



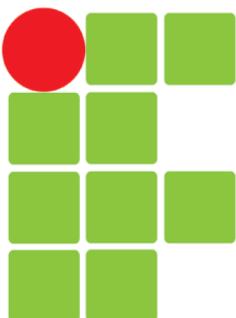
**MINISTÉRIO DA EDUCAÇÃO**  
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA CATARINENSE  
CAMPUS LUZERNA

# Engenharia de Controle e Automação

## Algoritmos e Estrutura da Informação

Professor: Ricardo Kerschbaumer

8 de novembro de 2024



**Aluno:** \_\_\_\_\_

INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
CATARINENSE

# Sumário

<b>1</b>	<b>Revisão de linguagem C</b>	<b>9</b>
1.1	Introdução	9
1.2	Estrutura de um programa em C	10
1.2.1	Arquivos de cabeçalho	10
1.2.2	A função principal	11
1.3	Sintaxe básica	11
1.3.1	Comentários	11
1.3.2	Indentação	12
1.3.3	Identificadores	12
1.3.4	Palavras reservadas	13
1.4	Variáveis e tipos de dados	13
1.4.1	Tipos de dados	13
1.4.2	Variáveis	14
1.4.3	Declaração de variáveis	14
1.4.4	Tipos de variáveis	15
1.4.5	O tipo void	16
1.5	Constantes	16
1.5.1	Caracteres e cadeias de caracteres	17
1.5.2	Definindo constantes	17
1.6	Classes de armazenamento	18
1.7	Operadores	18
1.7.1	Operadores aritméticos	19
1.7.2	Operadores relacionais	19
1.7.3	Operadores lógicos	19
1.7.4	Operadores de bits	20
1.7.5	Operadores de atribuição	20
1.7.6	Operadores diversos	20
1.7.7	Precedência e Associatividade de Operadores	21
1.8	Tomada de decisão	22
1.8.1	O comando if	22
1.8.2	O comando if - else	23
1.8.3	O comando switch - case	24
1.8.4	O operador ?:	25
1.9	Laços de repetição	26
1.9.1	O laço while	26
1.9.2	O laço do - while	27
1.9.3	O laço for	28
1.9.4	Declarações de controle	28
1.10	Funções	29
1.10.1	Declaração de funções em C	29
1.11	Regras de escopo	30
1.12	Vetores e matrizes	31

1.12.1 Vetores . . . . .	31
1.12.2 Acessando elementos de um vetor . . . . .	31
1.12.3 Matrizes . . . . .	32
1.13 Ponteiros . . . . .	32
1.14 Cadeias de caracteres . . . . .	33
1.15 Estruturas . . . . .	34
1.16 Uniões . . . . .	36
1.17 Definição de tipos . . . . .	37
1.18 Enumerações . . . . .	38
1.19 Entradas e saídas . . . . .	38
1.19.1 A função scanf . . . . .	39
1.19.2 A função printf . . . . .	39
1.20 Manipulação de arquivos . . . . .	40
1.20.1 Abertura de Arquivos . . . . .	41
1.20.2 Gravando Informações em um Arquivo . . . . .	42
1.20.3 Lendo Informações de um Arquivo . . . . .	43
1.21 Pré processador . . . . .	45
1.22 Conversão de tipos . . . . .	46
1.23 Recursividade . . . . .	46
1.24 Argumentos de linha de comando . . . . .	47
1.25 Alocação de memória . . . . .	47
<b>2 Ponteiros</b> . . . . .	<b>49</b>
2.1 Introdução . . . . .	49
2.2 O que são ponteiros . . . . .	49
2.3 Ponteiros na linguagem C . . . . .	49
2.3.1 Operadores utilizados com ponteiros . . . . .	49
2.3.2 Ponteiros NULL . . . . .	50
2.3.3 Atribuição de endereços aos ponteiros . . . . .	50
2.3.4 Aritmética de ponteiros . . . . .	51
2.3.5 Comparações de ponteiros . . . . .	52
2.4 O uso de ponteiros com vetores e matrizes . . . . .	52
2.5 Vetores e matrizes de ponteiros . . . . .	53
2.6 Ponteiros para ponteiros . . . . .	53
2.7 O uso de ponteiros com registros . . . . .	54
2.8 Exemplo de utilização de ponteiros . . . . .	55
<b>3 Abstração procedural</b> . . . . .	<b>56</b>
3.1 Introdução . . . . .	56
3.2 A importância da abstração procedural . . . . .	56
3.3 Como fazer a abstração procedural . . . . .	56
3.4 Quando utilizar a abstração procedural . . . . .	57
3.5 A abstração procedural na linguagem C . . . . .	57
3.5.1 Funções padrão do C . . . . .	57
3.5.2 Funções definidas pelo usuário . . . . .	57
3.6 Funções na linguagem C . . . . .	58
3.6.1 Protótipo de função . . . . .	59

3.6.2	Chamando uma função . . . . .	59
3.6.3	Definição de função . . . . .	59
3.6.4	Passando argumentos para uma função . . . . .	60
3.6.5	Declaração de retorno . . . . .	60
3.7	Passagem de parâmetros . . . . .	61
3.7.1	Passagem de parâmetros por valor . . . . .	61
3.7.2	Passagem de parâmetros por referência . . . . .	63
3.8	Exercícios . . . . .	66
<b>4</b>	<b>Recursividade e iteratividade</b>	<b>67</b>
4.1	Introdução . . . . .	67
4.2	Iteratividade . . . . .	67
4.3	Recursividade . . . . .	68
4.4	Algumas conclusões . . . . .	69
4.5	Ordem de crescimento . . . . .	70
4.6	Exercícios . . . . .	71
<b>5</b>	<b>Abstração de dados: pilhas e filas</b>	<b>72</b>
5.1	Introdução . . . . .	72
5.2	Pilhas . . . . .	72
5.2.1	Implementação da pilha em C . . . . .	73
5.2.2	Um exemplo de implementação de pilha . . . . .	75
5.3	Filas . . . . .	77
5.3.1	Implementação da fila em C . . . . .	77
5.3.2	Um exemplo de implementação de fila . . . . .	80
5.4	Filas circulares . . . . .	82
5.4.1	Implementação da fila circular em C . . . . .	83
5.4.2	Um exemplo de implementação de fila circular . . . . .	85
5.5	Exercícios . . . . .	88
<b>6</b>	<b>Abstração de dados: listas</b>	<b>89</b>
6.1	Introdução . . . . .	89
6.2	Listas . . . . .	89
6.2.1	Implementação de uma lista . . . . .	90
6.2.2	Percorrendo uma lista . . . . .	91
6.2.3	Inserindo nós na lista . . . . .	92
6.2.4	Inserindo um nó no início da lista . . . . .	92
6.2.5	Inserindo um nó após uma posição da lista . . . . .	92
6.2.6	Anexando um nó no final da lista . . . . .	93
6.2.7	Um exemplo mais completo . . . . .	93
6.2.8	Apagando um nó de uma lista . . . . .	95
6.2.9	Apagando um nó em uma posição da lista . . . . .	96
6.2.10	Exemplo de exclusão de nós de uma lista . . . . .	97
6.2.11	Apagando uma lista . . . . .	99
6.2.12	Exemplo de programa que apaga uma lista . . . . .	99
6.2.13	Encontrados nós na lista . . . . .	101
6.2.14	Lendo nós da lista . . . . .	102

---

6.2.15	Alterando nós da lista . . . . .	102
6.2.16	Exemplo com as três últimas funções . . . . .	103
6.3	Exercícios . . . . .	107
<b>7</b>	<b>Abstração de dados: Árvores binárias</b>	<b>108</b>
7.1	Introdução . . . . .	108
7.2	As árvores binárias . . . . .	108
7.2.1	Criando uma árvore binária . . . . .	109
7.2.2	Pesquisa em árvores binárias . . . . .	110
7.2.3	Exclusão de árvore binária . . . . .	111
7.2.4	Impressão de árvore binária . . . . .	111
7.2.5	Um exemplo de árvore binária . . . . .	112
7.3	Exercícios . . . . .	116
<b>8</b>	<b>Algoritmos de ordenação</b>	<b>117</b>
8.1	Introdução . . . . .	117
8.2	BubbleSort . . . . .	117
8.2.1	Funcionamento do BubbleSort . . . . .	117
8.2.2	Ordem de crescimento do BubbleSort . . . . .	120
8.3	InsertSort . . . . .	120
8.3.1	Funcionamento do InsertSort . . . . .	121
8.3.2	Ordem de crescimento do InsertSort . . . . .	123
8.4	QuickSort . . . . .	123
8.4.1	Funcionamento do QuickSort . . . . .	124
8.4.2	Ordem de crescimento do QuickSort . . . . .	126
8.5	Exercícios . . . . .	127

# Lista de Figuras

1.1	Processo de compilação . . . . .	9
1.2	O comando If . . . . .	22
1.3	O comando If - else . . . . .	23
1.4	O comando Switch - Case . . . . .	24
1.5	O laço while . . . . .	26
1.6	O laço do - while . . . . .	27
1.7	O laço for . . . . .	28
2.1	Exemplo de operador “&” . . . . .	50
2.2	Aritmética de ponteiro . . . . .	51
2.3	Relação entre vetores e ponteiros . . . . .	52
2.4	Ponteiros para ponteiros . . . . .	54
3.1	Funções na linguagem C . . . . .	58
3.2	Passagem de parâmetros . . . . .	60
3.3	Declaração de retorno . . . . .	61
3.4	Passagem de parâmetros por valor . . . . .	62
3.5	Passagem de parâmetros por referência . . . . .	64
4.1	Recursividade . . . . .	68
4.2	Ordens de crescimento . . . . .	71
5.1	Operações push e pop . . . . .	72
5.2	Funcionamento da pilha . . . . .	73
5.3	Operações enQueue e deQueue . . . . .	77
5.4	Funcionamento da fila . . . . .	79
5.5	Fila Circular . . . . .	82
5.6	Funcionamento da fila circular . . . . .	83
6.1	Listas . . . . .	89
7.1	Exemplo de árvore binária . . . . .	109
8.1	Primeiro passo do BubbleSort . . . . .	117
8.2	Segundo passo do BubbleSort . . . . .	118
8.3	Terceiro passo do BubbleSort . . . . .	118
8.4	Quarto passo do BubbleSort . . . . .	118
8.5	Primeiro passo do InsertSort . . . . .	121
8.6	Segundo passo do InsertSort . . . . .	121
8.7	Terceiro passo do InsertSort . . . . .	122
8.8	Quarto passo do InsertSort . . . . .	122
8.9	Primeiros passos do QuickSort . . . . .	124

# Lista de Códigos

1.1	Estrutura de um programa em C . . . . .	10
1.2	Estrutura da função main . . . . .	11
1.3	Exemplo de expressão . . . . .	11
1.4	Exemplos de comentários . . . . .	12
1.5	Exemplo de indentação . . . . .	12
1.6	Exemplo de definição de constante . . . . .	17
1.7	Exemplo de declaração de variável como constante . . . . .	18
1.8	Exemplo de comando if . . . . .	23
1.9	Exemplo de comando if - else . . . . .	23
1.10	Exemplo de comando switch - case . . . . .	25
1.11	Exemplo de comando ?: . . . . .	26
1.12	Exemplo de laço while . . . . .	27
1.13	Exemplo de laço do - while . . . . .	27
1.14	Exemplo de laço for . . . . .	28
1.15	Exemplo de função . . . . .	30
1.16	Exemplo de vetor . . . . .	31
1.17	Exemplo de matriz . . . . .	32
1.18	Exemplo de ponteiros . . . . .	33
1.19	Exemplo de cadeias de caracteres . . . . .	33
1.20	Exemplo de estruturas . . . . .	35
1.21	Exemplo de união . . . . .	36
1.22	Exemplo de definição de tipos . . . . .	37
1.23	Exemplo de enumerações . . . . .	38
1.24	Exemplo das funções scanf e printf . . . . .	40
1.25	Exemplo de abertura de arquivo . . . . .	42
1.26	Exemplo de gravação em arquivo . . . . .	42
1.27	Exemplo de leitura de arquivo . . . . .	43
1.28	Exemplo de leitura de arquivo com fgets . . . . .	43
1.29	Exemplo de gravação de vetor em arquivo . . . . .	44
1.30	Exemplo de leitura de vetor em arquivo . . . . .	44
1.31	Exemplo de conversão de tipo . . . . .	46
1.32	Exemplo de recursividade . . . . .	46
1.33	Exemplo de argumentos de linha de comando . . . . .	47
1.34	Exemplo de alocação de memória . . . . .	48
2.1	Exemplo de ponteiros para registros . . . . .	54
2.2	Exemplo de utilização de ponteiros . . . . .	55
3.1	Função definida pelo usuário . . . . .	58
3.2	Função com protótipo . . . . .	59
3.3	Passagem de parâmetro por valor . . . . .	62
3.4	Passagem de parâmetros maiores . . . . .	63
3.5	Passagem de parâmetros por referência . . . . .	64

