FELIPE CAVALARO

PROGRAMAÇÃO DE COMPUTADORES

USF | EAD
UNIVERSIDADE SÃO FRANCISCO

PROGRAMAÇÃO DE COMPUTADORES

Felipe Cavalaro

2022

CASA NOSSA SENHORA DA PAZ – AÇÃO SOCIAL FRANCISCANA, PROVÍNCIA FRANCISCANA DA IMACULADA CONCEIÇÃO DO BRASIL – ORDEM DOS FRADES MENORES

RESIDENTE

Frei Thiago Alexandre Hayakawa, OFM

DIRETOR GERAL

Jorge Apóstolos Siarcos

REITOR

Frei Gilberto Gonçalves Garcia, OFM

VICE-REITOR

Frei Thiago Alexandre Hayakawa, OFM

PRÓ-REITOR DE ADMINISTRAÇÃO E PLANEJAMENTO

Adriel de Moura Cabral

PRÓ-REITOR DE ENSINO, PESQUISA E EXTENSÃO

Dilnei Giseli Lorenzi

COORDENADOR DO NÚCLEO DE EDUCAÇÃO A DISTÂNCIA - NEAD

Renato Adriano Pezenti

GESTOR DO CENTRO DE SOLUÇÕES EDUCACIONAIS - CSE

Franklin Portela Correia

REVISÃO TÉCNICA

Fábio Andrijauskas

PROJETO GRÁFICO

Impulsa Comunicação

DIAGRAMADORES

Lucas Ichimaru Testa
Daniel Landucci

CAPA

Daniel Landucci

Avenida São Francisco de Assis, 218
CEP 12916-900 – Bragança Paulista/SP

O AUTOR

FELIPE CAVALARO

Mestre em Educação pelo Programa de Pós-Graduação Stricto Sensu da Universidade São Francisco. Possui pós-graduação em nível de Especialização (Lato sensu) em Docência Universitária pela FAE Centro Universitário (2013) e graduação em Engenharia de Computação pela Universidade São Francisco (2008). Atualmente é professor do curso de Engenharia de Computação da Universidade São Francisco Campus Itatiba, Bragança Paulista e Campinas. Coordenador da Câmara Temática “Tecnologia e Inovação” do Comitê Interno de Educação, Tecnologia e Inovação. Membro do Grupo Análise de Linguagem, Trabalho Educacional e suas Relações: Letramento, Gêneros Textuais e Ensino (ALTER-LEGE) e Grupo Relações de ensino e Trabalho docente.

O REVISOR

FÁBIO ANDRIJAUSKAS

Possui graduação em Ciência da computação pela Universidade São Francisco (2007). Mestre pela Faculdade de Tecnologia da Unicamp, na área de processamento de alto desempenho e processamento de imagens astronômica e Doutorando na Faculdade de Tecnologia da Unicamp na área de computação de alto desempenho e dinâmica molecular. Professor da Universidade São Francisco campi Campinas e Itatiba na área de Computação para os cursos de Computação ministrando as disciplinas de Computação Gráfica, Sistemas Multimídia, Sistemas Operacionais, Programação Orientada a Objeto, Algoritmos e Programação de Computadores, dentre outras. Tem experiência na área de Ciência da Computação, com ênfase em Computação de Alto Desempenho, atuando principalmente nos seguintes temas: astronomia, processamento de imagem, processamento de alto desempenho e reconhecimento de padrão.

SUMÁRIO

UNIDADE 1

ARRAYS E MODULARIZAÇÃO

INTRODUÇÃO

“Quando eu estava na escola, o computador era uma coisa muito assustadora. As pessoas falavam em desafiar aquela máquina do mal que estava sempre fazendo contas que não pareciam corretas. E ninguém pensou naquilo como uma ferramenta poderosa.” Disse Bill Gates em uma palestra na Universidade de Illinois, nos Estados Unidos, em 2004 (SARTORI *et al*., 2016, p. 1). Nos tempos atuais, o computador já pode ser visto como uma ferramenta poderosa, que traz benefícios para diversas áreas e setores, em algumas situações, é até essencial.

Mas não só o computador (equipamento físico) evoluiu, mas sua programação também vem evoluindo, propiciando melhor interação e recursos para o usuário. Importante destacar que mesmo diante tantas mudanças a base da programação não sofreu alteração, com isso, para apresentar conceitos de *arrays* e modularização será adotada a linguagem C, mas, para isso, precisa-se recordar comandos que darão suporte para iniciar os novos temas.

Importante lembrar que para programar na linguagem C é necessário utilizar um compilador e uma ambiente de desenvolvimento como: Code::blocks: (disponível em http://www.codeblocks.org/downloads/binaries/), Dev-C++ (disponível em https://sourceforge.net/projects/orwelldevcpp/) ou ferramentas online como OnlineGDB (disponível em: https://www.onlinegdb.com/online_c_compiler).

1. REVISÃO DOS CONCEITOS DE LÓGICA DE PROGRAMAÇÃO

Criada em 1972, a linguagem C é antiga, mas robusta. O que comprova isso é que seu uso continua até os tempos atuais. Ela tem algumas características que podemos observar no Código 01 abaixo:

Código 01. Exemplo Alô mundo

```
01  #include <stdio.h>
02  void main(){
03          printf("Alo mundo\n");
04  }
```

Fonte: elaborado pelo autor.

No Código 1 é um clássico da programação para o primeiro programa, no qual temos a declaração da biblioteca na linha 01 por meio do #include, em seguida, na linha 02 está

o programa principal chamado de main, na linha 03, o comando printf está mostrando na tela o texto “Alo mundo” e pulando uma linha. Repare que nessa linguagem os blocos de códigos são separados por chaves e os comandos são finalizados com ponto e vírgula (;). A linguagem C também faz distinção entre letras maiúsculas e minúsculas, ou seja, se o comando printf estiver com a letra inicial maiúscula o seu código não funcionará. O código apresenta elementos simples, nem ao menos faz o uso de variáveis.

1.1. COMANDOS DE ENTRADA E SAÍDA, VARIÁVEIS E CONSTANTES

Os comandos de saída e entrada é o que possibilita a interação do usuário com o programa, permitindo que ele informe um valor por meio do teclado ou apresentar uma informação na tela do computador. Já as variáveis permitem armazenar essas informações coletadas na memória RAM do computador e alterá-la assim que necessário, diferentemente da constante que também possui um valor, mas não sofre alteração (como boa prática utiliza todas as letras em maiúsculo no nome da constante). No Código 02 apresenta alguns exemplos dos diversos tipos de variáveis existentes na linguagem C o texto depois de duas barras (//) são comentários, texto que não fazem parte da programação que ajudam a explicar o código.

Código 02. Exemplos de declaração de variáveis

```
01 #include<stdio.h>
02 #define PI 3.14 //declara uma constante
03 void main(){
04   char var_c;//declara uma variável char
05   int var_i; //declara uma variável int
06   float var_f; //declara uma variável float
07   double var_d; //declara uma variável double
08   scanf("%c",&var_c); //lê uma variável char
09   scanf("%i",&var_i); //lê uma variável int
10   scanf("%f",&var_f); //lê uma variável float
11   scanf("%lf",&var_d); //lê uma variável double
12   printf("%c\n",var_c); //mostra uma variável char e pula uma linha
13   printf("%i\n",var_i); //mostra uma variável int e pula uma linha
14   printf("%f\n",var_f); //mostra uma variável float e pula uma linha
15  printf("%lf\n",var_d); //mostra uma variável double e pula uma linha
16   printf("%f\n",PI); //mostra uma constante e pula uma linha
17 }
```

Fonte: elaborado pelo autor.

1

No Código 02, é apresentado na linha 02 um exemplo de como criar uma constante, neste caso, chamado PI com seu valor 3.14, repare que o separador de casas decimais na linguagem C é o ponto (.), diferente do sistema brasileiro que utiliza a vírgula (,). Logo abaixo, na linha 3 é apresentado como declarar uma variável do tipo caractere chamada var_c, isso se repete nas 4, 5 e 6, exemplificando os tipos *int, float e double*. Entre as linhas 8 e 11, é demonstrado como coletar os dados e armazenar em variáveis char, *int, float e double*, respectivamente. Entre as linhas 12 e 15, é apresenta como mostrar os dados armazenados nas variáveis *char, int, float e double,* respectivamente, em seguida, pulando para a linha seguinte da tela por meio do "\n". Por fim, a linha 16 exemplifica como mostrar o valor de uma constante.

Após coletar os dados e armazená-los em variáveis, é possível fazer operações e/ou expressões aritméticas, relacionais e lógicas com essas informações para as diversas finalidades.

1.2. OPERADORES E EXPRESSÕES

Operadores e expressões são vastamente utilizados em qualquer linguagem, pois as operações necessárias para resolver simples algoritmos. Um exemplo disso são os operadores aritméticos e de atribuição apresentados no Código 03.

Código 03. Exemplos de operadores aritméticos e de atribuição

```
01  #include<stdio.h>
02  void main(){
03    int a=2,b;
04    b=1;
05    printf("%i\n",a+b); //2+1=3
06    printf("%i\n",a-b); //2-1=1
07    printf("%i\n",a*b); //2*1=2
08    printf("%i\n",a/b); //2/1=2
09    printf("%i\n",a%b); //2%1=0
10  }
```

Fonte: elaborado pelo autor.

No Código 03, como podemos observar na linha 3 e 4, o operador de atribuição é o símbolo igual (=), que pode ser utilizado na declaração da variável ou após ela ser declarada. Entre as linhas 5 e 8 são apresentados os operadores de soma, subtração, multiplicação e divisão. Por fim, é demonstrado o operador % que coleta o resto da divisão (só funciona com números inteiros). Os operadores listados são os tradicionais, mas haverá outros operadores explorados ainda neste texto.

1.3. COMANDOS CONDICIONAIS

Os comandos condicionais são utilizados muitas vezes com os operadores relacionais como == (igualdade), != (diferente), >= (maior ou igual), <= (maior ou igual), > (maior)

ou < (menor). Algumas vezes, utiliza-se os operadores lógicos como && (E), II (OU) e o inversor ! (NÃO). O Código 4 abaixo, exemplifica alguns operadores relacionais e lógico utilizando os comandos condicionais.

Código 04. Exemplos de comandos condicionais com operadores relacionais e lógico

```
01  #include<stdio.h>
02  void main(){
03    int a;
04    printf("Digite um numero par de 0 a 10: ");
05    scanf("%i",&a);
06    if(a>=0 && a<=10){
07          if(a%2==0) printf("Numero valido!\n");
08          else printf("Numero impar!\n");
09    }else{
10         printf("Intervalo invalido!\n");
11    }
12  }
```

Fonte: elaborado pelo autor.

No código 04, na linha 4 é mostrada a exigência de digitar um número par em um intervalo de 0 a 10, em seguida, na linha 5 é coletado o número e armazenada na variável a declarada na linha 3. Depois, na linha 6 há um comando condicional que possui duas comparações realizadas pelos operadores <= e >= (repare que o igual sempre é o segundo símbolo), essas duas comparações terão como resposta verdadeira ou falsa, que será tratada pelo operador E (&&), com isso, se a variável for maior ou igual a zero e também menor ou igual a 10 será executado o bloco de comando entre as linhas 7 e 8, caso contrário, será executado a linha 10. Observe que essa condicional é encadeada, ou seja, tem um comando condicional dentro do outro. Sendo assim, verificado se o número é par, em outras palavras, conferindo se o resto da divisão por dois é igual a zero, se isso ocorrer é mostrado "Número valido!", caso contrário, mostra-se "Número ímpar!". Atente que nas linhas 7 e 8 o comando condicional não foi utilizado as chaves, pois só possui um comando para ser executado, portanto, o else da linha 9 também poderia ter removido as chaves. Neste exemplo, não foi apresentado a situação em que se tem somente o if, sem o else, mas isso pode ocorrer. Entretanto, nunca poderá ter um else sem um if.

Lembrando que o comando condicional seleciona uma porção de instruções para serem executadas de acordo com a condição imposta, mas não permite a execução diversas vezes do mesmo código, isso ocorre nos comandos de repetição.

1.4. COMANDOS DE REPETIÇÃO

Os comandos de repetição na linguagem C são for, while e do-while, que são apresentados no Código 05, que mostra uma contagem de 1 a 100 (executa 100 vezes).

1

Código 05. Exemplos de comandos de repetição

```
01 #include<stdio.h>
02 void main(){
03   int i;
04   for(i=1;i<=100;i++){
05     printf("%i\n",i);
06   }
07 }
08
09
```

```
#include<stdio.h>
void main(){
  int i;
  i=1;
  while(i<=100){
    printf("%i\n",i);
     i++;
  }
}
```

```
#include<stdio.h>
void main(){
  int i;
  i=1;
  do{
    printf("%i\n",i);
    i++;
  }while(i<=100);
}
```

Fonte: elaborado pelo autor.

Repare que o comando for é condensado a uma linha (linha 4) informando o valor que a variável contadora (i) se inicia (em 1), em seguida, a condição de repetição (que se limita a se repetir quando o valor da variável contadora for menor ou igual a 100) e a iteração, neste caso o operador ++ soma o valor 1 ao valor atual da variável contadora e atribui o valor à ela mesma. Já no comando *while*, o valor inicial da variável contadora é inserido na linha 4, a condicional está na linha 5 junto com o comando *while* e a iteração já ocorre na linha 7, sendo a última linha de instrução antes de encerrar o bloco de comando. Por fim, o comando do-while ocorre uma inversão do comando *while*, mudando o *while* e a condição para a linha 8 e na linha 5, seu lugar de origem, passa a ter o comando do. Lembrando que os três códigos fazem a mesma coisa, entretanto foram montados em comandos diferentes de repetição.

SAIBA MAIS

Mesmo os comandos de repetição tendo a mesma função, eles podem ser selecionados perante suas características para se adequar a determinada situação, pois o comando for é utilizado em situações contáveis (quando é possível descobrir quantas vezes vai se repetir), o comando *while* e *do-while* para situações incontáveis ou contáveis, sendo que o comando *while* é feita uma verificação antes do ciclo de repetição e o *do-while* é verificado após a execução de pelo menos uma vez o ciclo de repetição. Portanto, uma situação pode ser mais adequada ser efetuada em um comando de repetição que nos outros.

2. *ARRAY* UNIDIMENSIONAL

Ao aprender a declarar uma variável na linguagem C, descobre-se como reservar um espaço em memória RAM do computador que pode armazenar um valor do tipo que se declarou. Entretanto, na programação, há situações que é necessário armazenar um volume maior de informações, por exemplo, ao armazenar a nota dos alunos de uma sala com aproximadamente 100 alunos, você terá que criar esse número de variáveis, inserindo em cada variável uma nota, essa tarefa se torna trabalhosa. Com isso, para se declarar mais de um espaço rapidamente na programação criou-se os *arrays*, também conhecidos como vetores e matrizes.

“Um vetor é uma estrutura que armazena vários dados de mesmo tipo, ao contrário das variáveis comuns, que só podem armazenar um valor de cada vez. Em programação, é das estruturas mais simples” (SOFFNER, 2013, p. 88).

Um vetor, ou também chamado de *array* unidimensional, possui capacidade de armazenar mais de um valor do tipo informado em sua declaração, essa quantidade é informada em sua declaração, como podemos ver no exemplo apresentado no Código 6. Atente-se que para percorrer esses espaços é utilizado um índice que se inicia do valor erro (0).

Código 06. Exemplo de declaração de um vetor

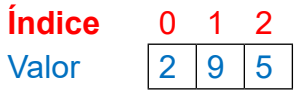
```
01  #include<stdio.h>
02  void main(){
03    int v[3]; //declaração de um vetor com 3 posições
04    v[0]=2; //atribuição na posição 0 (primeira posição)
05    v[1]=v[0]+7; //atribuição com operação de soma
06    scanf("%i",&v[2]); //entrada de dados e armazena no vetor
07    printf("posicao 0 valor %i",v[0]); //mostra primeira posição
08    printf("posicao 1 valor %i",v[1]); //mostra segunda posição
09    printf("posicao 2 valor %i",v[2]); //mostra terceira posição
10  }
```

Fonte: elaborado pelo autor.

No código 6, é declarado um vetor com o nome v que armazena três valores do tipo int, observe que a quantidade é informada entre colchetes após o nome do vetor. Na linha 4 é um exemplo de como fazer uma atribuição do valor 2 na posição 0 (primeira posição) do vetor. Na linha 5 também está fazendo uma atribuição do resultado da soma do valor 7 com o valor contido na posição 0 do vetor, portanto será atribuído o valor 9 (2+7) na segunda posição do vetor. Na linha 6 está demonstrando como usar o comando scanf para armazenar na posição 2 do vetor uma informação digitada pelo usuário. Por fim, as linhas 7 a 9, mostram na tela os valores das três posições do vetor.

Sabendo que a primeira posição em um vetor sempre será o índice zero e neste caso, a última posição deste vetor tem índice 2, pode-se observar que a última posição sempre terá o índice equivalente a quantidade de elementos menos um, como pode-se verificar na Figura 1 apresentada abaixo se o usuário se digitasse o valor 5 no comando *scanf*.

Figura 01. Ilustração do vetor preenchido



Fonte: elaborada pelo autor.

É possível declarar o vetor e fazer a atribuição dos valores em um única linha, basta colocar o símbolo de atribuição (=), em seguida, colocar dentro das chaves os valores entre parênteses, como por exemplo, int v[3]={2,9,5}; para declarar o vetor ilustrado com os seus respectivos valores.

1

Como dito anteriormente, o vetor é utilizado para facilitar a manipulação de um volume maior de informações, por isso, é comum o uso dessa estrutura com comandos de repetição para percorrer cada posição, seja para inserir ou para coletar os dados, como é apresentado no Código 07 que faz a coleta de 10 alturas, armazenando cada altura em uma posição do vetor, em seguida, mostra essas informações coletadas na tela.

Código 07. Exemplo de vetor com estrutura de repetição

```
01  #include<stdio.h>
02  void main(){
03    float alturas[10];
04    int i;
05    for(i=0;i<10;i++){//coleta as alturas
06         printf("Digite a %ia altura: ",i+1);
07         scanf("%f",&alturas[i]);
08    }
09    for(i=0;i<10;i++){//mostra as alturas
10         printf("%.1f\t",alturas[i]);
11    }
12  }
```

Fonte: elaborado pelo autor.

No Código 07, foi declarado um vetor com 10 posições do tipo float para armazenar as alturas. Na linha 4 foi declarado a variável i que representará o índice do vetor (posição). Entre as linhas 5 e 8 está o primeiro comando de repetição que inicia de zero (primeira posição do vetor) e executa enquanto o valor da variável i for menor que 10 (vai executar até 9) e faz a iteração de um em um por meio do comando i++, lembrando que esse comando faz com que a variável i seja adicionado o valor atual mais um a cada vez que é executado, dessa maneira, os comandos das linhas 6 e 7 se repetirão 10 vezes, sendo solicitado a primeira altura e armazenado na primeira posição do vetor (índice 0), em seguida, solicitado a segunda altura e armazenado na segunda posição, assim por diante. Observe que o índice que era inserido entre colchetes no Código 6, no Código 7 foi substituído pela variável i.

Por fim, nas linhas 9 a 11 está o segundo comando de repetição que está mostrando os valores contidos no vetor. A variável i também se inicia de zero, faz a execução até que ela seja menor que dez, fazendo a interação de um em um, ou seja, a estrutura de repetição é a mesma da anterior, mudando o que é executado. Na linha 10 é mostrado cada valor armazenado no vetor (alturas), sendo que o "%f" foi substituído por "%.1f" para se limitar a uma casa decimal depois da vírgula (o padrão são 6 casas decimais), assim como o "\n" foi substituído pelo "\t", desse jeito, ao invés de pular uma linha após sua execução, é efetuado uma tabulação.

Abaixo, no Código 08, faz a coleta de 15 pesos, armazena em um vetor esses dados, em seguida, mostra esses dados juntamente com a média desses pesos.

Código 08. Exemplo de vetor com quantidade constante

```
01  #include<stdio.h>
02  #define QUANT 15
03  void main(){
04    float peso[QUANT],soma=0;
05    int i;
06    for(i=0;i<QUANT;i++){
07         printf("Digite o %io peso: ",i+1);
08         scanf("%f",&peso[i]);
09         soma=soma+peso[i];
10    }
11    printf("\nPesos= \n");
12    for(i=0;i<QUANT;i++){
13         printf("%.1f\t",peso[i]);
14         if(i%10==9) printf("\n");
15    }
16    printf("\nMedia= %.2f\n",soma/QUANT);
17  }
```

Fonte: elaborado pelo autor.

No Código 08, foi criado uma constante (QUANT) na linha 2, essa constante permite fazer uma rápida mudança no tamanho da quantidade de dados coletados e armazenados neste programa, tanto que, neste momento está o valor 15, mas se alterar somente essa linha para 150, o programa passa a fazer a mesma coisa para 150 pesos. Isso foi possível por ter trocado a quantidade de elemento do vetor em diversas partes do programa pela constante QUANT.

Em detalhe, na linha 4 é declarado um vetor do tipo float com o tamanho da constante QUANT (que neste momento tem valor 15) e uma variável soma que já é atribuído o valor zero (ela será uma variável acumuladora). Em seguida, é declarada a variável i que será utilizada como índice do vetor.

Entre as linhas 6 e 10 é realizado a coleta dos pesos e a cada coleta é armazenado em uma posição diferente do vetor, sendo que a cada coleta também é somando do valor atual da variável soma com o peso digitado pelo usuário (linha 9), com isso, fazendo que a variável tenha o acúmulo da soma dos pesos atuais.

Na linha 11 é mostrado um texto informando que será exibido os pesos, adiante, entre as linhas 12 a 15 é exibido na tela os pesos armazenados no vetor com uma casa decimal (linha 13) e realizado uma tabulação, já na linha 14 é verificado se o resto da divisão

1

por 10 do índice é igual 9, ou seja, verifica se é o décimo número para pular uma linha a cada 10 valores exibido. Por fim, na linha 16, é exibido na tela a média com duas casas decimais, ou seja, o resultado da divisão da variável soma (soma de todos os números digitados) com a quantidade de dados (QUANT).

Os vetores na programação são muito utilizados, mas há situações que acaba sendo dificultosa, principalmente quando precisa-se criar vários vetores, como por exemplo usar vetores para armazenar a temperatura das 27 Unidades Federativas (26 Estados mais o Distrito Federal) a cada hora do dia (24 horas), será necessário criar 27 vetores com 24 posições ou 24 vetores com 27 posições. Para resolver essa situação é possível uma matriz que será abordado a seguir.

Desafio

Faça um programa na linguagem C que colete 20 números inteiros, armazene cada valor em um vetor. Ao final, mostre todos os valores, em seguida, quais são os números pares e quantos são pares.

3. ARRAY BIDIMENSIONAL

Um array multidimensional é conhecimento também com o nome de matriz. “Uma matriz é um vetor multidimensional. Alguns autores consideram ambas as expressões a mesma coisa, e o que muda é apenas o número de dimensões” (SOFFNER, 2013, p. 95). Neste tópico será tratado somente de arrays bidimensionais, ou seja, será usado somente duas dimensões. Neste caso, usamos dois índices para identificar as posições de uma matriz, que referenciamos como linhas e colunas, igual uma tabela. Assim como no vetor, a matriz tem o início do seu índice no valor zero, no Código 9 pode-se notar como declarar, atribuir e obter uma informação em uma matriz.

