Ponteiros

Algoritmos e Estruturas de Dados I

Prof. Cristiano Rodrigues

Prof. Lucas Astore

Ponteiro

- Ponteiros são variáveis cujos valores são endereços de memória
- Seu valor indica onde uma variável está armazenada, não o que está armazenado
- Um ponteiro proporciona um modo de acesso a uma variável sem referenciá-la diretamente

Ponteiro

- Ponteiros são utilizados em situações em que o uso do nome de uma variável não é permitido ou é indesejável
- Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem – passagem por referência
- Ponteiros alocam e desalocam memória dinamicamente no sistema

Ponteiro

- Para conhecermos o endereço ocupado por uma variável usamos o operador de endereços (&)

```c
#include <stdio.h>

int main()
{
        int i = 1, j = 33, k = 790;
        printf(" %p\n %p\n %p\n", &i, &j, &k);
        return 0;
}
```

Resultado:

Ponteiro

- Para conhecermos o endereço ocupado por uma variável usamos o operador de endereços (&)

```c
#include <stdio.h>

int main()
{
        int i = 1, j = 33, k = 790;
        printf(" %p\n %p\n %p\n", &i, &j, &k);
        return 0;
}
```

Resultado:

```
0x16f94f298
0x16f94f294
0x16f94f290
```

Ponteiro

- Um ponteiro, diferentemente de uma variável comum, contém um endereço de uma variável que contém um valor específico. Um ponteiro referencia um valor indiretamente

- Ponteiros devem ser definidos antes de sua utilização, como qualquer outra variável

- Exemplo:

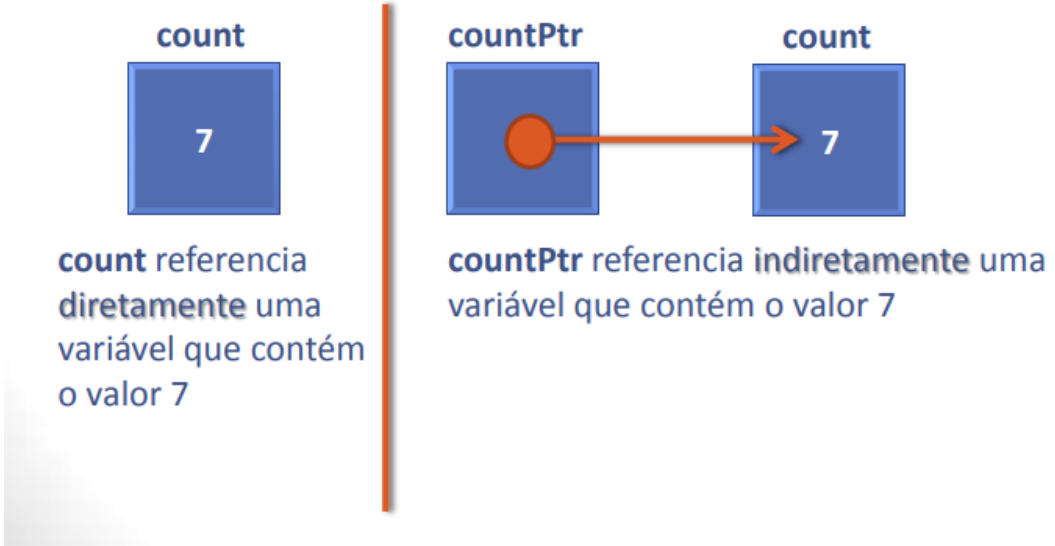
```
int *countPtr, count;
```

Ponteiro

- A variável count é definida para ser um inteiro, e não um ponteiro para int
- Se quisermos que count seja um ponteiro, devemos alterar a declaração int count para int *count

Exemplo:

int *countPtr, count;



Ponteiro

- Para atribuir um valor ao ponteiro, usamos apenas seu nome de variável. Esse valor deve ser um endereço de memória, portanto obtido com o operador &:

```
int a;
int *p;
p = &a;
```

Se a uma variável e p vale &a então dizer *p é o mesmo que dizer i.