3.6	Retorno da função como ponteiro . . . . .	65
3.7	Passagem de parâmetros maiores por referência . . . . .	65
4.1	Exemplo de iteratividade . . . . .	67
4.2	Exemplo de recursividade . . . . .	69
5.1	Criando uma pilha . . . . .	73
5.2	Verificando se a pilha está cheia . . . . .	74
5.3	Verificando se a pilha está vazia . . . . .	74
5.4	Armazenando dados na pilha . . . . .	74
5.5	Extraindo dados na pilha . . . . .	75
5.6	Exemplo de pilha . . . . .	75
5.7	Criando uma fila . . . . .	77
5.8	Inserindo dados na fila . . . . .	79
5.9	Extraindo dados na fila . . . . .	80
5.10	Exemplo de fila . . . . .	80
5.11	Criando uma fila circular . . . . .	84
5.12	Inserindo dados na fila circular . . . . .	84
5.13	Extraindo dados da fila circular . . . . .	84
5.14	Exemplo de fila circular . . . . .	85
6.1	Exemplo de nó de lista . . . . .	90
6.2	Exemplo de lista simples . . . . .	90
6.3	Percorrendo uma lista . . . . .	91
6.4	Inserindo dados no início da lista . . . . .	92
6.5	Inserindo dados no após de um nó . . . . .	92
6.6	Anexando dados no final da lista . . . . .	93
6.7	Exemplo de inserção de dados na lista . . . . .	93
6.8	Apagando um nó de uma lista . . . . .	95
6.9	Apagando um nó em uma posição da lista . . . . .	96
6.10	Apagando elementos da lista . . . . .	97
6.11	Apagando uma lista . . . . .	99
6.12	Exemplo de função para apagar uma lista . . . . .	99
6.13	Encontrando um nó na lista . . . . .	101
6.14	Lendo um nó da lista . . . . .	102
6.15	Alterando um nó da lista . . . . .	103
6.16	Encontrando, lendo e alterando nós da lista . . . . .	103
7.1	Exemplo de estrutura para árvore binária . . . . .	108
7.2	Inserindo nó na árvore . . . . .	110
7.3	Pesquisando na árvore . . . . .	110
7.4	Excluindo uma árvore . . . . .	111
7.5	Imprimindo uma árvore . . . . .	111
7.6	Exemplo de árvore binária . . . . .	112
8.1	O algoritmo do BubbleSort . . . . .	119
8.2	O algoritmo do BubbleSort otimizado . . . . .	119
8.3	O algoritmo InsertSort . . . . .	122

8.4 O algoritmo QuickSort . . . . . 125

# Aula 1 Revisão de linguagem C

## 1.1 Introdução

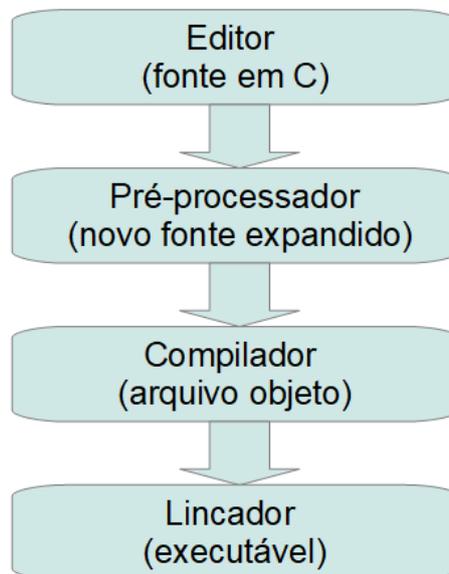
Nesta primeira aula será realizada uma revisão da linguagem de programação C. Esta revisão é importante pois todos os exemplos de algoritmos e estruturas de dados apresentados nas aulas seguintes, bem como os exercícios propostos estão em linguagem C.

A linguagem de programação C foi desenvolvida por Dennis Ritchie dos Laboratórios Bell da AT & T no início da década de 70 para desenvolver o sistema operacional UNIX. Trata-se de uma linguagem imperativa e procedural de baixo nível e de propósito geral. A linguagem C ainda é uma das linguagens de programação mais utilizadas, isso devido a características como acesso de baixo nível a memória, conjunto simplificado de palavras-chave, capacidade de manipular bits, bytes e endereços, suporte a diversas plataformas etc. Estas características também tornam a linguagem C interessante para a automação e o controle de processos.

Muitas linguagens modernas de programação tiveram origem ou herdaram características da linguagem C, como por exemplo, C++, Java, PHP, JavaScript entre outras.

A geração do programa executável através da linguagem C inicia através do desenvolvimento do código fonte do programa em um editor de texto. Este código fonte é então compilado para gerar o código executável do programa, para a plataforma desejada. Este processo de compilação é apresentado na figura 1.1.

Figura 1.1: Processo de compilação



## 1.2 Estrutura de um programa em C

Um programa em linguagem C deve obedecer a uma estrutura formal para ser corretamente compilado. No início devem ser inseridas as bibliotecas de funções através dos arquivos de cabeçalho, seguidas pelas diretivas de compilação. Na sequência podem ser inseridas as variáveis globais e as demais funções ou procedimentos.

A linguagem C exige que se tenha uma função principal chamada “main”, é através desta função que o programa inicia sua execução. No interior desta função é que ficam as declarações de variáveis locais, o corpo da função e as chamadas a outras funções.

O código 1.1 apresenta a estrutura básica de um programa em linguagem C.

Código 1.1: Estrutura de um programa em C

```
#include <stdio.h> //Cabeçalho
int x; //Variáveis globais
int main() //Função principal
{
    int y; //Variáveis locais
    x=5; //Corpo da função
    y=6;
    printf("Resultado = %d",x+y);
    return 0; //Retorno
}
```

### 1.2.1 Arquivos de cabeçalho

A inclusão de bibliotecas ou arquivos de cabeçalho é o mecanismo pelo qual conjuntos de funções ou procedimentos podem ser incluídos e utilizados em nossos programas.

Um arquivo de cabeçalho é um arquivo com extensão .h que contém declarações de funções e definições de macros compartilhadas entre programas. Alguns arquivos de cabeçalho são distribuídos com os compiladores de linguagem C, outros são desenvolvidos pelos próprios programadores.

Algumas das bibliotecas mais utilizadas são listadas a seguir.

- **stdio.h**: contém as principais funções de entrada e saída relacionadas a leitura de teclado, impressão na tela e manipulação de arquivos.
- **stdlib.h**: contém funções relacionadas ao funcionamento do programa, conversão de tipos de dados alocação de memória etc.
- **math.h**: contém as principais funções matemáticas e trigonométricas.

A sintaxe para incluir um arquivo de cabeçalho em um programa em C é:

```
#include <(nome do arquivo).h>
```

### 1.2.2 A função principal

Como foi dito, todo programa em C deve possuir uma função principal por onde o programa inicia. Na linguagem C esta função se chama “main”. A sintaxe desta função inicia pela declaração de tipo `int`, seguida pela palavra `main` e por um par de parênteses. O corpo da função `main` é delimitado por um par de colchetes. A função `main` deve terminar retornando um valor numérico inteiro. O código 1.2 apresenta a estrutura básica da função `main`.

Código 1.2: Estrutura da função `main`

```
int main() //Função principal
{
    return 0; //Retorno
}
```

## 1.3 Sintaxe básica

A partir da estrutura de um programa em C apresentada na seção anterior podemos começar a apresentar os detalhes da programação em linguagem C.

A linguagem C trabalha com os chamados tokens ou palavras chaves, cada instrução do programa é composta por um conjunto de tokens. Estes tokens são identificadores, constantes cadeias de caracteres, símbolos etc. Para que o compilador possa compilar corretamente o programa estes conjuntos de tokens devem estar compostos corretamente. Estes conjuntos de tokens compõe as chamadas expressões. Na linguagem C todas as expressões devem ser finalizadas com ponto e vírgula. Veja um exemplo de expressão no código 1.3.

Código 1.3: Exemplo de expressão

```
printf("Olá Mundo");
```

Este exemplo de expressão utiliza o comando `printf` para imprimir `Olá Mundo` na tela do computador.

### 1.3.1 Comentários

A linguagem C permite que se faça comentários dentro dos programas. Estes comentários são textos que não são interpretados pelo compilador e servem para ajudar o programador a documentar seu código.

Comentários são muito importantes para facilitar o desenvolvimento, a manutenção e o reaproveitamento dos códigos, e devem ser amplamente utilizados.

Na linguagem C existem dois tipos de comentários. Os comentários de uma única linha e os comentários de várias linhas. Para comentários de uma única linha utiliza-se o token `//`. O compilador ignora tudo que estiver entre o token `//` e o final da linha. Para comentários de várias linhas utiliza-se o token `/*` no início do comentário e o token `*/` no final. Os comentários de várias linhas são o padrão do C, mas a maioria dos compiladores também aceita a sintaxe dos comentários de uma linha. O código 1.4 apresenta alguns exemplos de comentários.

Código 1.4: Exemplos de comentários

```
#include <stdio.h>

/* Este é um exemplo de comentário
de várias linhas. Todo este bloco de
texto é ignorado pelo compilador.*/

int main(void)
{
    // Este é um comentário de uma linha.
    printf("Olá Mundo");
    return 0;
}
```

### 1.3.2 Indentação

Indentação é uma forma de organizar o código fonte do programa para torná-lo mais legível. O uso de regras de consistentes de indentação permite visualizar a estrutura global do programa através de uma simples inspeção visual.

A ideia básica da indentação é adicionar uma ou mais tabulações no início da linha das expressões que estão incluídas no interior de outras expressões. Quando várias expressões são aninhadas esta estratégia permite visualizar facilmente esta estrutura.

Existem várias estratégias de indentação, cada programador tem liberdade de escolher sua forma de trabalhar pois o compilador ignora a indentação do programa. O importante é escolher uma estratégia de indentação e mantê-la consistente durante o desenvolvimento do código.

O código 1.5 apresenta um exemplo de indentação.

Código 1.5: Exemplo de indentação

```
#include <stdio.h>

int main(void)
{
    int cont;
    for(cont=1; cont<=10; cont++)
    {
        printf("cont=%d\n",cont);
    }
    return 0;
}
```

### 1.3.3 Identificadores

Na linguagem C os nomes de variáveis, funções e quaisquer outras estruturas definidas pelo programador são chamadas de identificadores. Os nomes de identificadores possuem regras rígidas em sua composição. Na linguagem C um identificador deve necessariamente iniciar com uma letra de A a Z ou uma letra de a a z ou ainda com o caractere `_`, seguidos de

zero ou mais letras ou dígitos de 0 a 9 ou caracteres `_`. Não é permitido utilizar espaços ou caracteres especiais.

A linguagem C diferencia letras maiúsculas de minúsculas, assim o identificados Contador é diferente do identificador contador, por exemplo.

Linhas em branco, assim como os comentários são ignoradas pelo compilador.

Espaços em branco, tabulações e quebras de linha são utilizados pelo compilador para separar as partes das expressões, o número destes elementos não importa.

### 1.3.4 Palavras reservadas

A linguagem C possui 32 palavras reservadas, todas minúsculas, que não podem ser usadas para nenhum outro propósito durante o desenvolvimento de um programa. A tabela 1.1 mostra as palavras reservadas da linguagem C conforme o padrão ANSI.

Tabela 1.1: Palavras Reservadas

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

## 1.4 Variáveis e tipos de dados

### 1.4.1 Tipos de dados

Um programa de computador utiliza a memória para armazenar todas as informações que ele está processando. Cada tipo de informação utiliza uma determinada quantidade de memória, esta quantidade de memória e o tipo de informação relacionado é definido pelo tipo de dado utilizado.

Os tipos de dados na linguagem C são utilizados para declarar variáveis e funções de diferentes tipos. O tipo de dados utilizado na variável também determina como o padrão de bits armazenado na memória deve ser interpretado pelo processador.

A linguagem C tem um grupo grande de tipos de dados pré-definidos, e permite que o programador crie seus próprios tipos de dados. Os tipos de dados empregados na linguagem C podem ser descritos da seguinte forma.

- **Tipos básicos:** São tipos aritméticos e são classificados em: tipos inteiros e tipos de ponto flutuante.
- **Tipos enumerados:** São também tipos aritméticos e são usados para definir variáveis com certos valores inteiros discretos.

- **O tipo void:** É um tipo vazio que indica que nenhum valor está disponível.
- **Tipos derivados:** São tipos especiais que incluem ponteiro, vetores, matrizes, estruturas, uniões e funções.

### 1.4.2 Variáveis

As variáveis são utilizadas para manter o controle sobre os dados armazenados pelo programa na memória do computador. Com as variáveis é possível dar um nome a um determinado local na memória, de forma que fique mais fácil manipular os dados ali armazenados. As variáveis também são responsáveis por informar ao compilador os tipos de dados armazenados naquele local.

