

NÃO PODE FALTAR

COMPONENTES E ELEMENTOS DE LINGUAGEM DE PROGRAMAÇÃO

Vanessa Cadan Scheffer

Ver anotações

COMO OS DADOS SÃO REPRESENTADOS E UTILIZADOS EM UMA LINGUAGEM DE PROGRAMAÇÃO

Você estudará os conceitos de variáveis e constantes, seus tipos, suas características e sua utilização em uma linguagem de programação.



Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro estudante, pegue um papel e uma caneta ou abra o bloco de notas. Olhe no relógio e anote a hora, o minuto e o segundo. Anotou? Novamente, olhe no relógio e faça a mesma anotação. Pronto? O valor foi o mesmo? E se anotasse novamente, seria igual? Certamente que não. Agora considere um computador que tem uma massa de 3 quilogramas, se você mudá-lo de mesa, sua massa altera? E se colocá-lo no chão? Se esses elementos (tempo e massa) fizessem parte de uma solução computacional, eles seriam representados da mesma forma em um algoritmo?

Ver anotações

Nesta seção, veremos como os dados podem ser classificados em algoritmos implementados na linguagem C. Se existem diferentes tipos de dados, é natural que existam diferentes formas de representá-los em uma linguagem de programação.

A fim de colocarmos em prática os conhecimentos que serão adquiridos nesta seção, vamos analisar a seguinte situação-problema: você é um dos programadores de uma empresa responsável por criar um software de locação de filmes on-line, e foi incumbido de criar uma nova funcionalidade para o software, que consiste em detectar se um filme pode ou não ser locado pelo cliente com base na idade dele e na classificação indicativa do filme.

Até então, você já construiu um algoritmo capaz de receber como entradas a idade do cliente e a classificação indicativa do filme que ele pretende locar e, logo após, mostrar na tela um dos possíveis resultados: “Este filme não é indicado para sua faixa etária” ou “Este filme é indicado para sua faixa etária”.

Agora é hora de tirar essa ideia do papel e colocar para funcionar em um computador. Contudo, você ainda não dispõe de todos os conhecimentos necessários para implementar essa solução na linguagem C. Por isso, seu chefe lhe passou outra tarefa. Ele quer que seu programa seja capaz de ler a idade e o nome do cliente, bem como a classificação do filme que ele deseja locar.

Posteriormente, seu programa deve imprimir todas essas informações na tela, conforme o padrão a seguir:

Cliente: José das Couves
Idade: 18 anos
Classificação do filme: 12 anos

Seu chefe, que também é analista de sistemas, informou que você deve utilizar os conceitos de *struct*, variáveis e constantes para resolver esse problema.

CONCEITO-CHAVE

Durante nossa formação escolar, aprendemos a realizar diversos cálculos, por exemplo, a área de uma circunferência $A = \pi r^2$, em que π (pi) equivale a, aproximadamente, 3,14 e r é o raio. Pensando ainda nesse problema da área da circunferência, vamos analisar o valor da área para alguns raios. Observe a Tabela 1.1.

Tabela 1.1 | Alguns valores para área da circunferência

$\pi(\text{pi})$	Raio (r)	Área
3,14	2 cm	12,56 cm ²
3,14	3 cm	28,26 cm ²
3,14	4 cm	50,24 cm ²

Fonte: elaborada pela autora.

Após aplicar a fórmula para alguns valores, percebemos que, conforme o valor do raio **varia**, a área também varia, mas o *pi* permanece **constante**. São inúmeros os exemplos que poderiam ser dados, nos quais se observam valores que variam ou que são fixos, isso porque nosso mundo é composto por situações que podem ou não variar, de acordo com certas condições. Sabendo que os sistemas computacionais são construídos para solucionar os mais diversos problemas, eles

Ver anotações

devem ser capazes de lidar com essa característica, ou seja, efetuar cálculo com valores que podem ou não variar. No mundo da programação, esses recursos são chamados de **variáveis** e **constantes**. A principal função desses elementos é armazenar temporariamente dados na memória de trabalho.

o

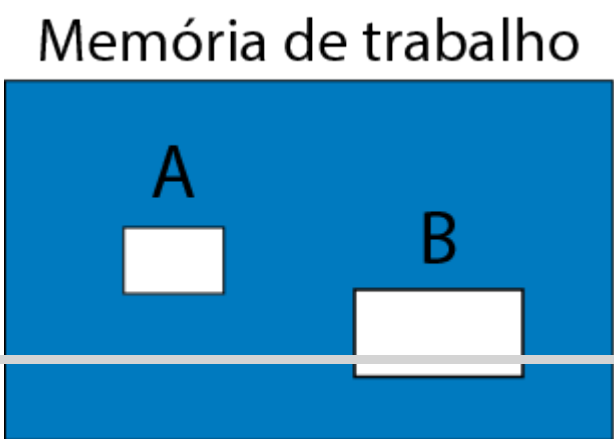
VARIÁVEIS

Ver anotações

Deitel e Deitel (2011, p. 43) nos trazem a seguinte definição: “uma variável é uma posição na memória onde um valor pode ser armazenado para ser utilizado por um programa”. Soffner (2013, p. 33) incrementa dizendo que “variáveis são endereços de memória de trabalho que guardam, temporariamente, um valor utilizado pelo programa”. A associação que os autores fazem entre as variáveis e os endereços de memória nos ajuda a compreender que esse elemento, quando criado, existe de fato na memória do computador e, portanto, ocupa um espaço físico. O mesmo se aplica às constantes, porém, nesse caso, o valor armazenado nunca será alterado.

Na Figura 1.6 foi criada uma representação simbólica para a memória de um computador e dentro dessa memória foram alocadas duas variáveis, A e B. O espaço alocado para B é maior que o alocado para A, mas como é feita essa especificação? A quantidade de espaço que será alocada para uma variável pode ser especificada de duas maneiras: a primeira refere-se ao tipo de dado que será armazenado no espaço reservado, caso em que o programador não consegue manipular o tamanho alocado; a segunda é feita de maneira manual pelo programador, por meio de funções específicas.

Figura 1.6 | Alocação de variáveis na memória de trabalho



Fonte: elaborada pela autora.

Todas as linguagens de programação têm **tipos primitivos** (ou básicos) e **compostos**. No grupo dos primitivos estão os seguintes tipos:

o

Ver anotações

- **Numérico inteiro:** são valores inteiros que podem ser positivos, negativos ou zero. Alguns exemplos são as variáveis que armazenam idade, quantidade de produtos e código de identificação.
- **Numérico de ponto flutuante:** esse tipo armazena valores pertencentes ao conjunto dos números reais, ou seja, valores com casas decimais. Como exemplo, temos as variáveis que armazenam peso, altura e dinheiro, entre outras.
- **Caractere:** é o tipo usado para armazenar um caractere alfanumérico (letra, número, símbolo especial e outros). Como exemplo de uso, podemos citar o armazenamento do gênero de uma pessoa; caso seja feminino, armazena F, caso masculino, armazena M.
- **Booleano:** variáveis desse tipo só podem armazenar um dos dois valores: **verdadeiro** ou **falso**. Geralmente são usados para validações, por exemplo, para verificar se o usuário digitou um certo valor ou se ele selecionou uma determinada opção em uma lista, entre outros.