Ponteiro

Cuidado:

- A notação *, usada para declarar variáveis de ponteiro, não distribui para todos os nomes de variáveis em uma declaração
- Cada ponteiro precisa ser declarado com o * prefixado ao nome
- Exemplo:

```
int *xPtr, *yPtr;
```

Ponteiro

Dica:

- Inclua as letras Ptr nos nomes de variáveis de ponteiros para deixar claro que essas variáveis são ponteiros

- Inicialize os ponteiros para evitar resultados inesperados

Ponteiro

- Como o ponteiro contém um endereço, podemos também atribuir um valor à variável guardada nesse endereço, ou seja, à variável apontada pelo ponteiro. Para isso, usamos o operador * (asterisco), que basicamente significa "o valor apontado por"

```c
#include <stdio.h>
int main()
{
  int i = 10;
  int *p = &i;
  *p = 5;
  printf("%d\t%d\t%p\n", i, *p, p);
  return 0;
}
```

Saída:

Ponteiro

- Como o ponteiro contém um endereço, podemos também atribuir um valor à variável guardada nesse endereço, ou seja, à variável apontada pelo ponteiro. Para isso, usamos o operador * (asterisco), que basicamente significa "o valor apontado por"

```c
#include <stdio.h>
int main()
{
  int i = 10;
  int *p = &i;
  *p = 5;
  printf("%d\t%d\t%p\n", i, *p, p);
  return 0;
}
```

Saída:

```
5	5	0x16fafb298
```

Ponteiro

- Ponteiros devem ser inicializados quando são definidos ou então em uma instrução de atribuição
- Ponteiros podem ser inicializados com NULL, zero, ou um endereço

- NULL: não aponta para nada, é uma constante simbólica
- Inicializar um ponteiro com zero é o mesmo que inicializar com NULL

```
int *xPtr = NULL;
int *yPtr = 0;
```

Ponteiro

Suponhamos dois ponteiros inicializados p1 e p2. Podemos fazer dois tipos de atribuição entre eles:

p1 = p2; ou *p1 = *p2;

Ponteiro

p1 = p2

- Esse primeiro exemplo fará com que p1 aponte para o mesmo lugar que p2. Ou seja, usar p1 será equivalente a usar p2 após essa atribuição

Ponteiro

```
*p1 = *p2;
```

- Nesse segundo caso, estamos a igualar os valores apontados pelos dois ponteiros: alteraremos o valor apontado por p1 para o valor apontado por p2

Ponteiros

Para dominar a linguagem C, é essencial dominar ponteiro. Algumas razões para o uso de ponteiros:

- Manipular elementos de matrizes;
- Receber argumentos em funções que necessitem modificar o argumento original;
- Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro;
- Alocar e desalocar memória do sistema;
- Passar para uma função o endereço de outra.

```c
#include <stdio.h>

int main()
{
  int *A, *B, x = 10, y = 20;
  A = &x;
  B = &y;
  printf("\nA = %d", *A);
  printf("\nB = %d", *B);
  A = B;
  *B = 5;
  printf("\nA = %d", *A);
  printf("\nB = %d\n", *B);
  return 0;
}
```

Saída:

```c
#include <stdio.h>

int main()
{
  int *A, *B, x = 10, y = 20;
  A = &x;
  B = &y;
  printf("\nA = %d", *A);
  printf("\nB = %d", *B);
  A = B;
  *B = 5;
  printf("\nA = %d", *A);
  printf("\nB = %d\n", *B);
  return 0;
}
```

Saída:

A = 10

B = 20

A = 5

B = 5

```c
#include <stdio.h>
int main()
{
  int x = 4, y = 7, *px, *py;
  printf("\n &x=%p e x=%d", &x, x);
  printf("\n &y=%p e y=%d", &y, y);
  printf("\n");
  px = &x;
  py = &y;
  printf("\n px=%p e *px=%d", px, *px);
  printf("\n py=%p e *py=%d", py, *py);
  printf("\n");
  return 0;
}
```

Saída:

```c
#include <stdio.h>
int main()
{
  int x = 4, y = 7, *px, *py;
  printf("\n &x=%p e x=%d", &x, x);
  printf("\n &y=%p e y=%d", &y, y);
  printf("\n");
  px = &x;
  py = &y;
  printf("\n px=%p e *px=%d", px, *px);
  printf("\n py=%p e *py=%d", py, *py);
  printf("\n");
  return 0;
}
```