O tipo de dado associado a variável determina o tamanho, o formado e o range dos dados armazenados na posição de memória associada.

Com base nos tipos de dados apresentados anteriormente a linguagem C apresenta os seguintes tipos básicos de variáveis.

- **char:** tipo inteiro composto normalmente por 1 byte.
- **int:** tipo inteiro cujo tamanho depende da arquitetura para a qual o programa é compilado.
- **float:** Ponto flutuante de precisão simples.
- **double:** Ponto flutuante de precisão dupla.
- **void:** Representa a ausência do tipo.

### 1.4.3 Declaração de variáveis

Na linguagem C a declaração das variáveis é feita da seguinte forma.

```
tipo_da_variável nome_da_variável;
```

A seguir temos um exemplo de declaração de variável em C.

```
int x;
```

Neste exemplo “int” é o tipo da variável e “x” é o nome da variável. Como todas as expressões em C a declaração de uma variável termina com um ponto e vírgula. Na linguagem C é necessário declarar qualquer variável antes de utilizá-la.

É possível declarar várias variáveis do mesmo tipo de uma única vez, separando seus nomes por vírgula. Veja alguns exemplos a seguir.

```
char linha, coluna;  
int x=0, y=0;
```

A regra para a formação de nomes de variáveis segue a regra para formação de identificadores discutida anteriormente

### 1.4.4 Tipos de variáveis

A linguagem C tem vários tipos de dados pré-definidos, e permite que o programador crie seus próprios tipos de dados. A seguir são apresentados os dois principais tipos de dados, os tipos inteiros e os tipos de ponto flutuante.

**Tipos de dados inteiros:** ocupam de 8 a 64 bits na memória, e podem conter valores numéricos ou letras (caracteres). Como existem várias arquiteturas, com processadores de barramentos com número de bits diferentes, também existem variações na definição dos tipos de dados na linguagem C. A tabela 1.2 apresenta os principais tipos de variáveis inteiras utilizadas na linguagem C e seus respectivos ranges.

Tabela 1.2: Tipos de variáveis

Tipos	Tamanho na memória	Range
char	1 byte (8 bits)	-128 a 127
unsigned char	1 byte (8 bits)	0 a 255
signed char	1 byte (8 bits)	-128 a 127
int	2 ou 4 bytes (16 ou 32 bits)	-32768 a 32767 ou -2147483648 a 2147483647
unsigned int	2 ou 4 bytes (16 ou 32 bits)	0 a 65535 ou 0 a 4294967295
short	2 bytes (16 bits)	-32768 a 32767
unsigned short	2 bytes (16 bits)	0 a 65535
long	8 bytes (64 bits)	-9223372036854775808 a 9223372036854775807
unsigned long	8 bytes (64 bits)	0 a 18446744073709551615

A seguir temos alguns exemplos de declaração de variáveis do tipo inteiro.

```
int contador;
unsigned int y;
char temperatura;
char velocidade=0;
char letra='A';
long int Z=126543;
```

Observe que é possível atribuir um valor a variável utilizando o operador “=”. O tipo de dado “char” também é utilizado para armazenar uma letra, para atribuir uma letra a uma variável a letra deve estar entre aspas simples.

**Tipos de dados de ponto flutuante:** servem para armazenar números reais, ou seja, valores fracionários. A tabela 1.3 apresenta os principais tipos de variáveis de ponto flutuante utilizadas na linguagem C e seus respectivos ranges.

Tabela 1.3: Tipos de variáveis

Tipo	Tamanho na memória	Range	precisão
float	4 bytes (32 bits)	$\pm 1.2E-38$ a $\pm 3.4E+38$	6 casas decimais
double	8 bytes (64 bits)	$\pm 2.3E-308$ a $\pm 1.7E+308$	15 casas decimais
long double	10 bytes (80 bits)	$\pm 3.4E-4932$ a $\pm 1.1E+4932$	19 casas decimais

A seguir temos alguns exemplos.

```
float x;  
double y=12.124;
```

É importante salientar que o compilador utiliza a notação inglesa, assim o número fracionário utiliza o ponto (.) e não a vírgula (,).

### 1.4.5 O tipo void

O tipo void especifica que nenhum valor está disponível e é usado em três casos.

- Quando uma função retorna como vazia ou seja, quando uma função não necessita retornar nenhum valor ela é declarada como void.
- Quando uma função não recebe argumentos.
- Ponteiros de tipo nulo. Um ponteiro do tipo void \* representa o endereço de um objeto, mas não o seu tipo.

## 1.5 Constantes

Constantes são valores fixos que não podem ser alterados durante a execução do programa.

Uma constante inteira, por exemplo, pode ser declarada de forma decimal, hexadecimal ou octal. Para números em hexadecimal deve-se utilizar o prefixo 0x ou 0X e para números em octal deve-se utilizar o prefixo 0 (zero). veja alguns exemplos a seguir.

```
2019 // inteiro no formato decimal  
0xFE23 // inteiro no formato hexadecimal  
0253 // inteiro no formato octal
```

As constantes inteiras também podem ter dois sufixos, que indicam se o valor é um inteiro sem sinal (unsigned) representado pela letra U ou u e que indicam se é um inteiro longo (long) representado pela letra L ou l. Os prefixos e sufixos podem ser combinados sem restrições. Veja alguns exemplos a seguir.

```
432345L // inteiro longo  
54U // inteiro sem sinal  
0x3F2UL // inteiro longo, sem sinal e em formato hexadecimal
```

As constantes de ponto flutuante por sua vez podem ser representadas no formato decimal ou no formato exponencial. No formato decimal basta utilizar o ponto decimal (.). Já no formato exponencial é necessário utilizar uma letra e ou E para separar o expoente da parte decimal do número. Vaja alguns exemplos a seguir.

```
12.5 // representação decimal (12,5)  
1.34e-3 // representação exponencial (0,00134)  
8e6 // representação exponencial (8000000,0)
```

A notação exponencial é muito útil para representar números muito grandes ou muito pequenos.

### 1.5.1 Caracteres e cadeias de caracteres

Na linguagem C os caracteres (letras) também podem ser constantes fixas, para inserir um caractere em um programa utiliza-se aspas simples ( ' '). Já para cadeias de caracteres (textos) utiliza-se aspas duplas ( " "). Veja alguns exemplos a seguir.

```
'a' // um caractere
"olá mundo" // uma cadeia de caracteres
```

Existem também alguns caracteres especiais na linguagem C. A seguir são apresentados alguns deles.

```
\\ caractere \
\' caractere '
\" caractere "
\b Backspace
\n Nova linha
\r Retorno de carro
\t tabulação
```

### 1.5.2 Definindo constantes

A linguagem C permite a definição de constantes de duas formas diferentes. Utilizando a diretiva do pré-compilador "#define" ou utilizando a palavra chave "const".

A diretiva "#define" define um identificador que representa um valor constante em todo o programa. Sua utilização é definida a seguir.

```
#define identificador valor
```

O código 1.6 apresenta um exemplo da utilização de "#define".

Código 1.6: Exemplo de definição de constante

```
#include <stdio.h>

#define PI 3.14 // definição de uma constante

int main()
{
    float raio=10.0;
    float per;
    per=2.0*PI*raio;
    printf("perímetro = %f", per);
    return 0;
}
```

A segunda forma de definir uma constante em C é declarar uma variável como constante usando a palavra chave "const". A seguir é apresentada a forma de utilização da palavra chave "const" na declaração de uma variável.

```
cont tipo_da_variável identificador = valor;
```

É uma prática comum entre os programadores declarar constantes em letras maiúsculas, isso ajuda na clareza do programa.

O código 1.7 apresenta um exemplo de programa que utiliza a palavra chave “const” para declarar uma variável como constante.

Código 1.7: Exemplo de declaração de variável como constante

```
#include <stdio.h>

int main()
{
    const float PRECO = 7.5; // declaração de uma constante
    int quantidade;
    printf("Digite a quantidade: ");
    scanf("%d",&quantidade);
    printf("\nTotal = %.2f",quantidade*PRECO);
    return 0;
}
```

## 1.6 Classes de armazenamento

Na linguagem C as classes de armazenamento dizem respeito a visibilidade e ao tempo de vida das variáveis e funções em um programa. Estas classes podem ser “auto”, “register”, “static” ou “extern”, e devem ser atribuídas, assim como a palavra chave “const”, na declaração das variáveis.

- **auto:** é a classe padrão do C, e não necessita ser declarada explicitamente.
- **register:** instrui o compilador a colocar esta variável em um registrador da CPU e não na memória RAM. Isto permite um acesso muito mais rápido a esta variável, porém é dependente de plataforma e disponibilidade de registradores.
- **static:** instrui o compilador a manter o conteúdo da variável durante toda a atividade do programa, forçando esta variável a manter seu valor entre as chamadas de função. A classe “static” também pode ser aplicada a variáveis globais, fazendo com que o escopo desta variável seja restrito ao arquivo no qual ela está declarada.
- **extern:** serve para indicar uma variável global que é visível em todos os arquivos fonte do programa. Normalmente a variável declarada como “extern” referencia uma variável global declarada em outro arquivo do programa.

## 1.7 Operadores

Existem diversos tipos de operadores na linguagem C, eles podem ser divididos em: operadores aritméticos, operadores relacionais, operadores lógicos, operadores de bits, operadores de atribuição e operadores diversos.

### 1.7.1 Operadores aritméticos

A tabela 1.4 apresenta estes operadores.

Tabela 1.4: Operadores aritméticos

Operador	Exemplo	Descrição
+	a + b	Soma a com b
-	a - b	Subtrai b de a
*	a * b	Multiplica a por b
/	a / b	Divide a por b
%	a % b	O resto da divisão de a por b
++	a++	Incrementa a
--	a--	Decrementa a

### 1.7.2 Operadores relacionais

Operadores relacionais são utilizados nas tomadas de decisão e retornam verdadeiro ou falso de acordo com o resultado da comparação. A tabela 1.5 apresenta estes operadores.

Tabela 1.5: Operadores relacionais

Operador	Exemplo	Descrição
==	a == b	Verifica se a é igual a b
!=	a != b	Verifica se a é diferente de B
>	a > b	Verifica se a é maior que b
<	a < b	Verifica se a é menor que b
>=	a >= b	Verifica se a é maior ou igual a b
<=	a <= b	Verifica se a é menor ou igual a b

### 1.7.3 Operadores lógicos

Operadores lógicos são utilizados para concatenar ou alterar comparações e retornam verdadeiro ou falso de acordo com o resultado da operação lógica realizada com os resultados das comparações. A tabela 1.6 apresenta estes operadores.

Tabela 1.6: Operadores lógicos

Operador	Exemplo	Descrição
&&	(a > b) && (c == d)	Retorna verdadeiro se a for maior que b e se c for igual a d
	(a > b)    (c == d)	Retorna verdadeiro se a for maior que b ou se c for igual a d
!	!(a == b)	Retorna verdadeiro se a não for igual a b

### 1.7.4 Operadores de bits

Operadores de bits realizam operações E, OU e OU exclusivo entre os bits de variáveis. A tabela 1.7 apresenta estes operadores.

Tabela 1.7: Operadores de bits

Operador	Exemplo	Descrição
&	a & b	Retorna a operação lógica E (AND) entre os bits das variáveis a e b
	a   b	Retorna a operação lógica OU (OR) entre os bits das variáveis a e b
^	a ^ b	Retorna a operação OU exclusivo entre os bits das variáveis a e b
~	~a	Retorna a inversão de todos os bits de a
<<	a << 3	Retorna os bits de a deslocados 3 bits para a esquerda
>>	a >> 2	Retorna os bits de a deslocados 2 bits para a direita

### 1.7.5 Operadores de atribuição

Operadores de atribuição realizam a atribuição dos dados às variáveis. A tabela 1.8 apresenta estes operadores.

Tabela 1.8: Operadores de atribuição

Operador	Exemplo	Descrição
=	x = b	x recebe o conteúdo de b
+=	x += b	Equivalente a x = x + b
-=	x -= b	Equivalente a x = x - b
*=	x *= b	Equivalente a x = x * b
/=	x /= b	Equivalente a x = x / b
%=	x %= b	Equivalente a x = x % b
<<=	x <<= b	Equivalente a x = x << b
>>=	x >>= b	Equivalente a x = x >> b
&=	x &= b	Equivalente a x = x & b
=	x  = b	Equivalente a x = x   b
^=	x ^= b	Equivalente a x = x ^ b

### 1.7.6 Operadores diversos

Existem alguns outros operadores que não se encaixam nas categorias anteriores. A tabela 1.9 apresenta estes operadores.