Código 09. Exemplo de declaração de uma matriz

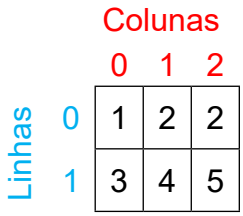
```
01 #include<stdio.h>
02 void main(){
03   int m[2][3]; //declara uma matriz de 2 linhas e 3 colunas
04   m[0][0]=1;
05   m[0][1]=m[0][2]=2;
06   m[1][0]=3;
07   m[1][1]=4;
08   scanf("%i",&m[1][2]);
09   printf("%i %i %i\n",m[0][0],m[0][1],m[0][2]);
10   printf("%i %i %i\n",m[1][0],m[1][1],m[1][2]);
11 }
```

Fonte: elaborado pelo autor.

No código 09, linha 3, é declarada uma matriz do tipo int que possui 2 linhas e 3 colunas (o primeiro número entre colchetes informa a quantidade de linhas e o segundo valor informa a quantidade de colunas). A seguir, na linha 4 é atribuído o valor 1 na primeira coluna e linha da matriz (linha 0 e coluna 0), na linha 5 é atribuído o valor dois em duas posições da matriz, na posição de linha 0 e colunas 1 e 2 (na linguagem C é possível fazer a atribuição dessa forma, mas nem todas as linguagens aceitam esse formato). Na linha 6 é atribuído o valor 3 na linha 1 e coluna 0 (segunda linha e primeira coluna), em seguida, é atribuído o valor 4 na mesma linha (linha 1), mas na coluna 1. Na linha 8 tem-se um exemplo de como fazer a coleta de dados digitado pelo usuário e armazenar em uma posição da matriz (neste exemplo, posição da linha 1 e coluna 1). Por fim, está exibindo na tela os seis valores da matriz, mas especificamente na linha 9 os valores da primeira linha da matriz e na linha 10 os valores da segunda linha da matriz.

Igualmente a um vetor, na matriz, o maior valor dos seus índices é a sua quantidade menos um, por iniciar em zero, a Figura 2 apresentada abaixo ilustra a matriz com 2 linhas e 3 colunas do Código 9, simulando que o usuário digita o valor 5 no comando scanf.

Figura 02. Ilustração da matriz preenchida



Fonte: elaborada pelo autor.

Resgatando o exemplo dado anteriormente, no qual se deseja armazenar a temperatura a cada hora do dia das 27 Unidades Federativas, é possível criar uma matriz com 27 linhas e 24 colunas, entretanto, será reduzido esse exemplo para não dificultar o teste do programa (muita informação para digitar), dessa forma, será escolhida somente a região Sudeste que temos 4 Estados (Espírito Santo, Minas Gerais, Rio de Janeiro e São Paulo) e somente as horas do dia (das 6h às 18h), portanto, será uma matriz que possui 4 linhas (os Estados) e 13 colunas (a quantidade de temperatura por Estado no período delimitado). No Código 10 abaixo mostramos como ficaria o armazenamento e exibição dos dados utilizando comandos de repetição para percorrer as linhas e colunas da matriz gerada.

Código 10. Exemplo de matriz com estrutura de repetição

```
01  #include<stdio.h>
02  void main(){
03    float temp[4][13];
04    int i,j;
05    for(i=0;i<4;i++){ //coleta as temperaturas dos estados
06          for(j=0;j<13;j++){
07            printf("Digite a temp. do Estado %i e hora %i: ",i+1,j+6);
08            scanf("%f",&temp[i][j]);
09          }
10    }
11    for(i=0;i<4;i++){ //mostra as temperaturas dos estados
12          printf("\nEstado %i:\n",i+1);
13          for(j=0;j<13;j++){
14            printf("%ih=%.1f\t",j+6,temp[i][j]);
15          }
16    }
17  }
```

Fonte: elaborado pelo autor.

No código 10, linha 3, está sendo declarado uma matriz com 4 linhas e 13 colunas do tipo *float* para armazenar as temperaturas dos 4 Estados, em seguida, na linha 4 está sendo declarado duas variáveis (i e j) que servirão de índice para percorrer as linhas (i) e as colunas (j). Entre as linhas 5 e 10 está sendo feita a coleta e armazenamento dos dados na matriz, mais especificamente, a linha 5 é o comando de repetição que contém o índice das linhas, ou seja, inicia em 0, somando de um em um, quando o valor do índice for menor que 4 (quantidade de linhas), na linha 6 há outro comando de repetição dentro do anterior, que possui o índice das colunas, ou seja, inicia em 0, de um em um, até o valor do índice for menor que 13 (quantidade de colunas). Por ser um comando de repetição dentro do outro, será feito 4 x 13 (quantidade de dados contida na matriz), ou seja, 52 vezes os comandos das linhas 7 e 8 que mostra uma mensagem pedindo o Estado e a hora a qual se deseja a temperatura, em seguida, armazena esse dado na posição correspondente da matriz.

Entre as linhas 11 e 16 está mostrando os dados para cada Estado, mais detalhadamente, na linha 11 está usando o comando de repetição novamente com o índice para as linhas (i), em seguida, mostrando o Estado e o número correspondente a ele (1 a 4). Seguindo, na linha 13, há outro comando de repetição que representa o índice das colunas (j) fazendo que mostre as 13 temperaturas dos 4 Estados, onde será mostrado a hora do dia e a temperatura correspondente.

Repare nesse código que cada Estado foi representado por uma linha da matriz (0 a 3) e cada hora do dia foi representada por uma coluna (0 a 12). Tanto na linha 7 como na linha 14 o valor da variável j é somado ao valor 6, assim exibindo a hora da coleta da temperatura, assim como nas linhas 7 e 12, o valor da variável i é somado ao valor 1 que representou o número de um Estado (1 a 4), se considerar em ordem alfabética, será 1 para Espírito Santo, 2 para Minas Gerais, 3 para Rio de Janeiro e 4 para São Paulo. Importante destacar também as linhas 8 e 14 que armazenam e coletam informações da matriz fazendo uso de posições das linhas e colunas entre colchetes representadas pelas variáveis i (linhas) e j (colunas).

O uso de matriz para armazenar as informações necessariamente precisa-se utilizar um comando de repetição dentro do outro, em algumas situações um comando de repetição é suficiente, como é o caso do Código 11 que coleta a altura, peso e calcula o IMC (Índice de Massa Corporal) de 10 pessoas, em seguida, mostra todos esses dados que foram armazenados em uma matriz e mostra também qual o valor do maior IMC.

Código 11. Exemplo de matriz

```
01  #include<stdio.h>
02  #define LIN 10
03  #define COL 3
04  void main(){
05    float dados[LIN]][COL],m=0;
06    int i;
07    for(i=0;i<LIN;i++){
08      printf("Digite o peso da pessoa %i: ",i+1);
09      scanf("%f",&dados[i][0]);
10      printf("Digite a altura da pessoa %i: ",i+1);
11      scanf("%f",&dados[i][1]);
12      dados[i][2]=dados[i][0]/(dados[i][1]*dados[i][1]);
13      if(m<dados[i][2]) m=dados[i][2];
14    }
15    printf("\nPessoa    Peso\t Alt.\t IMC\n");
16    for(i=0;i<LIN;i++){
17         printf("Pessoa %i: %.2f\t %.2f\t %.2f\n",
18                i+1,dados[i][0],dados[i][1],dados[i][2]);
19    }
20    printf("Maior IMC: %.2f\n",m);
21  }
```

Fonte: elaborado pelo autor.

No código 11, linha 2, criou-se uma constante chamada LIN que possui o valor 10, que neste caso representa a quantidade de pessoas (linhas) que serão coletadas e armazenadas na matriz. Em seguida, na linha 3, criou-se mais uma constante que se chama COL com o valor 3, que representará a quantidade de colunas (para armazenar o peso, altura e IMC).

Na linha 5 é criada uma matriz com 10 linhas (valor de LIN) e 3 colunas (valor de COL) do tipo *float*, assim como é declarada uma variável m e atribuído nela o valor zero. Na linha 6 é criada uma variável que será utilizada como índice das linhas da matriz.

Entre as linhas 7 e 14 são coletados dos dados das pessoas, calculado o IMC e verificado o maior IMC. De uma forma mais detalhada, na linha 8 é solicitado para digitar o peso da pessoa, na linha 9 é coletado esse valor digitado e armazenado na coluna 0 e linha da pessoa correspondente (cada linha possui o dado de cada pessoa), em seguida, na linha 10 é solicitado a altura e na linha 11 é coletado essa informação e armazenada na coluna 1 e linha correspondente a pessoa. Na linha 12 é efetuado o cálculo do IMC (IMC=Peso/(Alt*Alt)) e o resultado é armazenado na coluna 2 da linha correspondente a pessoa. Na linha 13 é verificado se o valor contido na variável m é menor que o valor calculado para aquela pessoa, caso seja, esse valor é copiado para a variável m e passa a ser o maior valor do IMC.

Na linha 15 é mostrado um cabeçalho para exigir as informações das pessoas em formato de uma tabela (linhas e colunas) na tela. Entre as linhas 16 e 19, são exibidas essas informações, na linha 16 contem o comando de repetição que permite percorrer todas as linhas (informações de todas as pessoas), em seguida, nas linhas 17 e 18, é exibido na tela qual é a pessoa (1 a 10), em seguida, seu peso, altura e o IMC, todos os valores estão formatados para exibirem com 2 casas decimais. Observe também que as linhas 17 e 18 poderia estar em somente uma linha, pois representa um único comando *printf* que exibe o índice da pessoa, peso (coluna 0), altura (coluna 1) e IMC (coluna 2), mas foi dividida para não ficar tão extensa. Por fim, na linha 20 é exibido o maior valor de IMC contido na variável M, também no formato de duas casas decimais.

Desafio

Faça um programa na linguagem C que colete a nota de 3 avaliações de 10 alunos e armazene em uma matriz 10 x 3. Depois, mostre todas as notas na tela, em seguida, a maior e a menor de cada uma das três avaliações.

4. MODULAÇÃO

Na programação existem diversos problemas complexos que já podemos solucionar, mas para que isso aconteça usou-se o princípio de dividir esses problemas complexos em problemas menores que poderiam ser solucionados, após isso, chegar à solução de problemas mais complexos.

Sem ao mesmo saber, já foi realizado esse conceito quando estávamos utilizando os comandos printf, scanf, etc. esses comandos são trechos de códigos que são chamados de funções (módulos, subprogramas ou sub-rotinas) que permitem o seu reuso. Funções são trechos (ou blocos) de código que diminuem a complexidade de um

programa, ou evitam a repetição excessiva de determinada parte do aplicativo. A isso chamamos de modularização (SOFFNER, 2013, p. 100).

Claro que esses comandos são funções que são fornecidos pela biblioteca padrão da linguagem C (stdio.h), assim como existem outras que não se trabalhou até o momento como a biblioteca math.h que possui funções matemática como raiz quadrada (sqrt), potência (pow), seno (sin), entre outras. Observe o Tabela 1 abaixo que mostra algumas bibliotecas da linguagem C e suas descrições.

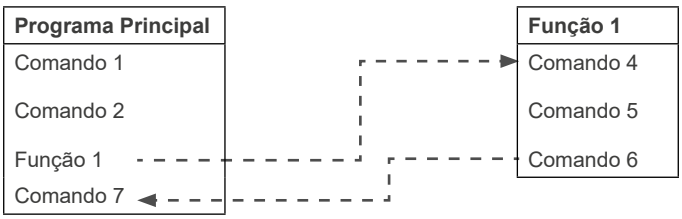
Tabela 01. Bibliotecas da linguagem C

BIBLIOTECA	DESCRIÇÃO
stdio.h	Essa biblioteca é a mais utilizada na programação em linguagem C, pois é a padrão, na qual estão embutidas as funções printf(), puts(), gets(), scanf(), entre outras.
math.h	Possui as funções matemáticas usadas pela linguagem. Encontram-se funções trigonométricas, hiperbólicas, exponenciais, logarítmicas, entre outras.
string.h	Esta possui as rotinas de tratamento de strings e caracteres, na qual se encontram as funções strcmp() e strcpy(), entre outras
time.h	Essa biblioteca possui as funções de manipulação de data e hora do sistema.
stdlib.h	Possui um conjunto de funções que não se enquadra em outras categorias. As funções dessa biblioteca são conhecidas como “funções miscelâneas”
locale.h	Utilizada para uso de constantes de identificação de idiomas e países.
conio.h	Manipula caracteres na tela, especificando cor e fundo.

Fonte: adaptada de Manzano (2002, p. 145).

Uma função é um trecho identificado de um programa, com início e fim bem definidos e que pode ser acionado de outros pontos desse programa. Quando uma função é chamada o fluxo de execução é desviado para o primeiro comando da função e prossegue a partir desse ponto até que o seu fim seja atingido, ou um comando explicito de retorno seja executado, quando então o fluxo de execução retorna ao ponto imediatamente posterior ao ponto de chamada. A Figura 3 ilustra esse desvio de fluxo do programa para a função e seu retorno.

Figura 03. Desvio do Fluxo de Execução



Fonte: elaborado pelo autor.

Como podemos observar na Figura 3, o programa principal (main) tem uma sequência de comandos executados pelo programa durante o seu processamento, normalmente sequencial, podendo ser desviados por chamadas de funções. Neste caso, o programa

1

principal executa os comandos 1 e 2 em sequência, em seguida, faz um desvio de fluxo (devido a chamada da Função 1 contida após o comando 2), assim, o programa passa a executar os comandos 4, 5 e 6 em sequência, como o comando 6 é último comando a ser executado na Função 1, ele retorna no programa principal e continua a sequência, executando o comando 7. Esse desvio de fluxo pode ocorrer em mais de uma vez no programa principal e também pode acontecer um desvio em uma função que chama outra função (ou até ela mesma).

Alguns autores distinguem a modularização em dois tipos: funções que são caracterizadas por retornarem um valor como resultado do seu processamento, e procedimentos, caracterizados por não produzirem valor de retorno.

As funções são facilmente exemplificadas com funções aritméticas que realizam cálculos como seno, cosseno, entre outros resultados obtidos. Já os procedimentos são as rotinas como por exemplo, para ler registros de um arquivo e armazená-los em alguma estrutura da memória, para ordenar valores armazenados em memória, entre outros. Entretanto, a linguagem C utiliza um único mecanismo para implementar tanto as funções quanto os procedimentos, sendo estes especificados como funções que possuem void como o tipo do valor de retorno.

Na linguagem C, para se criar uma função utiliza-se o seguinte formato:

```
<tipo_do_retorno> <nome_função>(<parâmetros>){
      <comandos>
      <retorno>
}
```

Em que cada item tem as seguintes definições:

`<tipo_de_retorno>` é o tipo da informação que deseja retornar como resultado (int, float, char, etc), caso seja um procedimento (não tenha retorno), informe void;

`<nome_da_função>` é o nome que chamará a função (comando), respeitando as regras para criar nomes para variáveis (pois ambos são identificadores);

`<parâmetros>` também conhecido como argumentos, são os elementos que precisam ser informados para executar aquele processo, caso não necessite, basta deixar sem essa informação;

`<comandos>` os comandos que deverão ser executados para efetuar aquele processo.

`<retorno>` resultado esperado pela função, caso seja um procedimento (o tipo do retorno está void) não é necessário a sua inserção.

O Código 12 abaixo, apresenta um procedimento que apresenta uma saudação.

Código 12. Exemplo de procedimento

```
01  #include<stdio.h>
02  void msg_oi(){
03          printf("Oie!\n");
04  }
05  void main(){
06          printf("Inicio\n");
07          msg_oi();
08          printf("Fim\n");
09  }
```

Fonte: elaborado pelo autor.

No código 12, linha 2, é criado um procedimento chamado msg_oi sem parâmetro (pois não possui nenhuma informação entre parênteses), em seguida, na linha 3 é solicitado para mostrar na tela o texto "Oie!" e pula-se uma linha. Entre as linhas 2 e 4, são a declaração do procedimento, ou seja, informar que agora existe um comando msg_oi e ele fará essas ações. Já na linha 6, dentro do programa principal, mostra-se na tela o texto "Início" e pula-se uma linha, em seguida, na linha 7 é realizada a chamada ao procedimento, fazendo o desvio de fluxo, mostrando o texto "Oie" e pulando uma linha, adiante, o fluxo volta para o programa principal e mostra na tela o texto "Fim" e pulando uma linha. Portanto, terá como saída os seguintes dados:

```
Inicio

Oie

Fim
```

Observe que o procedimento foi inserido antes do programa principal (main), pois dessa forma o programa principal reconhece a declaração, mas esse problema pode ser resolvido inserindo (declarando) o cabeçalho do procedimento antes do programa principal (main) e depois definir o que será feito (ações) no procedimento. Isso está ocorrendo no Código 13, em que o procedimento é declarado antes e definido após o programa principal.

Código 13. Exemplo de procedimento com declaração de cabeçalho

```
01  #include<stdio.h>
02  void imprime_idade(int idade);
03  void main(){
04          printf("Inicio\n");
05          imprime_idade(20);
06          printf("Fim\n");
07  }
08  void imprime_idade(int idade){
09          printf("Voce tem %i anos.\n",idade);
10          if(idade>=18) printf("Voce e maior de idade.\n");
11          else printf("Voce e menor de idade.\n");
12  }
```

Fonte: elaborado pelo autor.

No código 13, linha 2, está sendo declarado o cabeçalho do procedimento sem sua definição, com isso, a definição do procedimento pode estar após o programa principal, em alguns compiladores, se a linha 2 não existisse o programa iria acusar alguma advertência ou até erro antes de sua execução.

No programa principal, entre a linha 3 e 7 está mostrando assim como o Código 12, o texto "Inicio" e "Fim" antes e depois da chamada do procedimento. Mas nesse procedimento chamado na linha 5 é passado o valor 20 por parâmetro (argumento).

Na definição do procedimento, entre as linhas 8 e 12, está recebendo como parâmetro a idade que é do tipo int, em seguida, na linha 09, é mostrado o texto junto com o valor do parâmetro, exibindo o texto "Voce tem 20 anos.", por fim, se o valor da idade for maior ou igual a 18 será mostra do texto "Voce e maior de idade", caso contrário, exibirá "Voce e menor de idade". Com isso, a saída da tela para esse código será:

```
Inicio
Voce tem 20 anos.
Voce e maior de idade.
Fim
```

Observe que o procedimento pode executar mais de um comando, neste caso foi realizado também um comando condicional, que informava se a idade informada é de uma pessoa de maior ou menor de idade.

Desafio

Faça um procedimento na linguagem C que receba como parâmetro dois números reais e mostre na tela o maior valor entre esses dois números. Mostre o seu funcionamento no programa principal.

Todos os exemplos de funções mostrado até o momento são sem retorno (procedimento), ou seja, no espaço do tipo de retorno está a palavra void. Já no Código 14 é mostrado um exemplo de função com retorno.

Código 14. Exemplo de função

```
01  #include<stdio.h>
02  float multiplicacao(float a, float b){
03          return a*b;
04  }
05  void main(){
06          float n1,n2,r;
07          printf("Digite um numero: ");
08          scanf("%f",&n1);
09          printf("Digite outro numero: ");
10          scanf("%f",&n2);
11          r=multiplicacao(n1,n2);
12          printf("Resultado= %f\n",r);
13  }
```

Fonte: elaborado pelo autor.

No código 14, na linha 2, é declarada uma função chamada multiplicacao que recebe como parâmetro dois valores do tipo float (a e b) e retorna um valor float, na linha 3 mostra-se o returno da função que é o resultado da multiplicação do parâmetro a com o parâmetro b.

Entre as linhas 5 e 13 está o programa principal, sendo que na linha 6 são declaradas três variáveis (n1, n2 e r), na linha 7 é solicitado para o usuário digitar um número, adiante, na linha 8 é coletado esse valor e armazenado na variável n1, já nas linhas 9 e 10 ocorrem situações similares, entretanto, está sendo armazenado o valor na variável n2. Na linha 11 é chamada a função multiplicacao passando como parâmetros n1 e n2, em que n1 será o valor do parâmetro a e n2 será o valor do parâmetro b (por conta da ordem que foi inserido). Após a chamada, ocorrerá o desvio de fluxo fazendo que a sequência do programa se desloque para a linha 2, executando a multiplicação na linha 3 e retornando o resultado que será atribuído na variável r, conforme mostrado na linha 11. Por fim, será mostrado o resultado contido na variável r.

Um outro exemplo de função com retorno é o Código 15, que recebe como parâmetro as três variáveis de uma equação de segundo grau (a, b e c) e retorna a quantidade de raízes que a equação possui. É verificado também se a equação é válida, pois ela só pode ser uma equa-

1

ção de segundo grau se a variável a for diferente de zero. Importante destacar que o exercício não quer saber o valor do delta (da fórmula de Bhaskara) como retorno, ou tampouco quais são os valores das raízes, mas sim a quantidade de raízes que a equação possui.

Código 15. Exemplo de função com equação de 2º grau

```
01  #include<stdio.h>
02  int quant_raizes(float a, float b, float c){
03        float d;
04        if(a==0) return -1;
05        else{
06              d=(b*b)-(4*a*c);
07              if(d<0) return 0;
08              else{
09                    if(d==0) return 1;
10                    else return 2;
11              }
12        }
13  }
14  void main(){
15        float a,b,c;
16        int r;
17        printf("Digite o valor de a: ");
18        scanf("%f",&a);
19        printf("Digite o valor de b: ");
20        scanf("%f",&b);
21        printf("Digite o valor de c: ");
22        scanf("%f",&c);
23        r=quant_raizes(a,b,c);
24        if(r==-1){
25          printf("O valor a nao pode ser zero!\n");
26        }else{
27          printf("A equacao tem %i raiz(es)!\n",r);
28        }
29  }
```

Fonte: elaborado pelo autor.

No código 15, na linha 2, é declarada a função quant_raizes que recebe como parâmetro três variáveis (a, b e c) e o retorno da função é int. Dentro da função, na linha 3 é declarada uma variável d do tipo float que irá armazenar o resultado do delta. Em seguida, na linha 4 é verificado se o valor da variável a é igual a zero, se for, será retornado o valor -1 (valor referenciado como equação inválida), caso contrário, é calculado o valor do delta (segundo a fórmula de Bhaskara) e armazenado na variável d. Na linha 7 é verificado se o valor da variável d é menor que zero (delta negativo não possui raiz real), se for será retornado o valor 0, representando que não possui raiz. Adiante, se o valor da variável d não for menor que zero, será verificado na linha 9 se o valor da variável d é igual a zero, se for será retornado o valor 1 (representando que a equação possui uma raiz real), caso contrário, resta somente a opção do valor da variável d ser maior que zero, com isso é retornado o valor 2 (representando que a equação possui duas raízes reais).

