo preenchido com cadeias de caracteres, as diferenças significativas no código se concentram no uso da função `fgets()`, que, segundo Ascencio (2012, p. 428), recebe como parâmetro uma variável do tipo *string* (que será responsável por retornar a cadeia extraída do arquivo), seguido do número de caracteres a serem retornados na *string* e do ponteiro para o arquivo. No código, a função `fgets()` recebe cadeia – 5 e `p`, respectivamente.

```
#include<stdio.h>

int main()
{
    FILE *p;
    char cadeia[5];
    p=fopen("cadeias.txt","r");
    if(p==NULL)
        printf("\nErro na abertura");
    else
    {
        while(!feof(p))
        {
            fgets(cadeia,5,p);
            if(ferror(p))
                printf("\nErro na leitura da cadeia");
            else
            {
                printf("\nLeitura realizada com sucesso");
                printf("Cadeia lida: %s",cadeia);
            }
        }
    }

    fclose(p);
    return 0;
}
```



Saiba mais

É importante lembrar que uma *string* em C sempre é declarada e lida com palavras de um caractere a menos que o declarado. Isso acontece por causa do caractere de fim de *string*, que ocupa uma posição no vetor de *char*.

8.1.5 Fechamento de arquivos

Para Mizrahi (2008, p. 343), um arquivo, quando aberto para manipulação em alto nível, é utilizado por meio de um buffer, que armazena as informações que estão sendo alteradas. No fim do processamento, o buffer deve ser fechado para que as informações sejam gravadas no arquivo, e a memória temporária, liberada. A autora ainda apresenta a função conforme a seguir:

```
int fclose(FILE *ponteiro_arquivo);
```

Em todos os códigos já apresentados nesta apostila, a função `fclose()` foi utilizada, pois era necessária para o correto funcionamento. O exemplo apresentado aqui está da mesma forma que o já utilizado:

```
fclose(p);
```

Onde `p` é uma variável ponteiro do tipo `FILE` que recebeu anteriormente a função `fopen()`.

8.1.6 Apagando arquivos

Os arquivos do tipo texto podem ser excluídos utilizando a função `remove`, conforme mostrado no código a seguir. Basta que se passe o nome do arquivo ou o nome com o caminho do arquivo caso ele esteja em outro diretório.

Para Ascencio (2012, p. 433), a função `remove` é responsável por apagar arquivos, e seu protótipo é:

```
int remove(char *nome_arquivo);
```

No protótipo apresentado, o `nome_arquivo` representa o nome físico do arquivo que será excluído, podendo ser também o caminho do arquivo. Se a função for executada com êxito, retornará o valor 0; caso contrário, retornará outro valor inteiro diferente de 0 (zero).

No código, é excluído o `arquivo.txt` criado no primeiro exemplo desta apostila, sobre abertura de arquivos.

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

```
int main()
```

```
{
```

```
    if(remove("arquivo.txt")==0)
```

```
        printf("Arquivo removido com sucesso!");
```

```

else

    printf("Ocorreu algum erro na remoção do arquivo!");

return 0;

}

```

De acordo com o exposto até aqui, pôde-se perceber que a manipulação de arquivos é bastante simples de ser feita. Na sequência desta unidade, serão apresentadas aplicações utilizando arquivos.

8.2 Exemplo de aplicação para arquivos

Para entender melhor como funcionam os arquivos, imagine uma situação em que seja necessário o arquivamento do código de acesso de várias pessoas a um sistema. É importante nessa situação que o usuário possa inserir dados e realizar a impressão do arquivo quando quiser, e que isso possa acontecer por quantas vezes for necessário.