Saída:

&x=0x16d2f7298 e x=4

&y=0x16d2f7294 e y=7

px=0x16d2f7298 e *px=4

py=0x16d2f7294 e *py=7

```c
#include <stdio.h>
int main()
{
  int x, y, *px = &x;
  *px = 14;
  y = *px;
  printf("\ny=%d ", y);
  printf("\nx=%d ", x);
  return 0;
}
```

Saída:

```c
#include <stdio.h>
int main()
{
  int x, y, *px = &x;
  *px = 14;
  y = *px;
  printf("\ny=%d ", y);
  printf("\nx=%d ", x);
  return 0;
}
```

Saída:

y=14

x=14

Memória e ponteiros

Sintaxe

Ponteiro é uma variável que armazena o endereço de outra variável.

```
int a = 10;

int *ptr; // Declaração de um ponteiro para int

ptr = &a; // Atribuição do endereço de 'a' ao
          ponteiro 'ptr'
```

Exemplo:

```c
int main()
{
    int x = 100;
    int *px = &x;
    printf("valor de x     = %d\n", x);
    printf("endereço de x = %p\n", &x); // %p: formato para ponteiro
    printf("endereço de x = %p\n", px); // %p: formato para ponteiro
    printf("valor de x     = %d\n", *px);
    return 0;
}
```

Exemplo de saída (computador com 64 bits):

```
valor de x     = 100
endereço de x = 0x7ffedfc1e378
endereço de x = 0x7ffedfc1e378
valor de x     = 100
```

Memória

- Quais são as características da variável x ?

```
int x = 9;
```

- Tipo: int
- Nome: x
- Endereço de memória: 0xbfd267c4
- Valor: 9

Para acessar o endereço de memória :

```
printf("O endereço de memória de x é %p\n", &x);
```

Memória

Endereço	Valor
00010000	??
00010001	??
00010002	??
00010003	??
00010004	??
00010005	??
00010006	??
00010007	??
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??

- A memória é formada por várias células
- Cada célula contém um endereço e um valor
- O tamanho do endereço e do valor dependem da arquitetura (32 ou 64 bits)

Memória

Endereço	Valor
00010000	??
00010001	??
00010002	??
00010003	??
00010004	??
00010005	??
00010006	??
00010007	??
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??

i

- Exemplo: O caractere i ocupa 1 byte na memória

```
1  int main()
2  {
3      char i;
4      return 0;
5  }
```

Memória

Endereço	Valor
00010000	??
00010001	
00010002	
00010003	
00010004	??
00010005	??
00010006	??
00010007	??
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??

i

- Exemplo: O inteiro i ocupa 4 bytes na memória (float tbm ocupa)

```
1  int main()
2  {
3      int i;
4      return 0;
5  }
```

Memória

Endereço	Valor
00010000	??
00010001	
00010002	
00010003	
00010004	
00010005	
00010006	
00010007	
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??

i

- Exemplo: O double i ocupa 8 bytes na memória (float tbm ocupa)

```
1  int main()
2  {
3      double i;
4      return 0;
5  }
```

Memória

Endereço	Valor	
00010000	??	c
00010001		
00010002		
00010003		
00010004	??	i
00010005		
00010006		
00010007		
00010008	??	f
00010009		
0001000A		
0001000B		
0001000C	??	d
0001000D		
0001000E		
0001000F		

- Exemplo: Note os quatro ponteiros. Todos requerem o mesmo tamanho de memória.
- Um ponteiro armazena um endereço de memória, independente do tipo

```
1  int main()
2  {
3      char *c;
4      int *i;
5      float *f;
6      double *d;
7      return 0;
8  }
```

Ponteiros e memória

Endereço	Valor
00010000	??
00010001	??
00010002	??
00010003	??
00010004	??
00010005	??
00010006	??
00010007	??
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??

