

Estrutura de Dados I

2025/2

Nicolas Ramos Carreira

Sumário

1	Intuito	3
2	Fundamentos em C	4
2.1	Sobre a linguagem C	4
2.2	Estrutura de um programa em C	4
2.3	Aspectos da linguagem C	5
2.3.1	Variaveis	5
2.3.2	Tipos de dados	6
2.3.3	Input e output	7
2.3.4	Contantes	10
2.3.5	Operadores	11
2.3.6	Coerção de tipos	13
2.3.7	Condicionais	13
2.3.8	Loops	15
2.3.9	Arrays	15
2.3.10	Struct - Criação de tipos	16
2.3.11	Comando typedef	16
3	Acerca de ponteiros	17
3.1	O porquê de estudar esse topico	17
3.2	O que são e como usá-los	17
3.2.1	O que é	17
3.2.2	Declaração de ponteiros	17
3.2.3	Detalhe após a declaração	18
3.2.4	Exemplo de uso	18
4	Falando de funções	19
4.1	O que é	19
4.2	Estrutura	19
4.2.1	Corpo	20
4.2.2	Parametros	20
4.2.3	Retorno	20
4.3	Passagem de parâmetros	20
4.3.1	Passagem por valor	20

4.3.2	Passagem por referência	20
4.3.3	Arrays como parametro	20
4.3.4	Struct como parametro	20
4.4	Recursão	20
5	Introdução aos algoritmos	21
5.1	Algoritmos de busca	21
5.1.1	Busca linear - Não ordenada	21
5.1.2	Busca linear - Ordenada	22
5.1.3	Busca Binária	23
5.1.4	Busca em array de struct	24
5.2	Algoritmos de ordenação	25
5.2.1	Bubblesort	25
5.2.2	Selection sort	30
5.2.3	Insertion sort	30
5.2.4	Shell sort	30
5.2.5	Merge sort	30
5.2.6	Quicksort	30
6	Alocação dinâmica	31

Capítulo 1

Intuito

O intuito deste documento é documentar o meu aprendizado da disciplina de estrutura de dados 1. Nesta disciplina começamos estudando sobre a linguagem C até entrar nas principais estruturas de dados.

Capítulo 2

Fundamentos em C

2.1 Sobre a linguagem C

A linguagem C é uma das linguagens de programação mais influentes e utilizadas da história da computação. Criada na década de 1970 por Dennis Ritchie nos laboratórios Bell, ela foi projetada para ser uma linguagem de propósito geral, eficiente e próxima do hardware, permitindo alto desempenho.

C é considerada uma linguagem de médio nível, pois combina características de linguagens de baixo nível (como manipulação direta de memória) com recursos de alto nível (como funções e estruturas). Sua sintaxe influenciou muitas outras linguagens modernas, como C++, Java, Csharp e até mesmo Python em alguns aspectos.

É amplamente usada em sistemas operacionais, softwares embarcados, drivers e aplicações que exigem alto desempenho. Além disso, aprender C é um ótimo ponto de partida para entender conceitos fundamentais de programação e arquitetura de computadores.

2.2 Estrutura de um programa em C

A code editor window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is written in a light green monospace font. It includes the standard input/output header, defines the main function, prints "Hello world!\n", and returns 0.

```
#include <stdio.h>

int main(){
    printf("Hello world!\n");
    return 0;
}
```

A imagem acima mostra um programinha extremamente simples em C, um Hello, world. Para iniciar um programa em C, nós sempre começamos declarando a biblioteca principal, que é a `stdio.h` (poderíamos ter outras bibliotecas inclusive, mas essa é a principal e DEVE estar lá).

Depois disso, nós declaramos o local do programa principal, onde fazemos o programa em si.

Um detalhe é que ao final de cada coisa SEMPRE temos que ter o ponto e vírgula (;), pois se não o nosso programa não compila.

2.3 Aspectos da linguagem C

2.3.1 Variáveis

O que são e pra que são usadas

Varível, em linguagens de programação, é basicamente uma posição alocada da memória para guardar uma informação. Variáveis podem ser modificadas pelo programa e devem ser definidas antes de ser utilizadas

Declaração de variáveis em C

Para definir variáveis em C, nós precisamos passar o tipo de dado e nome da variável, no formato:

<tipo de dado> nome-da-variavel;

Obs: ao fazer da forma acima, estamos apenas declarando a variável, sem atribuir um valor a ela

O tipo de dado deve ser aqueles que são aceitos pela linguagem (inteiro, decimais, caracteres, booleanos..), mas como falaremos sobre tipos de dados mais pra frente, não entraremos em detalhes agora. O nome da variável é algo bem importante a se considerar, pois existem algumas regras e boas práticas importantes quanto a isso:

- Nomes de variáveis devem iniciar com letras ou underscore
- Os caracteres da variável devem ser letras, números ou underscore (não utilizar acentos ou símbolos)
- Não utilizar espaço em nomes de variáveis
- Palavras chaves (palavras que são reservadas pela linguagem para fazer determinadas coisas) não podem ser usadas como nomes
- Letras maiúsculas e minúsculas são consideradas diferentes

Só para deixar totalmente claro, as palavras chaves que a linguagem C usa são:

auto	break	case	char	const	continue	do	double
else	for	int	union	static	default	void	return
enum	goto	long	unsigned	struct	extern	while	sizeof
float	if	short	volatile	switch	register	typeof	

Atribuição de valores em variáveis

Tendo o formato <tipo de dado> nome-da-variavel, podemos atribuir valores a elas (ou seja, armazenar valores dentro da memória). Para isso, basta fazer:

<tipo de dados> nome-variavel = valor;

2.3.2 Tipos de dados

Como falamos anteriormente na parte de variáveis, quando vamos defini-las, nós temos que declarar o tipo de dado da variável. O tipo de dado define os valores que aquela variável pode assumir e as operações que podem ser realizadas com ela. Os tipos de dados principais são: char, int, float e double

Char

Um byte que armazena

Int

Um inteiro cujo o tamanho do número que pode ser alcançado depende do processador (tipicamente 16 ou 32 bits)

Float

Basicamente números decimais com precisão simples (em C a parte decimal usa ponto e não vírgula)

Double

Também números decimais, mas com precisão dupla. É usados para números muito pequenos (científicos por exemplos) ou muito grandes

Bool

Esse tipo de dados é muito interessante, pois ele pode assumir dois valores: verdadeiro ou falso (true ou false). Em outras linguagens, nós temos literalmente um valor True e False. No entanto, na linguagem C nós não temos True e False, mas podemos representá-los como 1 e 0, respectivamente.