USO DE VARIÁVEIS EM LINGUAGENS DE PROGRAMAÇÃO

Para se usar uma variável na linguagem de programação C é preciso criá-la e, para isso, usa-se a seguinte sintaxe:

```
<tipo> <nome_da_variavel>;
```

Esse padrão é obrigatório e podemos usar os seguintes tipos primitivos: `int` (inteiro), `float` ou `double` (ponto flutuante) e `char` (caractere). O tipo booleano é representado pela palavra-chave `bool`, entretanto, para seu uso, é necessário

incluir a biblioteca `<stdbool.h>`.

Uma biblioteca é um conjunto de funções e tipos de dados que podem ser reutilizadas em nossos próprios programas.

Ver anotações

Veja no Código 1.1 a criação de algumas variáveis na linguagem C.

Código 1.1 | Criação de variáveis na linguagem C

```
1  #include <stdbool.h>
2  int main(){
3      int idade;
4      float salario = 1250.75;
5      double porcentagem_desconto = 2.5;
6      bool estaAprovado = false;
7      char genero = 'M';
8      return 0;
9  }
```

Fonte: elaborado pela autora.

É importante ressaltar que todo programa em C deve conter uma função específica, denominada “main”. Ela representa o ponto inicial do seu programa, ou seja, o ponto a partir do qual o computador começará a executá-lo. Não se preocupe, por enquanto, com o conceito de função, pois teremos toda uma seção para discutirmos sobre ele.

Voltando ao assunto das variáveis, destaca-se que, ao criar uma variável, o programador pode optar por já atribuir ou não um valor (por exemplo, para a variável “idade”, nenhum valor foi atribuído a priori).

ATENÇÃO

Inicialização de variáveis

É uma boa prática de programação sempre inicializar as variáveis com algum valor específico, evitando que recebam dados de processamentos anteriores e que estejam na memória (a esses dados nós damos o nome de lixo de memória). Portanto, quando a variável for numérica, sempre vamos iniciar com zero; quando booleana, com falso; quando do tipo caractere, usaremos ' ' para atribuir vazio.

o
Ver anotações

Nome da variável

Outro ponto importante é o nome da variável, que deve ser escolhido de modo a identificar o dado a qual se refere. Por exemplo, em vez de usar “ida” para criar uma variável que armazenará a idade, opte pelo termo completo.

Outro aspecto importante a ser ressaltado é que a maioria das linguagens de programação são *case sensitive*, o que significa que letras maiúsculas e minúsculas são tratadas com distinção. Então, a variável “valor” é diferente da variável “Valor”. Além disso, no nome de uma variável não podem ser usados acentos nem caracteres especiais, como interrogação e espaços em brancos, entre outros. Nomes de variáveis também não podem iniciar com números. Por exemplo, “1telefone” **não** é um nome válido para variável, enquanto “telefone1” **é** um nome válido.

TIPOS DE VARIÁVEIS E SUA CAPACIDADE

Como já mencionado, a quantidade de espaço que será alocada para uma variável depende do tipo de variável. Por exemplo, para uma variável `int` serão alocados 4 bytes na memória (MANZANO; MATOS; LOURENÇO, 2010). O tamanho alocado na memória pelo tipo de variável limita o valor que pode ser guardado naquele espaço. Por exemplo, não seria possível guardar o valor 10 trilhões dentro da variável do tipo `int`. Vejamos, 4 bytes são 32 bits. Cada bit só pode armazenar zero

ou um. Portanto, nós temos a seguinte equação: valor máximo de uma variável inteira = $2^{32}=4.294.967.296$. Porém, esse valor precisa ser dividido por dois, pois um inteiro pode armazenar números negativos e positivos. Logo, uma variável `int` poderá ter um valor entre -2.147.423.648 e 2.147.423.648.

o

Para suprir parte da limitação dos valores que uma variável pode assumir pelo seu tipo, foram criados modificadores de tipos, os quais são palavras-chave usadas na declaração da variável que modifica sua capacidade-padrão. Os três principais modificadores são:

Ver anotações

- **Unsigned**, usado para especificar que a variável armazenará somente a parte positiva do número.
- **Short**, que reduz o espaço reservado pela memória.
- **Long**, que aumenta a capacidade padrão.

A Tabela 1.2 mostra o tamanho e os possíveis valores de alguns tipos de dados na linguagem C, inclusive com os modificadores.

Tabela 1.2 | Tipos de variáveis e sua capacidade

Tipo	Tamanho (byte)	Valores
int	4	-2.147.423.648 até 2.147.423.648
float	4	$-3,4^{38}$ até $3,4^{38}$
double	8	$-1,7^{308}$ até $1,7^{308}$
char	1	-128 até 127
unsigned int	4	4.294.967.296
short int	2	-32.768 até 32.767

Tipo	Tamanho (byte)	Valores
long double	16	-3,4 ⁴⁹³² até 1,1 ⁴⁹³²

0

Fonte: adaptada de Manzano, Matos e Lourenço (2015, p. 35).

ESPECIFICADOR DE FORMATO

Na linguagem de programação C, cada tipo de variável usa um especificador de formato para fazer a impressão do valor que está guardado naquele espaço da memória (PEREIRA, 2017). Veja no Código 1.2, que foram criadas cinco variáveis (linhas 4 a 8) e para cada tipo, usou-se um especificador na hora de fazer a impressão com a função `printf()`. A função “`printf`” pertence à biblioteca `<stdio.h>`, por isso, é necessário inclui-la no início do programa. Na linha 10, para o tipo inteiro, foi usado `%d`. Nas linhas 11 e 12, para os pontos flutuantes, foi usado `%f`. Na linha 13, para o caractere, foi usado o `%c`, e, na linha 14, o especificador de ponto flutuante ganhou o adicional `.3`, o que significa que o resultado será impresso com três casas decimais. O símbolo `\n` é usado para gerar uma quebra de linha na saída do programa.

Ver anotações

EXEMPLIFICANDO

Código 1.2 | Impressão de variáveis

```
1  #include <stdio.h>
2
3  int main(){
4      int idade = 18;
5      float salario = 1250.75;
6      double porcentagem_desconto = 2.5;
7      char genero = 'F';
8      float altura = 1.63;
9
10     printf("\n Idade: %d",idade);
11     printf("\n Salário: %f",salario);
12     printf("\n Desconto (%): %f", porcentagem_desconto);
13     printf("\n Gênero: %c",genero);
14     printf("\n Altura: %.3f",altura);
15     return 0;
16 }
17
```

0
Ver anotações

Fonte: elaborado pela autora.

Vamos utilizar a ferramenta Paiza.io para testar as declarações das variáveis e a impressão de cada uma delas.