No programa principal entre as linhas 14 e 29, são declaradas 3 variáveis float (a, b e c) que armazenarão as variáveis da equação de segundo grau. Em seguida, na linha 16 é declarada uma variável r do tipo int para armazenar o retorno da função. Entre as linhas 17 e 22 são mostradas as instruções para o usuário e armazenadas os valores nas três variáveis. Já na linha 23, é realizada a chamada da função passando como parâmetro as variáveis que receberam os valores informados pelo usuário. A programa irá fazer o fluxo para a função acima explicada e retornará um valor inteiro (-1, 0, 1 ou 2) que será armazenado na variável r. Na linha 24 está sendo verificado se o valor de r é igual a -1 (a função retornou informando que a equação é inválida), se ocorrer, será mostrada a mensagem “O valor a não pode ser zero!”, caso contrário, será mostrada a mensagem “A equação tem X raiz(es)!”, na qual X será o valor que foi armazenado na variável r após a sua execução.

Para fixar o conceito de função, abaixo é apresentado o Código 16, que possui uma função que retorna o fatorial de um número natural.

Código 16. Exemplo de função com fatorial

```
01  #include<stdio.h>
02  int fatorial(int n){
03          int i,fat=1;
04          for(i=1;i<=n;i++)
05          fat=fat*i;
06          return fat;
07  }
08  void main(){
09          int num;
10          printf("Digite um numero: ");
11          scanf("%i",&num);
12          printf("Fatorial de %i e %i\n",num,fatorial(num));
13  }
```

Fonte: elaborado pelo autor.

1

No Código 16, linha 2 é declarada a função fatorial que recebe como parâmetro um número int chamado n, em seguida, na linha 3 são declaradas duas variáveis, a variável i que será uma contadora e a variável fat que irá acumular a multiplicação do valor atual da variável fat e o valor atual da variável i. Na linha 4 há um comando de repetição que se inicia do valor 1 (não pode iniciar em zero, pois estamos usando multiplicação, se multiplicar por zero irá zerar todas as outras multiplicações), de um em um, até o valor n passado por parâmetro, na linha 5, está sendo executado o cálculo da multiplicação do valor atual de fat com o valor atual da variável i e armazenando novamente na variável fat. Com isso, será calculado o valor do fatorial do número informado no parâmetro n e armazenado na variável fat, por isso, na linha 6 é realizado o retorno dessa variável.

No programa principal, é criada variável num do tipo inteira, pedido para o usuário digitar um número que armazenado nesta variável num, por fim, é chamada a função fatorial passando como parâmetro a variável num e seu retorno é mostrado com o valor da variável num.

Para ilustrar o funcionamento desta função, a Tabela 2 traz o teste de mesa quando o valor passado no parâmetro n for o valor 5.

Tabela 02. Teste de mesa para o n=5

LINHA EM EXECUÇÃO	n	fat	i	COMENTÁRIO
3	5	1		Recebe o parâmetro n=5 e atribui fat=1
4			1	Inicia a estrutura de repetição com i=1
5		1		fat=1*1
4			2	Incrementa o valor da variável i
5		2		fat=1*2
4			3	Incrementa o valor da variável i
5		6		fat=2*3
4			4	Incrementa o valor da variável i
5		24		fat=6*4
4			5	Incrementa o valor da variável i
5		120		fat=24*5
4			6	Valor da variável i > n, com isso termina o loop
6				Retorna o valor atual de fat (120)

Fonte: elaborado pelo autor.

Desafio

Faça uma função na linguagem C que passe por parâmetro as informações necessárias para se obter como retorno o resultado da taxa média de uma equação de 1º grau. Sabendo que fórmula obedece a seguinte regra: taxa_media=(y-y0)/(x-x0), onde y, y0, x e x0 são valores reais. Mostre o funcionamento da função no programa principal fazendo a passagem de variáveis por parâmetro.

CONCLUSÃO

Os arrays permitem fazer a manipulação de um volume maior de informações de forma mais rápida, os vetores (arrays unidimensionais) são mais adequados para dados que não há distinção dos dados, agora os dados que exigem uma maior organização podem-se utilizar as matrizes (arrays bidimensionais) para ter uma similaridade de uma planilha ou tabela, onde os dados podem ser organizados em linhas e colunas.

A modularização é um dos princípios da programação, permitindo reutilizar código já existem ou dividir problemas complexos em menores, para resolver os menores em seguida, resolver os mais complexos. Por isso, criar procedimentos e funções tem objetivo de organizar os códigos do programa, assim como trazer produtividade, no reaproveitamento de códigos já existentes em outras partes do programa ou em outros projetos já desenvolvidos.

1

REFERÊNCIAS BIBLIOGRÁFICAS

MANZANO, José Augusto Navarro G. **Estudo Dirigido de Linguagem C**. São Paulo: Saraiva, 2002.

SARTORI, Rodrigo *et a*l. Hacktivismo e suas origens na cybercultura. Materializando Conhecimentos. **Memorizando conhecimentos**, [on-line], v. 7, set. 2016. Disponível em: https://www.redeicm.org.br/revista/wp-content/uploads/sites/36/2019/06/a3_hacktivismo.pdf. Acesso em: 4 maio 2022.

SOFFNER, Renato. **Algoritmos e Programação em Linguagem C**. São Paulo: Saraiva, 2013.

UNIDADE 2

ESCOPO DE VARIÁVEIS E RECURSIVIDADE

INTRODUÇÃO

Em outra ocasião foi visto o que é função e algumas variações que elas podem ter como funções contidas em bibliotecas que já estão prontas, onde basta declarar a biblioteca no programa e usá-las ou também funções que podem ser criadas no programa que podem ou não ter retorno.

Nesta parte iremos continuar o assunto referente às funções falando sobre o escopo de variáveis que podem ser locais, globais e parâmetros formais. Será tratado também de funções recursivas que são bem utilizadas na computação em temas como busca de dados, estrutura de dados, ordenação de dados, entre outros. Será mostrado também um comparativo do método iterativo que utiliza estrutura de repetição como comando for, *while* e *do-while* em relação ao método recurso que utiliza funções recursivas para a solução dos problemas.

1. ESCOPO DE VARIÁVEIS

Como foi visto em outros momentos, uma variável possui um nome pelo qual é referida, representa uma localização específica da memória e possui um valor associado. Em um programa, sempre que utilizamos o nome de uma variável estamos fazendo uma referência à localização de memória associada à variável e, portanto, ao valor que está armazenado nessa localização (PINHEIRO, 2012, p. 87).

Para ser utilizada uma variável dever ser declarada, na linguagem C, é necessário informar também o seu tipo. As variáveis são declaradas basicamente em três locais: dentro das funções, no cabeçalho dos parâmetros da função ou fora de todas as funções. Com isso, o local da declaração define o escopo da variável, na qual sua visibilidade que pode ser classificado como globais, locais e parâmetros formais.

1.1 VARIÁVEIS LOCAIS

As variáveis locais são aquelas declaradas dentro de um bloco de código, sendo que os blocos de códigos na linguagem C se iniciam com abre-chaves ({) e termina com fecha-chaves (}). Sendo que elas podem ser referenciadas por comandos que estão dentro do bloco no qual as variáveis foram declaradas, no caso da linguagem C, as variáveis locais não são reconhecidas fora do seu próprio bloco de código. Essa variável só existe enquanto o bloco de código estiver em execução, em outras palavras, ela é

criada no momento da execução do bloco e destruída após o seu encerramento. Para exemplificar, observe o Código 1.

Código 01. Exemplo de variáveis locais

```
01  #include <stdio.h>
02  int func1(){
03          int x;
04          x=10;
05          return x;
06  }
07  int func2(){
08          int x;
09          x=20;
10          return x;
11  }
12  void main(){
13     printf("%i\n",func1());
14     printf("%i\n",func2());
15  }
```

Fonte: elaborado pelo autor.

No Código 1, a variável inteira x é declarada duas vezes (linhas 3 e 8), uma vez dentro da função func1 e outra vez na função func2. Por ser declarada dentro da função, a variável x da função func1 não tem nenhuma relação com a variável x da função func2. Isso acontece porque cada x é reconhecido apenas dentro da função (bloco de código) que foi declarada a variável.

A grande maioria dos programadores declararam as variáveis usadas logo após o cabeçalho da função (após o abre-chaves), entretanto, as variáveis locais podem ser declaradas dentro de qualquer bloco de código como podemos ver no Código 2.

Código 02. Exemplo de variável local no meio do código

```
01  #include <stdio.h>
02  void main(){
03     int x;
04     printf("Digite um numero inteiro: ");
05     scanf("%i",&x);
06     if(x>0){
07        float y;
08        printf("Digite um numero real: ");
09        scanf("%f",&y);
10        printf("X= %i e Y= %f\n",x,y);
11     }
12  }
```

Fonte: elaborado pelo autor.

No Código 2, a variável local y é criada na entrada do bloco de código do *if* (linha 7) e destruída na sua saída (linha 11). Com isso, a variável y só é reconhecida dentro do bloco de código do *if* (linhas 6 a 11), dessa forma, não podendo ser referenciada em qualquer outro lugar (por exemplo, após a linha 11).

A principal vantagem de declarar uma variável local em bloco condicional (igual ocorre na variável y no Código 2) é que a memória para ela só será alocada se necessário (quando entrar no bloco de código do if). Isso ocorre porque as variáveis locais não existirão se o bloco que elas estão não forem executados, sendo algo muito relevante quando está se programando em um dispositivo que tem um espaço reduzido da memória de trabalho. Outro benefício de declarar as variáveis dentro do bloco de código que as utilizam, é evitar a alteração dos valores delas acidentalmente.

Um ponto a ser lembrando, é que todas as variáveis locais são criadas e destruídas a cada entrada e saída do bloco em que elas são declaradas, com isso, seu conteúdo é perdido quando o bloco deixa de ser executado. Portanto, as variáveis locais não podem reter seus valores entre chamadas da função, a não ser que que seja informado algum parâmetro adicional como *static* para modificar seu armazenamento.

SAIBA MAIS

A linguagem C possui modificadores de armazenamento que precede sua declaração para indicar ao compilador de que forma as variáveis no programa devem ser armazenadas. Esses especificadores de armazenamento são: auto- define uma variável como local, mas é raramente utilizado, pois as variáveis já são definidas como locais por padrão; *extern*- define

para o compilador que uma variável foi declarada em outro arquivo de código; *static*- define como uma variável permanente, podendo ser usado em variáveis globais e locais; e *register*: define para o compilador que a variável deve ser, se possível, usada em um registrador da CPU ao invés de ser usada na memória principal, com isso, tornando o acesso mais rápido.

Pode-se iniciar a variável local com um valor conhecido (atribuir um valor no momento da declaração) e ele será atribuído toda vez que o bloco de código for executado, como mostra o exemplo no Código 3.

Código03. Exemplo de variável local com chamadas da função

```
01  #include <stdio.h>
02  void func(){
03          int x=2;
04          printf("%i\n",x);
05          x++;
06  }
07  void main(){
08     int i;
09     for(i=0;i<10;i++) func();
10  }
```

Fonte: elaborado pelo autor.

No Código 3, mostrará na tela dez (10) vezes o número 2, isso ocorrerá porque será chamada a função func dez (10) vezes, a cada vez que é chamada a função func, é declarada a variável local x e atribuído o valor 2, em seguida, mostra-se na tela o número, após é somado o valor 1 a variável x (ficando com 3), linha que não tem nenhum efeito nesse código, por fim, é destruída a variável x.

1.2 PARÂMETROS FORMAIS

As variáveis denominadas de parâmetros formais são as variáveis declaradas em uma função que usam argumentos (parâmetros), ou também conhecidas como entradas da função. Essas variáveis possuem o mesmo comportamento de uma variável local dentro de uma função como é mostrado no Código 4.

Código04. Exemplo de variáveis de parâmetros formais

```
01  #include <stdio.h>
02  int func(int a, float b){
03     if(a>0 && b>0) return 1;
04     else return 0;
05  }
06  void main(){
07     printf("%i\n",func(1,3.5));
08  }
```

Fonte: elaborado pelo autor.

No código 4, na linha 2, as declarações das variáveis a e b ocorrem dentro dos parênteses, logo após o nome da função func, nesta função, se o valor das variáveis a e b forem maiores que zero (0) será retornado um (1), caso contrário, retornará o valor zero (0).

Observe que na linguagem C é necessário informar o tipo da variável quando são declaradas como parâmetros formais e assim como qualquer outra variável local, será destruída quando a função encerrar sua execução.

1.3 VARIÁVEIS GLOBAIS

As variáveis globais, ao contrário das variáveis locais, são reconhecidas pelo programa inteiro e podem ser referenciadas por qualquer pedaço de código, com isso, guardam os valores durante toda a execução do programa. Para se declarar uma variável global basta declará-la fora de qualquer função, dessa forma, ela poderá ser acessada por qualquer expressão independente de qual bloco a expressão pertence, como é mostrado no Código 5.

Código 05. Exemplo de variável global

```
01  #include <stdio.h>
02  int cont;
03  void func1();
04  void func2();
05  void main(){
06     cont=100;
07     func1();
08  }
09  void func1(){
```

```
10      int temp;
11      temp=cont;
12      func2();
13      printf("Cont eh %i\n",cont);
14  }
15  void func2(){
16      int cont;
17      cont=10;
18  }
```

Fonte: elaborado pelo autor.

No Código 5, é declarada uma variável global cont na linha 2, observe que a variável global cont está sendo utilizadas em dois blocos de códigos que ela não foi declarada (*main* e func1). Os subprogramas func1 e func2 são procedimentos, mas também poderiam ser chamados de função, pois também são funções sem retorno. Esse código indica que no programa principal (*main*) é atribuído o valor 100 à variável cont e chamado o procedimento func1. No procedimento func1 é declarada uma variável local temp, é atribuído nela o valor contido na variável global cont e se chama o procedimento func2. No procedimento func2 é declarado uma variável local cont (mesmo nome da variável global), neste caso, cont na func2 será referenciada como a variável local, se comportando neste procedimento como não existisse a variável global cont. Dessa forma, o valor 10 foi atribuído à variável local cont e a variável global cont continua com o valor 100. Ao encerrar o procedimento func2 a variável local cont também será destruída e retornará para a linha 13 do procedimento func1 e será mostrado o texto "Cont eh 100" e pulado para a próxima linha, pois o valor da variável global cont continua com o valor 100 que foi atribuído no programa principal.

As variáveis globais são armazenadas pelo compilador C em uma região fixa da memória, sendo úteis quando o mesmo dado é usado em muitas funções em seu programa. Porém, evite usar variáveis globais desnecessárias, pois elas ocupam espaço de memória durante todo o tempo de execução do programa. Até porque, ao usar uma variável global onde uma variável local poderia ser utilizada torna-se a função menos geral, por ter que usar alguma coisa definida fora dela. Outro ponto que enfatiza o mínimo uso de variáveis globais, é que o uso elevado delas podem provocar erros desconhecidos e mudanças acidentais de valores.

DESAFIO

Analise o código na linguagem C abaixo:

```
01  #include <stdio.h>
02  int a;
03  void func1();
04  int func2(int b, int c);
05  void main(){
06     a=8;
07     func1();
08     a=func2(a,3);
09
10     printf("a=%i\n",a);
11  }
12  void func1(){
13     int a;
14     a=a+5;
15  }
16  int func2(int b, int c){
17     int d;
18     d=b*(c+a);
19     return d;
20  }
```

Após analisar o código, informe o escopo de cada variável declarada no código e mostre os valores assumidos por elas durante elas durante a execução do programa através do teste de mesa.

2. RECURSIVIDADE

Como já visto em outros momentos, uma função pode chamar qualquer outra função e essa função também pode chamar outra, e assim sucessivamente, por exemplo, se tivermos as funções f1, f2 e f3, a função f1 pode chamar a função f2 e a função f2 pode f3. Mas será possível a função f1 chamar a função f1 novamente?

Sim, as funções podem chamar a si mesmas. A função é denominada de recursiva quando um comando no corpo da função chama ela mesma. A recursão é o processo de definir algo em termos de si mesmo e também é chamado de definição circular (SCHILDT, 1996, p. 160).

A recursividade pode ser direta, quando a própria função a chama, e indireta, quando a função chama outra função que chama outra, em uma sequência que eventualmente resulta em uma chamada à função inicial.

Antes de partir para a aplicação da recursividade em programação, será mostrado como resolver um problema recursivamente. Imagine o seguinte cenário, tem-se um saco com três bolas e o objetivo é esvaziar o saco. Como devo fazer para esvaziar um saco com três bolas? Primeiro, verificamos se o saco está vazio, se não está vazio, tira-se uma bola, com isso, agora tem-se duas bolas no saco. Como devo fazer para esvaziar um saco com duas bolas? Primeiro, verificamos se o saco está vazio, se não está vazio, tira-se uma bola, com isso, agora tem-se uma bola no saco. Como devo fazer para esvaziar um saco com uma bola? Primeiro, verificamos se o saco está vazio, se não está vazio, tira-se uma bola, com isso, agora tem-se um saco sem bolas. Como devo fazer para esvaziar um saco com uma bola? Primeiro, verificamos se o saco está vazio. Como ele já está vazio, o processo termina. Veja que o processo é bastante repetitivo, então será que não é possível fazer uma regra para qualquer quantidade de bolas no saco?

Como devo fazer para esvaziar um saco com N bolas? Primeiro, verificamos se o saco está vazio. Se o vaso não está vazio, tiramos uma bola. Temos agora que esvaziar um vaso contendo N − 1 bolas. Perceba que a solução para esvaziar N bolas do saco foi definida em termos de si mesma, portanto, trata-se de uma função recursiva.

Para resolver esse problema criou-se um procedimento que vai se reduzindo o problema até chegar em uma solução. Para torna-se um pouco associativo o entendimento, observe a Figura 1 que apresenta as bonecas russas Matrioska. Essas bonecas são ocas e podem ser abertas para serem colocadas uma dentro da outra. Assim como na recursividade, a boneca russa tem dentro de si uma boneca menor, que tem dentro de si uma boneca ainda menor, assim por diante, até chegarmos a uma boneca pequena demais para ter outra dentro de si, que no caso da programação, é um problema que seja tão pequeno que possa ser resolvido diretamente.

Figura 01. Bonecas russas Matrioska

Fonte:Wikimedia.org. Acesso em: 07 jul. 2022.

Um problema clássico de recursividade na computação é encontrar o fatorial de um número natural, sabendo que um fatorial respeita a seguinte regra:

0! = 1

1! = 1 * 0! = 1 * 1 = 1

2! = 2 * 1! = 2 * 1 = 2

3! = 3 * 2! = 3 * 2 * 1 = 6

4! = 4 * 3! = 4 * 3 * 2 * 1 = 24

5! = 5 * 4! = 5 * 4 * 3 * 2 * 1 = 120

6! = 6 * 5! = 6 * 5 * 4 * 3 * 2 * 1 = 720

Dessa forma, conclui-se que por regra geral temos N! = N * (N-1). Para se resolver qualquer algoritmo recursivo é necessário possui um caso base (problema menor que se conhece a solução) e um caso geral que modifique o estado atual (local onde a função chama a si mesmo) para um estado que se aproxime do caso base. Levando em consideração essa necessidade e o comportamento de um número fatorial, sendo n é o número natural que deseja o seu fatorial, podemos dizer que:

Caso base: n=0 → 1

Caso geral: n>0 → n * (n-1)!

Observe que o caso base é um problema menor que se consegue a solução facilmente, provocando a parada da recursividade e o caso geral é uma solução que encontra uma solução que depende da solução de um problema menor do atual e em algum momento essa recursividade chegará no caso base.

Perante essas informações, pode-se desenvolver o Código 6, que apresenta uma solução recursiva para encontrar o fatorial de um número natural.

Código06. Função recursiva para encontrar o fatorial

```
01  #include <stdio.h>
02  int fatorial(int n){
03    if (n==0) return 1;
04    else return n*fatorial(n-1);
05  }
06  void main(){
07    int num,f;
08    printf("Digite um numero: ");
09    scanf("%d",&num);
10    f = fatorial(num);
11    printf("Fatorial de %d = %d",num,f);
12  }
```

Fonte: elaborado pelo autor.

No Código 6, criou-se uma função fatorial que recebe como parâmetro um número inteiro (n), na linha 3 é verificado se o número for igual a 0 (zero) será retornado o valor 1 (um), caso contrário, na linha 4, será realizado o retorno do resultado da multiplicação do valor da variável n com o resultado da chamada da função fatorial passando como parâmetro o valor da variável n subtraindo o valor 1 (um). No programa principal que se inicia na linha 6, são declaradas duas variáveis inteiras, em seguida, é solicitado para o usuário digitar um número que é armazenado na variável num, na linha 10 é chamada a função fatorial passando por parâmetro a variável num e seu retorno é salvo na variável f, por fim, na linha 11 é mostrado o resultado do fatorial do número solicitado.

Para elucidar melhor o comportamento da função, o Quadro 1 mostra o comportamento da função quando o usuário digita o valor 4, por meio de um teste de mesa.

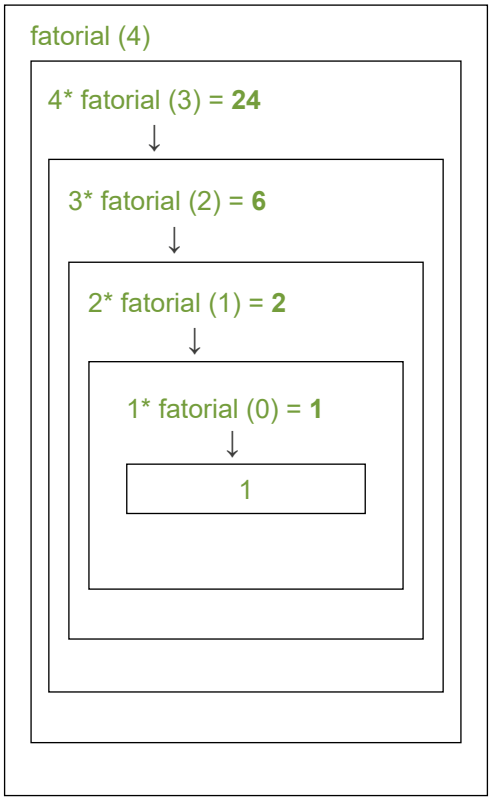
Quadro 01. Teste de mesa para a função fatorial com n=4

LINHA DE EXECUÇÃO	n	COMENTÁRIO
2	4	Função é chamada pela primeira vez com n=4
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Retorna o resultado da multiplicação de 4 pelo fatorial de 3
2	3	Função é chamada pela segunda vez com n=3
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Retorna o resultado da multiplicação de 3 pelo fatorial de 2
2	2	Função é chamada pela terceira vez com n=2
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Retorna o resultado da multiplicação de 2 pelo fatorial de 1
2	1	Função é chamada pela quarta vez com n=1
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Retorna o resultado da multiplicação de 1 pelo fatorial de 0
2	0	Função é chamada pela quinta vez com n=0
3		Verifica se o valor de n é igual a zero, portanto, retorna 1
4	1	Finaliza a quarta chamada retornando o valor 1*1 = 1
4	2	Finaliza a terceira chamada retornando o valor 2*1 = 2
4	3	Finaliza a segunda chamada retornando o valor 3*2 = 6
4	4	Finaliza a primeira chamada retornando o valor 4*6 = 24

Fonte: elaborado pelo autor.