Outros tipos

Na imagem abaixo, você poderá ver alguns outros que são utilizados:

Tipo	Bits	Intervalo de valores
char	8	-128 A 127
unsigned char	8	0 A 255
signed char	8	-128 A 127
int	32	-2.147.483.648 A 2.147.483.647
unsigned int	32	0 A 4.294.967.295
signed int	32	-32.768 A 32.767
short int	16	-32.768 A 32.767
unsigned short int	16	0 A 65.535
signed short int	16	-32.768 A 32.767
long int	32	-2.147.483.648 A 2.147.483.647
unsigned long int	32	0 A 4.294.967.295
signed long int	32	-2.147.483.648 A 2.147.483.647
float	32	1,175494E-038 A 3,402823E+038
double	64	2,225074E-308 A 1,797693E+308
long double	96	3,4E-4932 A 3,4E+4932

2.3.3 Input e output

Input e output é basicamente a entrada e a saída de dados. As vezes, podemos querer receber do usuário alguns valores, para fazer alguma coisa com eles e depois entregá-los com modificações. É basicamente isso. Um detalhe é que para o output, não necessariamente nós precisamos ter recebido algo.

Especificadores de formato

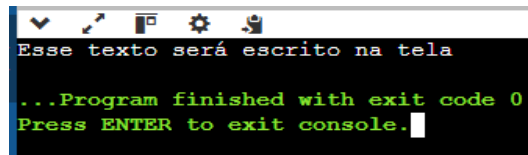
Saída com printf()

Vamos começar com a saída de dados. Para exibir algo na tela. Fazemos:

```
#include <stdio.h>

int main(){
    printf("Esse texto será escrito na tela");
    return 0;
}
```

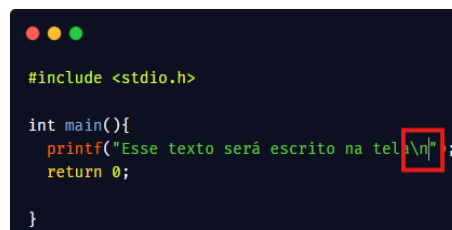
Ao fazer isso, em nosso terminal será exibido o texto que digitamos dentro do printf ("Esse texto será escrito na tela). Veja:



```
Esse texto será escrito na tela
...Program finished with exit code 0
Press ENTER to exit console.
```

Uso do escape no printf()

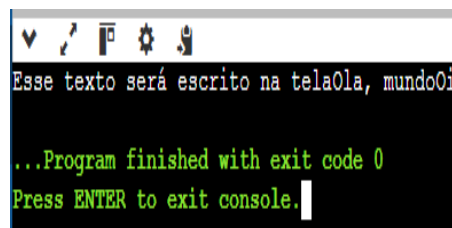
Um detalhe é que algo que podemos utiliza no printf é caracter de escape . Esse caracter é utilizado sempre ao final do que que queremos escrever na saída e ele serve para quebrar a linha após a saída. Veja:



```
#include <stdio.h>

int main(){
    printf("Esse texto será escrito na tela\n");
    return 0;
}
```

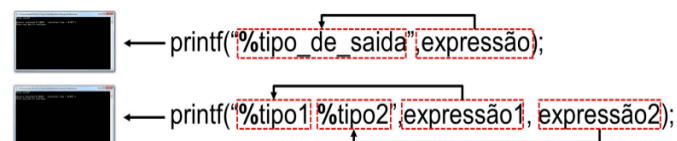
Se fizermos vários printf, por exemplo, e não usarmos o caracter de escape em nenhum deles, o que escrevemos nos prints, ficará tudo junto. Veja:



```
Esse texto será escrito na telaOla, mundoOi
...Program finished with exit code 0
Press ENTER to exit console.
```

Exibindo valores de variaveis no output

Se quisermos que em nosso output seja usada alguma variavel, temos que utilizar o seguinte formato:



Isso acima significa que se quisermos passar no output uma variavel que tenha o tipo int, nós teríamos que passar o tipo de saída dentro das aspas duplas e depois separar por virgula passando a nossa variavel. Mas você deve

estar se perguntando: Como assim tipos de saída? Veja abaixo os tipos de saída que usaremos no output (printf):

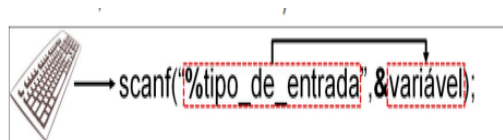
Alguns tipos de saída	
%c	escrita de um caractere (char)
%d ou %i	escrita de números inteiros (int ou char)
%u	escrita de números inteiros sem sinal (unsigned)
%f	escrita de número reais (float ou double)
%s	escrita de vários caracteres
%p	escrita de um endereço de memória
%e ou %E	escrita em notação científica

Ou seja, seguindo o exemplo da variável de tipo int que tínhamos dado, se quiséssemos exibi-la no output (printf), faríamos:

```
printf("porcentagemd", variavel);
```

Entrada com scanf()

Agora, falando sobre entrada de dados, o comando que utilizamos para passar dados para o nosso programa é o scanf(). Esse comando permite realizar a leitura de dados da entrada padrão (teclado). Sua estrutura é a seguinte:



Sendo que, os tipos de entrada são praticamente os mesmos que vimos nos tipos de saída. Veja:

Alguns tipos de saída	
%c	leitura de um caractere (char)
%d ou %i	leitura de números inteiros (int ou char)
%f	leitura de número reais (float ou double)
%s	leitura de vários caracteres

Podemos ainda realizar a leitura de mais de um valor (assim como podemos fazer o output de mais de um valor). É bem parecido com o output também. Veja:



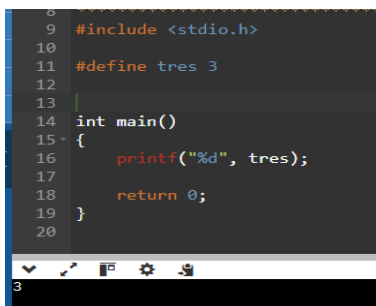
2.3.4 Contantes

Assim como variaveis, constantes também armazenam um valor na memória do computador. A principal diferença para as variaveis é que esse valor não será alterado. Outra coisa é que para as constantes é obrigatoria a atribuição de valor, diferente das variaveis que podemos simplesmente declará-las sem dar um valor

Declaração de constantes

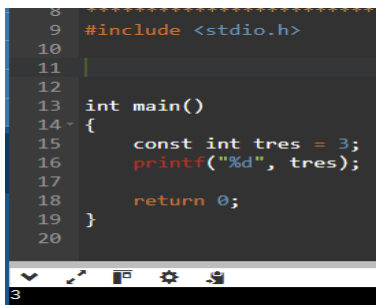
Para declarar uma constante existem duas formas. Na primeira, devemos utilizar `define nome-costante <valor>` no começo do programa. Uma detalhe é que neste caso, não usaremos ponto e virgula no final. Veja:

```
8
9  #include <stdio.h>
10
11  #define tres 3
12
13
14  int main()
15  {
16      printf("%d", tres);
17
18      return 0;
19  }
20
```



Outra forma é fazer: `const <tipo> nome = valor;`. Como você pôde ver, nesse caso, temos que usar o ponto e vírgula. Veja:

```
8
9  #include <stdio.h>
10
11
12
13  int main()
14  {
15      const int tres = 3;
16      printf("%d", tres);
17
18      return 0;
19  }
20
```



Curiosidade sobre constantes

Já chegamos a falar sobre caracteres de escape (no caso, falamos apenas do barran). Os caracteres de escape são constantes pre-definidas. Veja cada um deles:

Código	Comando
\a	som de alerta (bip)
\b	retrocesso (backspace)
\n	nova linha (new line)
\r	retorno de carro (carriage return)
\v	tabulação vertical
\t	tabulação horizontal
\'	apóstrofe
\"	aspa
\\	barra invertida (backslash)
\f	alimentação de folha (form feed)
\?	símbolo de interrogação
\0	caractere nulo (cancela a escrita do restante)

2.3.5 Operadores

Os operadores são usados para desenvolver diferentes tipos de operações. Com eles podemos fazer operações matemáticas, comparativas, lógicas e etc. Veremos acerca de cada um dos operadores a seguir

Operadores aritméticos

Os operadores aritméticos são aqueles que operam sobre números e/ou sobre expressões que tem como resultado valores numéricos. Veja os operadores:

Operador	Significado	Exemplo
+	Adição de dois valores	$z = x + y$
-	Subtração de dois valores	$z = x - y$
*	Multiplicação de dois valores	$z = x * y$
/	Quociente de dois valores	$z = x / y$
%	Resto de uma divisão	$z = x \% y$

Um detalhe é que as operações seguem a mesma ordem da matemática. A prioridade são as operações de multiplicação e divisão em detrimento das de soma e subtração.

Outro detalhe é que na divisão, se o numerador e denominador forem inteiros, o compilador retornará apenas a parte inteira da divisão

Operadores relacionais

São aqueles que verificam a magnitude (maior/menor) e/ou igualdades entre dois valores e/ou expressões. Esses operadores retornam verdadeiro (1) ou falso (0) (ou seja, um valor booleano). Veja cada um deles:

Operador	Significado	Exemplo
>	Maior do que	X > 5
>=	Maior ou igual a	X >= Y
<	Menor do que	X < 5
<=	Menor ou igual a	X <= Z
=	Igual a	X == 0
!=	Diferente de	X != Y

Operadores lógicos

Os operadores lógicos nos permitem representar situações lógicas unindo duas mais expressões relacionais simples em uma composta e nos retornam verdadeiro (1) ou falso (0). Veja cada um deles:

Operador	Significado	Exemplo
&&	Operador E	(x > 0) && (x < 10)
	Operador OU	(a == 'F') (b != 32)
!	Operador NEGAÇÃO	!(x == 10)

Operadores de atribuição simplificada

Muitas vezes em nosso código nós temos que atribuir valores a nossa variável. Uma forma de fazer isso de maneira mais fácil é utilizando os operadores de atribuição simplificada. Com eles, podemos adicionar valores a nossa variável de forma muito mais simples. Veja cada um deles:

Operador	Significado	Exemplo	
+=	Soma e atribui	x += y	igual a x = x + y
-=	Subtrai e atribui	x -= y	igual a x = x - y
*=	Multiplica e atribui	x *= y	igual a x = x * y
/=	Divide e atribui o quociente	x /= y	igual a x = x / y
%=	Divide e atribui o resto	x %= y	igual a x = x % y

Operadores de pré e pós incremento

Esses operadores podem ser utilizados sempre que for necessário somar uma unidade (incremento) ou subtrair uma unidade (decremento) a determinado valor. Veja cada um deles:

Operador	Significado	Exemplo	Resultado
++	incremento	++x ou x++	x = x + 1
--	decremento	--x ou x--	x = x - 1

Um detalhe é que como você pode ver na imagem acima, podemos utilizar o operador antes de depois da variável, mas qual a diferença? Veja abaixo:

Operador	Significado	Resultado
<code>++x</code>	pré-incremento	soma +1 à variável x antes de utilizar seu valor
<code>x++</code>	pós-incremento	soma +1 à variável x depois de utilizar seu valor
<code>--x</code>	pré-decremento	subtrai -1 da variável x antes de utilizar seu valor
<code>x--</code>	pós-decremento	subtrai -1 da variável x depois de utilizar seu valor