O código a seguir apresenta as funcionalidades necessárias para a resolução do problema proposto e ainda possibilita a criação ou abertura de um arquivo que tem o nome digitado pelo usuário. Isso é feito com a digitação do nome do arquivo e, na sequência, a concatenação com a extensão .txt para gerar um arquivo de texto. O código está comentado com as explicações sobre seu funcionamento.

```

#include<stdio.h>

#include<string.h>

int main()

{

    FILE *p;

    char cadeia[6], nomeArquivo[25];

    int op;

    printf("Digite o nome para o arquivo de texto que será " "gerado ou aberto (caso já exista:");

    scanf("%s",nomeArquivo);//lê o nome do arquivo

    strcat(nomeArquivo,".txt");//concatena o nome com o (.txt)

    op=1;//inicializa com 1 para entrar a primeira vez no flag

    while(op!=0)//repetirá enquanto não for digitado 0

    {

        p=fopen(nomeArquivo,"a");//Abre o arquivo

        if(p==NULL)//Se o arquivo não foi aberto corretamente

            printf("\nErro na abertura");
    }

```

```

else
{
    //Se o arquivo foi aberto corretamente
        //Mostra o menu de opções
    printf("\n0 que deseja fazer com o Arquivo:");

        "\n1 - Inserir Valores,"
        "\n2 - Ler Os Valores do Arquivo,"
        "\n0 - Sair e terminar o programa: ";

    scanf("%i",&op);
    if(op==1)//Inserir valores no arquivo
    {
        printf("\nDigite o código de 5 caracteres: ");
        fflush(stdin);//Libera o buffer de char
        gets(cadeia);//Faz a leitura do código
        fputs(cadeia,p);//Insere o código no arquivo
        if(ferror(p))//Confere se houve erro
            printf("\nErro na gravação da cadeia");
        else
            printf("\nGravação realizada com sucesso");
    }else if(op==2)
    {
        //Imprimir valores do arquivo
        do
        {
            //cadeia recebe uma sequência de 4 caracteres no arquivo
            fgets(cadeia,6,p);
            if(ferror(p))//Confere se houve erro
                printf("\nErro na leitura da cadeia");
            else if (!feof(p)) printf("\n%s",cadeia);
        }while(!feof(p));

        //repete até a última cadeia cadastrada.
    }

    getchar();
}

```

```

    }

//A cada vez que é feita uma interação com o arquivo //ele é fechado e aberto novamente para
que os //valores no buffer sejam salvos no arquivo.

    fclose(p);

}

return 0;

}

```

O código apresentado como exemplo manipula os arquivos, inserindo e recuperando as informações armazenadas de forma a demonstrar como o arquivo pode ser manipulado em uma aplicação real da programação.

Síntese da unidade

Nesta unidade, você aprendeu sobre a manipulação de arquivos, viu que arquivos são uma das formas de se fazer persistência de dados quando se programa em linguagem C e, também, que a manipulação do arquivo no programa é feita por meio de buffer.

Você também aprendeu sobre ponteiros para arquivos, que são a forma de se manipular o arquivo e seu buffer dentro da programação.

Sobre a manipulação dos arquivos pelo programa, você entendeu como é feita a abertura de arquivos, com a função `fopen()` e os modos de abertura possíveis na linguagem C.

Estudou ainda sobre a escrita e leitura em arquivos com caracteres e cadeias de caracteres, utilizando as funções `fputc()` e `fputs()` para escrita e `fgetc()` e `fgets()` para leitura dos valores no arquivo.

Foram explicados também o fechamento de arquivos (`fclose()`) e a função `remove()` para apagar arquivos.

Por último, foi apresentada uma implementação que faz a leitura e escrita em um arquivo, além de criar ou abrir o arquivo com o nome digitado pelo usuário.



Considerações finais

Você chegou ao final desta unidade. Foram aprendidas muitas coisas novas; dentre elas, vetores e matrizes, ponteiros e a alocação dinâmica de memória, estruturas ou registros, programação modular com funções em C, recursividade e, agora, manipulação de arquivos.