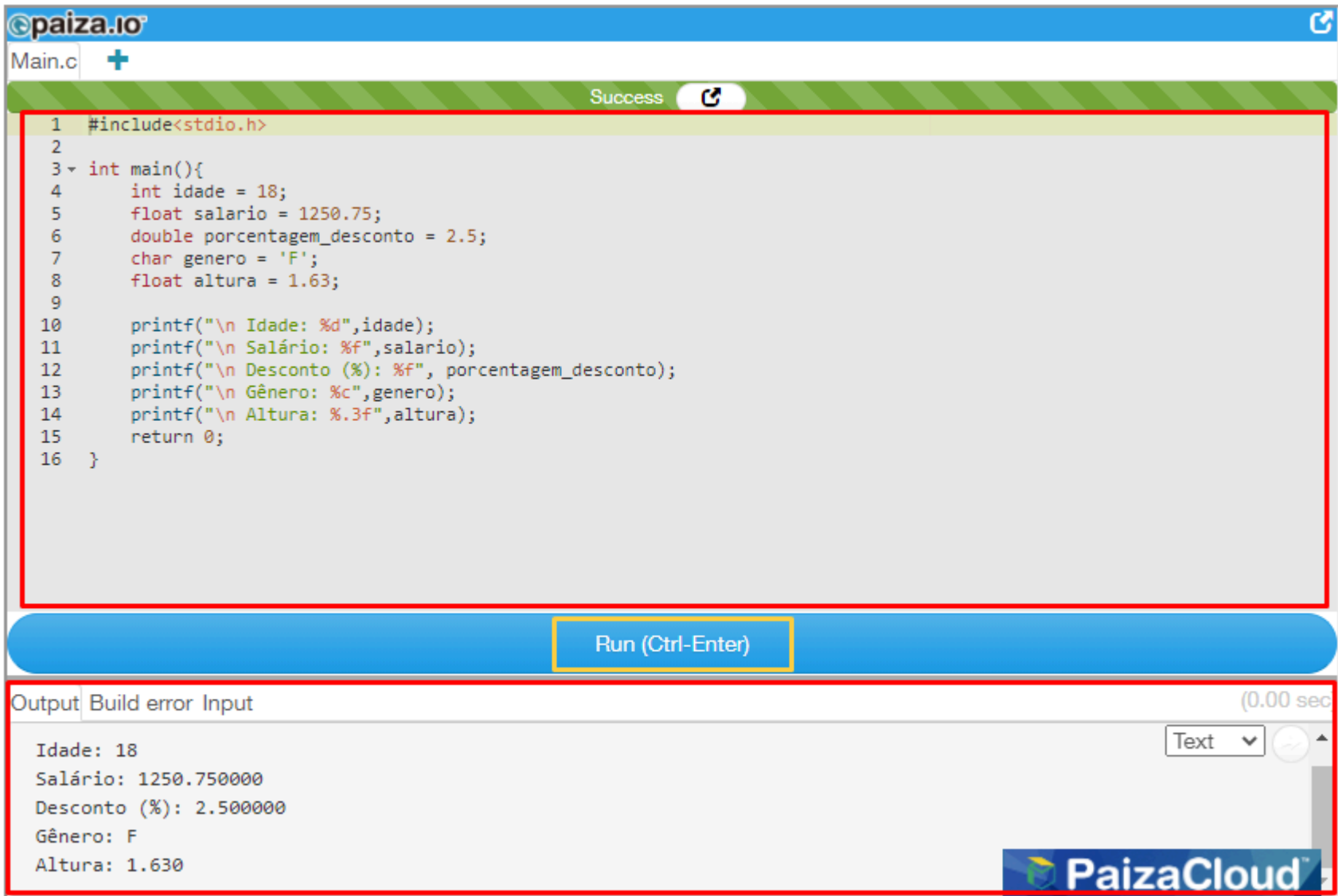
O **Paiza.io** é um software gratuito e disponível para acesso via Web (PAIZA.IO, 2020). Ele permite a execução de códigos de várias linguagens de programação, dentre elas, a linguagem C.



Teste o Código 1.2 utilizando a ferramenta Paiza.io.

A Figura 1.7 mostra a captura de tela do Paiza dividida em 2 blocos.

Figura 1.7 | Orientação Paiza



Ver anotações

Fonte: captura de tela do software Paiza.io elaborada pela autora.

No primeiro bloco, observe que temos os códigos que serão executados.

No segundo bloco, você poderá realizar a entrada de dados (**INPUT**) ou visualizar as saídas (**OUTPUT**) , ou seja, os resultados da execução do código. Mas para isso, precisamos utilizar o botão “**Run**” ou as teclas “**Ctrl-Enter**” para realizar a execução dos códigos.

1 O ENDEREÇO DE MEMÓRIA DE UMA VARIÁVEL

A memória de um computador é dividida em blocos de bytes (1 byte é um conjunto de 8 bits) e cada bloco tem um endereço que o identifica. Podemos fazer uma analogia com os endereços das casas, já que cada casa detém uma localização, e se existissem dois endereços iguais, certamente seria um grande problema. Já sabemos que as variáveis são usadas para reservar um espaço temporário na memória, que tem um endereço único que o identifica. Será que conseguimos

saber o endereço de alocação de uma variável? A resposta é sim. Para sabermos o endereço de uma variável basta utilizarmos o **operador &** na hora de imprimir a variável.

ASSIMILE

O operador & é muito importante na linguagem C, justamente por permitir acessar diretamente os endereços de memória das variáveis.

o

Ver anotações

Conforme avançarmos em nossos estudos, usaremos esse operador & para atribuímos valor a uma variável, a partir do teclado do computador.

Para armazenar valores digitados pelo usuário em uma variável, podemos usar a função `scanf()`, com a seguinte estrutura:

```
scanf("especificador",&variavel);
```

A Figura 1.8 apresenta um exemplo no qual se utilizou o especificador `"%d"` para indicar ao compilador que o valor a ser digitado será um inteiro, e esse valor será guardado no endereço de memória da variável `x`.

Figura 1.8 | Armazenamento em variáveis

```
scanf ("%d", &x);
```

usado para armazenar valores inteiros

será armazenado no endereço da variável x

Fonte: elaborada pela autora.

EXEMPLIFICANDO

Criando programas em C

1. Vamos criar um programa em C que armazena dois valores, um em cada variável. Para isso, o usuário terá que informar as entradas, que

deverão ser armazenadas nas variáveis `valor1` e `valor2`. O Código 1.3 apresenta a solução. Observe que na linha 4, como todas as variáveis eram do mesmo tipo, foram declaradas na mesma linha, separadas por vírgula, e todas foram inicializadas com zero. Nas linhas 7 e 9, os valores digitados pelo usuário serão armazenados nos endereços das variáveis `valor1` e `valor2`, respectivamente.

Ver anotações

Código 1.3 | Impressão de valores armazenados

```
1  #include <stdio.h>
2
3  int main(){
4      float valor1=0, valor2=0;
5
6      printf("\n Digite o primeiro valor:");
7      scanf("%f",&valor1);
8      printf("\n Digite o segundo valor:");
9      scanf("%f",&valor2);
10     printf("Variável 1 = %.2f",valor1);
11     printf("Variável 2 = %.2f",valor2);
12     return 0;
13 }
```

Fonte: elaborado pela autora.

Vamos utilizar a ferramenta Paiza.io para testar impressão dos valores armazenados.

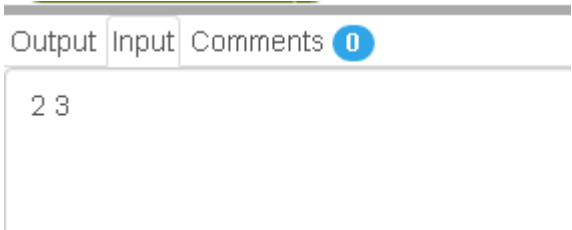


Veja como fica a saída do programa apresentado, após sua execução no Paiza.io.

No Paiza.io, quando o programa exige entrada de valores por meio do teclado, você deve especificá-los na aba **“Input”**.

Para o exemplo anterior (informar dois números reais), você poderia especificar os valores em uma mesma linha, com os valores separados por um espaço em branco, conforme ilustrado a seguir:

Figura 1.9 | Entrada de dados com valores em uma mesma linha e separados por espaço



Fonte: captura de tela do software Paiza.io elaborada pela autora.