2.3.6 Coerção de tipos

Lembra quando falamos anteriormente que se dividirmos um numero inteiro por outro inteiro seu resultado sempre será inteiro, desconsiderando assim a parte decimal? Podemos contornar isso utilizando o casting. O casting é aplicado sobre uma expressão aritmética e força o resultado da expressão a ser de um tipo especificado. Veja as diferentes formas de utilizar o casting:

Type casting explícito

Nós faremos a conversão de tipo no resultado da expressão:

```
int a = 5, b = 2;
float resultado = (float)a / b;
printf("%f\n", resultado); // saída: 2.500000
```

Type casting nos operandos

Faremos o casting nos dois operandos da operação para obter o resultado no tipo que queremos

```
double resultado = (double)a / (double)b;
printf("%lf\n", resultado); // saída: 2.500000
```

2.3.7 Condicionais

Certo. Agora falaremos sobre condicionais. Condicional é basicamente uma mudança de fluxo em nosso código. Caso uma determinada expressão atenda determinada condição, nosso código seguirá por um fluxo e caso contrário, seguirá para outro fluxo. Existem diferentes maneiras de se utilizar as condicionais em nosso código. Veremos cada uma delas abaixo.

If-else

A estrutura do if-else é feita da seguinte forma em nosso código:

```

if (condicao){
    sequencia de comandos 1;
}
else{
    sequencia de comandos 2;
}

```

O que acontece acima é que se a condição for satisfeita, ou seja, for verdadeira (tiver valor 1), nosso programa entrará nesse fluxo e executará o código dentro da condição. Caso contrário, ou seja, caso a condição não for satisfeita (for falsa (ter valor 0)), entraremos no fluxo do else.

Um detalhe é que além do if e do else, podemos ter ainda o else if, onde caso a condição do if não for satisfeita, haverá a condição do else if a ser satisfeita e aí se ela não for satisfeita também, iremos para o else. Veja:

```

if (condicao){
    sequencia de comandos 1;
}
else if (condicao){
    sequencia de comandos 2;
}
else{
    sequencia de comandos 3;
}

```

Switch-case

O switch-case é outra estrutura de controle de fluxo de código. Sua estrutura é a seguinte:

```

switch(expressao){
    case valor 1:
        sequencia de comandos 1;
        break;

    case valor k:
        sequencia de comandos k;
        break;

    ...
    default:
        sequencia de comandos padrao;
        break;
}

```

O switch, como podemos ver acima é próprio para testar uma variável em relação a diversos valores pré-estabelecidos. Além disso, como podemos ver acima o default irá desempenhar o valor que o else desempenha na estrutura if-else

2.3.8 Loops

Agora falando sobre loops, o nome já entrega. Os loops serão responsáveis por repetir um bloco de código a partir de uma condição. Enquanto a condição for verdadeira, o bloco de código permanecerá se repetindo. Existem diferentes tipos de loops. Vamos a cada um deles.

While

Do-While

For

2.3.9 Arrays

Quando vimos sobre variáveis, estudamos que elas podem armazenar um valor. Sempre que tentamos armazenar um novo valor dentro da variável o antigo valor é sobrescrito (e portanto, perdido). Agora, pense: E se quiséssemos armazenar mais de um valor em uma variável? Para isso, usamos os arrays, que é basicamente uma sequência de elementos do mesmo tipo, onde cada elemento é identificado por um índice. Ou seja, quando criamos um array, nós alocamos um espaço na memória (onde, quanto maior o tamanho do array, que é a quantidade de elementos que ele pode armazenar, maior o espaço de memória alocado) e podemos armazenar dentro dele vários valores do mesmo tipo (os valores podem ser acessados por meio do índice do elemento dentro do array, que é basicamente a posição do elemento lá dentro). Um exemplo que pode fazer você entender melhor é: suponhamos que queiramos armazenar em um local a nota de 5 alunos. Para isso, poderíamos usar um array.

Declaração de arrays

Para declarar um array, nós fazemos da seguinte forma:

$$\langle \text{tipo-array} \rangle \text{ nome-array}[\text{tamanho}];$$

Ou seja, primeiro nós precisamos declarar o tipo do array, que será o tipo dos valores que aquele array irá armazenar, depois passamos o nome do array e depois passamos o tamanho do array, ou seja, a quantidade de elementos que ele poderá armazenar.

Inserindo e acessando valores dentro de arrays

Com o array declarado, caso quisermos inserir algum valor no array, basta fazer:

$$\text{nome-array}[\text{índice}] = \text{valor};$$

Lembrando que o índice é a posição do elemento dentro do array. Se tivéssemos um array de tamanho 10 e quiséssemos inserir um valor no sexto elemento, faríamos: `nome-array[5] = valor;` (uma vez que os índices começariam do 0 e iriam até o 9).

Para acessar valores de um array, basta fazer:

`nome-array[indice];`

Observações

Em C e C++, se tivermos um array que pode armazenar 10 elementos e tentarmos armazenar 11 elementos, o elemento que sobrar irá ser armazenado em um espaço da memória que não pertence ao array, o que causa comportamento indefinido (pode sobrescrever dados, travar o programa e entre outros)

Outro detalhe é que se tivermos um array de 10 elementos (ou seja, teremos 10 índices, do 0 ao 9) e tentarmos acessar o índice 10 (11º elemento) o que acontecerá (em C e C++) é que iremos acessar um elemento da memória que não pertence oficialmente ao array, o que pode retornar "Lixos de memória".

2.3.10 Struct - Criação de tipos

Agora, falaremos sobre structs, estamos falando de uma composição de variáveis de outros tipos que formam um "novo tipo de dado". Assim como temos o tipo inteiro, float e etc, podemos "criar" um outro tipo que será um agrupamento de dados.

Declaração

Inserindo valores e acessando valores

2.3.11 Comando typedef

O comando `typedef` nos permite "dar um alias" para os tipos de dados existentes na linguagem C. Se temos, por exemplo, o tipo `float`, mas queremos que ele se chame flutuante, poderíamos fazer isso.

Como usar

Para usar, basta fazer o seguinte:

`typedef <tipo-de-dado> <alias>;`

O comando acima "dá um alias" a um tipo de dado existente. Um detalhe é que o comando acima deve estar no topo do programa, juntamente com a inclusão das bibliotecas.