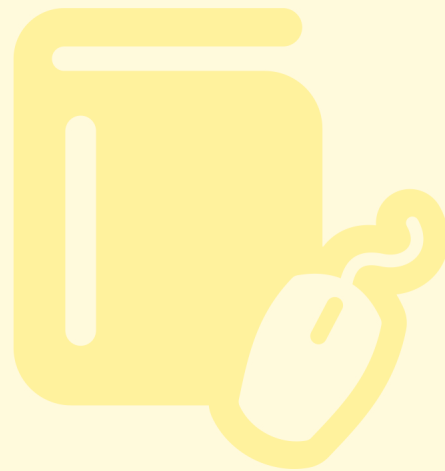
Tudo foi apresentado com conceituação e demonstração de aplicações práticas. Com todo o conhecimento obtido até aqui, você está se tornando um ótimo programador, com domínio de recursos diversos da linguagem, além de ter informações sobre as boas práticas de programação.

Para o entendimento completo desta unidade, é fundamental que você acesse todos os objetos disponíveis para ela, não deixe de assistir à videoaula e de utilizar os materiais de resumo e aprofundamento de conteúdo. Resolva também as questões propostas e o desafio com o máximo de atenção. Agora, siga em frente!





Referências bibliográficas



ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores:** algoritmos, PASCAL, C/C++ (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson Education do Brasil, 2012.

CASTRO, J. **Linguagem C na prática.** São Paulo: Ciência Moderna, 2008.

CORMEN, Thomas H.; LEISERSON, Charles. E.; RIVEST, Ronald L.; STEIN, Clifford. **Algoritmos:** teoria e prática 1. Tradução da segunda edição [americana]: Vandenberg D. de Souza. Rio de Janeiro: Elsevier, 2002.

DALMAS, Luis. **Linguagem C.** Trad. João Araújo Ribeiro, Orlando Bernardo Filho. 10. ed. Rio de Janeiro: LTC, 2016.

DEITEL, Paul; DEITEL, Harvey. **C: como programar.** São Paulo: Pearson Prentice Hall, 2011.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação: a construção de algoritmos e estruturas de dados.** 3. ed. São Paulo: Prentice Hall, 2005.

KOFFMAN, Elliot B. **Objetos, abstração, estruturas de dados e projeto usando C++.** Trad. Sueli Cunha. Revisão técnica Orlando Bernardo Filho, João Araújo Ribeiro. Rio de Janeiro : LTC, 2008.

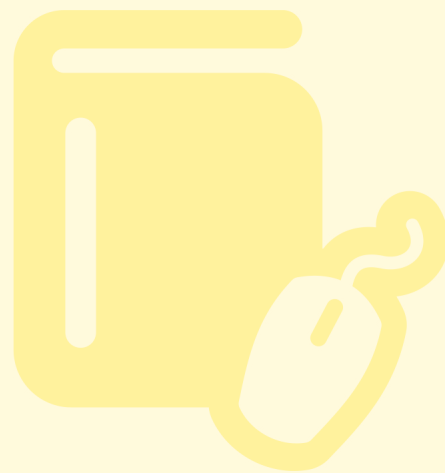
LORENZI, Fabiana; MATTOS, Patrícia Noll de; CARVALHO, Tanisi Pereira de. **Estruturas de dados.** São Paulo: Thomson Learning, 2007.

MIZRAHI, Victorine Viviane. **Treinamento em Linguagem C.** 2. ed. São Paulo: Pearson Prentice Hall, 2009.

PEREIRA, Silvio do Lago. **Estruturas de dados em C**: uma abordagem didática. São Paulo: Erica, 2016. ISBN Digital: 9788536517254.

SHILDT, Herbert. **C, Completo e Total**. Trad. Roberto Carlos Mayer. 3. ed. São Paulo: Makron Books, 1996.

SOFFNER, Renato. **Algoritmos e programação em linguagem C**. 1. ed. São Paulo: Saraiva, 2013.



NewtonVirtual