Ou também poderia especificar cada valor em uma linha, conforme ilustrado a seguir:

Figura 1.10 | Entrada de dados com valores separados em diferentes linhas



Fonte: captura de tela do software Paiza.io elaborada pela autora.

2. Vamos criar um programa que armazena as coordenadas x, y com os seguintes valores (5,10). Em seguida, vamos pedir para imprimir o endereço dessas variáveis. Veja que, nas linhas 7 e 8, a impressão foi feita com a utilização de &x, &y, o que resulta na impressão dos endereços de memória de x e y em hexadecimal, pois usamos o especificador %x.

 Veja como fica a saída do programa apresentado, utilizando a ferramenta Paiza.io.

CONSTANTES

Entende-se por constante um valor que nunca será alterado. Na linguagem C, existem duas formas de criar valores constantes. A primeira delas é usar a diretiva `#define`, logo após a inclusão das bibliotecas. Nesse caso, a sintaxe será da seguinte forma:

```
#define <nome_da_constante> <valor>
```

Veja que não há ponto e vírgula no final da declaração. Outro aspecto interessante é que a diretiva não utiliza espaço em memória, ela apenas cria um rótulo associado a um valor (MANZANO; MATOS; LOURENÇO, 2015). Assim, durante a compilação do programa, esse valor é substituído nos pontos em que o nome da constante é usado do código.

A outra forma de se criar valores constantes é similar à criação de variáveis, porém, antes do tipo, usa-se a palavra-chave `const`. Portanto, a sintaxe ficará como segue:

```
const <tipo> <nome_da_constante>;
```

Quando se utiliza a segunda forma de declaração, a alocação de espaço na memória segue o mesmo princípio das variáveis, ou seja, `int` alocará 4 bytes; `char`, 1 byte, entre outros. A principal diferença entre as constantes e as variáveis é que o valor de uma constante **nunca** poderá ser alterado. Caso você crie uma constante, por exemplo, `const int x = 10` e tente alterar o valor no decorrer do código, o compilador acusará um erro e não será gerado o arquivo executável.

No Código 1.4, há um exemplo das duas formas de sintaxes para constantes. Na linha 3 definimos uma constante (rótulo) chamada *pi* com valor 3.14. Na linha 6, criamos uma constante usando a palavra-chave `const`. Nas linhas 8 e 9 imprimimos o valor de cada constante. Observe que nada difere da impressão de variáveis.

Código 1.4 | Sintaxe para criação de constantes em C

o
Ver anotações

```
1  #include <stdio.h>
2
3  #define PI 3.14
4
5  int main(){
6      const float G = 9.80;
7
8      printf("\n PI = %f", PI);
9      printf("\n G = %f", G);
10     return 0;
11 }
12
```

Ver anotações

Fonte: elaborado pela autora.

VARIÁVEIS COMPOSTAS

Como já explicitado, as variáveis são usadas para armazenar dados na memória do computador e esses dados podem ser de diferentes tipos (inteiro, decimal, caractere ou booleano), chamados de tipos primitivos. Vimos que podemos armazenar a idade de uma pessoa em uma variável do tipo `int` e a altura em um tipo `float`, por exemplo. Mas, e se fosse necessário armazenar quinze medidas da temperatura de um dispositivo, usaríamos quinze variáveis? Com nosso conhecimento até o momento teríamos que criar as quinze, porém muitas variáveis podem deixar o código maior, mais difícil de ler, entender e manter e, portanto, não é uma boa prática de programação. A melhor solução para armazenar diversos valores dentro de um mesmo contexto é utilizar variáveis compostas. Esse recurso permite armazenar diversos valores utilizando um mesmo nome de variável (MANZANO; MATOS; LOURENÇO, 2015).

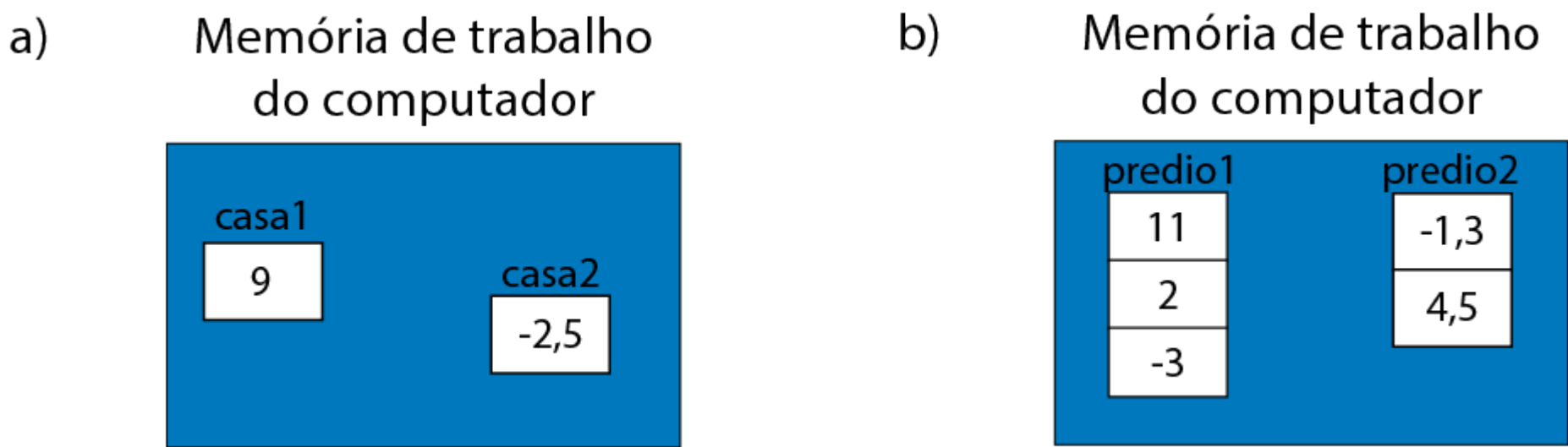
Quando alocamos uma variável primitiva, por exemplo um `int`, um espaço de 4 bytes é reservado na memória, ou seja, **um bloco** é reservado e seu endereço é usado para armazenamento e leitura dos dados. Quando alocamos uma variável

composta do tipo int, **um conjunto de blocos** de 4 bytes será reservado. O tamanho desse conjunto (1, 2, 3 ...N blocos) é especificado pelo programador.

Para entendermos as variáveis compostas, vamos fazer uma analogia entre casas, prédios e as variáveis. Uma casa tem um endereço único que a identifica (que seria equivalente ao nome da variável em um programa), composto por rua, número, bairro, cidade, estado e CEP. Considerando que uma casa é construída para uma família morar, podemos compará-la a uma variável primitiva que armazena um único valor. Por outro lado, um prédio apresenta um endereço composto pelos mesmos parâmetros que a casa (ou seja, uma variável composta tem um nome único, assim como uma variável primitiva), porém nesse mesmo endereço moram várias famílias. Assim são as variáveis compostas: em um mesmo endereço (nome de variável) são armazenados diversos valores. Esses conceitos estão ilustrados na Figura 1.11: veja que do lado esquerdo, a variável casa1 armazena o valor 9 e a variável casa2 armazena o valor -2,5; já no lado direito, a variável predio1 armazena 3 valores inteiros e a variável predio2 armazena dois valores decimais.

Ver anotações

Figura 1.11 | a) Variáveis primitivas b) Variáveis compostas



Fonte: elaborada pela autora.