Exemplo

Capítulo 3

Acerca de ponteiros

3.1 O porquê de estudar esse topico

Agora, iniciaremos um tópico mais avançado, que são os ponteiros. É muito importante entendermos sobre esse conceito porque várias das estruturas de dados que aprenderemos nesta disciplina dependem deles (listas, pilhas, filas, árvores e grafos), então sem entender isso, não iremos para frente

3.2 O que são e como usá-los

3.2.1 O que é

Conceitualmente, um ponteiro é uma variavel que armazena o endereço de memoria de outra variavel. Ou seja, diferentemente das variaveis comuns, um ponteiro não irá armazenar um valor como um caracter, por exemplo, mas sim, um endereço de memória.

3.2.2 Declaração de ponteiros

Para criar um ponteiro, a estrutura lembra bastante a forma como nós criamos as variaveis, mas com pequenas mudanças. Veja como declaramos um ponteiro:

`<tipo de dado> *nome ponteiro;`

Perceba que para declarar um ponteiro, assim como nas variaveis, nós temos que usar um tipo de dado. Isso acontece porque nós estamos indicando para o ponteiro que estamos criando o tipo de dado do lugar da memória que ele vai apontar. Isso é importante porque não é muito aconselhavel você ter um ponteiro inteiro e apontar para um char, por exemplo.

Um detalhe é que podemos criar o nosso ponteiro apontando ele para NULL, para que ele não aponte para nenhum lugar (para que consigamos administrar para onde ele aponta depois). Fazendo isso, a declaração ficaria:

<tipo de dado> *nome ponteiro = NULL;

Se não declararmos da maneira acima e utilizarmos a primeira versão de declaração (<tipo de dado> *nome ponteiro;), o que acontece é que nosso ponteiro irá apontar para um endereço de memória aleatório.

Um outro detalhe bem interessante é que podemos fazer com que nosso ponteiro aponte para o endereço de memória de uma variável já existente. Veja:

<tipo de dado> *nome ponteiro = a;

Ou seja, o endereço de memória que nosso ponteiro irá apontar, será o endereço da variável a (esse significa que estamos nos referindo ao endereço de memória da variável a. Desta forma, o ponteiro b apontará para o endereço de memória de a).

3.2.3 Detalhe após a declaração

Quando declaramos um ponteiro(exemplo: int *a), algo importante de se dizer é que se utilizarmos *a em qualquer outro trecho do nosso código, nós não vamos estar usando o ponteiro em si, mas sim o valor que está no endereço apontado pelo ponteiro a. Veja um exemplo:

3.2.4 Exemplo de uso

Veja abaixo um exemplo de uso de ponteiros:

```
main.c
1  #include <stdio.h>
2
3  int main()
4  {
5      int a = 5;
6      int *b = &a;
7
8      printf("a = %d | b = %d\n", a, *b);
9
10     return 0;
11 }
```

Acima, o que acontece é que:

- int a = 15; -> Você está colocando o valor 15 dentro da variável a.

Capítulo 4

Falando de funções

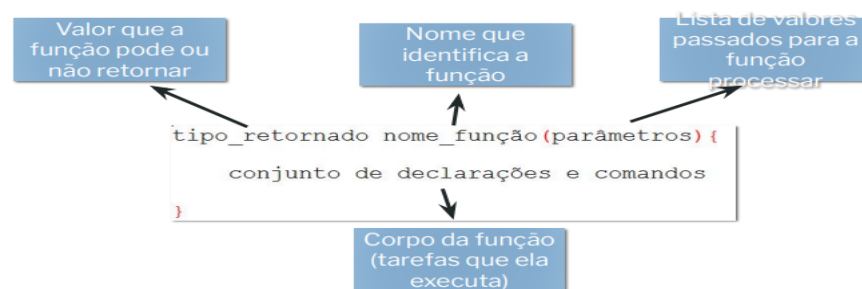
4.1 O que é

Funções são blocos de código que podem ser nomeados e chamados dentro de um programa. Elas facilitam a estruturação e reutilização do código, pois:

- Estruturação: programas grandes e complexos são construídos bloco a bloco.
- Reutilização: o uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros

4.2 Estrutura

A forma geral de uma função é:



As funções devem ser declaradas ANTES de serem utilizadas, ou seja, antes da cláusula `main`

4.2.1 Corpo

O corpo é a alma da função e é composto pelos comandos que a função deve executar. Ele processa os parametros (se houver), realiza tarefas e gera saídas (se necessário)

Um detalhe é que nós evitamos operações de leitura e escrita dentro de uma função. Essas operações devem ser feitas em quem chamou a função (o `main()`), por exemplo.

4.2.2 Parametros

A declaração de parametros

4.2.3 Retorno

4.3 Passagem de parâmetros

4.3.1 Passagem por valor

4.3.2 Passagem por referência

4.3.3 Arrays como parametro

4.3.4 Struct como parametro

4.4 Recursão

Capítulo 5

Introdução aos algoritmos

5.1 Algoritmos de busca

Quando temos um conjunto de dados, podemos querer procurar por um elemento. Exemplo: Temos um array de números inteiros, pode ser que queiramos buscar algum valor dentro desse array.

Existem varios tipos de busca e a utilização dos tipos dependerá de como são os dados (se eles estão estruturados, ordenados e se existem valores duplicados). Com isso em mente, veremos cada um desses tipos de busca

5.1.1 Busca linear - Não ordenada

Como funciona?

Esse é o algoritmo de busca mais simples que existe. O que ele faz é percorrer o array que contém os dados desde sua primeira posição até a última comparando cada valor dele com o valor buscado. Se os valores forem iguais, a busca termina e caso contrário, continua até o fim do array.