- Exemplo:

```
1  int main()
2  {
3  →  int i;
4      i = 15;
5      char c = 's';
6      int *p = &i;
7      *p = 25;
8      return 0;
9  }
```

Ponteiros e memória

Endereço	Valor	
00010000	??	i
00010001		
00010002		
00010003		
00010004	??	
00010005	??	
00010006	??	
00010007	??	
00010008	??	
00010009	??	
0001000A	??	
0001000B	??	
0001000C	??	
0001000D	??	
0001000C	??	
0001000D	??	

- Exemplo:

```
1  int main()
2  {
3   → int i;
4      i = 15;
5      char c = 's';
6      int *p = &i;
7      *p = 25;
8      return 0;
9  }
```

- A memória para o inteiro i foi alocada

Ponteiros e memória

Endereço	Valor
00010000	15
00010001	
00010002	
00010003	
00010004	??
00010005	??
00010006	??
00010007	??
00010008	??
00010009	??
0001000A	??
0001000B	??
0001000C	??
0001000D	??
0001000C	??
0001000D	??

i

- Exemplo:

```
1  int main()
2  {
3      int i;
4   →  i = 15;
5      char c = 's';
6      int *p = &i;
7      *p = 25;
8      return 0;
9  }
```

- Atribuímos 15 ao conteúdo de memória da variável i

Ponteiros e memória

Endereço	Valor	
00010000	15	i
00010001		
00010002		
00010003		
00010004	s	c
00010005	??	
00010006	??	
00010007	??	
00010008	??	
00010009	??	
0001000A	??	
0001000B	??	
0001000C	??	
0001000D	??	
0001000C	??	
0001000D	??	

- Exemplo:

```
1  int main()
2  {
3      int i;
4      i = 15;
5   →  char c = 's';
6      int *p = &i;
7      *p = 25;
8      return 0;
9  }
```

- A memória para o char c foi alocada e inicializada com ‘s’

Ponteiros e memória

Endereço	Valor	
00010000	15	i
00010001		
00010002		
00010003		
00010004	s	c
00010005	00	
00010006	01	p
00010007	00	
00010008	00	
00010009	??	
0001000A	??	
0001000B	??	
0001000C	??	
0001000D	??	
0001000C	??	
0001000D	??	

- Exemplo:

```
1  int main()
2  {
3      int i;
4      i = 15;
5      char c = 's';
6   →  int *p = &i;
7      *p = 25;
8      return 0;
9  }
```

- O ponteiro p foi declarado e inicializado com endereço de memória de i

Ponteiros e memória

Endereço	Valor	
00010000	25	i
00010001		
00010002		
00010003		
00010004	s	c
00010005	00	
00010006	01	p
00010007	00	
00010008	00	
00010009	??	
0001000A	??	
0001000B	??	
0001000C	??	
0001000D	??	
0001000C	??	
0001000D	??	

- Exemplo:

```
1  int main()
2  {
3      int i;
4      i = 15;
5      char c = 's';
6      int *p = &i;
7   →  *p = 25;
8      return 0;
9  }
```

- O conteúdo de memória apontada por p é atualizado pra 25

Exemplo:

```c
int main()
{
    int x = 0;
    int *px;
    px = &x;
    *px = x - 5;
    printf("x = %d\n", x);
    return 0;
}
```

O que será impresso?

Exemplo:

```c
int main()
{
    int x = 0;
    int *px;
    px = &x;
    *px = x - 5;
    printf("x = %d\n", x);
    return 0;
}
```

O que será impresso?

```
x = -5
```

Exemplo:

```c
int main()
{
    int x = 0;
    int *px = &x;
    int *py;
    py = &(*px);
    *py = 10;
    printf("x = %d\n", x);
    return 0;
}
```

O que será impresso?

Exemplo:

```c
int main()
{
    int x = 0;
    int *px = &x;
    int *py;
    py = &(*px);
    *py = 10;
    printf("x = %d\n", x);
    return 0;
}
```

O que será impresso?

```
x = 10
```

Exemplo:

```c
int main()
{
    int x = 1000;
    int *px = &x;
    int y = *&*px; // ou *(&(*px))
    printf("y = %d\n", y);
    return 0;
}
```

O que será impresso?