Tabela 1.9: Operadores diversos

Operador	Exemplo	Descrição
sizeof	sizeof(a)	Retorna o tamanho em bytes da variável a na memória
&	&a	Retorna o endereço da variável a
*	*a	Caracteriza um ponteiro chamado a
?:	a > 20 ? 5 : 10	Se a for maior que 20 retorna 5 senão retorna 10
.	dados.a	Seleciona o membro a de dados
->	dados->a	Seleciona o membro apontado por a de dados

### 1.7.7 Precedência e Associatividade de Operadores

A tabela 1.10 apresenta os operadores da linguagem C por ordem de precedência, da maior para a menor. A associatividade indica em que ordem operadores de mesma precedência são avaliados em uma expressão.

Tabela 1.10: Precedência e Associatividade de Operadores

Operador	Associatividade
()	esquerda para direita
[]	esquerda para direita
.	esquerda para direita
->	esquerda para direita
++ e -- (pós-fixado)	esquerda para direita
++ e -- (pré-fixado)	direita para esquerda
+ e - (unário)	direita para esquerda
! e ~	direita para esquerda
(type)	direita para esquerda
* (ponteiro)	direita para esquerda
& (endereço)	direita para esquerda
sizeof	direita para esquerda
*, / e %	esquerda para direita
+ e -	esquerda para direita
<< e >>	esquerda para direita
< e <=	esquerda para direita
> e >=	esquerda para direita
== e !=	esquerda para direita
& (operação e de bits)	esquerda para direita
^(ou exclusivo de bit)	esquerda para direita
(ou de bit)	esquerda para direita
&&	esquerda para direita

	esquerda para direita
?:	direita para esquerda
=	direita para esquerda
+= e -=	direita para esquerda
*= e /=	direita para esquerda
%= e &=	direita para esquerda
^= e  =	direita para esquerda
<<= e >>=	direita para esquerda
,	esquerda para direita

## 1.8 Tomada de decisão

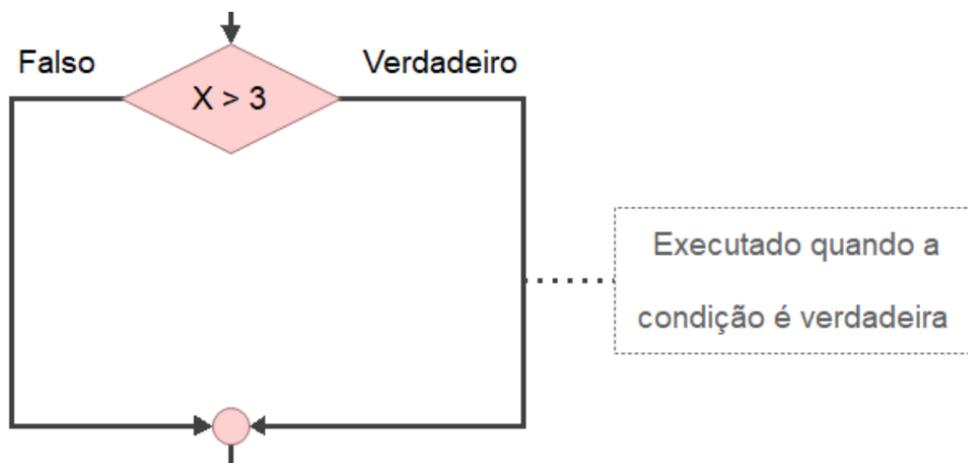
Na programação em linguagem C o programador deve especificar uma ou mais condições a serem avaliadas pelo programa. Um conjunto de instruções é então executado ou não dependendo do resultado das avaliações das condições impostas.

### 1.8.1 O comando if

O comando “if” é utilizado quando queremos que um determinado trecho de código somente seja executado se uma determinada condição ou conjunto de condições for verdadeiro.

A figura 1.2 apresenta o fluxograma de execução do comando “if”.

Figura 1.2: O comando If



O trecho de código 1.8 apresenta um exemplo de utilização do comando “if”.

Código 1.8: Exemplo de comando if

```
int x;
printf("Digite um número\n");
scanf("%d",&x);
if(x>10)
{
    printf("o número é maior que 10!");
}
```

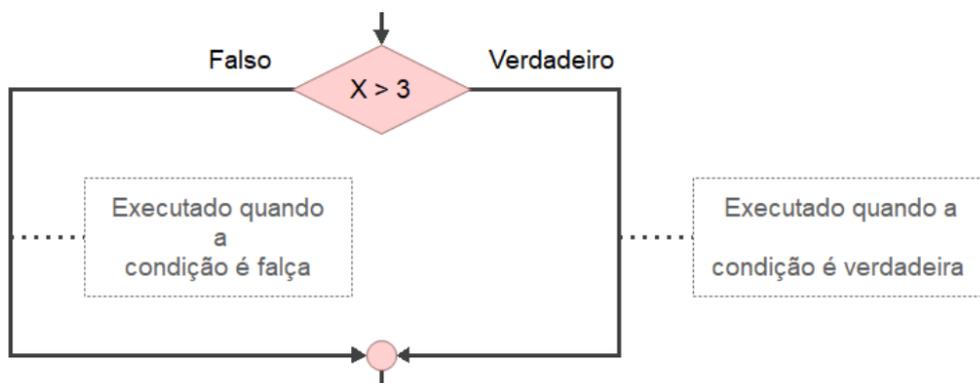
Quando apenas uma expressão é utilizada no interior do comando “if” a utilização de chaves ( { } ) não é obrigatória.

### 1.8.2 O comando if - else

O comando “if - else” é um complemento do comando “if” e é utilizado quando queremos escolher entre dois trechos de código baseado no resultado de um ou mais testes.

A figura 1.3 apresenta o fluxograma de execução do comando “if - else”.

Figura 1.3: O comando If - else



O trecho de código 1.9 apresenta um exemplo de utilização do comando “if - else”.

Código 1.9: Exemplo de comando if - else

```
int x;
printf("Digite um número\n");
scanf("%d",&x);
if(x>10)
{
    printf("o número é maior que 10!");
}
else
{
    printf("o número é menor ou igual a 10!");
}
```

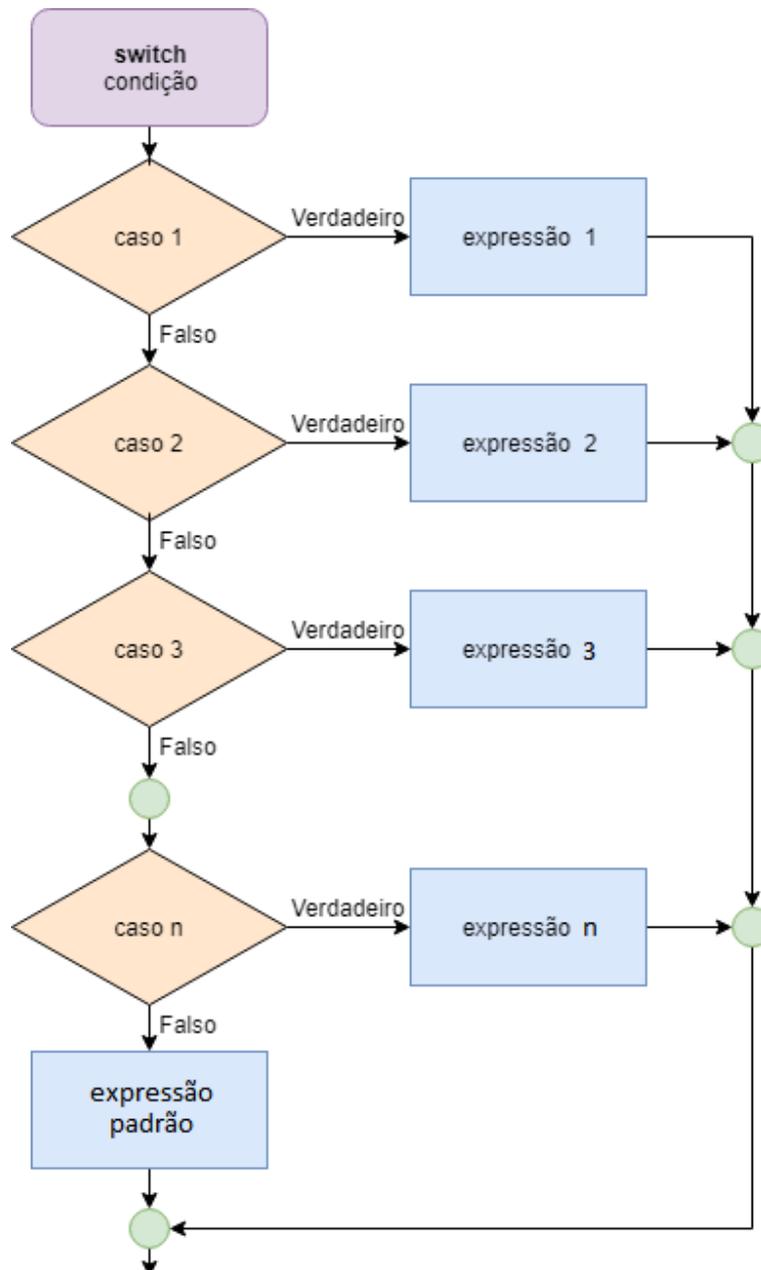
Quando apenas uma expressão é utilizada no interior do comando “if” ou do comando “else” a utilização de chaves ( { } ) não é obrigatória.

Os comandos “if” e “if - else” podem ser aninhados uns no interior dos outros de forma a compor conjuntos de decisões mais complexos.

### 1.8.3 O comando switch - case

O comando “switch - case” é utilizado quando queremos escolher uma entre várias condições. A figura 1.4 apresenta o fluxograma de execução do comando “switch - case”.

Figura 1.4: O comando Switch - Case



O trecho de código 1.10 apresenta um exemplo de utilização do comando “switch - case”.

Código 1.10: Exemplo de comando switch - case

```
int dia;
printf ("Digite o dia de 1 a 7: ");
scanf ("%d", &dia);
switch (dia)
{
    case 1:
        printf ("Domingo\n");
        break;

    case 2:
        printf ("Segunda\n");
        break;

    case 3:
        printf ("Terça\n");
        break;

    case 4:
        printf ("Quarta\n");
        break;

    case 5:
        printf ("Quinta\n");
        break;

    case 6:
        printf ("Sexta\n");
        break;

    case 7:
        printf ("Sábado\n");
        break;

    default:
        printf ("Valor inválido!\n");
}
```

Para compor códigos com várias expressões no interior do “switch - case” podem ser usadas chaves ({}).

#### 1.8.4 O operador ?:

O operador ?: também chamado de operador ternário já foi abordado na seção anterior, mas sua utilização está relacionada com as tomadas de decisão. Este operador pode ser utilizado para substituir um comando “if - else” simples.

A sintaxe é: **Condição ? verdadeiro : falso**

Veja um exemplo no código 1.11.

Código 1.11: Exemplo de comando ?:

```
#include <stdio.h>

int main(void)
{
    int a, b, maior;
    printf("Digite 2 valores ");
    scanf("%d",&a);
    scanf("%d",&b);
    maior = (a > b) ? a : b;
    printf("\n0 maior número é %d", maior);
    return 0;
}
```

Neste exemplo a variável “maior” vai receber o valor de a se a for maior que b, senão ela vai receber o valor de b.

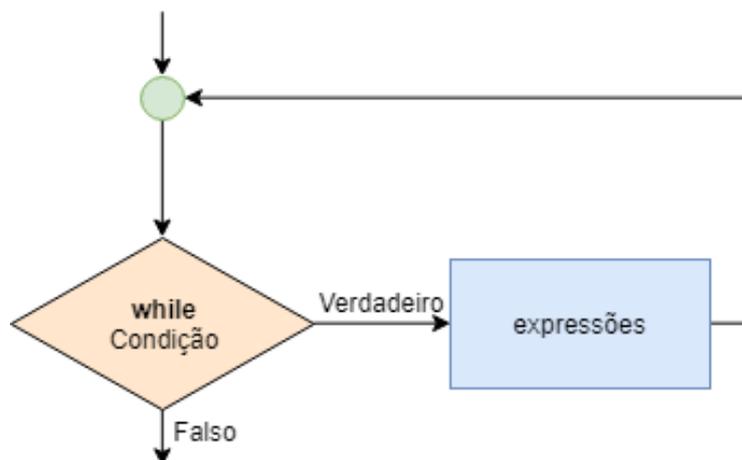
## 1.9 Laços de repetição

### 1.9.1 O laço while

O laço “while” é utilizado quando se quer repetir um conjunto de expressões enquanto uma determinada condição for verdadeira.

A figura 1.5 apresenta o fluxograma de execução do laço “while”.

Figura 1.5: O laço while



O trecho de código 1.12 apresenta um exemplo de utilização do laço “while”.