	0	1	2	3	4	5	6	
v	23	4	67	-8	54	90	21	
elem	54	Elemento procurado						
i=0	23	4	67	-8	54	90	21	Valor diferente: continua a busca
i=1	23	4	67	-8	54	90	21	Valor diferente: continua a busca
i=2	23	4	67	-8	54	90	21	Valor diferente: continua a busca
i=3	23	4	67	-8	54	90	21	Valor diferente: continua a busca
i=4	23	4	67	-8	54	90	21	Valor igual: termina a busca

Apesar de ser intuitivo, o motivo pelo qual ele tem que percorrer o array inteiro é o fato dele não estar ordenado. No nosso exemplo, suponhamos que o array estivesse ordenado, mas que o número 54 que está sendo buscado não

existisse. Ao fazer nossa busca, quando chegarmos no número 67, ele iria parar a busca, pois o valor procurado não poderia estar depois de 67 (o array está ordenado).

Complexidade

A complexidade do algoritmo de busca linear não ordenada pode ser analisada conforme os seguintes casos:

- Melhor caso: $O(1)$. Acontece quando o elemento buscado é o primeiro do array.
- Pior caso: $O(N)$. Acontece quando o elemento é o último do array ou não existe.
- Caso médio: $O(N/2)$.

Implementação

```

13
14 int buscaLinear(int *V, int N, int elem){
15     int i;
16     for(i = 0; i<N; i++){
17         if(elem == V[i])
18             return i;//elemento encontrado
19     }
20     return -1;//elemento não encontrado
21 }

```

5.1.2 Busca linear - Ordenada

Como funciona?

A busca sequencial ordenada funcionará da mesma forma que a busca sequencial não ordenada. A diferença é que, com o array ordenado, caso o valor do array seja maior que o valor buscado, ele parará a busca.

	0	1	2	3	4	5	6	
V	-8	4	21	23	54	67	90	

elem	34	Elemento procurado	
------	----	--------------------	--

	0	1	2	3	4	5	6	
i=0	-8	4	21	23	54	67	90	Valor diferente: continua a busca
i=1	-8	4	21	23	54	67	90	Valor diferente: continua a busca
i=2	-8	4	21	23	54	67	90	Valor diferente: continua a busca
i=3	-8	4	21	23	54	67	90	Valor diferente: continua a busca
i=4	-8	4	21	23	54	67	90	Valor é maior: elemento não existe

Complexidade

A complexidade do algoritmo de busca linear ordenada pode ser analisada conforme os seguintes casos:

- Pior caso: $O(n)$. Acontece quando o valor é o maior valor do array, ou seja, está na última posição do array

Implementação

```
14
15 int buscaOrdenada(int *V, int N, int elem){
16     int i;
17     for(i = 0; i < N; i++){
18         if(elem == V[i])
19             return i; //elemento encontrado
20         else
21             if(elem < V[i])
22                 return -1; //para a busca
23     }
24     return -1; //elemento não encontrado
25 }
```

5.1.3 Busca Binária

Como funciona?

Esse algoritmo é uma das formas mais "especializadas" de se realizar uma busca em um array. Para utilizá-lo o array DEVE estar ordenado. O que ele faz é calcular o meio do array e utilizar o valor desse meio para comparar com o valor buscado. Se o valor buscado for menor que o valor do meio, ele descarta a segunda metade do array e fica com apenas a primeira metade com os valores menores. Caso o valor buscado for maior, ele fará o contrário

	0	1	2	3	4	5	6	7	8	9	
V	-8	-5	1	4	14	21	23	54	67	90	
elem	4	Elemento procurado									
meio=4	-8	-5	1	4	14	21	23	54	67	90	Valor é menor: buscar no início
meio=1	-8	-5	1	4	14	21	23	54	67	90	Valor é maior: buscar no final
meio=2	-8	-5	1	4	14	21	23	54	67	90	Valor é maior: buscar no final
meio=3	-8	-5	1	4	14	21	23	54	67	90	Valor é igual: terminar a busca

A parte em azul na imagem acima representa a parte do array que foi descartada por conta da condição

Um detalhe é que ao realizar a comparação entre o valor buscado e o valor do meio do array, ele vai verificar se o valor buscado é igual ao valor do meio do array e, caso for, ele já encerra a busca ali mesmo.

Complexidade

A complexidade do algoritmo de busca linear ordenada pode ser analisada conforme os seguintes casos:

- Melhor caso: $O(1)$. O elemento está exatamente no meio do array
- Caso médio: $O(\log_2 N)$.
- Pior caso: $O(\log_2 N)$. O elemento não existe

Implementação

Abaixo, a sua implementação no código:

```
27 |
28 | int buscaBinaria(int *V, int N, int elem){
29 |     int i, inicio, meio, final;
30 |     inicio = 0;
31 |     final = N-1;
32 |     while(inicio <= final){
33 |         meio = (inicio + final)/2;
34 |         if(elem < V[meio])
35 |             final = meio-1; //busca na metade da esquerda
36 |         else
37 |             if(elem > V[meio])
38 |                 inicio = meio+1; //busca na metade da direita
39 |             else
40 |                 return meio;
41 |     }
42 |     return -1; //elemento não encontrado
43 | }
```

5.1.4 Busca em array de struct

Como funciona?

Aqui, estamos lidando com algo mais complexo. Quando queremos buscar algo em um array de struct, nós usaremos uma das chaves da struct para realizar a busca, como usar a matricula para buscar de uma struct para realizar uma busca

```
struct aluno V[6];
```

matricula;	matricula;	matricula;	matricula;	matricula;	matricula;
nome[30];	nome[30];	nome[30];	nome[30];	nome[30];	nome[30];
n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;	n1,n2,n3;
V[0]	V[1]	V[2]	V[3]	V[4]	V[5]

Complexidade

Como estamos falando de uma busca linear, sua complexidade será a mesma

Implementação

```
28 int buscaLinearMatricula(struct aluno *V, int N, int elem){
29     int i;
30     for(i = 0; i < N; i++){
31         if(elem == V[i].matricula)
32             return i; //elemento encontrado
33     }
34     return -1; //elemento não encontrado
35 }
```

5.2 Algoritmos de ordenação

Após uma base de dados estar construída pode ser necessário ordená-la. A ordenação dos dados PODE ser um passo preliminar para pesquisá-los (para utilizar o algoritmo de busca binária, por exemplo, precisamos que os dados estejam ordenados). Dada essa introdução, veremos alguns algoritmos de ordenação de dados.