A Figura 2 ilustra cada chamada da função fatorial. Primeira etapa, chamada para o fatorial de n=4, para se obter o resultado, será multiplicado o valor do n que é 4 nessa chamada e a função é chamada pela segunda vez para descobrir o valor de fatorial de n=3. Segunda etapa, será multiplicado o valor do n que é 3 nessa chamada e a função é chamada pela terceira vez para descobrir o valor de fatorial de n=2. Terceira etapa, será multiplicado o valor do n que é 2 nessa chamada e a função é chamada pela quarta vez para descobrir o valor de fatorial de n=1. Quarta etapa, será multiplicado o valor do n que é 1 nessa chamada e a função é chamada pela quinta vez para descobrir o valor de fatorial de n=0. Quinta etapa, retorna um (1), pois o fatorial de n=0 é um (1). Por fim, se obtém o valor da quinta chamada (fatorial de 0 é 1) e o resultado da multiplicação de 1 por 1 é 1, depois, se obtém o valor da quarta chamada (fatorial de 1 é 1) e o resultado da multiplicação de 2 por 1 é 2, em seguida, se obtém o valor da terceira chamada (fatorial de 2 é 2) e o resultado da multiplicação de 3 por 2 é 6, para finalizar, se obtém o valor da segunda chamada (fatorial de 3 é 6) e o resultado da multiplicação de 4 por 6 é 24 e retornando o valor 24 para o fatorial de 4.

Figura 02. Ilustração que representa as chamadas da função fatorial



Fonte: elaborado pelo autor.

> Outro exemplo típico de uma função recursiva é a série de Fibonacci, concebida originalmente como modelo para o crescimento de uma granja de coelhos (multiplicação de coelhos) pelo matemático italiano do século XVI, Fibonacci. A série é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 [...]. (AGUILAR, 2008, p. 219)

Esta série cresce muito rápido, começa no termo de ordem zero, e, a partir do segundo termo, seu valor é dado pela soma dos dois termos anteriores. A série de Fibonacci (fib) expressa-se assim:

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \text{ para } n > 2$$

Dessa forma, podemos dizer que o caso base são fib(0) e fib(1) e o caso geral é a regra gerada para fib(n). O Código 7 apresenta uma função que retorna o enésimo termo da sequência de Fibonacci e mostra a sua sequência no programa principal.

Código07. Função recursiva a sequência de Fibonacci

```
01  #include <stdio.h>
02  int fib(int n) {
03    if(n==0) return 0;
04    if(n==1) return 1;
05    return fib(n-1)+fib(n-2);
06  }
07  void main() {
08    int num,i;
09    printf("Digite um numero: ");
10    scanf("%d",&num);
11    for (i = 0; i <= num; i++) {
12      printf("%d ", fib(i));
13    }
14    printf("\n");
15  }
```

Fonte: elaborado pelo autor.

No código 7, foi declarada uma função fib, na linha 2, que recebe como parâmetro um valor inteiro (n), na linha seguinte verifica se o valor de n é igual a zero, se for retorna o valor 0 (zero) e finaliza a chamada da função, caso contrário, na linha 4, é verificado se o valor de n é igual a 1 (um), se for retorna o valor 1 (um) e finaliza a chamada da função, senão, executa a linha 5 que retorna a soma do retorno da chamada da função para o valor de n subtraído por 1 (um) e a chamada da função para o valor de n subtraído por 2 (dois). Já no programa principal, que inicia na linha 7 é coletado um valor numérico do usuário e armazenado na variável num, em seguida, é utilizada uma estrutura de repetição (for) que inicia do valor 0 (zero) até o número digitado pelo usuário (num), fazendo que se repita a chamada da função e mostrando a sequência de Fibonacci até o enésimo número solicitado, por fim, na linha 14 há um comando para pular uma linha.

Buscando um melhor entendimento da função, o Quadro 2 mostra o comportamento da função quando o usuário digita o valor 3, por meio de um teste de mesa.

Quadro 02. Teste de mesa para a função fib (Fibonacci) com n=3

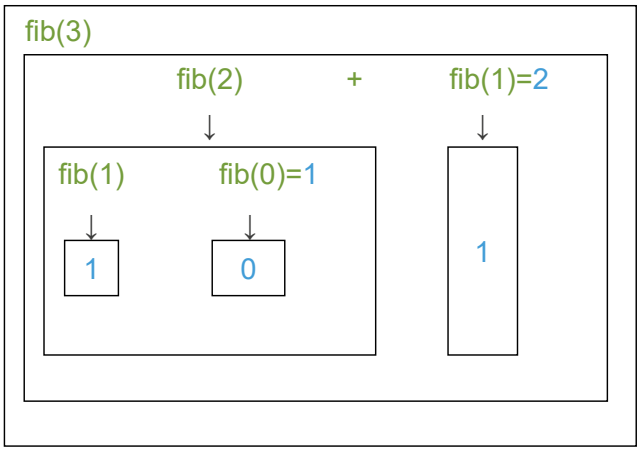
LINHA DE EXECUÇÃO	n	COMENTÁRIO
2	3	Função é chamada pela primeira vez com n=3
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Verifica se o valor de n é igual a um, portanto, não executa
5		Chama-se a função para n=2 e aguarda para fazer a soma
2	2	Função é chamada pela segunda vez com n=2
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Verifica se o valor de n é igual a um, portanto, não executa
5		Chama-se a função para n=1 e aguarda para fazer a soma
2	1	Função é chamada pela terceira vez com n=1
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Verifica se o valor de n é igual a um, portanto, retorna 1
5	2	Retorna 1 para n=1 e chama a função para n=0
2	0	Função é chamada pela quarta vez com n=0
3		Verifica se o valor de n é igual a zero, portanto, retorna 0
5	2	Retorna a soma de 1 com o retorno de n=0, ou seja, retorna 1
5	3	Retorna 1 para n=2 e chama a função para n=1
2	1	Função é chamada pela quinta vez com n=1

LINHA DE EXECUÇÃO	n	COMENTÁRIO
3		Verifica se o valor de n é igual a zero, portanto, não executa
4		Verifica se o valor de n é igual a um, portanto, retorna 1
5	3	Retorna a soma de 1 com o retorno de n=1, ou seja, retorna 2

Fonte: elaborado pelo autor.

A Figura 3 ilustra cada chamada da função que encontra o enésimo número na sequência de Fibonacci. Primeira etapa, é chamada a função para n=3, com isso, será somado o retorno da chamada da função de n=2 e n=1, para encontrar a solução a função é chamada pela segunda vez para n=2. Segunda etapa, será somado o retorno da chamada da função de n=1 e n=0, para encontrar a solução a função é chamada pela terceira vez para n=1. Terceira etapa, será retornado 1, pois n=1 é 1. Volta-se para a segunda chamada e ocorre a quarta chamada da função para n=0. Quarta etapa, será retornado 0, pois n=0 é 0. No retorno para a segunda chamada com a solução de n=1 e n=0 é realizado a soma de 1 e 0 retornando o valor para a primeira chamada da função e ocorre a quinta chamada da função para n=1. Quinta etapa, será retornado 1, pois n=1 é 1. Por fim, retorna-se para a primeira chamada da função com os valores de n=2 e n=1, com isso, é somado o valor 1 e 1 retornando o valor 2.

Figura 03. Ilustração que representa as chamadas da função fib (Fibonacci)



Fonte: elaborado pelo autor.

DESAFIO

A sequência de Padovan é semelhante a sequência de Fibonacci com estrutura recursiva semelhante. A sequência de Padovan é uma sequência de naturais P(n) definida pelos valores iniciais P(0) = P(1) = P(2) = 1 e a seguinte relação recursiva: P(n) = P(n - 2) + P(n - 3) se n > 2

Alguns valores da sequência são: 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21, 28...

Com base nessas informações, faça uma função recursiva que receba um número N e retorne o N-ésimo termo da sequência de Padovan. Também mostre o funcionamento da função no programa principal.

As funções recursivas não são funções que obrigatoriamente tem que ter um retorno, ou seja, existem também procedimentos (funções sem retorno) recursivos. Um exemplo de procedimento recursivo é o Código 8 que mostra na tela uma sequência decrescente de números de N até 1, onde N é definido pelo usuário.

Código08. Função recursiva que mostra uma sequência decrescente

```
01  #include <stdio.h>
02  void imprime_decrescente(int n) {
03    if(n>0){
04        printf("%i\n",n);
05        imprime_decrescente(n-1);
06    }
07  }
08  void main() {
09    int num;
10    printf("Digite um numero: ");
11    scanf("%d",&num);
12    imprime_decrescente(num);
13  }
```

Fonte: elaborado pelo autor.

No Código 8, é declarada o procedimento imprime_descrecente, na linha 2, que recebe como parâmetro um valor inteiro (n), em seguida, é verificado se o número é maior que 0 (zero), se for, na linha 4 é mostrado na tela o valor de n e pula-se uma linha, em seguida, chama-se o procedimento novamente com o valor de n subtraído o valor 1. Caso o valor de n seja menor ou igual a 0 (zero) o procedimento não executa nada e finaliza a sua execução (condição de parada da recursividade). No programa principal é solicitado que o usuário digite um número e seu valor é armazenado na variável num que é passada por parâmetro na função recursiva imprime_decrescente.

O Quadro 3 mostra o comportamento do procedimento quando o usuário digita o valor 3, por meio de um teste de mesa demonstra os valores que serão mostrados na tela.

Quadro 03. Teste de mesa para o procedimento imprime_decrescente com n=3

LINHA DE EXECUÇÃO	n	TELA	COMENTÁRIO
2	3		Procedimento é chamado pela primeira vez com n=3
3			Verifica se n é maior que zero, como é, executa a condicional
4		3	Mostra na tela o valor de n que é 3 e pula uma linha
5			Chama o procedimento com o valor de n=2
2	2		Procedimento é chamado pela segunda vez com n=2
3			Verifica se n é maior que zero, como é, executa a condicional
4		2	Mostra na tela o valor de n que é 2 e pula uma linha
5			Chama o procedimento com o valor de n=1
2	1		Procedimento é chamado pela terceira vez com n=1
3			Verifica se n é maior que zero, como é, executa a condicional
4		1	Mostra na tela o valor de n que é 1 e pula uma linha
5			Chama o procedimento com o valor de n=0
2	0		Procedimento é chamado pela quarta vez com n=0
3			Verifica se n é maior que zero, como não é, finaliza a chamada
5	1		Finaliza a terceira chamada do procedimento
5	2		Finaliza a segunda chamada do procedimento
5	3		Finaliza a primeira chamada do procedimento

Fonte: elaborado pelo autor.

O Código 8 apresenta na tela uma sequência decrescente de números de N até 1, mas é possível fazer a mesma sequência em ordem crescente utilizando um procedimento recursivo? Observe o Código 9 apresentado abaixo.

Código 09. Função recursiva que mostra uma sequência crescente

```
01  #include <stdio.h>
02  void imprime_crescente(int n) {
03    if(n>0){
04         imprime_crescente(n-1);
05         printf("%i\n",n);
06    }
07  }
08  void main() {
09    int num;
10    printf("Digite um numero: ");
11    scanf("%d",&num);
12    imprime_crescente(num);
13  }
```

Fonte: elaborado pelo autor.

No Código 9, com relação ao Código 8, foi alterado o nome do procedimento de imprime_decrescente para imprime_crescente e também invertida a ordem da linha 4 e 5. Com isso, a ordem de execução muda, conforme pode-se notar no Quadro 4 que mostra o comportamento do procedimento quando o usuário digita o valor 3, por meio de um teste de mesa demonstra os valores que serão mostrados na tela.

Quadro 04. Teste de mesa para o procedimento imprime_crescente com n=3

LINHA DE EXECUÇÃO	n	TELA	COMENTÁRIO
2	3		Procedimento é chamado pela primeira vez com n=3
3			Verifica se n é maior que zero, como é, executa a condicional
4			Chama o procedimento com o valor de n=2
2	2		Procedimento é chamado pela segunda vez com n=2
3			Verifica se n é maior que zero, como é, executa a condicional
4			Chama o procedimento com o valor de n=1
2	1		Procedimento é chamado pela terceira vez com n=1
3			Verifica se n é maior que zero, como é, executa a condicional
4			Chama o procedimento com o valor de n=0
2	0		Procedimento é chamado pela quarta vez com n=0

LINHA DE EXECUÇÃO	n	TELA	COMENTÁRIO
3			Verifica se n é maior que zero, como não é, finaliza a chamada
5	1	1	Mostra na tela o valor de n que é 1 e pula uma linha Em seguida, finaliza a terceira chamada do procedimento
5	2	2	Mostra na tela o valor de n que é 2 e pula uma linha. Em seguida, finaliza a segunda chamada do procedimento
5	3	3	Mostra na tela o valor de n que é 3 e pula uma linha. Em seguida, finaliza a primeira chamada do procedimento

Fonte: elaborado pelo autor.

Ao se comparar o Quadro 3 (imprime_decrescente) e o Quadro 4 (imprime_crescente), pode-se perceber que a ordem de execução se altera, dessa forma, altera-se a ordem que apresenta-se os números na tela, passando da ordem decrescente para a ordem crescente alterando as linhas 4 e 5 do código, ou seja, mudando a ordem que chama o procedimento com o comando que mostra o valor de n na tela.

DESAFIO

Elabore uma função recursiva que mostre uma sequência de números pares em ordem crescente de 0 a N. Mostre no programa principal o funcionamento da função sendo passado por parâmetro um valor definido pelo usuário (N).

Viu-se programas recursivos que trazem o resultado do fatorial de um número natural, o enésimo número da sequência de Fibonacci e também uma sequência de números em ordem crescente e decrescente, mas esses as soluções só podem serem resolvidas por meio de recursividade? Se tem duas maneiras, qual é a melhor? Isso que irá se abordado na sequência.

3. MÉTODO ITERATIVO E RECURSIVO

Como visto em outras ocasiões, um dos recursos mais importantes na atividade de programação de computadores é a capacidade de trechos de programas poderem ser repetidos. O método iterativo utiliza comando na linguagem C como for, *while* e *do-while* que possibilitam repetir determinado trecho de programa diversas vezes. Possibilitando definir quantidades variadas de repetições, com laços determinísticos (quando se sabe o número de vezes que o laço será executado – laço iterativo) e laços indeterminados (quando não se sabe quantas vezes o laço será executado, dependendo de uma resposta positiva do usuário para fazê-lo – laço interativo).

Já o método recursivo busca solucionar os problemas dividindo-os em parte menores, em subproblemas reduzidos até que chegue em um problema ao qual consiga se obter a solução de uma forma trivial. Esse método se baseia no princípio de prova fundamental em matemática (o princípio da indução), ou seja, a solução de um problema pode ser expressa em dois casos: os básicos e demais casos em termos da solução para casos mais simples que o original.

Se compararmos os dois métodos, de uma forma geral, os códigos recursivos são considerados mais elegantes e curtos do que os iterativos, com isso, facilitam a interpretação do código. Entretanto, esses códigos podem dificultar a detecção de erros e acabam sendo mais ineficientes. É importante ressaltar que, conforme foi falado anteriormente, as funções recursivas precisam ter critérios de parada (caso base) e parâmetro da chamada recursiva (caso geral), se isso não for bem definido no código a execução pode provocar loop infinito, fazendo com que o programa não pare de ser executado.

Os códigos recursivos também tendem a necessitar de mais tempo e/ou espaço do que códigos iterativos, pois sempre que chamamos uma função, é necessário espaço de memória para armazenar os parâmetros, as variáveis locais e o endereço de retorno da função. Dessa forma, nas funções recursivas, as informações das variáveis (locais ou parâmetros formais) ficam armazenadas para cada chamada da recursividade, com isso, o uso da memória é multiplicada pela quantidade de chamadas da função. É importante lembrar que as instruções para alocação, liberação, entre outras instruções necessárias nesse processo exige um custo computacional, provocando assim um tempo maior na execução em um código equivalente no método iterativo.

Analise o Quadro 5 que apresenta lado a lado o método iterativo e recursivo para localizar o fatorial de um número natural.

Quadro 05. Fatorial no método iterativo e recursivo

	Método Iterativo	Método Recursivo
01	`#include<stdio.h>`	`#include <stdio.h>`
02	`int fatorial(int n){`	`int fatorial(int n){`
03	`  int i,fat=1;`	`  if (n==0) return 1;`
04	`  for(i=1;i<=n;i++) fat=fat*i;`	`  else return n*fatorial(n-1);`
05	`  return fat;`	`}`
06	`}`	
07	`void main(){`	`void main(){`
08	`  int num,f;`	`  int num,f;`
09	`  printf("Digite um numero: ");`	`  printf("Digite um numero: ");`
10	`  scanf("%d",&num);`	`  scanf("%d",&num);`
11	`  f = fatorial(num);`	`  f = fatorial(num);`
12	`  printf("%d! = %d",num,f);`	`  printf("%d! = %d",num,f);`
13	`}`	`}`

Fonte: elaborado pelo autor.

No Quadro 5, tanto no método iterativo como recursivo o programa principal não possui diferença, então será discutido os trechos das partes diferentes das funções que estão entre as linhas 3 e 5.

Observe que o método recursivo se torna mais elegante e simples (comando condicional composto) em comparado com o método recursivo (comando de estrutura de repetição), entretanto, o método iterativo utiliza 3 (três) variáveis (i, n e fat) independentemente do valor do fatorial que se deseja, já no método recursivo, a cada momento que é realizado a chamada da função. Se compararmos com os métodos para o fatorial de 4, o método iterativo terá 3 (três) variáveis e o método recursivo armazenará terá 4 (quatro) espaço para armazenar o valor de n, com isso, o equivalente a 3 (três) variáveis sendo armazenadas, portanto, a partir do fatorial de 4, incluindo ele, o método recursivo consome mais memória, em relação ao método iterativo.

Outro problema apresentado no método recursivo pode ser realizado no método iterativo é a busca pelo enésimo número da sequência de Fibonacci que são apresentados no Quadro 6.

Quadro 06. Fibonacci no método iterativo e recursivo

	Método Iterativo	Método Recursivo
01	`#include <stdio.h>`	`#include <stdio.h>`
02	`int fib(int n) {`	`int fib(int n) {`
03	`  int i,t,c,a=0,b=1;`	`  if(n==0) return 0;`
04	`  for(i=0;i<n;i++){`	`  if(n==1) return 1;`
05	`  c=a+b;`	`  return fib(n-1)+fib(n-2);`
06	`  a=b;`	`}`
07	`  b=c;`	
08	`  }`	
09	`  return a;`	
10	`}`	
11	`void main() {`	`void main() {`
12	`  int num,i;`	`  int num,i;`
13	`  printf("Digite um numero: ");`	`  printf("Digite um numero: ");`
14	`  scanf("%d",&num);`	`  scanf("%d",&num);`
15	`  for (i = 0; i <= num; i++) {`	`  for (i = 0; i <= num; i++) {`
16	`  printf("%d ", fib(i));`	`  printf("%d ", fib(i));`
17	`  }`	`  }`
18	`  printf("\n");`	`  printf("\n");`
19	`}`	`}`

Fonte: elaborado pelo autor.

Assim como o Quadro 5 que foi analisado as partes diferentes do código, aqui no Quadro 6 será observado as diferenças nos códigos que estão entre as linhas 3 e 9. O método recursivo para a sequência de Fibonacci é muito elegante. Porém, ela contém duas chamadas para si mesma, em outras palavras, ela contém duas chamadas recursivas. Portanto, sua elegância não significa eficiência. Por exemplo, para encontrar o n=3 no método recursivo a função é chamada cinco vezes, contendo 5 (cinco) valores de n diferente, aumentando rapidamente o uso da memória a cada valor incrementado, por exemplo, para n=4 passa para 9 chamadas da função, por outro lado, o método iterativo terá sempre as mesmas 6 variáveis (cinco locais e uma parâmetro formal), não aumentando o uso da memória independentemente do valor de n.

Analise também os códigos que mostram na tela uma sequência crescente de números de 1 até N apresentados no Quadro 7.

Quadro 07. Sequência crescente de números no método iterativo e recursivo

	Método Iterativo	Método Recursivo
01	`#include <stdio.h>`	`#include <stdio.h>`
02	`void imprime_crescente(int n) {`	`void imprime_crescente(int n) {`
03	`  int i;`	`  if(n>0){`
04	`  for(i=1;i<=n;i++)`	`  imprime_crescente(n-1);`
05	`  printf("%i\n",n);`	`  printf("%i\n",n);`
06	`}`	`  }`
07		`}`
08	`void main() {`	`void main() {`
09	`  int num;`	`  int num;`
10	`  printf("Digite um numero: ");`	`  printf("Digite um numero: ");`
11	`  scanf("%d",&num);`	`  scanf("%d",&num);`
12	`  imprime_crescente(num);`	`  imprime_crescente(num);`
13	`}`	`}`

Fonte: elaborado pelo autor.

Novamente será analisado as diferenças nos códigos que estão entre as linhas 3 e 6. O método iterativo sempre terá as duas variáveis (n que é parâmetro e i que é local), já no método recursivo a cada chamada da função será armazenado um valor de n. Portanto, para n=1 os dois códigos armazenarão duas variáveis, para os demais números maiores que um o método iterativo permanecerá com duas variáveis obter uma resposta, diante do método recursivo, que aumentará um valor de n para ser armazenado a cada incremento. É claro que o maior armazenamento prejudica somente o espaço de memória, mas também o desempenho do algoritmo, pois armazenar e ler em memória também exigem instruções para o processador, com isso, aumenta instruções a serem processadas, também comprometendo o desempenho.

Diante de toda a explicação até o momento vale refletir, então por quais razões devo escolher o método recursivo?

> A razão fundamental é que existem numerosos problemas complexos que possuem natureza recursiva e, consequentemente, são mais fáceis de implementar com algoritmos deste tipo. Entretanto, em condições críticas de tempo e de memória, ou seja, quando o consumo de tempo e memória é decisivo ou conclusivo para a resolução do problema, a solução escolhida deve ser, geralmente, a iterativa. Qualquer problema que pode ser resolvido por recursão também pode ser resolvido por iteração. Um enfoque recursivo é escolhido com preferência a um enfoque iterativo quando o enfoque recursivo é mais natural para a resolução do problema e produz um programa mais fácil de compreender e depurar. Outra razão para escolher uma solução recursiva é que uma solução iterativa pode não ser clara nem evidente (AGUILAR, 2008, p. 510).

Existem tipos de problemas cujas soluções precisam manter registros de estados anteriores, sendo inerentemente recursivas, um exemplo é o percurso de uma árvore (uma abstração clássica na computação que é utilizada de diversas formas). Outros exemplos que se pode citar é a função de Ackermann e algoritmos de divisão e conquista (*quicksort*).

SAIBA MAIS

Na computação existem área de complexidade de algoritmos (ou complexidade computacional) que estuda e define quanto eficiente é um algoritmo em relação ao número de operações necessárias para finalizar a tarefa. Se ficou curiosos para avaliar as diferentes formas de resolver um algoritmo, busque mais sobre essa área.