Exemplo:

```c
int main()
{
    int x = 1000;
    int *px = &x;
    int y = *&*px; // ou *(&(*px))
    printf("y = %d\n", y);
    return 0;
}
```

O que será impresso?

```
y = 1000
```

Funções – Passagem de parâmetros por referência

Passagem de Parâmetros por Referência

- Na aula anterior, exploramos o conceito de **ponteiros** e como eles permitem acessar diretamente o endereço de memória de uma variável.

- Veremos como utilizar ponteiros para passar parâmetros por referência em funções, permitindo modificar diretamente o valor das variáveis originais.

Passagem de Parâmetros por Referência

Por que é importante?

- A passagem por referência é fundamental para otimizar memória, evitar cópias desnecessárias e modificar valores fora do escopo da função.

Sintaxe

- Passagem por **Valor**:

```
void funcaoValor(int a); // Recebe uma cópia de 'a'
```

- **chamada**: funcaoValor(x);
- **efeito**: Não modifica o valor original de x.

Exemplo:

Função de Troca de Valores (swap)

```
1  void naoTroca(int a, int b)
2  {
3      int aux = a;
4      a = b;
5      b = aux;
6  }
```

- Os parâmetros são passados por valor

Sintaxe

- Passagem por **Referência**:

```
void funcaoReferencia(int *a); // Recebe um ponteiro para ‘a’
```

- **chamada**: funcaoReferencia(&x); (passa o endereço de x)
- **efeito**: Pode modificar o valor original de x.

Exemplo:

Função de Troca de Valores (swap)

```
1  void troca(int *a, int *b)
2  {
3      int aux = *a;
4      *a = *b;
5      *b = aux;
6  }
```

- Os parâmetros são ponteiros para duas variáveis
- Logo, há troca de conteúdo de memória

Exemplo

```c
int main()
{
    int a = 1;
    int b = 2;
    naoTroca(a, b); // valores a e b são passados (e não há troca)
    printf("a = %d, b = %d", a, b); // a = 1, b = 2
}
```

```c
int main()
{
    int a = 1;
    int b = 2;
    troca(&a, &b); // endereço de memória de a e b são passados
    printf("a = %d, b = %d", a, b); // a = 2, b = 1
}
```

```
1  void naoTroca(int x, int y)
2  {
3      int aux;
4      aux = x;
5      x = y;
6      y = aux;
7  }
8
9  void troca(int *px, int *py)
10 {
11     int aux;
12     aux = *px;
13     *px = *py;
14     *py = aux;
15 }
16
17 int main()
18 {
19     int x = 100, y = 200;
20     naoTroca(x, y);
21     printf("x=%d, y=%d\n", x, y);
22     troca(&x, &y);
23     printf("x=%d, y=%d\n", x, y);
24
25     return 0;
26 }
```

Endereço	Conteúdo	Nome
0x1000		
0x1004		
0x1008		
0x1012		
0x1016		
0x1020		
0x1024		
0x1028		
0x1032		
0x1036		
0x1040		
0x1044		
0x1048		
0x1052		
0x1056		

```
1   void naoTroca(int x, int y)
2   {
3       int aux;
4       aux = x;
5       x = y;
6       y = aux;
7   }
8
9   void troca(int *px, int *py)
10  {
11      int aux;
12      aux = *px;
13      *px = *py;
14      *py = aux;
15  }
16
17  int main()
18  {
19      int x = 100, y = 200;
20      naoTroca(x, y);
21      printf("x=%d, y=%d\n", x, y);
22      troca(&x, &y);
23      printf("x=%d, y=%d\n", x, y);
24
25      return 0;
26  }
```

Endereço	Conteúdo	Nome	
0x1000			
0x1004	100	x	main
0x1008	200	y	main
0x1012			
0x1016	100	x	naoTroca
0x1020	200	y	naoTroca
0x1024			
0x1028			
0x1032			
0x1036			
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			

```
1   void naoTroca(int x, int y)
2   {
3       int aux;
4       aux = x;
5       x = y;
6       y = aux;
7   }
8
9   void troca(int *px, int *py)
10  {
11      int aux;
12      aux = *px;
13      *px = *py;
14      *py = aux;
15  }
16
17  int main()
18  {
19      int x = 100, y = 200;
20      naoTroca(x, y);
21      printf("x=%d, y=%d\n", x, y);
22      troca(&x, &y);
23      printf("x=%d, y=%d\n", x, y);
24
25      return 0;
26  }
```