REFLITA

Se, nas variáveis compostas, em um mesmo nome de variável, por exemplo predio1 (veja o exemplo anterior), são guardados muitos valores, como diferenciar um valor do outro? Assim como os apartamentos em um prédio

têm números para diferenciá-los, as variáveis compostas têm **índices** que as diferenciam.

Ao contrário de uma variável primitiva, uma variável composta tem um endereço na memória e índices para identificar seus subespaços. Existem autores que usam a nomenclatura “variável indexada” para se referir às variáveis compostas, pois sempre existirão índices (index) para os dados armazenados em cada espaço da variável composta (PIVA JUNIOR *et al.*, 2012).

Ver anotações

As variáveis compostas são formadas a partir dos tipos primitivos e podem ser classificadas em homogêneas e heterogêneas. Além disso, podem ser unidimensionais ou multidimensionais, e a bidimensional é mais comumente usada (MANZANO; MATOS; LOURENÇO, 2015). Quando armazenam valores do mesmo tipo primitivo, são homogêneas, mas quando armazenam valores de diferentes tipos, elas são heterogêneas.

VARIÁVEIS COMPOSTAS HOMOGÊNEAS UNIDIMENSIONAIS (VETORES)

As variáveis compostas `predio1` e `predio2`, ilustradas na Figura 1.11, são homogêneas, pois a primeira armazena apenas valores do tipo inteiro, e a segunda, somente do tipo decimal. Além disso, elas também são unidimensionais, e isso quer dizer que elas apresentam a estrutura de uma tabela contendo apenas 1 coluna e N linhas (o resultado não se altera se pensarmos em uma estrutura como uma tabela de 1 linha e N colunas). Esse tipo de estrutura de dados é chamado de **vetor** ou matriz unidimensional (MANZANO; MATOS; LOURENÇO, 2015).

Como o nome **vetor** é o mais utilizado entre os profissionais, adotaremos essa forma no restante do livro. Assim, vetores são conjuntos de dados de um único tipo dispostos de forma contígua, ou seja, um após o outro na memória. A criação

de um vetor é similar a uma variável primitiva, tendo que acrescentar apenas um número entre colchetes indicando qual será o tamanho desse vetor (quantidade de blocos). Portanto, a sintaxe ficará da seguinte forma:

`<tipo> <nome_do_vetor>[tamanho];`

ASSIMILE

O vetor é uma estrutura de dados estática, ou seja, seu tamanho deve ser informado no momento da criação, pelo programador, e não é alterado por nenhum recurso em tempo de execução. Além disso, ao se criar um vetor com N posições, nem todas precisam ser utilizadas. Lembre-se, no entanto, de que quanto maior, mais espaço será reservado na memória de trabalho.

EXEMPLIFICANDO

Vamos criar um vetor em C para armazenar a altura (em metros) de 3 pessoas. Veja no Código 1.5 que, na linha 3, foi criado o vetor *altura*, que foi inicializado com valores. Para armazenar valores no vetor no momento da criação, colocamos os elementos entre chaves (`{ }`) separados por vírgula. Da linha 4 a 6 é feita a impressão dos valores guardados no vetor *altura*.

Código 1.5 | Inicializando um vetor utilizando operador chaves (`{ }`)

```
1  #include <stdio.h>
2  int main(){
3      float altura[3] = {1, 1.5, 1.7};
4      printf("\n Vetor altura[0] = %f",altura[0]);
5      printf("\n Vetor altura[1] = %f",altura[1]);
6      printf("\n Vetor altura[2] = %f",altura[2]);
7      return 0;
8  }
```

Fonte: elaborado pela autora.

0

Ver anotações



Veja a execução do código utilizando a ferramenta Paiza.io. Observe os valores do vetor altura sendo impressos.

Conforme pode ser visto no exemplo anterior, cada elemento no vetor é acessado por meio do seu índice, **que sempre começará pelo valor zero**. Por isso, no Código 1.5, na linha 4, usou-se `altura[0]` para imprimir o primeiro valor, `altura[1]` para imprimir o segundo e `altura[2]` para imprimir o terceiro.

o
Ver anotações

Ainda a respeito do exemplo anterior, o índice do código apresentado é usado tanto para leitura como para escrita.

Outra forma de atribuir valores a um vetor altura é utilizando colchetes ([]), da seguinte maneira:

Código 1.6 | Atribuindo valores a um vetor com colchetes

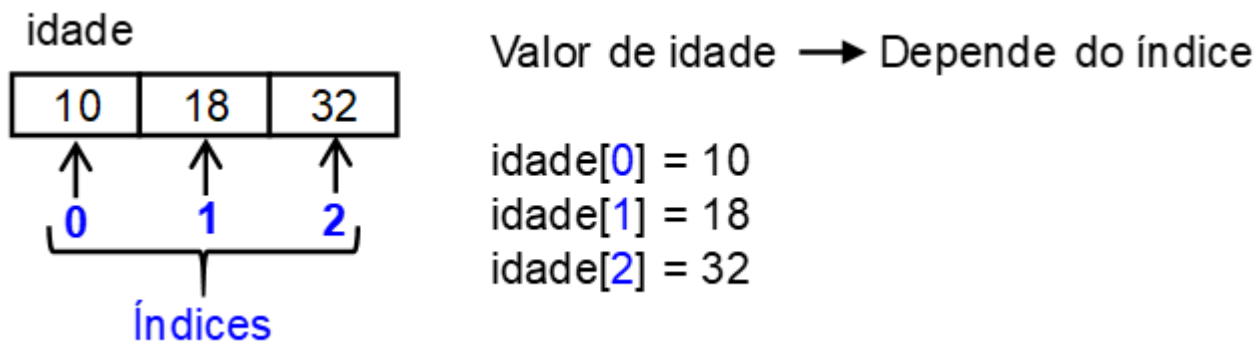
```
1  #include <stdio.h>
2  int main(){
3      float altura[3];
4      altura[0] = 1.50;
5      altura[1] = 1.80;
6      altura[2] = 2.02;
7      printf("\n Vetor altura[0] = %f",altura[0]);
8      printf("\n Vetor altura[1] = %f",altura[1]);
9      printf("\n Vetor altura[2] = %f",altura[2]);
10     return 0;
11 }
```

Fonte: elaborado pela autora.

Para ajudar a compreensão, observe a Figura 1.12, que representa um esquema para um vetor denominado *idade* na memória do computador. O valor do vetor depende da posição, ou seja, do índice. Ressalta-se que um vetor com N posições terá seus índices variando de 0 até N-1. Veja que o vetor *idade* tem capacidade de 3; com isso, seu índice varia de 0 até 2.

Ver anotações

Figura 1.12 | Vetor idade



Fonte: elaborada pela autora.

Na maioria dos programas, os dados a serem utilizados são digitados pelo usuário ou lidos de alguma base de dados. Para guardar um valor digitado pelo usuário em um vetor, usamos a função `scanf()`, porém, na variável, deverá ser especificado em qual posição do vetor se deseja guardar. Então, se quiséssemos guardar uma idade digitada pelo usuário na primeira posição do vetor, faríamos da seguinte forma:

```
printf("Digite uma idade: ");  
scanf("%d",&idade[0]);
```

EXEMPLIFICANDO

Vamos criar um vetor para armazenar a idade de 3 pessoas, cujos valores são informados via teclado. Logo após a leitura das idades, o programa deve calcular e imprimir na tela a média de idade dessas pessoas. Veja no Código 1.7 que, na linha 3, foi criado o vetor do tipo inteiro, denominado *idade*. Ele foi inicializado com valores nulos (zero), o que é uma boa prática, para se evitar a realização de cálculos com lixo de memória. Logo após, na linha 4, declarou-se ainda uma variável primitiva, denominada *media*. Para

armazenar valores no vetor, utilizou-se a função `scanf()` – linhas 6, 8 e 10. Nas linhas 11 a 12, são realizados, respectivamente, o cálculo da média das idades, bem como a impressão dela na tela.