CONCLUSÃO

O escopo de variáveis das variáveis permite o desenvolvedor criar subprogramas que não interferem em outros, declarando variáveis locais e por parâmetros formais, mas também é possível realizar a declaração de uso mais geral que é o caso da declaração global. É importante que o desenvolver entenda bem essa diferenciação e conceitos para não provocar inconsistência de informações no momento da execução do programa.

Definiu-se também o método recursivo para a solucionar problemas computacionais, assim como seus princípios que a função só recursiva quando ela chama si mesma diretamente ou indiretamente. É importante também lembrar que para que a função recursiva seja elaborada, é necessário produzir um caso básico, onde se já conhece a solução e o caso geral que muda o problema produzindo uma forma que caminhe para o caso básico, se isso não ocorrer pode-se cair em loop infinito.

Por fim, os métodos recursivo e iterativo têm suas diferenças que produzem vantagens e desvantagens para certos problemas computacionais, cabe também ao desenvolvedor conhecer as principais delas para que não selecione o método inadequado para solucionar o problema desejado, produzindo ineficiência e lentidão para os usuários.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILAR. Luis J. **Fundamentos de Programação**: algoritmos, estrutura de dados e objetos. 3. ed. Porto Alegre: AMGH, 2008.

PINHEIRO, Francisco de Assis C. **Elementos de Programação em C**. Porto Alegre: Bookman, 2012.

SCHILDT, Herbert. **C, Total e Completo**. 3. ed. São Paulo: Makron Books, 1996.

UNIDADE 3

PONTEIROS

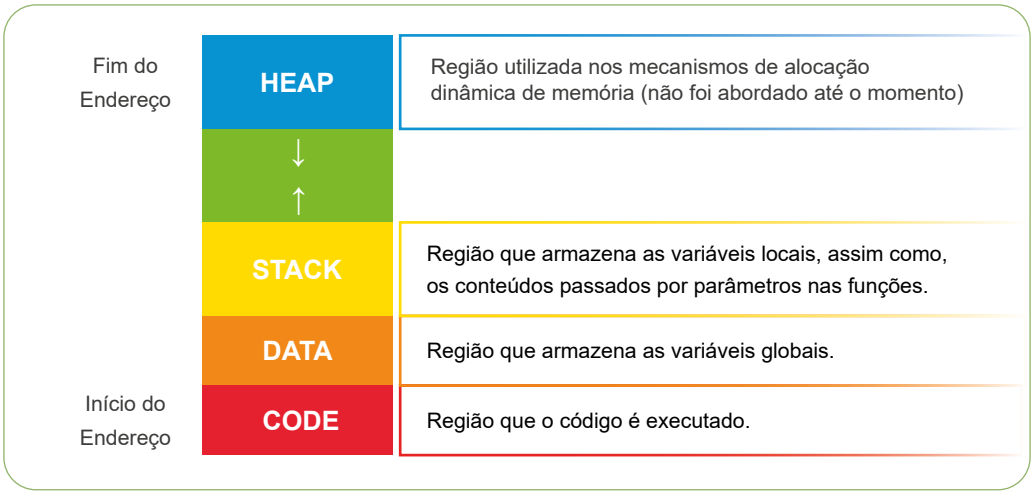
INTRODUÇÃO

Um dos recursos que a linguagem C permite para o programador é o acesso indireto as informações alocadas em memória, essa variável especial permite maiores possibilidades para o desenvolvimento, mas também traz muita responsabilidade, pois o uso incorreto desses recursos pode provocar erros inesperados ou até a interrupção da execução do programa. O conceito de ponteiro é um assunto que demora um pouco para ser compreendido, mas após sua compreensão acaba-se ampliando a visão do programador sobre o código, principalmente sobre o uso do recurso de armazenamento, ou seja, começa-se a ampliar algumas lacunas que não estavam explicadas no código sobre a memória principal.

1. PONTEIROS

Toda informação que manipulamos dentro de um programa obrigatoriamente está armazenada na memória principal do computador. Para a execução de um programa na memória principal do computador é necessário o uso adequado dos espaços existentes na memória, com isso, a linguagem C usa um mapa de memória em quatro regiões para realizar essa organização, conforme mostra a Figura 1.

Figura 01. Mapa de memória da linguagem C



Fonte: elaborado pelo autor.

Portanto, a região *Code* é onde fica armazenado o programa para sua execução, não podendo ter seu tamanho alterado quando seu programa está em execução. As variáveis estáticas (as quais viu-se até o momento) ficam armazenadas nas regiões *Data* (variáveis globais) e *Stack* (variáveis locais). Já as variáveis dinâmicas (que não foi abordado ainda) estão na região *Heap*, sendo criadas no momento da execução.

Os ponteiros são variáveis especiais que permitem acessar essas regiões (*Data*, *Stack* e *Heap*) da memória. Em outras palavras, ponteiros "[...] são um tipo especial de variáveis que permitem armazenar endereços de memória em vez de dados numéricos (como os tipos *int*, *float* e *double*) ou caracteres (como o tipo *char*) [...]" (BACKES, 2022, p. 195).

Ao cria-se uma variável, o computador reserva um espaço de memória no qual pode-se guardar o valor associado a ela, dessa forma, o nome da variável é associado ao endereço do espaço que o computador reservou na memória para guardá-la. Através de ponteiros pode-se ter acesso onde a variável está armazenada (endereço de memória) e fazer a manipulação do conteúdo armazenado neste espaço. Na linguagem C, para se declarar uma variável do tipo ponteiro basta colocar um asterisco (*), que é o operador de indireção, antes do nome dela e para obter o endereço da variável basta inserir o símbolo & (popularmente conhecido como E comercial), que é operador de referência, como pode-se ver no exemplo do Código 1.

Código 01. Exemplo de declaração e atribuição no ponteiro

```
01  #include <stdio.h>
02  void main(){
03          int a=3;
04          int *p;
05          p=&a;
06          printf("Conteudo apontado por p: %i\n",*p);
07          printf("Endereco em p: %p\n",p);
08  }
```

Fonte: elaborado pelo autor.

No código 1, linha 3, foi declarada uma variável inteira chamada "a" e foi atribuído o valor 3 (três), em seguida, na linha 4, foi declarado um ponteiro do tipo inteiro que pode armazenar o endereço da memória possuindo o conteúdo inteiro. Na linha 5 é atribuído o endereço da variável "a" através do operador &. Dessa forma, pode-se dizer que o ponteiro "p" aponta para a variável "a". Na linha 6 é mostrado o conteúdo apontado pelo ponteiro p por meio do operador asterisco (*). Esse operador tem função de multiplicação quando utilizado de forma binária (dois operandos), mas quando usado unitariamente (um operando), ele pode declarar um ponteiro (quando informado o tipo) e mostrar o conteúdo apontado pelo ponteiro. Por fim, na linha 7 é mostrado o endereço contido em p. Para se mostrar o endereço no comando printf utiliza-se o formatador %p ou %x, pois o endereço de memória é um número hexadecimal positivo. Observe a Tabela 1, que busca ilustrar a memória desse cenário do Código 1.

Tabela 02. Exemplo da memória para as variáveis no Código 1

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FE10	int a	3
62FE1C	int *p	62FE10

Fonte: elaborada pelo autor.

Na Tabela 1, o endereço que foi associado a variável “a” é 62FE10 e seu conteúdo é o valor 3 (três), pois foi atribuído esse valor quando ela foi declarada. Já na linha seguinte o ponteiro p está associado ao endereço 62FE1C e possui como conteúdo o valor 62FE10, ou seja, o endereço da variável “a”, dessa forma, o ponteiro “p” aponta para a variável “a”. Considerando esse cenário (informações da Tabela 1), os valores que serão mostrados na tela a execução do Código 1 serão:

```
Conteudo apontado por p: 3

Endereco em p: 62FE10
```

Isso ocorrerá porque na linha 6 será mostrado o conteúdo apontado pelo ponteiro p, ou seja, o valor da variável “a” que é 3 (três). Em seguida, na linha 7 faz com que será mostrado na outra linha o endereço em p que será o endereço da variável “a” que é 62FE10.

SAIBA MAIS

Existem três formas de colocar o asterisco na declaração de um ponteiro, são elas:

```
int * p; /* Asterisco separado do tipo e da variável */

int* p; /* Asterisco junto ao tipo */

int *p; /* Asterisco junto à variável */
```

As três formas são equivalentes, mas quando se coloca o asterisco junto com o tipo pode induzir o programador ao erro, por exemplo:

```
int* x,y,z;
```

Neste caso, parece que foi declarado três ponteiros, entretanto, foi declarado x como ponteiro e declarado como variáveis y e z.

O Código 2 apresenta como se comporta os ponteiros após a alteração do valor da variável que eles apontam e também a mudança do seu endereço que está apontando.

Código 02. Exemplo de mudança de conteúdo e endereço do ponteiro

```
01  #include <stdio.h>
02  void main(){
03        int a=3,b=1,c=2,*p1=NULL,*p2=NULL;
04        p1=&a;
05        p2=&b;
06        printf("Conteudo apontado por p1: %i\n",*p1);
07        printf("Endereco em p1: %p\n",p1);
08        printf("Conteudo apontado por p2: %i\n",*p2);
09        printf("Endereco em p2: %p\n",p2);
10        a=4;
11        p2=&c;
12        printf("Conteudo apontado por p1: %i\n",*p1);
13        printf("Endereco em p1: %p\n",p1);
14        printf("Conteudo apontado por p2: %i\n",*p2);
15        printf("Endereco em p2: %p\n",p2);
16  }
```

Fonte: elaborado pelo autor.

No Código 2, linha 3 está sendo declarado três variáveis (a, b e c) e atribuindo também os valores 3, 1 e 2, respectivamente. Também nessa linha são declarados dois ponteiros (p1 e p2) e atribuído o valor nulo. Quando não se quer que o ponteiro aponte para variável alguma, coloca-se apontando para NULL. A constante simbólica NULL, quando colocada em um ponteiro, indica que ele não aponta para nenhuma variável. Na linha 4 é atribuído o endereço da variável "a" para o ponteiro p1 e na linha 5 é atribuído ao ponteiro p2 o endereço da variável "b". Em seguida, nas linhas 6 a 9 são mostrados os conteúdos apontados e os endereços dos ponteiros p1 e p2. Que podemos ilustrar como na Tabela 2 com a execução até a linha 9 do Código 2.

Tabela 02. Exemplo da memória para as variáveis no Código 2 até a linha 9

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FE00	int *p2	62FE18
62FE08	int *p1	62FE1C
62FE1C	int a	3
62FE14	int c	2
62FE18	int b	1

Fonte: elaborada pelo autor.

Na Tabela 2, observe que o ponteiro p2 possui o endereço referente a variável “b” com o valor de endereço 62FE18 e o ponteiro p1 possui o endereço referente a variável “a” com o valor de endereço 62FE1C. Com isso, se considerar esse cenário (informações da Tabela 2), será mostrado na tela oriundo das linhas 6 a 9 os valores:

```
Conteudo apontado por p1: 3

Endereco em p1: 62FE1C

Conteudo apontado por p2: 1

Endereco em p2: 62FE18
```

Logo adiante, na linha 10 do Código 2 é atribuído o valor 4 na variável “a”, com isso, o conteúdo apontado por p1 será alterado, mas seu endereço continua o mesmo. Já na linha 11, o ponteiro p2 recebe o endereço da variável “c”, com isso, será mudado o valor do conteúdo e do endereço do ponteiro p2 mostrado na tela. Veja a Tabela 3 que ilustra essa mudança na execução após a linha 9.

Tabela 03. Exemplo da memória para as variáveis no Código 2 após a linha 9

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FE00	int *p2	62FE14
62FE08	int *p1	62FE1C
62FE1C	int a	4
62FE14	int c	2
62FE18	int b	1

Fonte: elaborada pelo autor.

Na Tabela 3, observe que agora o ponteiro p2 possui o endereço referente a variável “c” com o valor de endereço 62FE14 e o ponteiro p1 continua possuindo o endereço referente a variável “a” com o valor de endereço 62FE1C, entretanto o conteúdo da variável “a” mudou de 3 (três) para 4 (quatro). Com isso, se considerar esse cenário (informações da Tabela 3), será mostrado na tela oriundo das linhas 12 a 15 os valores:

```
Conteudo apontado por p1: 4

Endereco em p1: 62FE1C

Conteudo apontado por p2: 2

Endereco em p2: 62FE14
```

Também é possível alterar o valor do conteúdo apontado pelo ponteiro usando o ponteiro na atribuição, como podemos ver no Código 3

Código 03. Exemplo de mudança de conteúdo usando o ponteiro

```
01  #include <stdio.h>
02  void main(){
03          int a=3,b=1,c=2,*p1=NULL,*p2=NULL;
04          p1=&a;
05          p2=&b;
06          printf("Conteudo apontado por p1: %i\n",*p1);
07          printf("Endereco em p1: %p\n",p1);
08          printf("Conteudo apontado por p2: %i\n",*p2);
09          printf("Endereco em p2: %p\n",p2);
10          *p1=6;
11          (*p2)++;
12          printf("Conteudo apontado por p1: %i\n",*p1);
13          printf("Endereco em p1: %p\n",p1);
14          printf("Conteudo apontado por p2: %i\n",*p2);
15          printf("Endereco em p2: %p\n",p2);
16  }
```

Fonte: elaborada pelo autor.

No Código 3, até a linha 9 será mostrado os mesmos valores informados no Código 2 e ilustrados na Tabela 2. Já na linha 10 é atribuído o valor 6 (seis) no conteúdo apontado por p1, com isso o valor da variável "a" será alterado para o valor 6 (seis). Na linha 11, foi realizado o incremento (somando mais um ao valor atual) ao conteúdo apontado por p2, portanto o valor da variável "b" será incrementado. A Tabela 4, ilustra essa mudança na execução após a linha 9.

Tabela 04. Exemplo da memória para as variáveis no Código 3 após a linha 9

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FE00	int *p2	62FE18
62FE08	int *p1	62FE1C
62FE1C	int a	6
62FE14	int c	2
62FE18	int b	2

Fonte: elaborada pelo autor.

Na Tabela 4, observe que o ponteiro p1 continua possuindo o endereço referente a variável “a” com o valor de endereço 62FE1C, entretanto o conteúdo da variável “a” mudou de 3 (três) para 6 (seis), porque foi atribuído o valor 6 no conteúdo apontado por p1. O ponteiro p2 continua com o endereço referente a variável “b” com o valor de endereço 62FE18, entretanto o conteúdo da variável “b” mudou de 1 (um) para 2 (dois), porque foi incrementado no conteúdo apontado por p2, ou seja, somou mais um no valor atual que era 1 (um). Com isso, se considerar esse cenário (informações da Tabela 4), será mostrado na tela oriundo das linhas 12 a 15 os valores:

```
Conteudo apontado por p1: 6

Endereco em p1: 62FE1C

Conteudo apontado por p2: 2

Endereco em p2: 62FE18
```

Como dito em outros momentos, os ponteiros também possuem tipos e devem ser indicados no momento da sua declaração. No Código 4 são apresentados alguns exemplos dos tipos mais comuns de dados na linguagem C.

Código 04. Exemplo de ponteiros de outros tipos

```
01  #include <stdio.h>
02  void main(){
03        int a=3,*p_int;
04        float b=2.8,*p_float;
05        char c='a',*p_char;
06        double d=4.9,*p_double;
07        p_int=&a;
08        p_float=&b;
09        p_char=&c;
10        p_double=&d;
11        printf("Conteudo apontado por p_int: %i\n",*p_int);
12        printf("Endereco em p_int: %p\n",p_int);
13        printf("Conteudo apontado por p_float: %f\n",*p_float);
14        printf("Endereco em p_float: %p\n",p_float);
15        printf("Conteudo apontado por p_char: %c\n",*p_char);
16        printf("Endereco em p_char: %p\n",p_char);
17        printf("Conteudo apontado por p_double: %lf\n",*p_double);
18        printf("Endereco em p_double: %p\n",p_double);
19  }
```

Fonte: elaborado pelo autor.

No Código 4, mais precisamente, na linha 3 é declarada uma variável com o nome “a” e atribuído o valor 3 (três) e também na mesma linha é declarado um ponteiro chamado p_int do tipo inteiro. Na linha 4 é declarada uma variável chamada “b” do tipo *float* e atribuído o valor 2.8 (lembrando que os valores decimais na linguagem C são separados por ponto), do mesmo modo, na mesma linha é declarado um ponteiro chamado p_float do tipo *float*. Na linha 5 é declarada uma variável cujo nome é “c” do tipo char e foi atribuído o valor “c” (no código foi utilizado aspas simples por se tratar de somente um caractere), assim como, foi declarada na mesma linha um ponteiro do tipo *char* denominado p_char. Por fim, na linha 6 é declarada a variável “d” do tipo *double* que está sendo atribuído o valor “4.9”, bem como, a declaração do ponteiro p_double do tipo *float*.

SAIBA MAIS

Todo ponteiro possui um tipo para o qual aponta, entretanto é possível acontecer conversões de tipo. Existe o ponteiro do tipo *void*, esse ponteiro não pode ser aproveitado sem uma conversão de um tipo de dado, pois somente realizando essa conversão de tipo que o endereço do ponteiro pode ser manipulado diretamente. Um exemplo de uso são as funções para a alocação dinâmica de memória (*malloc*, *calloc* e *realloc*) retornam um ponteiro do tipo *void*, por isso, é preciso realizar o *cast* para que não acuse erro nas análises do compilador.

Entre as linhas 7 e 10 estão sendo atribuídos aos ponteiros os endereços das respectivas variáveis, ou seja, o ponteiro do tipo inteiro está recebendo o endereço da variável inteira e assim por diante. Vale alertar que alguns compiladores permitem inserir endereços de outro tipo nos ponteiros, mas é indicado uma advertência para o programador, pois não é uma atividade recomendada. Para ilustrar a sequência do Código 4 é utilizada a Tabela 4 que exemplifica a execução do código.

Tabela 05. Exemplo da memória para as variáveis no Código 4

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FDE0	double *p_double	62FDE8
62FDE8	double d	4.900000
62FDF0	char *p_char	62FDFF
62FDFF	char c	a
62FE00	float *p_float	62FE0C
62FE0C	float b	2.800000
62FE10	int *p_int	62FE1C
62FE1C	int a	3

Fonte: elaborada pelo autor.

Na Tabela 5, mostra-se os endereços armazenados nos ponteiros, assim como, os respectivos valores de cada variável. Observe que independentemente do tipo do ponteiro é armazenado um endereço de memória que nesse exemplo ocupam 8 bytes, já as variáveis têm valores diferentes para cada tipo de dados armazenadas. Com isso, se considerar esse cenário (informações da Tabela 5), será mostrado na tela oriundo das linhas 11 a 18 os valores:

```
Conteudo apontado por p_int: 3
Endereco em p_int: 62FE1C
Conteudo apontado por p_float: 2.800000
Endereco em p_float: 62FE0C
Conteudo apontado por p_char: a
Endereco em p_char: 62FDFF
Conteudo apontado por p_double: 4.900000
Endereco em p_float: 62FDE8
```

DESAFIO

Analise o código abaixo na linguagem C.

```
01  #include <stdio.h>
02  void main(){
03      int x, y=0, *p;
04      p = &y;
05      x = *p + 4;
06      x = 3;
07      x--;
08      *p = x + y;
09      printf("%i %i\n",x,y);
10  }
```

Mostre através do teste de mesa quais serão os valores de x e y ao final da execução do código acima.

2. ARITMÉTICA DE PONTEIROS

Como vimos em outros momentos, vetores são coleções de variáveis contínuas na memória principal do computador e os ponteiros também são utilizados na manipulação e tratamento dessa estrutura. Sendo que o nome do vetor está associado ao endereço do seu primeiro elemento, portanto, se vet é um vetor vet==&vet[0] (sabendo que o primeiro elemento está no índice zero do vetor).

Dessa forma, se o nome do vetor é um endereço de memória (número), o nome do vetor é um ponteiro constante (não tem alteração durante a execução) para o primeiro elemento deste vetor. Veja o Código 5 que apresenta um exemplo disso.

Código 05. Exemplo de ponteiros com vetor

```
01  #include <stdio.h>
02  void main(){
03    int vet[3] = {10,20,30};
04    int *ptr;
05    ptr = vet;
06    printf("Valor de vet[0]: %d\n",vet[0]);
07    printf("Valor de *ptr: %d\n",*ptr);
08    ptr = &vet[0];
09    printf("Valor de vet[0]: %d\n",vet[0]);
10    printf("Valor de *ptr: %d\n",*ptr);
11    ptr = &vet[2];
12    printf("Valor de vet[2]: %d\n",vet[2]);
13    printf("Valor de *ptr: %d\n",*ptr);
14  }
```

Fonte: elaborado pelo autor.

No Código 5, mais precisamente na linha 3 é declarado um vetor com 3 posições do tipo int e são atribuídos os valores 10, 20 e 30 nas posições 0, 1 e 2, respectivamente. Em seguida, na linha 4 é declarado um ponteiro chamado ptr do tipo inteiro que na linha 5 recebe o endereço do vetor, ou seja, receberá o endereço do primeiro elemento do vetor. A Tabela 6 ilustra como ficaria os valores das variáveis do Código 5 até a linha 10.

Tabela 06. Exemplo da memória para as variáveis no Código 5 até a linha 10.

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FE08	int *ptr	62FE10
62FE10	int v[0]	10
62FE14	int v[1]	20
62FE18	int v[2]	30

Fonte: elaborada pelo autor.

Na Tabela 6, o ponteiro está armazenando o endereço do primeiro elemento do vetor, pois na linha 5 o ponteiro recebe o endereço do vetor, após isso, nas linhas 6 e 7, são mostrados os valores da posição 0 do vetor e o valor do conteúdo apontado por ptr. Na linha 8, é inserido o mesmo endereço (do primeiro elemento do vetor), só que dessa vez de outra forma, com isso, os mesmos valores apresentados na tela nas linhas 6 e 7 serão os mesmos apresentados nas linhas 9 e 10. Já na linha 11, o ponteiro ptr recebe o endereço da posição 2 do vetor. Observe na Tabela 6, que cada posição do vetor tem um endereço em memória, sendo ele contínuo, uma sequência de 4 bytes em 4 bytes, pois o tipo inteiro tem o tamanho de 4 bytes. Após a mudança do endereço contido no ponteiro ptr, os valores mostrados nas linhas 12 e 13 será o valor da posição 2 do vetor nas duas linhas. Com isso, se considerar esse cenário (informações da Tabela 6), será mostrado na tela oriundo das linhas 6 e 7, 9 e 10 e também 12 e 13 os valores:

```
Valor de vet[0]: 10
Valor de *ptr: 10
Valor de vet[0]: 10
Valor de *ptr: 10
Valor de vet[2]: 30
Valor de *ptr: 30
```

Para facilitar esse descolamento entre as posições de memória são utilizadas algumas operações aritméticas (incremento, decremento, diferença e comparação) sobre os ponteiros. Para auxiliar na compreensão desse assunto é interessante mostrar a função sizeof() que ao passar por parâmetro o tipo da informação é retornado o tamanho desse tipo informado em bytes, ou seja, um número inteiro positivo. Essa função tem grande utilidade quando está acessando diretamente a memória principal, para descobrir o tamanho que foi reservado em memória de acordo com seu tipo. Veja o Código 6 que mostra o funcionamento da função sizeof() em quatros tipos da linguagem C.