5.2.1 Bubblesort

Como funciona?

O algoritmo de ordenação bubblesort é o mais simples dos algoritmos de ordenação. O que ele faz é:

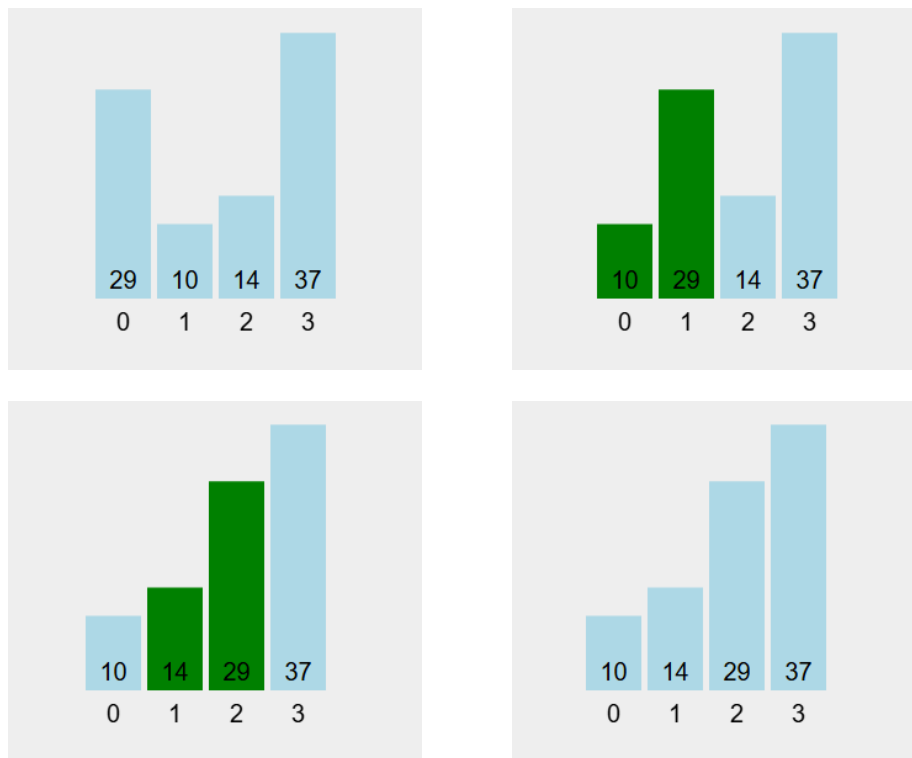
1. Comparação de dois números
2. Se o da esquerda for maior, os elementos devem ser trocados
3. Desloca-se uma posição à direita

Exemplo: Estamos percorrendo um vetor e na posição da esquerda nós temos o número 10 e na posição da direita nós temos o número 8. Como 8 é menor que 10, iremos fazer a troca. No lugar do 10, teremos o 8 e no lugar do 8 teremos o 10. Feito isso, ele estará na posição onde está o valor 8, então ele irá se deslocar uma posição à direita (que é onde estará o valor 10).

A medida que o algoritmo avança, os itens maiores "surgem como uma bolha" na extremidade superior do vetor (à direita do vetor). É por isso que o é o algoritmo da bolha (bolha = bubble)

Podemos observar esse algoritmo de forma visual a partir deste link. Aqui vai um exemplo rápido com algumas imagens:

Na primeira imagem, podemos ver o nosso vetor de forma desordenada, aí o que acontecerá com a aplicação do algoritmo de bubble sort é que pegaremos o número da esquerda (no caso 29) e iremos comparar com o segundo. Se o segundo número for menor, jogamos o maior para a direita (assim como podemos ver na segunda imagem, onde o número 29 foi para a direita e o 10, que era menor, foi para a esquerda). Após isso, faremos a comparação novamente entre o

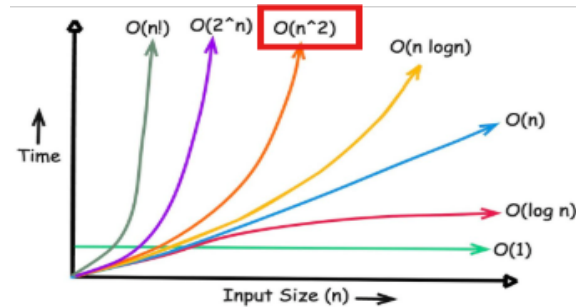


número da esquerda e o da direita. Agora, estamos comparando o 29 (esquerda) e o 14 (direita), então como o número da esquerda é maior (29), iremos jogá-lo para a direita e o número que estava na direita virá para a esquerda, assim como podemos ver na terceira imagem. Por fim, vamos fazer a comparação novamente entre o número da esquerda e o da direita. Dessa vez o número da esquerda é menor que o da direita, então não haverá troca, aí passaremos para o próximo número (37) para realizar novas comparações, mas como o vetor acabou, então finalizamos por aqui.

Portanto, o algoritmo vai percorrendo o vetor e fazendo as trocas, mas pode ser que ele tenha que percorrer o vetor mais de uma vez para fazer a ordenação (na maioria das vezes é o que acontece, apesar de termos dado sorte no exemplo que demos acima). Apesar disso, uma coisa que ele garante é que após a primeira rodada o maior elemento do array será movido para a última posição do array e isso faz com que, para um vetor com n elementos, o Bubble Sort precise de no máximo $n-1$ passagens. Isso acontece porque a cada passagem, o maior elemento restante é movido para sua posição final correta. Após a $n-1^{\text{a}}$ passagem, os $n-1$ maiores elementos já estarão ordenados. Por consequência, o único elemento que sobrou, o menor de todos, já estará automaticamente na primeira posição, que é a sua posição correta. Sendo assim, não há necessidade de uma n -ésima passagem.