Código 1.7 | Leitura de um vetor em C

```
1  #include <stdio.h>
2  int main(void){
3      int idade[3] = {0, 0, 0};
4      float media = 0.0;
5      printf("\n Informe a idade da pessoa 1: ");
6      scanf("%d", &idade[0]);
7      printf("\n Informe a idade da pessoa 2: ");
8      scanf("%d", &idade[1]);
9      printf("\n Informe a idade da pessoa 3: ");
10     scanf("%d", &idade[2]);
11     media = (idade[0] + idade[1] + idade[2]) / 3;
12     printf("\n Média de idade = %.2f", media);
13     return 0;
14 }
```

Ver anotações

Fonte: elaborado pela autora.



Utilize o Paiza.io para executar o código acima.

VETOR DE CARACTERES (*STRING*)

Como é formada uma palavra? Por uma sequência de caracteres, correto? Vimos que o caractere é um tipo primitivo nas linguagens de programação. Sabendo que uma palavra é uma cadeia de caracteres, podemos concluir que esta é, na verdade, um vetor de caracteres (MANZANO; MATOS; LOURENÇO, 2015). No mundo da programação, um vetor de caracteres é chamado de *string*, portanto adotaremos essa nomenclatura.

A declaração de uma *string* em C é feita da seguinte forma:

```
char <nome_da_string>[tamanho];
```

Exemplos:

```
char nome[16];
```

```
char sobrenome[31];
```

```
char frase[101];
```

o

Ver anotações

Ao criarmos uma *string* em C, temos que nos atentar ao seu tamanho, pois a última posição da *string* é reservada pelo compilador, que atribui o valor “\0” para indicar o final da sequência. Portanto, a *string* nome[16] apresenta 15 “espaços” disponíveis para serem preenchidos.

A atribuição de valores à *string* pode ser feita no momento da declaração de duas formas: (i) entre chaves informando cada caractere, separados por vírgula (igual aos outros tipos de vetores); e (ii) atribuindo a palavra (ou frase) entre aspas duplas.

Exemplos:

```
char nome[16]={'J','o','a','o'};
```

```
char sobrenome[31] = "Alberto Gomes";
```

Existem várias funções que fazem a leitura e a impressão de *string* em C.

Estudaremos duas delas. A primeira já é conhecida, a função scanf(), mas agora usando o especificador de formato %s para indicar que será armazenada uma *string*. Veja o código responsável por armazenar o nome digitado por um usuário:

```
char nome[16];
```

```
printf("\n Digite um nome:");
```

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

```
printf("\n Nome digitado: %s", nome);
```

Uma observação importante é que nesse caso o operador & não é usado na função `scanf()`. Isso ocorre porque, ao usarmos o nome de um vetor sem a especificação de um índice para ele, o compilador já entende que se trata de um endereço de memória.

Essa forma de atribuição tem uma limitação: só é possível armazenar palavras simples; compostas, não. Isso acontece porque a função `scanf()` interrompe a atribuição quando encontra um espaço em branco. Para contornar essa limitação, uma opção é usar a função `fgets()`, que também faz parte do pacote padrão `<stdio.h>`. Essa função apresenta a seguinte sintaxe:

```
fgets(destino,tamanho,fluxo);
```

O *destino* especifica o nome da *string* que será usada para armazenar o valor lido do teclado. O *tamanho* deve ser o mesmo da declaração da *string*. O *fluxo* indica de onde está vindo a *string*, no nosso caso, sempre virá do teclado, portanto usaremos `stdin` (*standard input*).

Exemplo:

```
char frase[101];  
printf("\n Digite uma frase:");  
fflush(stdin);  
fgets(frase, 101, stdin);  
printf("\n Frase digitada: %s", frase);
```

Veja que antes de usar o `fgets()`, usamos a função `fflush(stdin)`. Apesar de não ser obrigatório, isso é uma boa prática, pois garante que a entrada padrão (`stdin`) seja limpa (sem informações de leituras anteriores) antes de armazenar.

REFLITA

O uso da função `fflush()`, conforme comentado anteriormente, tem suas limitações:

o

Ver anotações

- O programador pode esquecer de chamar essa função antes de realizar a leitura de uma string.
- Para cada leitura com `fgets()`, o programador precisará fazer também uma chamada para `fflush()`.

Você consegue imaginar alguma forma de minimizar essas limitações? Apesar de você não ter visto ainda como declarar novas funções, você já sabe para que elas servem e como utilizá-las. Então, a ideia seria criar uma função, por exemplo `lerString()`, a qual executaria a função `fflush()` e depois `fgets()`. Assim, o programador poderia usar sempre a função `lerString()`, sem ter que se preocupar com detalhes de limpeza da entrada padrão (`stdin`).

Aguarde, pois voltaremos a esse assunto em uma seção apropriada.

Ver anotações

VARIÁVEIS COMPOSTAS HOMOGÊNEAS BIDIMENSIONAIS (MATRIZES)

Observe a Tabela 1.3, que apresenta alguns dados fictícios sobre a temperatura da cidade de São Paulo em uma determinada semana. Essa tabela representa uma estrutura de dados matricial com 7 linhas e 2 colunas. Para armazenarmos esses valores em um programa de computador, precisamos de uma variável composta bidimensional ou, como é mais comumente conhecida, uma matriz bidimensional.

Tabela 1.3 | Temperatura máxima de São Paulo na última semana

Dia	Temperatura (°C)
1	26,1
2	27,7
3	30,0

Dia	Temperatura (°C)
4	32,3
5	27,6
6	29,5
7	29,9

Ver anotações

Fonte: elaborada pela autora.

Para criarmos uma matriz em C usamos a seguinte sintaxe:

`<tipo_dado> <nome_da_matriz>[numero_linhas][numero_colunas];`

Observe o exemplo de declaração a seguir com uma matriz de inteiros e uma matriz de float.

- 1. `int coordenadas[3][2];`
- 2. `float temperaturas[7][2];`

Na linha 1 do código é criada uma estrutura com 3 linhas e 2 colunas. Na linha 2, é criada a estrutura da Tabela 1.3 (7 linhas e 2 colunas).

A criação e manipulação de matrizes bidimensionais exige a especificação de dois índices, portanto, a atribuição de valores deve ser feita da seguinte forma:

`matriz[M][N] = valor;`

M representa a linha que se pretende armazenar e **N**, a coluna. Assim como nos vetores, aqui os índices sempre iniciarão em zero.

Vamos criar uma matriz em C para armazenar as notas do primeiro e segundo bimestre de três alunos. Veja, na linha 3 do Código 1.8, que criamos uma matriz chamada notas, com 3 linhas e 2 colunas, o que significa que serão armazenados 6

valores (linhas x colunas). Nas linhas 6 e 7 do código, armazenamos as notas do primeiro aluno: veja que a linha não se altera (primeiro índice), já a coluna sim (segundo índice). O mesmo acontece para o segundo e terceiro alunos, que são armazenados respectivamente na segunda e terceira linhas da matriz.