Código 06. Exemplo da função sizeof

```
01  #include <stdio.h>
02  void main(){
03     printf("int: %i\n",sizeof(int));
04     printf("float: %i\n",sizeof(float));
05     printf("char: %i\n",sizeof(char));
06     printf("double: %i\n",sizeof(double));
07     printf("int*: %i\n",sizeof(int*));
08  }
```

Fonte: elaborado pelo autor.

No Código 6, foram passados por parâmetros os principais tipos da linguagem C, sendo mostrado os seguintes valores na tela:

```
int: 4
float: 4
char: 1
double: 8
int*: 8
```

Observe pela saída do programa que as variáveis possuem tamanhos diferentes e que uma variável pode ter tamanho diferente de um ponteiro do mesmo tipo como ocorre nas linhas 3 e 7 que apresentam o tamanho em bytes de uma variável inteira e um ponteiro do tipo inteiro.

Passado esse conceito sobre os tamanhos em bytes dos tipos por meio da função sizeof(), será mostrado no Quadro 1 alguns operadores já utilizados e outros que serão demonstrados ao longo desse texto.

Quadro 01. Operadores de ponteiros

OPERAÇÃO	EXEMPLO	DESCRIÇÃO
Atribuição	ptr=&x	Recebe o endereço de x.

OPERAÇÃO	EXEMPLO	DESCRIÇÃO
Incremento	ptr=ptr+2	Soma ao endereço atual com o valor de dois espaços do tamanho (em bytes) do tipo do ponteiro.
Decremento	ptr=ptr-3	Subtrai ao endereço atual com o valor de três espaços do tamanho (em bytes) do tipo do ponteiro.
Apontado por	*ptr	Acessa o valor apontado por ptr.
Endereço de	&ptr	Obtém o endereço de ptr.
Diferença	ptr1 – ptr2	Diferença entre os endereços contidos nos ponteiros, portanto, a quantidade de elementos presente entre os endereços.
Comparação	ptr1 > ptr2	Compara os valores de seus endereços, verificando se o endereço armazenado em ptr1 é maior que o endereço armazenado em ptr2.

Fonte: elaborado pelo autor.

Para exemplificar os primeiros operadores observe o Código 7 que mostra o incremento e decremento de um ponteiro.

Código 07. Exemplo de incremento e decremento de operadores

```
01  #include <stdio.h>
02  void main(){
03     float vet[5]={1.2,5.4,8.9,2.3,0.7};
04     float *p=vet;
05     printf("Valor da posicao 0: %f\n",*p);
06     printf("Endereco da posicao 0: %p\n",p);
07     p=p+2;
08     printf("Valor da posicao 2: %f\n",*p);
09     printf("Endereco da posicao 2: %p\n",p);
10     p++;
11     printf("Valor da posicao 3: %f\n",*p);
```

```
12      printf("Endereco da posicao 3: %p\n",p);
13      printf("Valor da posicao 4: %f\n",*(p+1));
14      printf("Endereco da posicao 4: %p\n",p+1);
15      printf("Valor da posicao 3 subtraindo 1: %f\n",*p-1);
16      printf("Endereco da posicao 2: %p\n",p-1);
17  }
```

Fonte: elaborado pelo autor.

Observe que no Código 7 é declarado um vetor de nome vet do tipo *float* com cinco posições e já atribuído valores em cada posição do vetor. Na linha 4 é declarado um ponteiro p que recebe o endereço do vetor (primeira posição do vetor). Para exemplificar com valores os endereços veja a Tabela 7 que contém os conteúdos e endereços das variáveis utilizada no Código 7 até a linha 6.

Tabela 07. Exemplo da memória para as variáveis no Código 7 até a linha 6

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FD08	int *p	62FE00
62FE00	int vet[0]	1.200000
62FE04	int vet[1]	5.400000
62FE08	int vet[2]	8.900000
62FE0C	int vet[3]	2.300000
62FE10	int vet[4]	0.700000

Fonte: elaborado pelo autor.

Na Tabela 7, observe que desde o primeiro endereço do vetor (posição 0), são espaçados os endereços de 4 em 4, pois um valor inteiro tem 4 bytes de tamanho. Se executar o Código 5, com o cenário da Tabela 7 teremos a seguinte saída:

```
Valor da posicao 0: 1.200000
Endereco da posicao 0: 62FE00
Valor da posicao 2: 8.900000
```

```
Endereco da posicao 2: 62FE08
Valor da posicao 3: 2.300000
Endereco da posicao 3: 62FE0C
Valor da posicao 4: 0.700000
Endereco da posicao 4: 62FE10
Valor da posicao 3 subtraindo 1: 1.300000
Endereco da posicao 2: 62FE08
```

Observe que as duas primeiras linhas mostradas são referentes a posição zero, porque o ponteiro está armazenando o endereço da posição zero. Logo após, na linha 7 é atribuído ao ponteiro p o valor atual do ponteiro somado a 2, com isso, ele deslocará para a posição 2 do vetor (endereço 62FE08), isso ocorre porque um inteiro (tipo do ponteiro) tem um tamanho de 4 bytes, dessa forma, cada número inteiro adicionado é multiplicado por 4 para se chegar no endereço desejado. Desse jeito, o ponteiro armazenando o endereço da posição 2 do vetor, nas linhas 8 e 9 são mostrados os dados referentes a essa posição. Adiante, na linha 10 é feito um incremento de um no valor atual do ponteiro, dessa forma, ele passa a ter o valor 62FE0C, isso acontece pois, foi somado o valor 62FE08 com mais 4, como o valor da memória está em hexadecimal (0 a F), chega-se no valor 62FE0C, ou seja, na posição 3 do vetor, logo, nas linhas 11 e 12 são mostrados os valores referentes a posição 3 do vetor.

Já na linha 13 é mostrado o conteúdo contido na posição 4 do vetor, isso ocorre porque o ponteiro está no endereço 62FE0C (posição 3) e é somado o valor 1, atente-se que ao executar esse comando o endereço armazenado no ponteiro não se altera, porque não houve a atribuição, somente a soma. Outro ponto importante a ser considerado é que foi colocado a operação de soma entre parênteses e o asterisco fora e também antes dos parênteses, para que ocorresse primeiro a soma do endereço e depois se coletasse o conteúdo no endereço apontado. Na linha 14, como não se tem o asterisco, nem os parênteses, é mostrado o endereço 62FE10 (o endereço 62FE0C somado a mais 4 bytes), mostrando o endereço da posição 4 do vetor.

Ao final do Código 5, na linha 15 é mostrado a subtração do conteúdo da posição 3 (endereço que o ponteiro está armazenando) e o valor, fazendo assim 2.3 – 1 e obtendo o resultado de 1.3. Considere que nesta linha 15 não foi inserido os parênteses, com isso, não ocorreu o deslocamento da posição de memória como ocorreu na linha 13. Já na linha 16 é mostrado o endereço da posição 2 do vetor, esse valor é resultado do deslocado feito com a subtração do endereço armazenado no ponteiro e 4 bytes (endereço 62FE0C menos 4 bytes), chegando no endereço 62FE08.

Para exemplificar os demais operadores restantes analise o Código 8 que mostra a diferença e comparação de um ponteiro.

Código 08. Exemplo de comparação e diferença de operadores

```
01  #include <stdio.h>
02  void main(){
03     char a='x',b='k';
04     char *p_a=&a,*p_b=&b;
05     if(p_a>p_b){
06        printf("p_a > p_b\n %p > %p\n",p_a,p_b);
07     }else{
08        printf("p_b > p_a\n %p > %p\n",p_b,p_a);
09     }
10     printf("Diferenca: %p\n",p_a-p_b);
11  }
```

Fonte: elaborado pelo autor.

Atente-se que no Código 8 é declarado na linha 3 a variável "a" e atribuído o valor 'x' e a variável "b" e atribuído o valor "k", ambas do tipo char. Na linha 4 são declarados o ponteiro p_a que é atribuído o endereço da variável "a" e o ponteiro p_b que é atribuído o endereço da variável "b". Para a melhor compreensão das linhas adiante, analise a Tabela 8 que traz uma exemplificação dos endereços de memória para as variáveis do Código 8.

Tabela 08. Exemplo da memória para as variáveis no Código 8

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FD08	char *p_b	62FE1E
62FE10	char *p_a	62FE1F
62FE1E	char b	k
62FE1F	char a	x

Fonte: elaborada pelo autor.

Na Tabela 8, observe que desde o endereço armazenado no ponteiro p_a é maior que o endereço armazenado no ponteiro p_b, por isso, se considerar esse cenário (informações da Tabela 8), será mostrado na tela as seguintes informações:

```
p_a > p_b

62FE1F > 62FE1E

Diferenca: 1
```

Atente-se que a comparação realizada na linha 5 é entre os endereços e não os conteúdos que apontam p_a e p_b, dessa forma, se os valores se alterassem, mas os endereços continuarem os mesmos, o fluxo não se alteraria. Caso deseje fazer a comparação entre os conteúdos que apontam esses ponteiros, basta inserir um asterisco antes do seu nome. Na linha 10 é mostrada a diferença dos ponteiros, novamente considerando os endereços contidos nos ponteiros, mostrando que a diferença é de 1, pois o valor de um char tem o tamanho de 1 byte.

Outros operadores aritméticos que podem ser utilizados com ponteiros são -= e += que estão sendo exemplificados no Código 9.

Código 09. Exemplo de outros operadores aritméticos com ponteiros

```
01  #include <stdio.h>
02  void main(){
03     double vet[5]={1.2,5.4,8.9,2.3,0.7};
04     double *p=vet;
05     p+=2;
06     printf("Valor da posicao 2: %lf\n",*p);
07     p-=1;
08     printf("Valor da posicao 1: %lf\n",*p);
09  }
```

Fonte: elaborado pelo autor.

No Código 9 é declarado um vetor do tipo *double* com 5 posições e já é atribuído valores em cada posição, em seguida, na linha 4 é declarado um ponteiro p que recebe o endereço do vetor vet. Na linha 5 é realizada uma abreviação por meio do operador += do comando p=p+2, ou seja, é realizado a soma do valor atual do endereço do ponteiro p e o valor de dois espaços de uma variável *double* em bytes, com isso, o endereço do ponteiro ficará na posição 2 do vetor, por isso, na linha 6 é mostrado o conteúdo que é apontado no endereço da posição 2 do vetor (valor 8.9). Na linha 7 é realizada uma abreviação por meio do operador -= do comando p=p-1, ou seja, é realizado a subtração do valor atual do endereço do ponteiro p e o valor de um espaço de uma variável *double* em bytes, com isso, o endereço do ponteiro ficará na posição 1 do vetor (estava na posição 2), por isso, na linha 8 é mostrado o conteúdo que é apontado no endereço da posição 1 do vetor (valor 5.4).

DESAFIO

Faça um programa que colete 20 valores do tipo *double* e armazene esses valores em cada posição de um vetor. Na sequência, crie um ponteiro que receba o endereço da primeira posição desse vetor e por meio dos recursos da aritmética de ponteiros, mostre na tela todos os dados desse vetor e o endereço de cada posição usando o ponteiro (acesso indireto).

3. PARÂMETROS DE FUNÇÕES

Como vimos em outros momentos as funções podem ter variáveis em seu cabeçalho que são as entradas dos dados para a função chamadas de parâmetros formais. Como os ponteiros são variáveis especiais, é possível também declarar um ponteiro como um parâmetro formal de uma função. Mas porque passar o endereço de memória pode ser útil? Analise o problema de criar uma função troca, onde o objetivo é trocar os valores entre as variáveis passadas por parâmetro, o Código 10 apresentado tenta fazer isso, mas não consegue.

Código 10. Função troca sem ponteiro

```
01  #include <stdio.h>
02  void troca(int a, int b){
03          int temp;
04          temp=a;
05          a=b;
06          b=temp;
07  }
08  void main(){
09     int num1=10,num2=20;
10     troca(num1,num2);
11     printf("num1: %i\n",num1);
12     printf("num2: %i\n",num2);
13  }
```

Fonte: elaborado pelo autor.

No Código 10 foi criada uma função chamada de troca que recebe como parâmetro duas variáveis inteiras (a e b) com objetivo de trocar seus valores entre si. Na linha 3 do código é criada uma variável local do tipo inteiro chamada “temp”, com o objetivo de ser uma variável temporária para auxiliar na troca dos valores. Com isso, na linha 4 é atribuído o valor da variável “a” para “temp”, em seguida, na linha 5 é atribuído o valor da variável “b” para “a”, por fim, na linha 6 é atribuído o valor da variável “temp” para “b”.

Se resgatarmos a informação que variáveis locais e parâmetros formais são criadas ao iniciar a execução da função e destruídas após a execução da mesma, todas as trocas de informações realizadas durante a função será perdida, pois as variáveis deixarão de existir. Portanto, os valores atribuídos nas variáveis num1 e num2 na linha 9 permanecerão sem alteração mesmo após a execução da linha 10 que chama a função troca.

O problema maior neste caso é que os valores das variáveis criados através dos parâmetros formais na função são armazenados em locais diferentes, ou seja, é feita uma cópia dos valores de “num1” e “num2” para as variáveis “a” e “b”, mas qualquer alteração realizada nas variáveis “a” e “b” não apresentam alteração nas variáveis locais “num1” e “num2”. Uma solução rápida, mas não recomendada é utilizar variáveis globais, mas lembrando que as variáveis globais são utilizadas em casos específicos. Para esse caso tem-se uma solução utilizando ponteiros com pequenas alterações, como é apresentado no Código 11.

Código 11. Função troca com ponteiro

```
01  #include <stdio.h>
02  void troca(int *a, int *b){
03          int temp;
04          temp=*a;
05          *a=*b;
06          *b=temp;
07  }
08  void main(){
09     int num1=10,num2=20;
10     troca(&num1,&num2);
11     printf("num1: %i\n",num1);
12     printf("num2: %i\n",num2);
13  }
```

Fonte: elaborado pelo autor.

Como pode-se perceber no Código 11, ocorreram alterações pontuais em relação ao Código 10, sendo que foram alterados os parâmetros da função de variáveis do tipo inteira para ponteiros do tipo inteiro. Havendo a troca de variável para ponteiro, as linhas 4 a 6 foi necessário acrescentar um asterisco antes do nome do ponteiro, fazendo que seja copiado e atribuído o conteúdo apontando pelo ponteiro “a” e “b”. Outra parte que teve alteração foi a linha 10, que agora passa por parâmetro o endereço das variáveis “num1” e “num2” por meio do operador &.

Essas alterações fazem com que seja enviado por parâmetro o endereço da variável, permitindo que a função faça a alteração indiretamente dos valores através do endere-

ço contido nos ponteiros, portanto, ao final da execução da função os valores alterados não serão perdidos.

Neste exemplo não foi utilizado o retorno da função (estava *void*), porque era necessário a alteração de dois valores e uma função na linguagem C só pode retornar uma informação do tipo informado, entretanto, se utilizar o ponteiro no parâmetro da função ele pode servir como entrada de dados, mas também como saída de dados. Isso torna-se mais claro no Código 12, onde é realizado o cálculo da área e do perímetro de um hexágono.

Código 12. Função calcula área e perímetro do hexágono

```
01  #include <stdio.h>
02  void calc_hexa(float L, float *area, float *perimetro){
03          *area=(3*L*L*sqrt(3))/2;
04          *perimetro=6*L;
05  }
06  void main(){
07     float a,p;
08     calc_hexa(2,&a,&p);
09     printf("Area: %f\n",a);
10     printf("Perimetro: %f\n",p);
11  }
```

Fonte: elaborado pelo autor.

No Código 12 foi criada uma função calc_hexa que recebe como parâmetro a medida do lado por meio da variável "L", em seguida, os ponteiros area e perímetro, todos do tipo *float*. Neste caso o ponteiro area e perímetro tem a papel de retorno da função, pois armazena o resultado do cálculo da área e do perímetro. Atente-se que na linha 7 as variáveis "a" e "p" do tipo *float* somente foram declaradas, não sabendo o valor que estão dentro dela, em seguida, na linha 8 foi passado o endereço delas por meio de parâmetro para a função calc_hexa para que seja armazenado por meio de uma atribuição indireta os resultados do cálculo da área e do perímetro. Dessa maneira, as linhas 9 e 10 mostrarão após a execução da função os valores da área e do perímetro referente a medida 2 passada na chamada da função.

Assim como as funções podem receber como parâmetros os ponteiros, elas podem também retornarem um ponteiro, como está sendo exemplificado no Código 13 que apresenta uma função para encontrar o número maior e retorna o endereço de onde está armazenado o maior número.

Código 13. Função maior número com retorno de um ponteiro

```
01  #include <stdio.h>
02  float* maior(float *n1, float *n2){
03         if(*n1>*n2) return n1;
04         else return n2;
05  }
06  void main(){
07     float num1=3,num2=4;
08     float *p;
09     p=maior(&num1,&num2);
10     printf("Maior: %f\n",*p);
11  }
```

Fonte: elaborado pelo autor.

No Código 13 foi criada uma função maior que recebe como parâmetro dois ponteiros do tipo *float* e retorna um ponteiro do tipo *float*. Na linha 3 é verificado se o conteúdo apontado por n1 é maior que o conteúdo apontado por n2, se sim é retornado o endereço armazenado no ponteiro n1, caso contrário, retorna o endereço armazenado no ponteiro n2. No programa principal é declarado duas variáveis do tipo *float* e atribuído valores nelas, na linha 8 é declarado um ponteiro do tipo *float* chamado "p". Já na linha 9 é chamada a função maior passando como parâmetro o endereço das variáveis e armazenando o retorno no ponteiro "p", que nesse caso será o endereço da variável num2 e será mostrado através do comando da linha 10 o maior valor que está no conteúdo apontado por p.

DESAFIO

Faça uma função que calcule as raízes de uma equação do segundo grau do tipo $ax^2 + bx + c=0$. Sabendo que:

$$\Delta = b^2 - 4ac \qquad X = \frac{-b \pm \sqrt{\Delta}}{2a}$$

Atente-se aos seguintes aspectos:

- a variável a deve ser diferente de zero.
- $\Delta < 0$ não possui raiz.
- $\Delta = 0$ possui uma raiz.
- $\Delta > 0$ possui duas raízes.

A função deve obedecer ao cabeçalho:

```
int raizes(float a, float b, float c, float *x1, float *x2);
```

E ter como valor de retorno o número de raízes reais e distintas da equação (0, 1 ou 2). Caso exista, os seus valores devem ser armazenados nas variáveis apontadas por x1 e x2.

4. PONTEIRO PARA PONTEIRO

O ponteiro para ponteiro, ou também é conhecido como apontamento múltiplo ou indireção múltipla, é uma técnica que visa ter um ponteiro que aponta para outro ponteiro, que pode apontar para uma variável ou outro ponteiro, não existindo limite para sua definição (MANZANO, 2015, p. 137).

Para indicar um ponteiro para ponteiro usa-se asteriscos à frente do nome do ponteiro. A forma mais usada são dois asteriscos (**) que significa “ponteiro de” e determina dois níveis de referência indireta de um ponteiro. Esse recurso permite fazer um efeito cascata entre ponteiros, tornando-se possível ter um extenso conjunto de ponteiro para ponteiros em diversos níveis. Isso ocorre no Código 14 de uma forma exemplificada.

Código 14. Ponteiro para ponteiro em diversos níveis

```
01  #include <stdio.h>
02  void main(){
03     int num=25;
04     int *p1 = NULL;
05     int **p2 = NULL;
06     int ***p3 = NULL;
07     int ****p4 = NULL;
08     p1 = &num;
09     p2 = &p1;
10     p3 = &p2;
11     p4 = &p3;
12  }
```

Fonte: elaborado pelo autor.

Observe que no Código 14, foi declarada uma variável inteira chamada num que é atribuído o valor 25 (vinte e cinco). Na linha 4, declarou-se um ponteiro chamado p1 que recebe o valor nulo como atribuição. Na linha 5, declarou-se um ponteiro para ponteiro titulado p2, ou seja, ele é um ponteiro de ponteiro que também é atribuído o valor nulo. Nas linhas 6 e 7 são declarados ponteiros para ponteiro com outros níveis. Já na linha

8 é atribuído ao ponteiro p1 o endereço da variável num. Em seguida, na linha 9, o ponteiro p2 recebe o endereço do ponteiro p1. Na linha 10 o ponteiro p3 recebe o endereço do ponteiro p2 e por fim, o ponteiro p4 recebe o endereço do ponteiro p3, provocando assim um efeito cascata de ponteiro, isso pode ser visualizado na Tabela 9, que ilustra as variáveis do Código 14.

Tabela 09. Exemplo da memória para as variáveis no Código 14

ENDEREÇO	VARIÁVEL	CONTEÚDO
62FD08	int *p4	62FE00
62FE00	int *p3	62FE08
62FE08	int *p2	62FE10
62FE10	int *p1	62FE1C
62FE1C	int num	25

Fonte: elaborado pelo autor.

Na Tabela 9, pode-se ver que o ponteiro p1 tem um nível de referência indireta, na sequência, o ponteiro p2 tem dois níveis de referência indireta, assim por diante.

Mesmo sendo algum muito abstrato e ainda com pouca utilidade até o momento, o ponteiro para ponteiro possui aplicação em matrizes alocadas dinamicamente na memória que é alguns assuntos que não é foco nesta ocasião, mas esta informação é apresentada aqui justamente para demonstrar a sua possibilidade de uso em ponteiros.

CONCLUSÃO

Os ponteiros trazem para o programador o recurso de acesso indireto a memória principal e com isso a possibilidade de retornar mais de um valor em uma função por meio de ponteiros declarados como parâmetros formais no cabeçalho da função, assim como o deslocamento em um vetor por meio do endereço de cada posição do vetor.

O uso do ponteiro adiciona possibilidade a ser explorada pelo desenvolver, mas com muito zelo, pois assim como o uso do ponteiro permite acessar a memória ou até coletar endereço ou outras informações que não eram conhecidas, também pode acessar posições de memória inválidas ou com dados que não conferem com o desejado. Portanto, sempre faça um teste rigoroso em seus programas desenvolvidos com ponteiros, pois pequenos deslizes podem acarretar em inconsistência nos dados gravados ou coletados da memória.

REFERÊNCIAS BIBLIOGRÁFICAS

BACKES, A. **Linguagem C - Completa e Descomplicada**. 2. ed. Rio de Janeiro: LTC, 2022.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015.

UNIDADE 4

MANIPULAÇÃO DE STRING E ARQUIVOS

INTRODUÇÃO

Cada vez mais as informações que estavam em papéis estão sendo digitalizadas, ou seja, o volume de dados digitais está aumentando rapidamente, por isso, manipular esses dados armazenados em diversos locais é algo necessário em qualquer setor da atividade econômica. Para manipular essas informações é importante saber que é necessário em muitas vezes fazer conversões de dados, busca por palavras ou caracteres, fazer comparações, copiar somente uma parte desse texto, entre outras exigências que podem surgir, por isso, serão mostradas as principais funções para realizar essas tarefas. Além disso, nada adianta fazer essas manipulações somente na memória e após o fechamento do programa esse processo se perder, por isso, fazer a escrita e leitura desses dados em dispositivos de armazenamento também é algo importante.