Código 1.12: Exemplo de laço while

```
int x=10;
while(x>=0)
{
    printf("%d\n",x);
    x--;
}
```

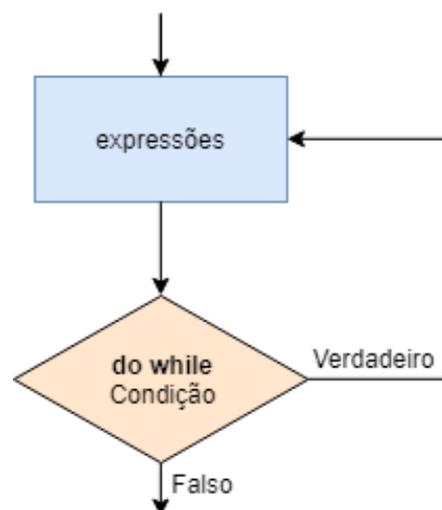
Quando apenas uma expressão é utilizada no interior do laço “while” a utilização de chaves ( { } ) não é obrigatória.

### 1.9.2 O laço do - while

O laço “do - while” é parecido com o laço while com a diferença de executar o código uma vez antes de testar a condição.

A figura 1.6 apresenta o fluxograma de execução do laço “do - while”.

Figura 1.6: O laço do - while



O trecho de código 1.13 apresenta um exemplo de utilização do laço “do - while”.

Código 1.13: Exemplo de laço do - while

```
double n, soma = 0;
do
{
    printf("Digite um número (0 para sair): ");
    scanf("%lf", &n);
    soma = soma + n;
}
while(n != 0.0);
printf("Soma = %lf",soma);
```

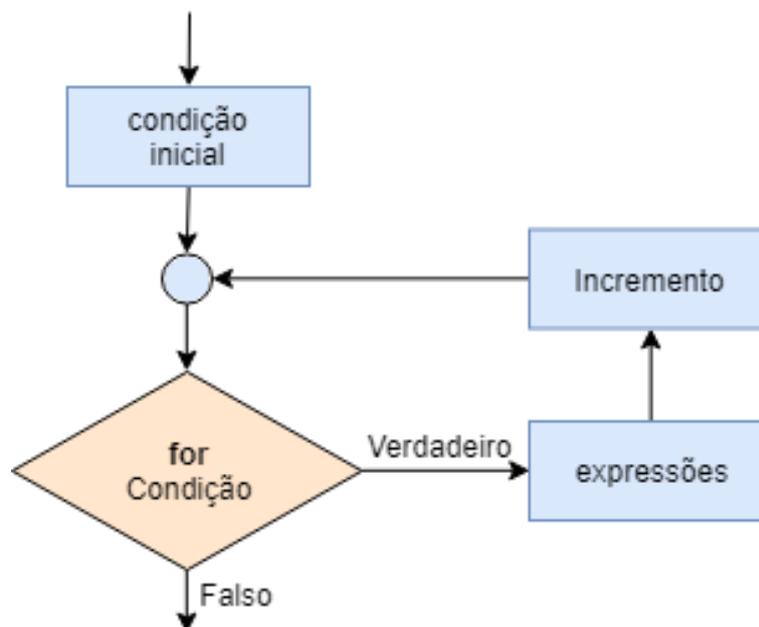
Quando apenas uma expressão é utilizada no interior do laço “do - while” a utilização de chaves ( { } ) não é obrigatória.

### 1.9.3 O laço for

O laço “for” é utilizado quando queremos repetir um trecho de código um determinado número de vezes. Sua estrutura facilita o gerenciamento da condição inicial, do incremento e do teste da variável de controle do laço.

A figura 1.7 apresenta o fluxograma de execução do laço “for”.

Figura 1.7: O laço for



O trecho de código 1.14 apresenta um exemplo de utilização do laço “for”.

Código 1.14: Exemplo de laço for

```
int cont;
for(cont=0; cont<=10; cont++)
{
    printf("%d\n",cont);
}
```

Quando apenas uma expressão é utilizada no interior do laço “for” a utilização de chaves ( { } ) não é obrigatória.

### 1.9.4 Declarações de controle

Declarações de controle são declarações que alteram o fluxo dos laços de repetição.

- **break:** o comando break encerra a execução do laço ou switch e transfere a execução para a instrução imediatamente após o laço ou o switch.

- **continue:** Faz com que o laço pule o restante das instruções e repita imediatamente sua condição antes de continuar a próxima iteração.
- **goto:** transfere o controle para a instrução marcada com o rótulo utilizado no comando.

## 1.10 Funções

Na linguagem C os programas são divididos em funções, todo programa deve possuir pelo menos uma função, a função “main”. Além da função principal é possível definir outras funções. Funções são grupos de instruções que juntas executam uma determinada tarefa.

Em C as funções devem informar ao compilador o nome de uma função, seu tipo de dados de retorno e os parâmetros que a função deve receber. É comum que se utilize outros nomes para as funções, como método, sub-rotina ou procedimento.

### 1.10.1 Declaração de funções em C

A declaração de funções na linguagem C possui a seguinte forma.

```
tipo identificador(parâmetros)
{
    corpo da função
}
```

- **tipo:** é o tipo de dados que a função retorna. Se a função não retorna nada utiliza-se o tipo “void”.
- **identificador:** é o nome da função e deve seguir as regras para a formação de identificadores discutida anteriormente.
- **parâmetros:** são os parâmetros que a função recebe. São listados os nomes dos parâmetros e seu tipo de dados.
- **corpo da função:** é onde ficam as instruções que compõe a função propriamente dita, podem ser utilizados quaisquer dos comandos da linguagem C.

Na linguagem C uma função deve ser definida antes de sua utilização no código do programa. Para isso é possível construir toda a função em uma posição do programa anterior a sua utilização ou então colocar toda a função em qualquer posição do código e criar uma declaração para esta função no início do programa. A declaração da função é um resumo da função, contendo seu tipo, nome e parâmetros.

Para utilizar uma função basta chamá-la pelo nome, passando os parâmetros necessários, e a função vai retornar o resultado devido.

O trecho de código 1.15 apresenta um exemplo que utiliza a declaração de uma função no início do programa e sua definição no final do mesmo.

Código 1.15: Exemplo de função

```
#include <stdio.h>

// função que retorna o maior de dois números
float maior(float a, float b); // declaração

int main(void)
{
    float n1,n2;
    printf("Digite dois números:\n");
    scanf("%f",&n1);
    scanf("%f",&n2);
    printf("O maior é %f",maior(n1, n2));
    return 0;
}

float maior(float a, float b) // definição
{
    float maior;
    maior=(a>b)?a:b;
    return(maior);
}
```

## 1.11 Regras de escopo

As regras de escopo são responsáveis por determinar onde cada variável pode ser acessada dentro do código. Na linguagem C o escopo de uma variável está relacionado ao local onde a variável foi declarada. Existem três locais para a declaração de variáveis na linguagem C, fora de todas as funções, na listagem de parâmetros de uma função e no interior de uma função. As variáveis declaradas fora de todas as funções são chamadas de **variáveis globais**. As variáveis declaradas como parâmetros de função, por sua vez, são chamadas de **parâmetros formais**. E finalmente as variáveis declaradas no interior de uma função, são chamadas **variáveis locais**.

Cada tipo de variável tem a seguinte regra de escopo.

- **Variáveis globais:** são geralmente definidas no início do programa e mantêm seu valor durante toda a execução do programa. Este tipo de variável pode ser acessado em qualquer parte do programa.
- **Variáveis locais:** são declaradas no interior de funções e seu valor é armazenado apenas durante a execução da função. Elas podem ser acessadas apenas do interior da função onde foram declaradas.
- **Parâmetros formais:** São interpretadas como variáveis locais e seguem a mesma lógica.

Se em um programa forem declaradas variáveis locais e globais com o mesmo nome, dentro da função a variável local tem precedência.

Variáveis globais são automaticamente inicializadas pelo sistema, normalmente com o valor zero. Já variáveis locais, não são inicializadas e o programador deve garantir que um valor seja atribuído a elas antes da sua utilização. Variáveis locais não inicializadas podem assumir qualquer valor dependendo do conteúdo armazenado anteriormente na mesma região de memória.

## 1.12 Vetores e matrizes

### 1.12.1 Vetores

Vetores são estruturas de dados que armazenam um determinado número de elementos do mesmo tipo. Vetores podem ser considerados como conjuntos de variáveis, todas do mesmo tipo.

Um vetor possui um nome único para todo o conjunto de elementos que o compõe. Para acessar um elemento individualmente é utilizado um índice. Cada elemento de um vetor possui um índice.

Os elementos de um vetor são alocados na memória sequencialmente, com o primeiro elemento possuindo o menor endereço de memória.

Na linguagem C a declaração de vetores é realizada da seguinte forma.

```
tipo identificador[número de elementos]
```

A inicialização de vetores em C é realizada da seguinte forma.

```
tipo identificador[número de elementos] = {valor1, valor2, ..., ValorN}
```

Se na declaração de um vetor o número de elementos não for informado o compilador irá criar um vetor com o número de elementos presentes nos dados de inicialização.

### 1.12.2 Acessando elementos de um vetor

Os elementos de um vetor podem ser acessados para leitura e escrita através de seu índice. Isso é feito colocando entre colchetes o índice do elemento após o nome do vetor.

O trecho de código 1.16 apresenta um exemplo de vetor.

Código 1.16: Exemplo de vetor

```
int vetor[5],fatores[5]={5, 6, 7, 8, 9}, cont;  
printf("Digite 5 números\n");  
for(cont=0; cont<5; cont++)  
{  
    scanf("%d",&vetor[cont]);  
    vetor[cont]=vetor[cont]*fatores[cont];  
}  
printf("Os valores multiplicados pelos fatores são:\n");  
for(cont=0; cont<5; cont++)  
{  
    printf("%d ",vetor[cont]);
```

```
}
```

É importante salientar que para um vetor de  $n$  elementos os índices vão de 0 a  $n-1$ .

### 1.12.3 Matrizes

Na linguagem C as matrizes são interpretadas como vetores de duas dimensões. Na realidade é possível criar vetores com muitas dimensões, a linguagem C não limita a duas.

O formato da declaração de uma matriz de duas dimensões na linguagem C é a seguinte.

```
tipo identificador[num_linhas][num_colunas]
```

O trecho de código 1.17 apresenta um exemplo de matriz.

Código 1.17: Exemplo de matriz

```
int i, j, matriz[3][4]={{11,12,13,14},{21,22,23,24},{31,32,33,34}};
printf("Conteúdo da matriz\n");
for(i=0; i<3; i++)
{
    for(j=0; j<4; j++)
    {
        printf("%d ",matriz[i][j]);
    }
    printf("\n");
}
```

## 1.13 Ponteiros

Ponteiros são tipos especiais de variáveis que servem para armazenar um endereço de memória.

Os ponteiros são importantes na programação em C porque algumas tarefas são executadas mais facilmente com ponteiros e outras, como alocação de dinâmica memória, não podem ser executadas sem o uso de ponteiros.

Como já foi mencionado, cada variável é um local de memória que tem seu endereço bem definido. É possível acessar o endereço de memória de uma determinada variável utilizando o operador "&".

Para que se possa utilizar um ponteiro ele deve ser primeiro definido, assim como se definem variáveis. A seguir esta a forma básica de definir um ponteiro.

```
tipo *identificador
```

Um ponteiro assim como uma variável normal necessita de um identificador, que dá a ele um nome. Um ponteiro também necessita de um tipo. Os tipos utilizados nos ponteiros são os mesmos tipo de dados utilizados na declaração de variáveis. O tipo de dado do ponteiro indica o tipo de dado armazenado no endereço de memória apontado por este ponteiro.

Vale destacar que na declaração de um ponteiro utiliza-se um asterisco (\*) antes do identificador, de forma a diferenciá-lo de uma variável comum.

A utilização comum para os ponteiros é o seguinte. Inicialmente é definido um ponteiro, em seguida um endereço é atribuído a este ponteiro. Para acessar o conteúdo de memória apontado pelo ponteiro utiliza-se o operador asterisco (\*) junto ao nome do ponteiro.

O código 1.18 apresenta um exemplo de programa que utiliza ponteiros.

Código 1.18: Exemplo de ponteiros

```
#include <stdio.h>

int main(void)
{
    int variavel=1234;
    int *ponteiro=null;
    ponteiro=&variavel; //atribui o endereço de variavel a ponteiro
    printf("Conteúdo da variável %d\n",variavel);
    printf("Endereço da variável 0x%lx\n",(unsigned long)&variavel);
    printf("Valor do ponteiro %p\n",ponteiro);
    printf("Valor apontado pelo ponteiro %d\n",*ponteiro);
    return 0;
}
```

Não é interessante permitir deixar os ponteiros sem inicialização. Quando não se possui o endereço do ponteiro no momento de criação do mesmo pode-se atribuir o valor “NULL” a ele. Neste caso diz-se que o ponteiro é um ponteiro nulo.

## 1.14 Cadeias de caracteres

Cadeias de caracteres na linguagem C são tratados como vetores do tipo “char”. Uma cadeia de caracteres é um vetor onde cada elemento deste vetor é ocupado por um caractere ou letra. Para finalizar uma cadeia de caracteres a linguagem C adiciona um “\0” após o último caractere. Assim a palavra teste é armazenada na memória pela linguagem C da seguinte forma.