No código, os textos que aparecem após o símbolo `//` são chamados de comentários. Eles não têm qualquer efeito sobre o comportamento do programa, pois são descartados durante o processo de compilação do código. Comentários servem, então, para melhorar a legibilidade do código para o programador que vai precisar, ler, entender e manter o programa futuramente.

o

Ver anotações

Código 1.8 | Matriz em C

```
1  #include <stdio.h>
2  int main(){
3      float notas[3][2];
4
5      //aluno 1
6      notas[0][0] = 10;
7      notas[0][1] = 8.5;
8
9      //aluno 2
10     notas[1][0] = 5.5;
11     notas[1][1] = 2.7;
12
13     //aluno 3
14     notas[2][0] = 4;
15     notas[2][1] = 10;
16     return 0;
17 }
```

Fonte: elaborado pela autora.

A Figura 1.13 ilustra a estrutura de dados que é criada na memória para o código do Código 1.8. Veja que as linhas foram usadas para representar os alunos e as colunas para as notas.

Figura 1.13 | Esquema de atribuição de notas

		Nota 1	Nota 2
		coluna 0	coluna 1
Aluno 1	linha 0 →	10,0	8,5
Aluno 2	linha 1 →	5,5	2,7
Aluno 3	linha 2 →	4,0	10,0

Fonte: elaborada pela autora.

Para armazenar valores digitados pelo usuário em uma matriz, usamos a função `scanf()`, indicando os dois índices para selecionar a posição que se deseja guardar. Para impressão de valores, também devemos selecionar a posição por meio dos dois índices.

Exemplo:

```
float notas[3][2];
printf("Digite uma nota: ");
scanf("%f", as[1][0]);
printf("Nota digitada: %.2f", notas[1][0]);
```

Nesse exemplo, a nota digitada será guardada na linha 1, coluna 0, e será exibida na tela usando duas casas decimais.

VARIÁVEIS COMPOSTAS HETEROGÊNEAS (*STRUCTS*)

Já sabemos como otimizar o uso de variáveis usando as estruturas compostas (vetor e matriz). Porém, só podemos armazenar valores de um mesmo tipo. Além das estruturas homogêneas, a linguagem de programação C oferece variáveis

0

Ver anotações

compostas heterogêneas chamadas de **estruturas** (*structs*) ou, ainda, de registros por alguns autores (SOFFNER, 2013).

Assim como associamos os vetores e as matrizes a tabelas, podemos associar uma estrutura a uma ficha de cadastro com diversos campos. Por exemplo, o cadastro de um cliente poderia ser efetuado a partir da inserção do nome, idade, CPF e endereço em uma *structs*.

Na linguagem C, a criação de uma estrutura deve ser feita antes da função `main()` e deve apresentar a seguinte sintaxe:

```
struct < nome >{  
    <tipo> <nome_da_variavel1>;  
    <tipo> <nome_da_variavel2>;  
    ...  
};
```

<nome> é o nome da estrutura, por exemplo, cliente, carro, fornecedor. As variáveis internas são os campos nos quais se deseja guardar as informações dessa estrutura. Na prática, uma estrutura funciona como um “tipo de dado” e seu uso sempre será atribuído a uma ou mais variáveis.

EXEMPLIFICANDO

Vamos criar uma estrutura para armazenar o modelo, o ano e o valor de um automóvel.

No Código 1.9, “Struct em C”, a estrutura `automove1` foi criada entre as linhas 3 e 7. Mas, para utilizar essa estrutura, na linha 10 foi criada a variável `dadosAutomove11`, que é do tipo `struct automove1`. Somente após essas especificações é que podemos atribuir algum valor para as variáveis dessa estrutura de dados.

```
1  #include<stdio.h>
2
3  struct automovel{
4      char modelo[20];
5      int ano;
6      float valor;
7  };
8
9  int main(){
10     struct automovel dadosAutomovel1;
11 }
```

Ver anotações 0

Fonte: elaborado pela autora.

O acesso aos dados da *struct* (leitura e escrita) é feito com a utilização do nome da variável que recebeu como tipo a estrutura com um ponto (.) e o nome do campo da estrutura a ser acessado. Por exemplo, para acessar o campo ano da struct `automovel` do Código 1.9, “Struct em C”, devemos escrever: `dadosAutomovel1.ano`.



Veja no Paiza.io como realizar “Acesso aos dados de uma estrutura em C” a implementação completa para guardar valores digitados pelo usuário na estrutura `automovel` e depois imprimi-los.

VARIÁVEL DO TIPO PONTEIRO

Além das variáveis primitivas e compostas, existe um tipo de variável muito importante na linguagem C. São os chamados **ponteiros**, por meio dos quais podemos manipular outras variáveis e outros recursos por meio do acesso ao endereço de memória (SOFFNER, 2013). Existe uma relação direta entre um

ponteiro e endereços de memória e é por esse motivo que esse tipo de variável é utilizado, principalmente para manipulação de memória, dando suporte às rotinas de alocação dinâmica (MANZANO; MATOS; LOURENÇO, 2015).

Variáveis do tipo ponteiro são usadas exclusivamente para armazenar endereços de memória. O acesso à memória é feito usando dois operadores: o asterisco (*), utilizado para declaração do ponteiro, e o “&”, que, como já vimos, é usado para acessar o endereço da memória, por isso é chamado de **operador de referência**.

A sintaxe para criar um ponteiro tem como padrão:

```
<tipo> *<nome_do_ponteiro>;
```

Exemplo: `int *idade;`

Nesse exemplo, é criado um ponteiro do tipo inteiro, e isso significa que ele deverá “apontar” para o endereço de uma variável desse tipo. A criação de um ponteiro só faz sentido se for associada a algum endereço de memória. Para isso, usa-se a seguinte sintaxe:

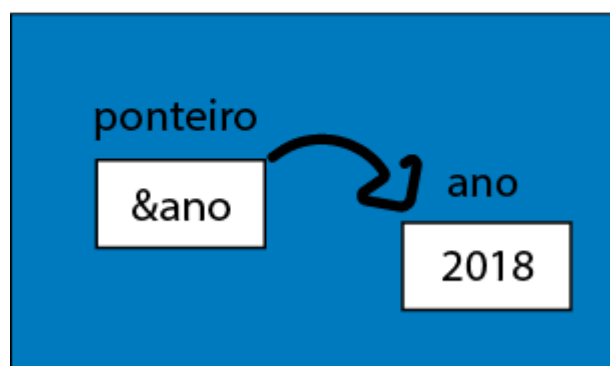
```
1. int ano = 2018;
```

```
2. int *ponteiro_para_ano = &ano;
```

Na linha 1 criamos uma variável primitiva inteira, denominada ano, com valor 2018, e na linha 2 associamos um ponteiro chamado ponteiro_para_ano ao endereço da variável primitiva ano. Agora tudo que estiver atribuído à variável ano estará acessível ao ponteiro ponteiro_para_ano. A Figura 1.15 ilustra, de forma simplificada, o esquema de uma variável com um ponteiro na memória. O conteúdo do ponteiro é o endereço da variável a que ele aponta e que ele também ocupa espaço na memória.

Figura 1.15 | Ponteiro e variável primitiva na memória

Memória de trabalho do computador



Fonte: elaborada pela autora.

0
Ver anotações

Para finalizar esta seção, vamos ver como imprimir as informações de um ponteiro. Podemos imprimir o conteúdo do ponteiro, que será o endereço da variável a que ele aponta. Utilizando o ponteiro criado anteriormente (`ponteiro_para_ano`), temos a seguinte sintaxe:

```
printf("\n Conteúdo do ponteiro: %p", ponteiro_para_ano);
```

O especificador de formato `%p` é usado para imprimir o endereço de memória armazenado em um ponteiro, em hexadecimal (o `%x` também poderia ser usado).