Complexidade

Com relação ao Big-O desse algoritmo, ele é um $O(n^2)$, ou seja, o tempo de execução dele é relativamente grande:



A razão que faz esse algoritmo ser um $O(n^2)$ é o fato de ter dois loops aninhados que o algoritmo usa para percorrer o vetor (veremos na implementação)

Implementação

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

```
//Funcao para trocar dois elementos de lugar
```

```
void trocar(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
//Funcao que implementa o Bubble Sort
```

```
void bubbleSort(int vetor[], int n) {  
    int i, j;  
    int houveTroca;
```

```
    //O algoritmo precisa repetir varias vezes
```

```
    //ate que nao haja mais trocas
```

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

```
        houveTroca = 0; //no comeco da rodada, nao houve troca ainda
```

```
        //Percorre o vetor ate a penultima posicao comparando os vizinhos
```

```
        for (j = 0; j < n - i - 1; j++) {
```

```
            if (vetor[j] > vetor[j + 1]) {
```

```
                trocar(&vetor[j], &vetor[j + 1]);
```

```
                houveTroca = 1; // se houve troca, marcamos
```

```
            }
```

```
        }
```

```

        //Se nao houve troca, significa que o vetor ja esta ordenado
        if (houveTroca == 0) {
            break;
        }
    }
}

//Funcao para imprimir o vetor
void imprimirVetor(int vetor[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}

int main() {
    int vetor[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(vetor) / sizeof(vetor[0]);

    printf("Vetor original: ");
    imprimirVetor(vetor, n);

    bubbleSort(vetor, n);

    printf("Vetor ordenado: ");
    imprimirVetor(vetor, n);

    return 0;
}

```

Explicação da implementação:

1. void trocar(int *a, int*b)

Essa é uma função auxiliar que serve para trocar os valores de duas variáveis inteiras.

- int *a e int *b: Os parâmetros são ponteiros para inteiros, e não as variáveis em si. Como sabemos, um ponteiro é uma variável que armazena o endereço de memória de outra variável. Usamos ponteiros aqui porque, para a função trocar realmente modificar os valores das variáveis que a chamaram, ela precisa de seus endereços de memória. Se passássemos apenas int a e int b, a função criaria cópias locais dos valores, e a troca não afetaria o vetor original.
- int temp = *a;; A variável temp (de "temporário") é usada para guardar o valor original do primeiro elemento. O asterisco (*) é

o operador de desreferenciação; ele "desempacota" o ponteiro para acessar o valor que está no endereço de memória. Então, `*a` é o valor da variável que o ponteiro `a` está apontando.

- `*a = *b;`: O valor da segunda variável (`*b`) é atribuído à primeira (`*a`)
- `*b = temp;`: O valor original da primeira variável (`*a`), que estava guardado em `temp`, é atribuído à segunda (`*b`).

2. `void bubbleSort(int vetor[], int n)`

Esta é a função principal que implementa o algoritmo de ordenação.

- `for (i = 0; i < n - 1; i++)`: Este é o laço externo. Ele controla o número de "passagens" que o algoritmo fará pelo vetor. Em cada passagem, o maior elemento "flutua" para a sua posição correta no final do vetor. O `n - 1` é usado porque, se temos `n` elementos, precisamos de no máximo `n - 1` passagens para ordená-los.
- `int houveTroca = 0;`: Esta variável de controle é uma otimização do Bubble Sort. Ela é inicializada como 0 (falso) no início de cada passagem. Se o laço interno não realizar nenhuma troca, significa que o vetor já está ordenado, e podemos parar o algoritmo mais cedo.
- `for (j = 0; j < n - i - 1; j++)`: Este é o laço interno. Ele é responsável por percorrer o vetor e comparar os pares de elementos vizinhos.
 - `n - i - 1`: A cada passagem do laço externo (`i`), o maior elemento já está na sua posição correta no final. Portanto, não precisamos mais comparar os elementos que já estão no lugar certo. Por exemplo, na primeira passagem (`i=0`), o maior elemento vai para a última posição. Na segunda passagem (`i=1`), o segundo maior elemento vai para a penúltima posição, e assim por diante. Essa otimização evita comparações desnecessárias, melhorando a eficiência do algoritmo.
- `if (vetor[j] > vetor[j + 1])`: Esta é a condição principal de comparação. Se o elemento atual (`vetor[j]`) for maior que o seu vizinho da direita (`vetor[j + 1]`), eles estão na ordem errada para uma ordenação crescente.
- `trocar(&vetor[j], &vetor[j + 1]);`: Com a condição sendo verdadeira, a função `trocar` é chamada. Note que usamos o operador `&` (operador de endereço) para passar o endereço de memória dos elementos do vetor, pois a função `trocar` espera ponteiros.
- `if (houveTroca == 0) break;`: Esta é a otimização comentada antes. Se, depois de uma passagem completa do laço interno, a variável `houveTroca` ainda for 0, significa que o vetor está totalmente ordenado. Nesse caso, usamos o comando `break` para sair do laço externo, encerrando o algoritmo.

3. `int main()`

Esta é a função principal do programa, onde a execução começa.

- `int vetor[] = {64, 34, 25, 12, 22, 11, 90};` Declara e inicializa um vetor de inteiros com os valores a serem ordenados.
- `int n = sizeof(vetor) / sizeof(vetor[0]);` Esta é uma forma padrão e portátil de calcular o número de elementos em um vetor em C.
 - `sizeof(vetor)`: Retorna o tamanho total do vetor em bytes.
 - `sizeof(vetor[0])`: Retorna o tamanho de um único elemento do vetor em bytes (neste caso, o tamanho de um `int`).
 - Ao dividir o tamanho total pelo tamanho de um elemento, obtemos o número exato de elementos no vetor, independentemente do tipo de dado ou da arquitetura do sistema. Isso é muito mais robusto do que simplesmente contar os elementos manualmente.
- `printf("Vetor original: ");` e `imprimirVetor(vetor, n);`: Exibe o vetor antes da ordenação.
- `bubbleSort(vetor, n);`: Chama a função para ordenar o vetor.
- `printf("Vetor ordenado: ");` e `imprimirVetor(vetor, n);`: Exibe o vetor após a ordenação.

5.2.2 Selection sort

5.2.3 Insertion sort

5.2.4 Shell sort

5.2.5 Merge sort

5.2.6 Quicksort

Capítulo 6

Alocação dinâmica