T	e	s	t	e	\0
---	---	---	---	---	----

A declaração de cadeias de caracteres em C segue a mesma lógica da declaração de vetores, com a exceção da inicialização. para inicializar uma cadeia de caracteres pode-se utilizar uma sequencia de caracteres entre aspas duplas (“ ”);

O trecho de código 1.19 apresenta exemplos de cadeias de caracteres.

Código 1.19: Exemplo de cadeias de caracteres

```
char palavra[6]="teste";
char frase[]="Utilização de cadeias de caracteres!";
printf("%s\n",palavra);
printf("%s\n",frase);
```

A linguagem C fornece um conjunto de funções para a manipulação de cadeias de caracteres. As principais funções para este fim são listadas a seguir, estas funções estão na biblioteca “string.h”.

- **strcpy(c1, c2)**: copia o conteúdo da cadeia de caracteres c2 na cadeia de caracteres c1.
- **strcat(c1, c2)**: concatena (anexa) o conteúdo da cadeia de caracteres c2 no final da cadeia de caracteres c1.
- **strlen(c1)**: retorna o tamanho da cadeia de caracteres c1.
- **strcmp(c1, c2)**: retorna zero se as duas cadeias de caracteres forem iguais. Se as duas cadeias de caracteres forem diferentes a função encontra o primeiro caractere diferente e retorna um valor maior que zero se este caractere tiver um valor numérico maior em C1 do que em C2, caso contrário retorna um valor menor que zero.
- **strchr(c, letra)**: retorna um ponteiro para a posição da primeira aparição da “letra” na cadeia de caracteres.
- **strstr(c1, c2)**: retorna um ponteiro para a posição da primeira aparição da cadeia de caracteres c2 na cadeia de caracteres c1.

A linguagem C também fornece um conjunto de funções para realizar a conversão de cadeias de caracteres em números e números em cadeias de caracteres. Estas funções pertencem a biblioteca “stdlib.h”. A seguir são listadas as principais.

- **int atoi(const char \*str)**; Converte uma cadeia de caracteres em um número inteiro (int).
- **long int atol(const char \*str)**; Converte uma cadeia de caracteres em um número inteiro (long int).
- **double atof(const char \*str)**; Converte uma cadeia de caracteres em um número de ponto flutuante (double).

Outra função interessante para a manipulação de cadeias de caracteres, presente na biblioteca “stdio.h”, é a função “sprintf”. Esta função é muito parecida com a função “printf”, porém sua saída é armazenada em uma cadeia de caracteres. Veja a descrição a seguir.

- **int sprintf(char \*str, const char \* format, ... )**; Recebe como parâmetros um ponteiro para uma cadeia de caracteres onde o resultado deve ser armazenado e uma cadeia de caracteres com o formato a ser utilizado. Opera de forma semelhante a função “printf” na formatação da cadeia de caracteres de saída.

## 1.15 Estruturas

Estruturas em linguagem C são tipos de dados definidos pelo programador que podem combinar outros tipos de dados. A forma de declarar uma estrutura em C é apresentada a seguir.

```
struct identificador
{
    tipo identificador_membro1;
    tipo identificador_membro2;
    tipo identificador_membro3;
} variaveisEstrutura;
```

Para declarar uma variável com o tipo de dado definido em uma estrutura a sintaxe é a seguinte.

```
struct nome_da_estrutura identificador;
```

Para acessar um membro de uma estrutura basta utilizar o operador “.”. Veja a seguir o formado para acesso a membros de estruturas.

```
nome_da_variavel_da_estrutura.nome_do_membro;
```

O código 1.20 apresenta um programa que utiliza estruturas.

Código 1.20: Exemplo de estruturas

```
#include <stdio.h>

int main(void)
{
    struct motor
    {
        float rpm;
        float corrente;
        float temperatura;
    } motor1;

    struct motor motor2;

    motor1.corrente=10.0;
    motor1.rpm=1800;
    motor1.temperatura=45.5;
    motor2.corrente=7.0;
    motor2.rpm=2450;
    motor2.temperatura=66.7;
    printf("Dados do motor 1\n");
    printf("Corrente %.2f\n",motor1.corrente);
    printf("RPM %.2f\n",motor1.rpm);
    printf("Temperatura %.2f\n",motor1.temperatura);
    printf("\nDados do motor 2\n");
    printf("Corrente %.2f\n",motor2.corrente);
    printf("RPM %.2f\n",motor2.rpm);
    printf("Temperatura %.2f\n",motor2.temperatura);
    struct motor *ponteiro_motor=NULL;
    ponteiro_motor=&motor1;
```

```
ponteiro_motor->corrente=12.8;
printf("\nDados do motor apontado pelo ponteiro\n");
printf("Corrente %.2f\n",ponteiro_motor->corrente);
printf("RPM %.2f\n",ponteiro_motor->rpm);
printf("Temperatura %.2f\n",ponteiro_motor->temperatura);
return 0;
}
```

Também é possível criar ponteiros para variáveis de estruturas. Estes ponteiros são úteis quando se deseja passar como parâmetro para uma função o endereço de uma estrutura.

A sintaxe para acessar membros de uma estrutura através de um ponteiro para esta estrutura é diferente da sintaxe utilizada para acessar membro de estrutura diretamente. Neste caso o ponto “.” é substituído pelo sinal “->”. Veja esta sintaxe a seguir.

```
ponteiro_para_variavel_da_estrutura->nome_do_membro;
```

É importante salientar que uma estrutura pode ser passada para uma função como parâmetro assim como uma variável comum.

## 1.16 Uniões

As uniões são tipos especiais de estruturas que permitem armazenar diferentes tipos de dados no mesmo local de memória. É possível definir uma união com muitos membros, mas **apenas um membro pode receber um valor por vez**. Uniões permitem utilizar o mesmo local de memória para múltiplos propósitos.

A definição de uma união é parecida com a definição de uma estrutura, basta substituir a palavra “struct” pela palavra union. A instrução union define um novo tipo de dados composto por mais de um membro. A sintaxe de declaração é a seguinte.

```
union identificador
{
    tipo identificador_membro1;
    tipo identificador_membro2;
    tipo identificador_membro3;
} variaveisUniao;
```

As definições de variáveis e as formas de acesso aos membros das uniões seguem a mesma sintaxe das estruturas.

O código 1.21 apresenta um programa que utiliza uniões.

Código 1.21: Exemplo de união

```
union exemploUniao
{
    int x;
    float y;
} ex_u;
```

```
ex_u.x=100;

printf("Valor de X = %d\n",ex_u.x);

ex_u.y=12.3;

printf("Valor de y = %.2f\n",ex_u.y);
// o valor de x foi perdido
printf("Valor inválido de x = %d\n",ex_u.x);
```

## 1.17 Definição de tipos

A linguagem C permite que se atribua um novo nome a um tipo de dados, para isso é utilizado o comando “typedef”. A sintaxe é a seguinte.

```
typedef tipo novoNome;
```

O typedef pode ser utilizados com tipos de dados normais, estruturas uniões etc. O código 1.22 apresenta um programa que utiliza definição de tipos.

Código 1.22: Exemplo de definição de tipos

```
#include <stdio.h>

int main(void)
{
    typedef int inteiro;
    inteiro x=4, y=6;
    printf("A soma dos inteiros é: %d\n",x+y);

    typedef struct structDado
    {
        int a;
        float b;
        inteiro c;
    } meuDado;

    meuDado dado;
    dado.a=1;
    dado.b=2;
    dado.c=3;

    printf("Os dados são: %d %.2f %d\n",dado.a, dado.b, dado.c);
    return 0;
}
```

## 1.18 Enumerações

Enumeração na linguagem C um tipo de dados definido pelo usuário e é utilizado principalmente para atribuir nomes a constantes inteiras, estes nomes tornam os programas mais fácil de ler e manter.

Uma enumeração é um conjunto de constantes inteiras que define todos os valores que uma variável deste tipo pode assumir.

Para criar uma enumeração em C utiliza-se o comando “enum”, cuja sintaxe é a seguinte.

```
enum identificador {lista_de_enumeração} lista_de_variaveis;
```

O código 1.23 apresenta um programa que utiliza enumerações.

Código 1.23: definição enumerações

```
#include <stdio.h>

int main(void)
{
    enum diasDaSemana {segunda, terca, quarta, quinta, sexta, sabado, domingo};

    enum diasDaSemana dia;

    dia=quarta;
    // imprime o número da quarta na enumeração
    printf("O dia é %d\n", dia);
    return 0;
}
```

## 1.19 Entradas e saídas

Existem diversas funções que proporcionam acesso a entrada e saída na linguagem C. Não é objetivo deste material explorar todas estas funções. A seguir são apresentadas resumidamente algumas delas.

- **getchar():** lê um caractere e espera o retorno de carro.
- **getche():** lê um caractere com eco na tela.
- **getch():** lê um caractere sem eco no tela.
- **putchar():** escreve um caractere na tela.
- **gets:** lê uma cadeia de caracteres do teclado.
- **puts:** escreve uma cadeia de caracteres na tela.

Além destas existem outras duas funções de entrada e saída que são as principais funções utilizadas para a entrada de dados pelo teclado e a saída de dados pela tela.

### 1.19.1 A função scanf

A função “scanf” é responsável pela leitura de dados da entrada padrão, no nosso caso, o teclado. Os dados são lidos e armazenados de acordo com os parâmetros informados. A sintaxe de utilização na função “scanf” é a seguinte.

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

A função “scanf” fornece um número enorme de especificadores de formato para a leitura de dados, a tabela 1.11 apresenta os principais.

Tabela 1.11: Especificadores de formato do scanf

Código	Descrição
%c	Lê um caractere
%d	Lê um inteiro decimal
%i	Lê um inteiro decimal
%e	Lê um ponto flutuante
%f	Lê um ponto flutuante
%g	Lê um ponto flutuante
%o	Lê um inteiro em formato octal
%s	Lê uma cadeia de caracteres
%x	Lê um número hexadecimal
%p	Lê um ponteiro
%u	Lê um inteiro sem sinal

É importante salientar que a função “scanf” espera receber o endereço da variável que vai receber o dado lido do teclado, assim o nome da variável deve ser precedido do operador “&”.

### 1.19.2 A função printf

A função “printf” é responsável pela gravação de dados na saída padrão, no nosso caso, a tela do computador. Os dados são gravados de acordo com os parâmetros informados. A sintaxe de utilização na função “printf” é a seguinte.

```
printf("texto e formatos", variavel1, variavel2, ...);
```

A função “printf” fornece um número enorme de especificadores de formato para a escrita de dados, a tabela 1.12 apresenta os principais.

O código 1.24 apresenta um programa que utiliza as funções “scanf” e “printf”.

Tabela 1.12: Especificadores de formato do printf

Código	Descrição
%c	Escreve um caractere
%d	Escreve um inteiro decimal
%i	Escreve um inteiro decimal
%e	Escreve um ponto flutuante em Notação científica (com e)
%E	Escreve um ponto flutuante em Notação científica (com E)
%f	Escreve um ponto flutuante
%g	Usa %e ou %f, o que for mais curto
%G	Usa %E ou %F, o que for mais curto
%o	Escreve um inteiro em formato octal
%s	Escreve uma cadeia de caracteres
%u	Escreve um inteiro sem sinal
%x	Escreve um número em hexadecimal (letras minúsculas)
%X	Escreve um número em hexadecimal (letras maiúsculas)
%p	Escreve um ponteiro

Código 1.24: Exemplo das funções scanf e printf

```
#include <stdio.h>

int main(void)
{
    char str [80];
    int i;

    printf ("Digite seu nome: ");
    scanf ("%79s",str);
    printf ("Digite sua idade: ");
    scanf ("%d",&i);
    printf ("Sr. %s, %d anos de idade.\n",str,i);
    printf ("Digite um número em hexadecimal: ");
    scanf ("%x",&i);
    printf ("Você digitou %#x (%d).\n",i,i);
    return 0;
}
```

## 1.20 Manipulação de arquivos

Há muitas situações nas quais precisamos ler ou gravar dados em arquivos a partir de nossos programas C, pois qualquer informação que não é armazenada em arquivos é perdida quando o programa é finalizado. A linguagem C fornece um conjunto de ferramentas para executar esta tarefa.

### 1.20.1 Abertura de Arquivos

Para que possamos ter acesso a um arquivo no computador é necessário inicialmente abrir ou criar este arquivo., isso é feito com a função `fopen()`. Esta função é a responsável por conectar um ponteiro de arquivo ao arquivo que queremos manipular.

A sintaxe da função “`fopen`” é a seguinte.

```
FILE *ponteiro_arquivo=fopen(const char *nome, const char *modo);
```

O parâmetro “`nome`” é uma cadeia de caracteres que indica o nome do arquivo a ser aberto ou criado. O parâmetro “`modo`” indica se o arquivo será aberto para leitura, escrita ou ambos, veja na tabela 1.13 os modos possíveis.