Também podemos acessar o conteúdo da variável que o ponteiro aponta, com a seguinte sintaxe:

```
printf("\n Conteúdo da variável pelo ponteiro: %d",  
*ponteiro_para_ano);
```

A diferença do comando anterior é o asterisco antes do nome do ponteiro.

Podemos, também, imprimir o endereço do próprio ponteiro, por meio da seguinte sintaxe:

```
printf("\n Endereco do ponteiro: %p", &ponteiro_para_ano);
```

EXEMPLIFICANDO

O nome de um vetor nada mais é do que um ponteiro para o endereço de memória do seu primeiro elemento. Para isso, faça o seguinte:



Execute o código no Paiza.io e veja que a saída é exatamente a mesma, ou seja, pode-se utilizar o índice 0 do vetor ou o operador * para obter o valor armazenado no endereço de memória apontado por num.

Agora, imagine como o compilador sabe qual endereço de memória acessar para um índice X do vetor?

Vejamos: se sabemos o endereço de memória do primeiro elemento e também sabemos qual é o tamanho do bloco de memória alocado para cada elemento (isso é obtido com base no tipo do vetor, por exemplo, um inteiro ocupa 4 bytes, um char ocupa 1 byte etc.), então fica fácil saber qual o endereço de memória do índice X.

*Endereço de memória do índice X = endereço inicial + (tamanho do bloco de memória de um elemento * X)*

Assim, supondo que o endereço de memória inicial do vetor num seja 1080, que um inteiro ocupe 4 bytes de memória, então a posição de memória do elemento da posição 1 do vetor é dada por:

*Endereço do num[1] = 1080 + (4 * 1) = 1084*

Por isso é tão rápido para um computador acessar qualquer elemento de um vetor.

o

Ver anotações

A compreensão do uso das variáveis é imprescindível, pois conheceremos métodos que podem ser usados para resolver problemas nos quais os dados são a matéria-prima. Continue seus estudos!

FAÇA VALER A PENA

Questão 1

Variáveis são usadas para guardar valores temporariamente na memória do computador. A linguagem C oferece recursos para que seja possível conhecer o endereço de memória que foi alocado. Durante a execução de um programa, uma variável pode assumir qualquer valor, desde que esteja de acordo com o tipo que foi especificado na sua criação.

A respeito dos tipos primitivos de variáveis, assinale a alternativa correta.

- a. Todas as linguagens de programação têm os mesmos tipos primitivos de dados.
- b. Para todos os tipos primitivos na linguagem C são alocados os mesmos espaços na memória.
- c. Os valores numéricos podem ser armazenados em tipos primitivos inteiros ou de ponto flutuante.
- d. O número 10 é inteiro e por isso não pode ser guardado em uma variável primitiva do tipo float.
- e. O número 12.50 é decimal e por isso não pode ser guardado em uma variável primitiva do tipo int, pois gera um erro de compilação.

Questão 2

Constantes são usadas para guardar temporariamente valores fixos na memória de trabalho. O valor armazenado em uma constante não pode ser alterado em tempo de execução e essa é a principal diferença com relação às variáveis.

A respeito das constantes na linguagem C, assinale a alternativa correta.

- a. Valores constantes podem ser declarados usando o comando `#define <nome> <valor>`.
- b. O comando `const int a = 10` alocará um espaço de 1 byte para a constante a.
- c. O comando `const int a = 13.4` resultará em um erro de compilação.
- d. O compilador trata o comando `const` da mesma forma que trata o comando `define`.
- e. A constante definida pelo comando `#define g 9.8` ocupará 4 bytes na memória.

Ver anotações

Questão 3

A linguagem C de programação utiliza especificadores de formato para identificar o tipo de valor guardado nas variáveis e constantes. Eles devem ser usados tanto para leitura de um valor como para a impressão. Quando um programa é executado, o compilador usa esses elementos para fazer as devidas referências e conexões, por isso o uso correto é fundamental para os resultados.

Considerando o código apresentado, analise as asserções em seguida e, depois, escolha a opção correta.

Ver anotações

```
1  #include <stdio.h>
2
3  int main(){
4      int idade = 0;
5      float salario = 0;
6      char a_letra = 'a';
7      char A_letra = 'A';
8  }
```

- I. O comando scanf(“%f”,&idade) guardará o valor digitado na variável idade.
- II. O comando printf(“%d”,a_letra) imprimirá a letra *a* na tela.
- III. O comando printf(“%c”,A_letra) imprimirá a letra *A* na tela.

Com base no contexto apresentado, é correto o que se afirma apenas em:

a. I.

b. II.

c. I e II.

d. III.

e. II e III.

REFERÊNCIAS

AGUILAR, L. J. **Fundamentos de programação**: algoritmos, estruturas de dados e objetos. 3. ed. Porto Alegre: AMGH, 2011.

BARANIUK, C. As falhas numéricas que podem causar desastres. **BBC News Brasil**, 14 maio 2015. Disponível em: <https://bbc.in/3llL7Xm>. Acesso em: 20 out. 2020.

CORDEIRO, T. O que foi o Bug do Milênio? **Super Interessante**, São Paulo, 4 jul. 2018. Disponível em: <https://bit.ly/35khiRI>. Acesso em: 23 out. 2020.

DEITEL, P.; DEITEL, H. C. **Como programar**. 6. ed. São Paulo: Pearson, 2011.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron, 2000.

GEEKS FOR GEEKS. **Errors in C/C++**. Geeks for Geeks, Noisa, Uttar Pradesh, 4 jan. 2019. Disponível em: <https://bit.ly/3phFzPZ>. Acesso em: 21 out. 2020.

LOPES, A.; GARCIA, G. **Introdução à programação**. 8. reimp. Rio de Janeiro: Elsevier, 2002.

MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. 2. ed. São Paulo: Érica, 2015.

MICROSOFT. **Math Constants**. Syntax. Visual Studio. Microsoft, 2016. Disponível em: <https://bit.ly/32ADhSd>. Acesso em: 11 mar. 2020.

ONLINE GDB BETA. Página inicial. GDB Online, 2020. Disponível em: <https://www.onlinegdb.com/>. Acesso em: 20 out. 2020.

PAIZA.IO. Página inicial. Paiza Inc., Tóquio, 2020. Disponível em: <https://paiza.io/en>. Acesso em: 29 out. 2020.

PEREIRA, S. do L. **Linguagem C**. Ime: USP, 2017. Disponível em: <https://bit.ly/3naTKEA>. Acesso em: 11 mar. 2020.

PEREIRA, A. P. **O que é algoritmo?** TecMundo, 12 maio 2009. Disponível em: <https://bit.ly/2GVDsjC>. Acesso em: 19 mar. 2018.

PIVA JUNIOR, D. *et al.* **Algoritmos e programação de computadores.** Rio de Janeiro: Elsevier, 2012.

SALIBA, W. L. C. **Técnica de programação:** uma abordagem estruturada. São Paulo: Makron, 1993.

SANTOS, D. **Processamento da linguagem natural:** uma apresentação através das aplicações. Organização: Ranchhod. Lisboa: Caminho, 2001.

SCHILDT, H. **C Completo e total.** 3. ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e programação em linguagem C.** São Paulo: Saraiva, 2013.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos.** Rio de Janeiro: LTC, 1994.