1. MANIPULAÇÃO DE STRING

> *String* é o nome que usamos para definir uma sequência de caracteres adjacentes na memória do computador. Essa sequência de caracteres, que pode ser uma palavra ou frase, é armazenada na memória do computador na forma de um *array* do tipo *char.* (BACKES, 2022, p. 130)

Com isso, na linguagem C uma *string* é um vetor (ou *array*) de caracteres, que segue as mesmas regras de um vetor tradicional, indicando entre colchetes após o nome a quantidade de elementos que podem ser armazenados no vetor, como pode está sendo mostrado no Código 1.

Código 01. Exemplo de declaração de *strings*

```
01  #include <stdio.h>
02  void main(){
03     char str1[10];
04     char str2[10];
05     char str3[]=”abcde”;
06     char str4[]={‘a’,’b’,’c’,’d’,’e’};
07     char *str5=”abcde”;
08     printf(“Digite uma string: “);
```

```
09      setbuf(stdin,NULL);
10      scanf("%s",str1);
11      printf("Digite outra string: ");
12      setbuf(stdin,NULL);
13      gets(str2);
14      printf("%s\n",str1);
15      printf("%s\n",str2);
16      putchar(str3[1]);
17      putchar('\n');
18      puts(str3);
19      puts(str4);
20      puts(str5);
21  }
```

Fonte: elaborado pelo autor.

No Código 1, exatamente nas linhas 3 e 4 foram declaradas duas *strings* (str1 e str2) que armazenam 10 caracteres cada uma. Um cuidado que deve ser tomado quando se trabalha com *string* na linguagem C é que as *strings* necessitam armazenar um caractere '\0' (barra zero) que representa o fim da sequência de caracteres, esse elemento vem em sequência da última letra da frase/palavra armazenada. Dessa forma, se armazenar uma palavra com quatro caracteres o quinto caractere será o '\0', com isso, essas *strings* devem armazenar no máximo 9 caracteres cada uma, pois o décimo caractere será o '\0'.

Os comandos scanf (na linha 10) e gets (na linha 13) que coletam o que o usuário digitou e armazena nas *strings* (str1 e str2) já fazem a inserção do caractere '\0' automaticamente, mas é necessário que a quantidade de caracteres digitados pelo usuário somando a esse caractere seja igual ou menor a quantidade de elementos da *string* (vetor de caracteres). Caso contrário acontecerá um overflow, em outras palavras, a quantidade de espaço reservado não é suficiente para armazenar as informações.

Em alguns casos, a leitura via teclado de caracteres ou *strings* pode ocasionar erros, dessa forma, para resolver isso é necessário inserido antes dos comandos scanf e gets o comando setbuf(stdin,NULL) para limpar o buffer do teclado, como pode ser visto nas linhas 9 e 12. Ambos os comandos possuem a mesma função, mas o que diferencia os dois é que o comando scanf faz a leitura de *strings* digitadas sem espaços, já o comando gets faz a leitura de *strings* digitadas com espaços, por exemplo, somente o comando gets consegue fazer a leitura de um nome completo e armazenar em uma *string* considerando que um nome completo tem o nome e o sobrenome separados por espaço.

SAIBA MAIS

Alguns programadores utilizam a função `fflush()` para limpar buffer de entrada (stdin), mas alguns documentações indicam que seu uso para esse fim pode causar comportamento indefinido, pois a finalidade correta dessa função é limpar o buffer de saída e mover os dados do buffer para o console ou disco, logo, se não deseja ter problemas de portabilidade, não use essa função para essa finalidade.

Continuando no Código 1, as linhas 5 a 7, mostram diferentes formas de declarar *strings* e atribuir valores. Na linha 5 é declarado a *string* str3 e atribuído a *string* "abcde", repare que nesse caso não foi informado a quantidade de posições dentro dos colchetes, mas nesta situação o caractere '\0' não é inserido automaticamente (é indispensável inserir ele antes de fechar as aspas). Se o tamanho for informado, deve-se considerar a quantidade de caracteres e mais um espaço para que o caractere '\0' seja inserido automaticamente. Na linha 6 é declarado a *string* str4, mas diferentemente da linha 5, os dados são inseridos caracteres por caractere em cada posição, repare que quando usa-se caracteres eles estão entre aspas simples e quando usa-se *string* ela está entre aspas duplas. Mas tanto na linha 5 como na linha 6, o caractere '\0' não é inserido automaticamente. Já na linha 7, a *string* é declarada como um ponteiro, pois um vetor de caracteres armazena exatamente o endereço da primeira posição do vetor, por isso também funciona igualmente as declarações anteriores.

Na linha 14 é mostrado a *string* str1 e pula-se uma linha por meio do comando printf, isso acontece na linha 15 com a *string* str2. Já na linha 16, mostra-se através do comando putchar o caractere na posição 1 da *string* str3 (caractere 'b'), na sequência, na linha 17 é usado novamente o comando para pular uma linha usando o caractere '\n'. Por fim, nas linhas 18 a 20 são mostradas em cada linha os valores das três *strings* (str3, str4 e str5).

Como foi mostrado no Código 1, para se manipular *strings* é necessário usar algumas funções, dessa forma, o Quadro 1 apresenta algumas funções que já foram utilizados e outros que serão utilizados nos próximos códigos.

Quadro 01. Relação de funções úteis para manipulação de *strings*.

BIBLIOTECA	FUNÇÃO	DESCRIÇÃO
stdio.h	scanf	Lê os dados de entrada padrão (teclado) e os armazena de acordo com o formato do parâmetro nos locais apontados pelos argumentos adicionais. A entrada não pode ter espaço em branco, quando se trata de uma *string*.
	printf	Grava uma *string* na saída padrão (tela), caso possua especificadores de formato, os argumentos adicionais após o formato são formatados e inseridos na *string* resultante substituindo seus respectivos especificadores.
	gets	Lê os caracteres da entrada padrão (teclado) e os armazena como uma *string* até que um caractere de nova linha ou o final do arquivo seja alcançado.
	puts	Grava uma *string* na saída padrão (tela) e acrescenta um caractere de nova linha ('\n').

BIBLIOTECA	FUNÇÃO	DESCRIÇÃO
stdio.h	getchar	Retorna o próximo caractere da entrada padrão (teclado).
	putchar	Grava um caractere na saída padrão (tela).
string.h	strcpy	Copia uma *string* para outra.
	strncpy	Copia uma quantidade de caracteres de uma *string* para outra.
	strcmp	Compara e determina a ordem (alfabética) de duas *strings*.
	strncmp	Compara uma quantidade de caracteres de duas *strings* e determina a ordem (alfabética) das duas.
	strcat	Concatena (junta) uma *string* para outra *string*.
	strncat	Concatena (junta) uma quantidade de caracteres de uma *string* para outra *string*.
	strlen	Retorna o número de caracteres na *string* (não conta o '\0')
	strchr	Localiza a primeira ocorrência de um caractere na *string*.
	strrchr	Localiza a última ocorrência de um caractere na *string*.
	strstr	Localiza a primeira ocorrência de uma string em outra *string*.
	strpbrk	Retorna um ponteiro para a primeira ocorrência em uma *string* de qualquer caractere de contido em outra *string*.
	strtok	Quebra uma *string* numa sequência de tokens delimitados por um ou mais caracteres de outra *string*.
	strlwr	Converte todos os caracteres da *string* para minúsculas.
	strupr	Converte todos os caracteres da *string* para maiúsculas.
stdlib.h	atoi	Converte uma *string* em um número inteiro.
	atof	Converte uma *string* em um número ponto flutuante (*float*).
	strtod	Converte uma *string* em um número ponto flutuante de precisão dupla (*double*).

BIBLIOTECA	FUNÇÃO	DESCRIÇÃO
ctype.h	isdigit	Verifica se o caractere é numérico.
	isalpha	Verifica se o caractere é uma letra do alfabeto.
	islower	Verifica se o caractere é minúsculo.
	isupper	Verifica se o caractere é maiúsculo.
	isspace	Verifica se o caractere é um espaço em branco.
	toupper	Converte o caractere para maiúsculo.
	tolower	Converte o caractere para minúsculo.

Fonte: elaborado pelo autor.

No Quadro 1 são apresentadas funções para coleta por meio do teclado de *strings* ou um único caractere, assim como saída de dado (mostrar na tela). Há também para manipulação de *strings* como comparar, copiar, concatenar, localizar caracteres ou *substrings*. É mostrado alguns validadores de caracteres e conversores de dados.

SAIBA MAIS

Os símbolos utilizados nos computadores como letras, números, pontuações, entre outros são armazenados na memória em códigos binários. Para que se tenha um código único para cada símbolo são criados padrões como EBCDIC, ASCII, ISSO-8859, UTF-8, UTF-16, etc. Na linguagem C o padrão utilizado é o ASCII (*American Standard Code for Information Interchange*).

Neste padrão, cada caractere ocupa 1 byte (8 bits) e existem 256 símbolos diferentes, dessa forma, uma variável do tipo char guarda exatamente 1 caractere.

O Código 2 apresenta um programa que coleta nome, sobrenome, data de nascimento, e-mail e sexo, em seguida, concatena o nome e sobrenome e também faz simples validações na data de nascimento, no e-mail e no sexo.

Código 02. Exemplo de manipulação e validação de *strings*.

```
01  #include <stdio.h>
02  #include <string.h>
03  void main(){
04          char nome[100],sobrenome[100],nomecompleto[200];
05          char datanasc[11],email[100],sexo[10];
06          printf("Digite o nome: ");
07          setbuf(stdin,NULL);
08          gets(nome);
09          printf("Digite o sobrenome: ");
10          setbuf(stdin,NULL);
11          gets(sobrenome);
12          printf("Digite a data de nasc. (DD/MM/AAAA): ");
13          setbuf(stdin,NULL);
14          gets(datanasc);
15          printf("Digite o e-mail: ");
16          setbuf(stdin,NULL);
17          gets(email);
18          printf("Digite o sexo: ");
19          setbuf(stdin,NULL);
20          gets(sexo);
21          strcpy(nomecompleto,nome);
22          strcat(nomecompleto," ");
23          strcat(nomecompleto,sobrenome);
24          printf("Nome: %s\n",nomecompleto);
25          if(strlen(datanasc)!=10) printf("Data de nasc.: invalida!\n");
26          else printf("Data de nasc.: %s\n",datanasc);
27          if(strchr(email,'@')==NULL) printf("E-mail: invalido!\n");
28          else printf("E-mail: %s\n",email);
29          if(strcmp(sexo,"Masculino")!=0 && strcmp(sexo,"Feminino")!=0)
30           printf("Sexo: invalido!\n");
31          else printf("Sexo: %s\n",sexo);
32  }
```

Fonte: elaborado pelo autor.

No código 2, verifique que na linha 2 foi incluído a biblioteca string.h para que algumas funções de manipulação de *strings* sejam reconhecidas pelo compilador. Nas linhas 4 e 5 são declaradas seis *strings* que serão utilizadas ao longo do programa. Entre as linhas 6 e 20 é realizada a coleta dos dados digitados pelo usuário (nome, sobrenome, data de nascimento, e-mail e sexo) através do comando gets, mas antes de cada coleta é feita a limpeza do buffer (memória) do teclado (stdin) para evitar possíveis erros.

Na linguagem C, a manipulação das *strings* é diferente das outras variáveis, por isso, comparação e atribuição necessitam de funções específicas, para realizar uma atribuição em qualquer outra variável bastava utilizar o símbolo de igual (=), mas em *strings* é necessário utilizar a função strcpy. Na linha 21 é realizado por meio da função strcpy a cópia do texto da *string* nome para a *string* nomecompleto, ou seja, o equivalente a uma atribuição da *string* nome para a *string* nome completo. Na linha 22, é concatenado com o texto contido na *string* nomecompleto um espaço em branco, na sequência, na linha 23, é concatenado o texto contido na *string* nomecompleto e a *string* sobrenome, fazendo com que a *string* nomecompleto, que é mostrada na linha 24, tenha o nome e sobrenome separado por espaço em branco. Observe que na linha 4 a *string* nomecompleto foi declarada com o dobro do nome e sobrenome para ter espaço suficiente para armazenar o nome e sobrenome.

Na linha 25 é verificado o comprimento da *string* datanasc é diferente de 10, pois o formato solicitado (DD/MM/AAAA) possui 10 caracteres, caso esse comprimento (quantidade de caracteres) não seja 10 (dez) será mostrado uma mensagem na tela informado sua invalida, senão, será mostrado a data de nascimento.

Na linha 27 é verificado se o caractere “@” está presente na *string e-mail*, pois a função strchr retornar o endereço de onde se localizar a primeira ocorrência desse caractere, caso não encontre ele será nulo, dessa forma, se o caractere não for encontrado será mostrado uma mensagem na tela informado sua invalidade, senão, será mostrado o e-mail.

Na linha 29 é verificado se o texto digitado na *string* sexo é “Masculino” ou “Feminino”, pois a função strcmp faz a comparação entre duas *strings* e retorna zero (0) caso as *strings* sejam iguais (caracteres maiúsculos e minúsculos são considerados diferentes nesta função). Ela também é útil para se colocar as *strings* em ordem alfabética, pois ela retorna -1, quando a primeira *string* é alfabeticamente menor que a segunda, e retorna 1, quando a segunda *string* é alfabeticamente menor que a primeira, dessa forma, se o texto for diferente de “Masculino” ou “Feminino” será mostrado uma mensagem na tela informado sua invalidade, senão, será mostrado o sexo.

Neste código, as validações dos dados são simples e não evitam todos os problemas, na data de nascimento, por exemplo, não foi verificado se foi digitado um número nos espaços que estão reservados para tal, assim como não foi verificado se as barras estão no local certo, ou até, verificar se o mês ou o dia está no intervalo numérico pelo qual eles são válidos, além de outras validações possíveis. Para facilitar essas operações é possível criar funções que validam esses dados específicos e ao passar a *string* por parâmetro será retornado um valor a qual informará se é válida ou não essa informação, isso permite sua reutilização e uma divisão em partes que facilita o entendimento do código.

Analise o Código 3, que faz a coleta e validação do IP na versão 4 no formato XXX.XXX.XXX.XXX, sendo que cada conjunto de XXX deve ter um número entre 0 e 255.

Código 03. Exemplo de validação de *strings*

```
01  #include <stdio.h>
02  #include <string.h>
03  int isIp(char *s){
04     if(strlen(s)!=15){
05           return -1;
06     }
07     if(s[3]!='.' || s[7]!='.' || s[11]!='.'){
08           return -2;
09     }
10     if(isdigit(s[0])==0 || isdigit(s[1])==0 ||
11        isdigit(s[2])==0 || isdigit(s[4])==0 ||
12        isdigit(s[5])==0 || isdigit(s[6])==0 ||
13        isdigit(s[8])==0 || isdigit(s[9])==0 ||
14        isdigit(s[10])==0 || isdigit(s[12])==0 ||
15        isdigit(s[13])==0 || isdigit(s[14])==0){
16           return -3;
17     }
18     char p1[4],p2[4],p3[4],p4[4];
19     strncpy(p1,s,3);
20     p1[3]='\0';
21     strncpy(p2,s+4,3);
22     p2[3]='\0';
23     strncpy(p3,s+8,3);
24     p3[3]='\0';
25     strncpy(p4,s+12,3);
26     p4[3]='\0';
27     if(atoi(p1)<0 || atoi(p1)>255 ||
```

```
28        atoi(p2)<0 || atoi(p2)>255 ||
29        atoi(p3)<0 || atoi(p3)>255 ||
30        atoi(p4)<0 || atoi(p4)>255){
31           return -4;
32     }
33     return 1;
34  }
35  void main(){
36         char ip[16];
37         printf("Digite o IP (XXX.XXX.XXX.XXX): ");
38         gets(ip);
39         if(isIp(ip)==1) printf("IP valido!\n");
40         else printf("IP invalido!\n");
41  }
```

Fonte: elaborado pelo autor.

No Código 3, na linha 3, foi definida uma função chamada IsIp que recebe como parâmetro uma *string* e retorna um valor inteiro. Dentro da função mais especificamente entre as linhas 4 e 6 é verificado o comprimento da *string*, que no padrão definido (XXX.XXX.XXX.XXX) é necessário conter 15 caracteres, dessa forma, se ele for diferente de 15 será retornado o valor -1.

Entre as linhas 7 a 9 é verificado se os pontos estão nos locais corretos, ou seja, nas posições 3, 7 e 11, caso essas posições tenham outro caractere, será retornado o valor -2. Já entre as linhas 10 a 17 é verificado se as posições 0, 1, 2, 4, 5, 6, 8, 9, 10, 12, 13 e 14 são caracteres numéricos, se alguma dessas posições não contém um valor numérico, será retornado o valor -3.

Na linha 18 declara-se quatro *strings* (p1, p2, p3 e p4) que serão as quatro partes numéricas (XXX), lembrando que elas possuem quatro posições para armazenar três caracteres e mais o caractere '\0' que age como terminador da sequência de caracteres.

A função strncpy está sendo usada na linha 19 para copiar três caracteres da *string* s para a *string* p1, ou seja, copiando da *string* s as posições 0, 1 e 2 para a *string* p3 e na linha seguinte está sendo colocado na posição 3 (última posição) da *string* p1 o caractere '\0', pois a função não faz a inserção desse caractere automaticamente como ocorre nas funções scanf e gets.

Nas linhas 21 a 26 ocorrem situações simulares, fazendo a cópia de também três caracteres para as *strings* p2, p3 e p4, porém agora ocorre o deslocamento da posição de

memória que fica na primeira posição para outro local (soma de ponteiro), para realizar a cópia a partir da posição 4 para a *string* p2, da posição 8 para a *string* p3 e da posição 12 para a *string* p4.

Como última verificação, entre as linhas 27 a 32, é convertido as *strings* para um número inteiro por meio da função atoi e verificado se cada uma das *strings* está com o valor entre 0 e 255 (intervalo válido de um IP), caso não esteja, é retornado o valor -4. Por fim, considerando que quando ocorre um retorno na função, ela não continua as instruções seguintes, dessa forma, o retorno na linha 33 só irá acontecer se as validações anteriores tenham sido concluídas sem problema, portanto, caso o endereço IP seja válido, será retornado o valor 1.

No trecho do programa principal é declarada uma *string* chamada ip que é coletada do usuário e essa informação é passada por parâmetro para a função isIp para validação, portanto, se ela retornar o valor 1 será mostrado a mensagem de IP válido, caso contrário, será mostrado que o IP está inválido.

Na linguagem C, por causa de uma *string* ser representada por um vetor de caracteres é comum o desenvolvedor pensar na forma iterativa (utilizando estrutura de repetição) para solucionar os mais diversos casos, iniciando da primeira posição até o comprimento estabelecido pela função strlen, entretanto, é possível desenvolver funções recursivas para trabalhar com *strings*, um exemplo disso é o Código 4, que apresenta uma função recursiva que conta a quantidade de vezes que o caractere se repete na *string*.

Código 04. Exemplo de função recursiva com *string*

```
01  #include <stdio.h>
02  int conta_caractere(char *s,char c){
03        if(*s!='\0'){
04              if(*s==c) return 1+conta_caractere(s+1,c);
05              else return conta_caractere(s+1,c);
06        }else{
07              return 0;
08        }
09  }
10  void main(){
11        char c,str[100];
12        puts("Digite um caractere: ");
13        c=getchar();
14        puts("Digite um texto: ");
15        setbuf(stdin,NULL);
16        gets(str);
17        printf("O caractere %c aparece %i vezes.\n",
18                    c,conta_caractere(str,c));
19  }
```

Fonte: elaborado pelo autor.

Observe que no Código 4, na linha 2, é definida uma função conta_caractere que recebe como parâmetro uma *string* (ponteiro do tipo caractere) e um caractere e retorna um valor inteiro. Se o conteúdo contido no endereço for diferente de terminador da sequência de caracteres (\0) será verificado se o conteúdo é igual ao caractere passado por parâmetro, se sim, será somado 1 ao retorno da chamada da função com o mesmo caractere, mas agora com o deslocamento para o próximo endereço (próxima posição do vetor), caso contrário, só retorna a mesma chamada da função sem somar qualquer valor. Caso o conteúdo que é apontado por s seja igual ao terminador da sequência de caracteres, será retornado o valor zero. Em suma, a função é recursiva, pois tem como caso base quando encontra o terminador da sequência de caracteres retornando zero e não chamando mais a função, caso contrário, é utilizado o caso geral que se chama a função novamente, somando o valor um quando é igual ao caractere e não somando quando não é igual.

No programa principal foi declarado uma variável do tipo caractere e mais uma *string*. Nas linhas 12 e 14 foi utilizada função puts para mostrar uma mensagem de saída (tela) para orientar o usuário o que deve ser digitado. A função getchar, na linha 13, faz a coleta de um caractere e armazena na variável c. Na linha 15 é limpo o buffer do teclado, em seguida, coletado a *string* por meio da função gets, na sequência, são passados esses dados por parâmetros para a função recursiva conta_caractere e o seu retorno é mostrado na tela com uma mensagem dizendo quantos desse caractere existe nesta *string*.

Esse exemplo foi utilizado o método recursivo ao invés do método iterativo para demonstrar outras formas de resolver problemas envolvendo *strings* apesar dos dois métodos conseguirem resolver o mesmo problema.

DESAFIO

Faça uma função que faça a validação de um CEP (Código de Endereçamento Postal) utilizando o seguinte formato XX.YYY-ZZZ, sendo que XX, YYY e ZZZ são valores numéricos inteiros. Retorne -1, caso o comprimento da *string* seja inválido. Retorne -2, se ponto (.) ou o hífen (-) não estejam na posição correta. Retorne -3, caso os valores XX, YYY ou ZZZ não sejam numéricos. Caso não apresente problemas nas validações, retorne 1. Mostre o funcionamento da função no programa principal.

2. MANIPULAÇÃO DE ARQUIVO

“Um arquivo, de uma forma abstrata, nada mais é do que uma coleção de bytes armazenados em um dispositivo de armazenamento secundário, que é geralmente um disco rígido, pen drive, DVD, etc. [...]” (BACKES, 2022, p. 235). Esse agrupamento de bytes pode ser interpretado como palavras, frases ou caracteres em um documento de texto; registros em uma tabela de banco de dados; pixels em uma imagem; números em uma planilha eletrônica; entre outras aplicações.

O que permite que um arquivo tenha um significado em particular é a forma que seus dados são organizados, permitindo a sua manipulação (leitura e escrita) através dos programas. A linguagem C possui um conjunto de funções que permite a manipulação de arquivos das mais variadas formas, admitindo criar novos arquivos, acrescentar no-

vos dados em arquivos existentes ou realizar a leitura das informações contidas em um arquivo já existente.