Tabela 1.13: Modos de abertura de arquivos

Modo	Descrição
“r”	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
“w”	Abre um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
“a”	Abre um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo (“append”), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
“rb”	Abre um arquivo binário para leitura. Igual ao modo “r” anterior, só que o arquivo é binário
“wb”	Cria um arquivo binário para escrita, como no modo “w” anterior, só que o arquivo é binário.
“ab”	Acrescenta dados binários no fim do arquivo, como no modo “a” anterior, só que o arquivo é binário.
“r+”	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
“w+”	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.
“a+”	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
“r+b”	Abre um arquivo binário para leitura e escrita. O mesmo que “r+” acima, só que o arquivo é binário.
“w+b”	Cria um arquivo binário para leitura e escrita. O mesmo que “w+” acima, só que o arquivo é binário.
“a+b”	Acrescenta dados ou cria um arquivo binário para leitura e escrita. O mesmo que “a+” acima, só que o arquivo é binário

A seguir temos um exemplo onde é criado um arquivo chamado “`teste.txt`” e em seu interior é gravada a frase “`escrevendo em um arquivo`”.

Este exemplo testa se a variável `arquivo` é igual a “`NULL`”, se for significa que houve algum erro em abrir o arquivo e o programa é finalizado apresentando uma mensagem de erro.

Todos os arquivos que são abertos devem ser fechados utilizando o comando “`fclose`” e passando como parâmetro o nome da variável que contém o ponteiro do arquivo.

O código 1.25 apresenta o programa do exemplo.

Código 1.25: Exemplo de abertura de arquivo

```
#include <stdio.h>

int main(void)
{
    FILE *arquivo = fopen("teste.txt", "w"); // cria ou abre o arquivo
    if(arquivo == NULL) // testa se o arquivo foi aberto com sucesso
    {
        printf("\n\nImpossivel abrir o arquivo!\n\n");
        return 0;
    }
    fprintf(arquivo,"escrevendo em um arquivo\n");
    fclose(arquivo);
    printf("Concluido!\n\n");
    return 0;
}
```

## 1.20.2 Gravando Informações em um Arquivo

Existem várias formas de gravar informações em um arquivo, a mais fácil delas é utilizar o comando “fprintf”, que é muito similar ao comando “printf”, com a única diferença de receber a variável que contém o ponteiro do arquivo como parâmetro.

O código 1.26 apresenta um exemplo de programa que grava informações em um arquivo.

Código 1.26: Exemplo de gravação em arquivo

```
#include <stdio.h>

int main(void)
{
    int cont;
    FILE *arq = fopen("teste.txt", "w"); // cria ou abre o arquivo

    if(arq == NULL) // testa se o arquivo foi aberto com sucesso
    {
        printf("\n\nImpossivel abrir o arquivo!\n\n");
        return 0;
    }
    cont=0;
    fprintf(arq,"Agora cont vale %d\n",cont);
    cont++;
    fprintf(arq,"Agora cont vale %d\n",cont);
    cont++;
    fprintf(arq,"Agora cont vale %d\n",cont);
    cont++;
    fprintf(arq,"Agora cont vale %d\n",cont);
}
```

```
fclose(arq);
printf("Concluido!\n\n");
return 0;
}
```

### 1.20.3 Lendo Informações de um Arquivo

Assim como existem comando para gravar nos arquivos existem também comandos para ler os arquivos. O principal deles é o comando “fscanf”. Este comando é muito similar ao comado “scanf”, com a única diferença de receber a variável que contem o ponteiro do arquivo como parâmetro.

É importante observar que o comando “fscanf” lê apenas uma palavra de cada vez, se desejarmos ler uma frase completa devemos usar o comando “fgets”. Este comando recebe três parâmetros, o primeiro é a variável onde a frase deve ser gravada, o segundo é o número máximo de letras que podem ser lidas e o terceiro é a variável que contem o ponteiro do arquivo.

O código 1.27 apresenta um exemplo de programa que lê informações de um arquivo.

Código 1.27: Exemplo de leitura de arquivo

```
#include <stdio.h>

int main(void)
{
    int x;
    FILE *arq = fopen("numero.txt", "r");
    if(arq == NULL) // testa se o arquivo foi aberto com sucesso
    {
        printf("\n\nImpossivel abrir o arquivo!\n\n");
        return 0;
    }
    fscanf(arq,"%d",&x);
    fclose(arq);
    printf("Valor lido = %d\n\n",x);
    return 0;
}
```

O código 1.28 apresenta um exemplo de programa que lê informações de um arquivo.

Código 1.28: Exemplo de leitura de arquivo com fgets

```
#include <stdio.h>

int main(void)
{
    char x[100];
    FILE *arq = fopen("frase.txt", "r");
    if(arq == NULL) // testa se o arquivo foi aberto com sucesso
    {
        printf("\n\nImpossivel abrir o arquivo!\n\n");
    }
}
```

```
    return 0;
}
fgets(x, 100, arq);
fclose(arq);
printf("\nfrase lida = %s\n\n", x);
return 0;
}
```

Existem ainda muitos outros comandos para manipular arquivos, porém estes que foram vistos são suficientes para nossos estudos. Contudo se for necessário construir programas que trabalhem com arquivos mais complexos é necessário utilizar outros comandos mais poderosos. Uma pesquisa na internet fornece mais informações nestes casos. A seguir temos mais dois exemplos, onde um vetor de 5 números reais é gravado em um arquivo e em seguida lido novamente.

O código 1.29 apresenta um exemplo de programa que grava um vetor em um arquivo.

Código 1.29: Exemplo de gravação de vetor em arquivo

```
#include <stdio.h>

int main(void)
{
    float vet[5]={1000.0,2000.0,3000.0,4000.0,5000.0};
    int cont;
    FILE *arq = fopen("vet.txt", "w+");
    if(arq == NULL) // testa se o arquivo foi aberto com sucesso
    {
        printf("\n\nImpossivel abrir o arquivo!\n\n");
        return 0;
    }
    for(cont=0; cont<5; cont++)
    {
        fprintf(arq, "%f\n", vet[cont]);
    }
    fclose(arq);
    printf("concluido\n\n");
    return 0;
}
```

O código 1.30 apresenta um exemplo de programa que lê um vetor em um arquivo.

Código 1.30: Exemplo de leitura de vetor em arquivo

```
#include <stdio.h>

int main(void)
{
    float vet[5];
    int cont;
    FILE *arq = fopen("vet.txt", "r");
    if(arq == NULL) // testa se o arquivo foi aberto com sucesso
```

```
{
    printf("\n\nImpossível abrir o arquivo!\n\n");
    return 0;
}
for(cont=0; cont<5; cont++)
{
    fscanf(arq,"%f",&vet[cont]);
    printf("%f\n",vet[cont]);
}
fclose(arq);
printf("\nconcluído\n\n");
return 0;
}
```

É importante salientar que para escrever nos arquivos usamos na função “fopen” o parâmetro “w+”, e para ler usamos o parâmetro “r”.

## 1.21 Pré processador

O processo de compilação de um programa escrito em linguagem C tem várias etapas, uma delas é chamada de pré processamento. O pré processador é responsável por exemplo pela inclusão de bibliotecas (#include) e pelas definições (#define), vistas anteriormente.

O pré processador tem várias outras tarefas, e possui vários outros operadores que podem ser utilizados na confecção dos programas. A relação a seguir apresenta os principais.

- **#define:** define constantes.
- **#include:** inclui uma biblioteca de funções.
- **#undef** desfaz uma definição.
- **#ifdef:** retorna verdadeiro se uma determinada macro estiver definida.
- **#ifndef** retorna verdadeiro se uma determinada macro não estiver definida.
- **#if:** testa se uma condição é verdadeira em tempo de compilação.
- **#else:** é o complemento do #if.
- **#elif:** #else e #if em uma única expressão.
- **#error:** apresenta uma mensagem de erro na hora da compilação.
- **#pragma:** para comandos para o compilador.

É importante salientar que todo o processamento das instruções do pré processador acontecem em tempo de compilação e não durante a execução do programa.

A linguagem C também possui um conjunto de macros que podem ser utilizadas durante o pré processamento. A seguir são listadas as principais.

- `__DATE__`: a data atual no formato de uma cadeia de caracteres.
- `__TIME__`: o horário atual no formato de uma cadeia de caracteres.
- `__FILE__`: o nome do arquivo no formato de uma cadeia de caracteres.
- `__LINE__`: o número da linha atual.
- `__STDC__`: definido como 1 se o compilador está compilando C ANSI padrão.

Não é objetivo deste material explicar o funcionamento e a utilização das diretivas do pré processador, apenas pretende-se apresentar uma breve introdução ao tema.

## 1.22 Conversão de tipos

A conversão de tipos na linguagem C é uma forma de converter um tipo de dado em outro. A sintaxe para a conversão de tipos de dados é a seguinte.

```
(novo_tipo) expressão;
```

O código 1.31 apresenta um exemplo de programa que utiliza a conversão de tipo.

Código 1.31: Exemplo de conversão de tipo

```
#include <stdio.h>

int main(void)
{
    int num=10, den=3;
    double resultado1, resultado2;
    resultado1=num / den;
    printf("resultado1=%f\n",resultado1);
    resultado2=(double)num / den; // força uma operação em ponto flutuante.
    printf("resultado2=%f\n",resultado2);
    return 0;
}
```

## 1.23 Recursividade

Recursividade é quando uma função chama a ela mesma. A linguagem C permite a utilização da recursividade no desenvolvimento de programas. É importante salientar que quando uma função utiliza recursividade o programador deve prever um ponto de saída nesta função.

O código 1.32 apresenta um exemplo de programa que utiliza recursividade.

Código 1.32: Exemplo de recursividade

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

```

unsigned long factorial(unsigned long i)
{
    if(i <= 1)
    {
        return 1;
    }

    return i * factorial(i - 1);
}

int main()
{
    int x;
    printf("Digite um número: ");
    scanf("%d",&x);
    printf("O fatorial de %d é %lu\n", x, factorial(x));
    return 0;
}

```

## 1.24 Argumentos de linha de comando

Quando um programa é iniciado pelo sistema operacional é possível passar para ele, através da linha de comando, um conjunto de argumentos. Estes argumentos podem então ser acessados pelo programa e utilizados em suas computações.

Na linguagem C estes argumentos são passados através de duas variáveis especiais chamadas “argc” e “\*argv[]”. estas variáveis são recebidas como argumentos da função “main”.

O código 1.33 apresenta um exemplo de programa que utiliza argumentos de linha de comando.

Código 1.33: Exemplo de argumentos de linha de comando

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    int cont;
    for(cont=0; cont<argc; cont++)
    {
        printf("Argumento %d = %s\n",cont,argv[cont]);
    }
    return 0;
}

```

## 1.25 Alocação de memória

Nem sempre é possível determinar de antemão quanta memória será necessária para a execução de um programa. Nestes casos é utilizada a alocação dinâmica de memória.

A linguagem C possui várias funções que ajudam nesta tarefa. A seguir são apresentadas as principais. estas funções estão na biblioteca "stdlib.h".

- **void\* calloc (size\_t num, size\_t size);** Aloca um bloco de memória de "num" elementos, cada elemento com "size" bytes. Este espaço de memória é zerado.
- **void free (void\* ptr);** Um bloco de memória previamente alocado com "malloc", "calloc" ou "realloc" é desalocado.
- **void\* malloc (size\_t size);** Aloca um bloco de memória de tamanho "size" e retorna um ponteiro para seu início.
- **void\* realloc (void\* ptr, size\_t size);** Altera o tamanho de um bloco de memória apontado por "ptr" para o tamanho "size" e retorna um ponteiro para seu início.

O código 1.34 apresenta um exemplo de programa que utiliza alocação de memória.

Código 1.34: Exemplo de alocação de memória

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

int main ()
{
    char *pDados;
    pDados = (char*) calloc (100, sizeof(char));
    if (pDados==NULL)
    {
        printf("Impossível alocar a memória");
        return 0;
    }
    else
    {
        strcpy(pDados, "Informação armazenada na memória alocada!");
        printf("%s", pDados);
    }
    free (pDados);
    return 0;
}
```

# Aula 2 Ponteiros

## 2.1 Introdução

Na aula anterior, na revisão de linguagem C, os ponteiros foram abordados rapidamente. Devido a sua importância e complexidade esta segunda aula será toda dedicada ao estudo dos ponteiros.

## 2.2 O que são ponteiros

Em ciências da computação os ponteiros são definidos como um objeto que armazena o endereço de memória de outro objeto. Normalmente um ponteiro faz referência a um determinado local na memória para que se possa recuperar ou alterar os dados armazenados neste local.

Os ponteiros são utilizados porque melhoram o desempenho de funções repetitivas, como por exemplo o acesso a tabelas ou vetores.

## 2.3 Ponteiros na linguagem C

Os ponteiros tem grande importância na linguagem C. Na realidade, os ponteiros são um dos aspectos mais fortes e mais perigosos da linguagem C. Se por um lado eles facilitam vários tipos de operações, por outro lado um ponteiro não inicializado ou “desgovernado” pode derrubar todo o programa.