Endereço	Conteúdo	Nome	
0x1000			
0x1004	100	x	main
0x1008	200	y	main
0x1012			
0x1016	200	x	naoTroca
0x1020	100	y	naoTroca
0x1024	100	aux	naoTroca
0x1028			
0x1032			
0x1036			
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			

```
1   void naoTroca(int x, int y)
2   {
3       int aux;
4       aux = x;
5       x = y;
6       y = aux;
7   }
8
9   void troca(int *px, int *py)
10  {
11      int aux;
12      aux = *px;
13      *px = *py;
14      *py = aux;
15  }
16
17  int main()
18  {
19      int x = 100, y = 200;
20      naoTroca(x, y);
21      printf("x=%d, y=%d\n", x, y);
22      troca(&x, &y);
23      printf("x=%d, y=%d\n", x, y);
24
25      return 0;
26  }
```

Endereço	Conteúdo	Nome	
0x1000			
0x1004	100	x	main
0x1008	200	y	
0x1012			
0x1016			
0x1020			
0x1024			
0x1028			
0x1032			
0x1036			
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			

```
1  void naoTroca(int x, int y)
2  {
3      int aux;
4      aux = x;
5      x = y;
6      y = aux;
7  }
8
9  void troca(int *px, int *py)
10 {
11     int aux;
12     aux = *px;
13     *px = *py;
14     *py = aux;
15 }
16
17 int main()
18 {
19     int x = 100, y = 200;
20     naoTroca(x, y);
21     printf("x=%d, y=%d\n", x, y);
22     troca(&x, &y);
23     printf("x=%d, y=%d\n", x, y);
24
25     return 0;
26 }
```

Endereço	Conteúdo	Nome	
0x1000			
0x1004	100	x	main
0x1008	200	y	main
0x1012			
0x1016			
0x1020			
0x1024			
0x1028			
0x1032	0x1004	px	troca
0x1036	0x1008	py	troca
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			

```
1  void naoTroca(int x, int y)
2  {
3      int aux;
4      aux = x;
5      x = y;
6      y = aux;
7  }
8
9  void troca(int *px, int *py)
10 {
11     int aux;
12     aux = *px;
13     *px = *py;
14     *py = aux;
15 }
16
17 int main()
18 {
19     int x = 100, y = 200;
20     naoTroca(x, y);
21     printf("x=%d, y=%d\n", x, y);
22     troca(&x, &y);
23     printf("x=%d, y=%d\n", x, y);
24
25     return 0;
26 }
```

Endereço	Conteúdo	Nome	
0x1000			
0x1004	200	x	main
0x1008	100	y	main
0x1012			
0x1016			
0x1020			
0x1024			
0x1028			
0x1032	0x1004	px	troca
0x1036	0x1008	py	troca
0x1040	100	aux	troca
0x1044			
0x1048			
0x1052			
0x1056			

Exemplo

1) Crie uma função que duplica o conteúdo da memória apontada por um ponteiro p

2) Faça uma única função que converte um valor em metros para: (i) jardas; (ii) pés; e (iii) polegadas. Use a função no método main().

1 metro é igual a aproximadamente 1094 jardas, 3281 pés, e 393701 polegadas.

Exemplo

```c
void dobrar(int *num) {
    *num = *num * 2;
}

int main() {
    int valor = 5;
    dobrar(&valor); // Passa o endereço de 'valor'
    printf("%d\n", valor); // Saída: 10
    return 0;
}
```

Observações

A biblioteca <string.h> fornece uma ampla gama de funções para manipulação de strings, incluindo:

- Manipulação e cópia: strcpy, strncpy para copiar strings de forma segura.
- Concatenação: strcat, strncat para juntar strings.
- Comparação: strcmp, strncmp para comparar conteúdos de strings.
- Busca: strchr, strrchr, strstr para localizar caracteres ou substrings.
- Tamanho: strlen para medir o comprimento de uma string.
- Tokenização: strtok para dividir uma string em partes menores (tokens).

Essas funções facilitam operações comuns com strings e ajudam a evitar a necessidade de escrever código do zero para tarefas básicas de manipulação de texto.