Resumidamente, a linguagem C aceita dois tipos de arquivos, os arquivos de texto e os arquivos binários. Sendo que os arquivos de texto armazenam dados que podem ser mostrando em um simples editor de texto como o Bloco de Notas do *Windows* ou *Emacs* do *Linux*. Entretanto, essa facilidade de leitura traz consigo a necessidade de mais espaço para armazenar essas informações, pois os dados em texto necessitam de uma conversão que acabam usando mais espaço de armazenamento se comparado ao arquivo binário. Já os arquivos binários são gravados exatamente como estão organizados na memória, dessa forma, fazendo com que suas operações de leitura e escrita sejam mais rápidas se comparado aos arquivos de texto, por não terem que passarem por conversões. Os arquivos executáveis, arquivos compactados e arquivos de registros são exemplos de arquivos binários.

Por meio da biblioteca stdio.h a linguagem C fornece funções para leitura e escrita de arquivos. Para isso, usa um tipo especial de ponteiro do tipo FILE para manipulação de arquivos, que quando aberto, aponta para o primeiro registro no arquivo. Esse ponteiro também controla o próximo byte a ser acessado para ser realizado a leitura, assim como, a indicação do final do arquivo, entre outras tarefas.

Veja o Código 5 que mostra um exemplo de abertura para a escrita e fechamento do arquivo.

Código 05. Exemplo de abertura para escrita de um arquivo

```
01  #include <stdio.h>
02  void main(){
03      FILE *fp;
04      fp=fopen ("arq.txt","w");
05      if (fp==NULL) printf ("Erro na abertura do arquivo.\n");
06      else printf("Arquivo aberto com sucesso.\n");
07      fclose(fp);
08  }
```

Fonte: elaborado pelo autor.

No Código 5, na linha 3, é declarado um ponteiro do tipo *FILE* chamado fp, em seguida, é utilizada a função fopen que abre o arquivo "arq.txt" com modo de acesso para escrita e armazena o endereço que retorna da função no ponteiro fp. Na linha 5, é verificado se o endereço do ponteiro é igual a nulo, isso ocorrerá se ocorrer algum problema na sua abertura, como por exemplo, um antivírus barrar o arquivo executável por questão de segurança ou o usuário que está executando não tenha permissão de criação no diretório. Se apresentar problema é mostrado uma mensagem na tela de erro, caso contrário, mostra-se uma mensagem de sucesso ao abrir. Por fim, na linha 7, ocorre o fechamento do arquivo aberto através da função fclose.

SAIBA MAIS

Fechar o arquivo é importante, pois com essa solicitação de fechamento, toda informação contida buffer é gravada no disco. O *buffer* é uma região da memória que armazena temporariamente as informações a serem gravados no disco e apenas quando o *buffer* se enche que seu conteúdo é gravado no disco, ao solicitar o fechamento do arquivo, essas informações são gravadas mesmo com o *buffer* não estando cheio. O uso de *buffer*, neste caso, aumenta a eficiência, pois a leitura e escrita no disco é mais lenta que na memória, caso fosse realizada diretamente cada escrita e leitura de caractere no disco provocaria uma enorme lentidão, com isso, é aguardado uma quantidade razoável de dados (ou o fechamento do arquivo) para que as operações de escrita e leitura sejam realizadas.

Sinteticamente, esse programa irá tentar abrir o arquivo de texto, se ele não existir, ele será criado. Como não foi gravado nada no arquivo, ele ficará em branco e será fechado.

Como pode-se observar, a função fopen possui dois parâmetros, o primeiro que é o nome do arquivo que será aberto e o modo de acesso. Em relação ao nome do arquivo, é possível ser trabalho com caminhos absolutos ou relativos. Os caminhos absolutos são quando indicamos desde o diretório raiz até o nome do arquivo como por exemplo, “C:\\Projetos\\Arquivos\\arq.txt”, observe que as separações dos diretórios são divididas por duas barras. Já os caminhos relativos dependem do local do programa se encontra como por exemplo, “arq.txt” ou “.\arq.txt” o arquivo deverá estar no mesmo diretório do executável.

Referente ao modo de acesso, o Quadro 2, mostra algumas possíveis formas de acesso para a abertura dos arquivos através da função fopen.

Quadro 02. Modo de acesso da função fopen

MODO	ARQUIVO	DESCRIÇÃO
r ou rt	Texto	Modo somente para leitura. O arquivo deve existir.
w ou wt	Texto	Modo somente para escrita. O arquivo criado, se não existir. Apaga os dados anteriores, se ele existir.
a ou at	Texto	Modo somente para escrita. O arquivo criado, se não existir. Os dados já existentes são adicionados no final do arquivo.
rb	Binário	Modo somente para leitura. O arquivo deve existir.
wb	Binário	Modo somente para escrita. O arquivo criado, se não existir. Apaga os dados anteriores, se ele existir.
ab	Binário	Modo somente para escrita. O arquivo criado, se não existir. Os dados já existentes são adicionados no final do arquivo.
r+	Texto	Modo para leitura e escrita. O arquivo deve existir e pode ser modificado.

MODO	ARQUIVO	DESCRIÇÃO
w+	Texto	Modo para leitura e escrita. O arquivo criado, se não existir. Apaga os dados anteriores, se ele existir.
a+	Texto	Modo para leitura e escrita. O arquivo criado, se não existir. Os dados já existentes são adicionados no final do arquivo.
r+b	Binário	Modo para leitura e escrita. O arquivo deve existir e pode ser modificado.
w+b	Binário	Modo para leitura e escrita. O arquivo criado, se não existir. Apaga os dados anteriores, se ele existir.
a+b	Binário	Modo para leitura e escrita. O arquivo criado, se não existir. Os dados já existentes são adicionados no final do arquivo.

Fonte: elaborado pelo autor.

Atente-se que no Quadro 2, que o "r" simboliza *read* de leitura, "w" simboliza *write* de escrita e "a" simboliza append de acrescentar em um arquivo de texto. Observe que pode se acrescentar um "t" nesses modos. Ao se acrescentar o "b" informa-se que é para manipular um arquivo binário e se acrescentar um "t" permanece sendo a manipulação de um arquivo de texto, ou seja, incluir ou não o "t" é opcional. Já o símbolo "+" faz com que o arquivo aberto seja lido ou escrito.

As funções fopen e fclose serão utilizadas necessariamente para manipulação de arquivo, mas existem outras que podem ser utilizadas. No Quadro 3 é mostrada as principais funções para leitura e escrita em um arquivo.

Quadro 03. Relação de funções úteis para manipulação de arquivos

FUNÇÃO	DESCRIÇÃO
fscanf	Lê os dados de entrada (arquivo) e os armazena de acordo com o formato do parâmetro nos locais apontados pelos argumentos adicionais. Similar a função scanf.
fprintf	Grava uma *string* na saída (arquivo), caso possua especificadores de formato, os argumentos adicionais após o formato são formatados e inseridos na *string* resultante substituindo seus respectivos especificadores. Similar a função printf.
fgets	Lê os caracteres da entrada (arquivo) e os armazena como uma *string* até que um caractere de nova linha ou o final do arquivo seja alcançado. Similar a função gets.
fgetc	Retorna o próximo caractere da entrada (arquivo). Similar a função getchar.
fputs	Grava uma *string* na saída (arquivo) e acrescenta um caractere de nova linha ('\n'). Similar a função puts.
fputc	Grava um caractere na saída (arquivo). Similar a função putchar.

FUNÇÃO	DESCRIÇÃO
fwrite	Grava um bloco de dados na saída (arquivo).
fread	Lê um bloco de dados na entrada (arquivo).
fseek	Move a posição atual de leitura ou escrita da entrada (arquivo), a partir de um ponto especificado.
ftell	Retorna a posição atual de leitura ou escrita da entrada (arquivo).
rewind	Move a posição atual de leitura ou escrita da entrada (arquivo) para o início.
feof	Verifica se é o fim do arquivo.
remove	Apaga um arquivo específico.

Fonte: elaborado pelo autor.

O Quadro 3 traz funções que permitem a leitura ou escrita de um único caractere por vez, assim como, dados formatados (inteiro, *string*, ponto flutuante, etc.) ou um bloco de dados (gravar um maior volume de dados em uma única vez) em um arquivo. Veja o Código 6 que apresenta um exemplo de como fazer gravação de três diferentes tipos de informação em um arquivo de texto, em seguida, fazer a leitura dessas mesmas três informações.

Código 06. Exemplo de leitura e gravação de dados no arquivo

```
01  #include <stdio.h>
02  void main(){
03      int id=1;
04      char descricao[50]=”Sabonete”;
05      float preco=2.48;
06      FILE *fp;
07      if(fp=fopen (“arq.txt”,”w”)){
08         fprintf(fp,”%i %s %f”,id,descricao,preco);
09         fclose(fp);
10      }else printf (“Erro na abertura do arquivo.\n”);
11      if(fp=fopen (“arq.txt”,”r”)){
12         fscanf(fp,”%i %s %f”,&id,&descricao,&preco);
13         printf(“ID: %i\n”,id);
14         printf(“Descricao: %s\n”,descricao);
15         printf(“Preco: %.2f\n”,preco);
16         fclose(fp);
17      }else printf (“Erro na abertura do arquivo.\n”);
18  }
```

Fonte: elaborado pelo autor.

No Código 6, entre as linhas 3 e 5 estão sendo declaradas três variáveis (id, descrição e preço) e atribuído um valor para cada uma, em seguida, na linha 6 é declarado um ponteiro do tipo *FILE* para ser usado na leitura e escrita do arquivo. Entre as linhas 7 e 10, tenta-se abrir o arquivo "arq.txt" para escrita, se a abertura tenha sido efetuada com sucesso será gravado através da função fprintf o conteúdo da variável id do tipo inteiro, da variável descrição do tipo *string* e da variável preço do tipo *float* separado por espaço em branco no arquivo e fecha-se o arquivo, caso não tenha sido aberto com sucesso, será mostrado uma mensagem na tela de erro.

Entre as linhas 11 e 17, tenta-se abrir o arquivo "arq.txt" para leitura, se a abertura tenha sido efetuada com sucesso será lido através da função fscanf de um valor do tipo inteiro, de um valor do tipo *string* e de um valor do tipo *float* separado por espaço em branco nas variáveis id, descrição e preço, respectivamente, após, será mostrado na tela os valores da variável id, descrição e preço em cada linha e fecha-se o arquivo, caso não tenha sido aberto com sucesso, será mostrado uma mensagem na tela de erro.

Importante destacar que a *string* descrição não poderá ter espaço em branco, pois o comando scanf e fscanf não conseguem fazer a leitura de *strings* com espaço em branco. Outro ponto, é que o arquivo foi aberto para a escrita e fechado, em seguida, aberto para leitura e fechado, mas poderia ser aberto e fechado uma única vez para leitura e escrita.

DESAFIO

Faça um programa que colete seu nome, idade, peso e sua altura, em seguida, grave esses dados em cada linha de um arquivo de texto. Por fim, faça a leitura desse arquivo de texto e mostre esses dados na tela.

O Código 7 apresenta uma outra forma de realizar a leitura do conteúdo do arquivo, através da leitura de caractere por caractere até encontrar o fim do arquivo.

Código 07. Exemplo de leitura caractere por caractere

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  void main(){
04     FILE *fp;
05     char c;
06     int n = 0;
07     if(!(fp=fopen ("file.txt","r"))) {
08        printf ("Erro na abertura do arquivo.\n");
09        exit(1);
```

```
10      }
11      while((c=fgetc(fp))!=EOF)
12          if (isdigit(c)) n++;
13      fclose(fp);
14      printf("Existem %d numeros no arquivo.\n",n);
15  }
```

Fonte: elaborado pelo autor.

No Código 7, foi incluído a biblioteca stdlib.h para se usar a função *exit*. Na linha 4 declarado um ponteiro do tipo *FILE*, em seguida, foi declarada uma variável do tipo caractere para armazenar o caractere que for lido no arquivo e na sequência, foi declarado uma variável inteira que é inicializada com o valor zero e será utilizada como contadora. Entre as linhas 7 a 10, está verificando se o arquivo não foi aberto com sucesso de uma forma diferente do que foi realizada no Código 6, ou seja, agora é verificado se não foi aberto e se isso ocorrer, será mostrado uma mensagem de erro e por meio da função *exit* será fechado o programa, sem dar continuidade.

Na linha 11 contém uma estrutura de repetição que irá coletar um caractere do arquivo através da função fgetc e será armazenado esse caractere na variável c. Neste caso a estrutura de repetição executará enquanto não for um fim de arquivo que é representado por EOF (*End of File*), em cada iteração da estrutura, será verificado se o caractere lido é um caractere numérico através da função isdigit, se for, será adicionado o valor um ao valor atual da variável n. Após o término da estrutura de repetição, ou seja, após fazer a leitura de todos os caracteres do arquivo um por um até chegar ao final do arquivo, será fechado o arquivo e será mostrado na tela quantos números existem no arquivo "file.txt".

Para a leitura e escrita de um arquivo é possível fazer caractere por caractere como foi visto no Código 7, mas também é possível fazer em bloco de dados, como é mostrado no Código 8.

Código 08. Exemplo de escrita e leitura com bloco de dados

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  void main(){
04      FILE *arq;
05      char texto[100], info[5][100];
06      int i;
07      if(!(arq=fopen("arqtexto.txt", "w+b"))){
```

```
08          printf ("Erro na abertura do arquivo.\n");
09          exit(1);
10      }
11      for(i=0;i<5;i++){
12              printf("Digite o texto %i: ",i+1);
13              setbuf(stdin,NULL);
14              gets(texto);
15              fwrite(texto,sizeof(texto),1,arq);
16      }
17      rewind(arq);
18      fread(info,sizeof(texto),5,arq);
19      fclose(arq);
20      for(i=0;i<5;i++) printf("Texto %i: %s\n",i+1,info[i]);
21  }
```

Fonte: elaborado pelo autor.

No Código 8, pode-se observar que assim como os outros programas, foi declarado um ponteiro do tipo FILE para armazenar o endereço do arquivo aberto. Foi declarado na linha 5 uma *string* texto com capacidade de armazenar no máximo 99 caracteres e mais o caractere '\0', e também, uma matriz com 5 linhas e 100 colunas, mas neste caso ela representa um vetor de *strings*, pois ela pode armazenar 5 informações do tamanho da *string* texto. Na linha seguinte foi declarada uma variável inteira para ser utilizada na iteração.

Na linha 7, verifica se o arquivo "arqtexto.txt" não foi aberto com sucesso para leitura e escrita de um arquivo binário, se isso ocorrer, será mostrado na tela uma mensagem de erro e finalizado o programa.

Entre as linhas 11 e 16 existe uma estrutura de repetição que irá fazer cinco vezes a coleta de um texto digitado pelo usuário e será armazenado na *string* texto, logo em seguida, é utilizado a função fwrite para gravar a *string* texto no arquivo, para isso são informados no parâmetro a *string* texto, o tamanho da *string* texto por meio da função sizeof, depois o valor 1 que indica que será armazenada somente uma dessa estrutura no arquivo que é representado pelo ponteiro arq. Neste caso foi optado em gravar uma *string* por vez, porque o usuário está digitando o texto e sobrepondo o dado contido anteriormente, entretanto, se optar em gravar todos os dados coletados (sem sobrepor), poderia ser realizada a gravação no arquivo uma única vez.

Na linha 17 é utilizada a função *rewind* para retornar o ponteiro arq para início do arquivo para realizar a leitura. Neste caso, por ter sido aberto o arquivo para leitura e escrita, o ponteiro se encontrava no final do arquivo, pois tinha passado pela função de escrita fwrite por cinco vezes.

Depois, na linha 18, é realizado a leitura do arquivo através da função *fread* que foi informado por parâmetro que as informações lidas serão gravadas em info, sendo informado o seu tamanho através da função sizeof, a quantidade de elementos desse tamanho, neste caso são 5 e por fim, em qual arquivo será realizado essa leitura. Em suma, essa linha é realizada a leitura de toda a matriz (cinco textos digitados pelo usuário) com apenas um comando. Por fim, é fechado o arquivo, pois as informações já foram carregadas na matriz de caracteres e na sequência, elas são mostradas na tela através da estrutura de repetição.

Apesar de, até o momento, ter utilizado somente arquivos com a extensão TXT, a linguagem C permite a manipulação de arquivos, independentemente de sua extensão, pois os arquivos são um mecanismo de abstração, proporcionando uma maneira de armazenar informações no disco e de lê-las posteriormente (TANENBAUM, 2008, p. 446).

Na verdade, as extensões dos arquivos são apenas convenções utilizadas em alguns sistemas operacionais, mas não são impostas. Por exemplo, um arquivo com o nome “arquivo.txt”, poderia ser algum tipo de arquivo de texto ou não, portanto, essa extensão do arquivo (parte do nome após o ponto) serve mais para lembrar o usuário, do que para transmitir qualquer informação para o computador.

Dessa forma, no Código 9 está manipulado um arquivo de texto com a extensão CSV que é um arquivo muito utilizado para a importação e exportação de informações entre sistemas.

Código 09. *Exemplo de manipulação de um arquivo de texto com a extensão CSV

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <string.h>
04  void main(){
05    FILE *arq;
06    char cidade[50],temp[80];
07    float altitude;
08    int id,op;
09    do{
10      system(“CLS”);
11      printf(“ 1-Incluir\n 2-Ler\n 3-Sair\n”);
12      scanf(“%i”,&op);
13      if(op==1){
14        printf(“Digite o ID: “);
15        scanf(“%i”,&id);
16        printf(“Digite a cidade: “);
17        setbuf(stdin,NULL);
```

```
18        gets(cidade);
19        printf("Digite a altitude: ");
20        scanf("%f",&altitude);
21        if(arq=fopen("altitude.csv","a")){
22          fprintf(arq,"%i;%s;%.2f\n",id,cidade,altitude);
23          fclose(arq);
24        }
25           printf("Inserido com sucesso!\n");
26           system("PAUSE");
27      }
28      if(op==2){
29           if(arq=fopen("altitude.csv","r")){
30             while(fgets(temp,80,arq)){
31               id=atoi(strtok(temp,";"));
32               printf("ID: %i\n",id);
33               printf("Cidade: %s\n",strtok(NULL,";"));
34               altitude=atof(strtok(NULL,";"));
35               printf("Altitude: %.2f\n",altitude);
36             }
37             fclose(arq);
38             system("PAUSE");
39           }
40      }
41    }while(op!=3);
42  }
```

Fonte: elaborado pelo autor.

No Código 9, está sendo incluída a biblioteca stdlib.h para usar funções de conversão de *string* para números inteiros e decimais, assim como, a biblioteca string.h para usar funções de busca em *string*. Entre as linhas 5 e 8, está sendo declarado as *strings*, variáveis e ponteiros que irão ser utilizadas durante o programa.

Na linha 9 até a linha 41 há uma estrutura de repetição que irá se repetir enquanto o valor da variável op for diferente de 3, isso foi desenvolvido dessa maneira para proporcionar um menu, onde o valor 3 (valor que interrompe a repetição) é utilizado para sair do programa.

Dentro da estrutura de repetição, na linha 10, através da função system é apagado as informações da tela (limpa tela), na sequência é mostrado as opções de 1 a 3 para o usuário escolher e na linha 12 é coletado a opção digitada e armazenada na variável op.

Entre as linhas 13 e 17 é verificado se a opção digitada é a 1 (incluir um item no arquivo), se for, serão coletados o id (número inteiro), a cidade (*string*) e a altitude (número real). Na linha 21 faz-se a abertura do arquivo "altitude.csv" para escrita em modo de acréscimo (append), com isso, o conteúdo do arquivo não será apagado e novo dado será adicionado no final do arquivo. Se o arquivo for aberto com sucesso, será gravada através da função fprintf as informações digitadas pelo usuário separadas por ponto e vírgula e ao final é pulado uma linha. Por fim, fecha-se o arquivo, mostra-se uma mensagem para usuário e por meio da função system é pedido para o usuário digitar qualquer tecla para continuar o programa.

Entre as linhas 28 e 40 é verificado se a opção digitada é a 2 (ler todos os itens no arquivo), se for, será aberto o arquivo "altitude.csv" para a leitura. Na linha 30 existe uma estrutura de repetição, que faz com que enquanto consiga fazer a leitura do arquivo (através da função fgets) e armazenar na *string temp*, será separado a primeira parte até o ponto e vírgula (ID) com a função strtok, convertido de *string* para inteiro por meio da função atoi e armazenado na variável id e mostrada na tela. Na linha 33, é realizado algo similar, sendo pego a segunda parte até o ponto e vírgula (cidade) e nesse caso, como é um valor *string* é mostrado na tela sem conversão. Na linha 34, é separada a terceira parte, neste caso buscando também o ponto e vírgula, mas a função interrompe por encontrar o terminador de sequência de caracteres ('\0') e também é mostrado na tela após a sua conversão através da função atof. Por fim, é fechado o arquivo e com auxílio da função system é pedido para o usuário digitar qualquer tecla para continuar o programa (impedindo que a tela seja apagada antes da leitura).

SAIBA MAIS

Os arquivos de texto com extensão CSV (Comma-Separated Values), chamados assim por ter os campos separados por vírgula (colunas) e cada dado novo é inserido em uma nova linha. No Brasil, por ser utilizado a vírgula como separador das casas decimais, diferentemente do formato americano que é separado por ponto, é utilizado o delimitador ponto e vírgula (;) ao invés de vírgula (,), entretanto, é possível utilizar outro delimitador como pipe ("|") e continuar seguindo o mesmo formato. Existem outros formatos para arquivos de texto como XML e JSON também usados na computação, mas o formato CSV pode ser facilmente aberto nas diversas planilhas eletrônicas e não exige bibliotecas sofisticado para sua manipulação tampouco recurso físico (*hardware*).

DESAFIO

Faça um programa que faça a leitura de um arquivo de texto no formato CSV contendo dados sobre carros como marca, modelo, ano e valor, conforme o exemplo abaixo:

```
Volkswagen;Fusca;1974;4900.00
Fiat;Uno;2010;24500.00
Chevrolet;Celta;2012;25200.00
Ford;Ka;2020;60850.00
```

Após a abertura do arquivo, colete um valor do usuário e mostre na tela os dados do arquivo somente dos carros abaixo do valor informado pelo usuário.

CONCLUSÃO

A manipulação de *strings* e arquivos é algo muito importante para o desenvolvedor, diversos dispositivos como relógio eletrônico de ponto (REP), *Data Logger*, entre outros dispositivos que fazem a gravação de dados em arquivos de texto e para esses dados se tornarem acessíveis precisam ser inseridos por meio de outros softwares nos mais diversificados banco de dados. Para fazer isso, esses dados necessitam de validação, conversão, comparação, entre outras ações para evitar inconsistência de dados. Além disso, os mais variados softwares também utilizam arquivos como complemento, como arquivos de linguagem (que contém as configurações do idioma), arquivos de personalização do usuário (tamanho da fonte, cor do fundo, etc.). Portanto, essa parte mostrada traz utilidade e benefício para a vida profissional do desenvolvedor, seja de jogos, sistemas comerciais, aplicativos, entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

BACKES, A. **Linguagem C**: completa e descomplicada. 2. ed. Rio de Janeiro: LTC, 2022.

TANENBAUM, A. S. **Sistemas operacionais**: projeto e implementação. 3. ed. Porto Alegre: Bookman, 2008.