Na linguagem C pode-se dizer que um ponteiro é uma variável especial que armazena um endereço de memória. Este endereço é normalmente a posição de uma outra variável na memória.

### 2.3.1 Operadores utilizados com ponteiros

Na linguagem C existem dois operadores especiais que são utilizados com ponteiros: o “&” e o “\*”.

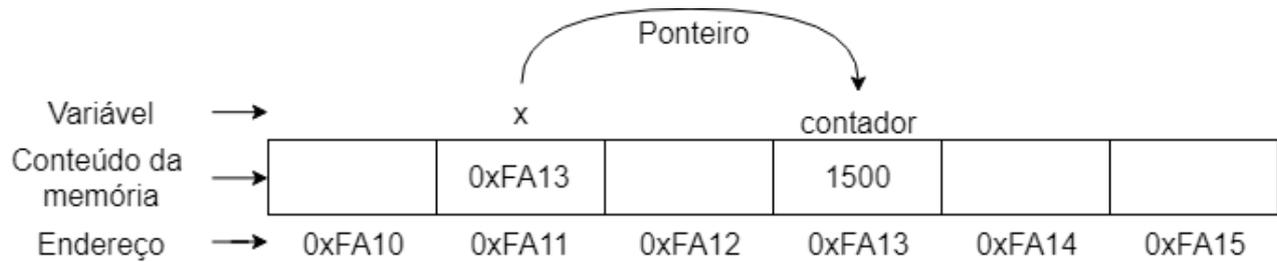
O operador “&” é um operador que retorna o endereço de seu operando. Veja um exemplo a seguir.

```
int *x;
int contador = 1500;
x = &contador;
```

Neste exemplo a variável “x” recebe o endereço de memória onde a variável “contador” está armazenada. Assim esta expressão deve ser lida como “x recebe o endereço de contador”

A figura 2.1 apresenta a configuração da memória após a execução do código do exemplo anterior.

Figura 2.1: Exemplo de operador “&amp;”



O operador “\*” por sua vez é utilizado para declarar variáveis do tipo ponteiros. A forma geral para a declaração de um ponteiro, como já visto anteriormente é.

```
tipo *identificador
```

O identificador é o nome do ponteiro e o tipo pode ser qualquer tipo válido na linguagem C.

Tecnicamente qualquer tipo de ponteiro pode apontar para qualquer lugar na memória, porém, para que as operações com ponteiros sejam possíveis o compilador necessita saber qual o tipo de dado envolvido.

O operador “\*” é utilizado para retornar o valor armazenado no endereço de memória apontado pelo seu operando.

No exemplo da figura 2.1 o ponteiro “\*x” aponta para o valor 1500 no endereço 0xFA13 da memória.

É importante salientar que uma variável do tipo ponteiro deve sempre apontar para o tipo correto, caso contrário as computações acontecerão incorretamente.

### 2.3.2 Ponteiros NULL

É uma boa prática atribuir o valor NULL aos ponteiros cujo valor definitivo ainda não foi atribuído. Normalmente isso é feito no momento da declaração do ponteiro. Um ponteiro que tem seu endereço definido como NULL é chamado de ponteiro nulo. Veja um exemplo a seguir.

```
int *pnt = NULL;
```

### 2.3.3 Atribuição de endereços aos ponteiros

Para que se possa utilizar um ponteiro é necessário atribuir um endereço a ele. Isso pode ser feito de duas formas. A primeira forma é atribuir o endereço de uma variável ao ponteiro, veja um exemplo a seguir.

```
int var;
int *pnt;
pnt=&var;
```

Neste caso o operador “&” é utilizado para atribuir o endereço de “var” ao ponteiro “pt”. Observe que neste caso o nome do ponteiro não é precedido do operador “\*”.

A segunda é atribuir o endereço armazenado em um ponteiro a outro ponteiro, veja um exemplo a seguir.

```
int var;
int *pt1, *pt2;
pt1=&var;
pt2=pt1;
```

Neste caso o endereço armazenado em “pt1” é armazenado também em “pt2”. Observe que o operador “\*” não é utilizado nesta atribuição, pois não se deseja acessar o valor apontado pelos ponteiros.

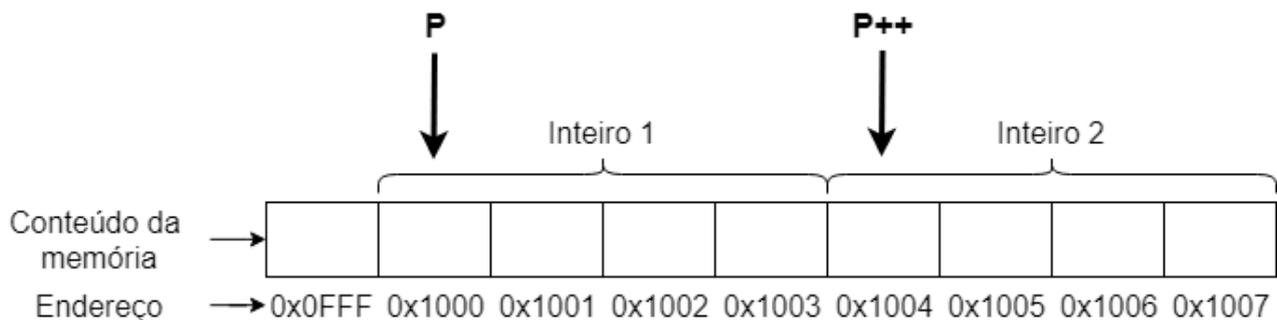
### 2.3.4 Aritmética de ponteiros

É possível realizar operações de soma ou subtração com os ponteiros, porém estas operações aritméticas realizadas com ponteiros são diferentes das operações aritméticas tradicionais.

Como exemplo imagine que se tenha um ponteiro “P” do tipo inteiro apontando para o endereço 0x1000. Assuma que no sistema em questão cada inteiro utilize 4 bytes (32 bits) na memória. Se realizarmos uma operação de incremento (somar 1) com este ponteiro o novo valor passa a ser 0X1004, ou seja o número de bytes ocupado por um inteiro é adicionado a ele.

A figura 2.2 ilustra esta situação.

Figura 2.2: Aritmética de ponteiro



É importante lembrar que toda a aritmética de ponteiros é realizada relativamente ao tipo de dados do ponteiro.

Além de incrementar ponteiros também é possível decrementá-los, ou ainda, é possível somar e subtrair números inteiros de ponteiros. Isso tudo levando sempre em consideração o tamanho do tipo de dados do ponteiro.

### 2.3.5 Comparações de ponteiros

Certos algoritmos necessitam de comparações entre ponteiros e a linguagem C permite tais comparações. Suponha dois ponteiros “P1” e “P2” o seguinte trecho de código é um exemplo de comparação entre ponteiros.

```
if(P1<P2)
```

Observe que não se utiliza o operador “\*” junto aos ponteiros. Se o operador “\*” for utilizado estaríamos comparando os valores apontados pelos ponteiros e não os ponteiros.

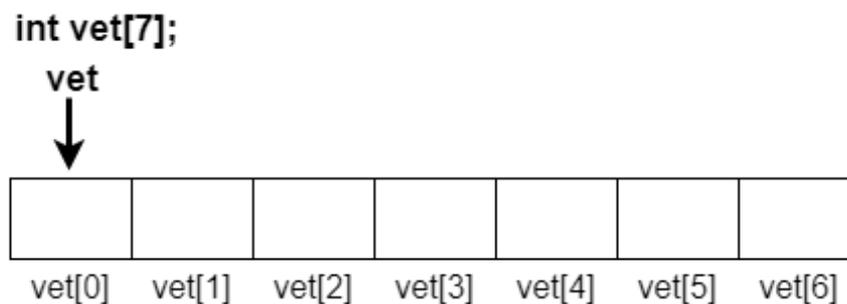
## 2.4 O uso de ponteiros com vetores e matrizes

Na linguagem C os vetores e os ponteiros estão intimamente relacionados. O nome de um vetor ou de uma matriz já é um ponteiro para o próprio vetor ou matriz.

A principal diferença é que os vetores e as matrizes, diferentemente dos ponteiros, já alocam o espaço necessário para os dados na memória.

A figura 2.3 ilustra a relação entre vetores e ponteiros.

Figura 2.3: Relação entre vetores e ponteiros



Na figura 2.3 é possível observar um vetor de 7 números inteiros declarado como “vet[7]”. Cada posição do vetor pode ser acessada através de seu índice, veja o exemplo a seguir.

```
x = vet[3];
```

Onde a variável “x” recebe o quarto elemento do vetor;

Também é possível acessar os elementos do vetor através do ponteiro “vet” que aponta para o primeiro elemento do vetor. Veja o exemplo a seguir.

```
x = *(vet+3);
```

Assim como no exemplo anterior a variável “x” recebe o quarto elemento do vetor, só que desta vez o operador “\*” foi utilizado junto ao ponteiro “vet” para acessar este elemento.

É possível armazenar o endereço de elementos de vetores em ponteiros, veja o exemplo a seguir.

```
int y[10];  
int *pnt;  
pnt = y;
```

Quando se trabalha com matrizes as coisas ficam um pouco diferentes. A linguagem C interpreta uma matriz como um vetor de vetores, assim não se pode utilizar o nome da matriz como um ponteiro diretamente. O que se costuma fazer é utilizar cada uma das linhas como um ponteiro. Veja o exemplo a seguir.

```
int *pnt;  
int mat[3][3] = {{11, 12, 13}, {21, 22, 23}, {31, 32, 33}};  
pnt = mat[0]+3;
```

Neste caso “mat[0]” é um ponteiro que aponta para o primeiro elemento da primeira linha da matriz. Como todas as linhas da matriz estão posicionadas na memória em sequência, pode-se dizer que ele aponta para o início da matriz. Assim a atribuição “pnt = mat[0]+3;” atribui o endereço do quarto elemento na memória, ou seja, o primeiro elemento da segunda linha, ao ponteiro “pnt”. Neste caso o ponteiro “\*pnt” aponta para o número 21.

## 2.5 Vetores e matrizes de ponteiros

É possível criar vetores e matrizes de ponteiros, de forma a armazenar conjuntos de ponteiros. Veja o exemplo a seguir.

```
int *vetPtr[5];
```

Este código cria um vetor de 5 ponteiros do tipo inteiro.

Para atribuir valores aos ponteiros no vetor de ponteiros faz-se como no exemplo a seguir.

```
vetPtr[2] = &var;
```

Neste código o endereço da variável “var” é armazenado no terceiro ponteiro do vetor de ponteiros.

Para acessar o valor da memória apontada por um ponteiro do vetor de ponteiros faz-se como no exemplo a seguir.

```
z = *vetPtr[2];
```

Neste código a variável “z” recebe o valor apontado pelo terceiro ponteiro do vetor de ponteiros.

## 2.6 Ponteiros para ponteiros

A linguagem C permite que se defina ponteiros que apontam para outros ponteiros.

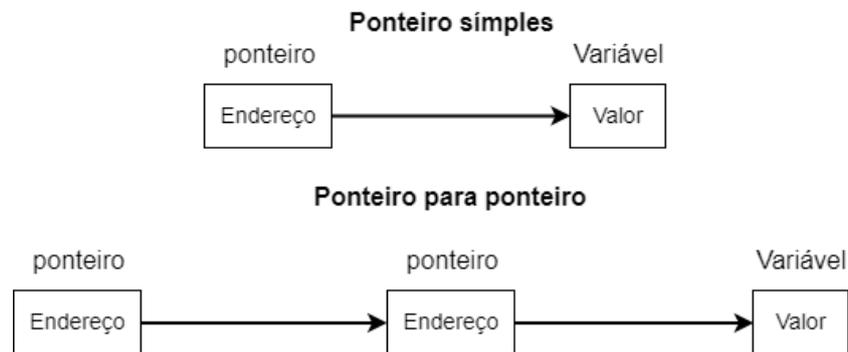
A figura 2.4 ilustra o conceito de ponteiros para ponteiros.

O exemplo a seguir mostra como são utilizados os ponteiros para ponteiros.

```
int a, *p, **pp;  
a = 5;  
p = &a;  
pp = &p;
```

Neste exemplo um ponteiro para ponteiro é declarado com o operador “\*\*”. Este ponteiro para ponteiro é chamado de “pp” e a ele é atribuído o endereço de um outro ponteiro chamado “p”;

Figura 2.4: Ponteiros para ponteiros



## 2.7 O uso de ponteiros com registros

A linguagem C prevê a utilização de ponteiros com outros tipos de estruturas além das variáveis. Um exemplo é a utilização de ponteiros com registros, ou estruturas.

É possível criar ponteiros para estruturas de forma semelhante a criação de ponteiros para variáveis, a principal diferença é que o operador “.” é substituído pelo operador “->” no acesso aos membros da estrutura.

O código 2.1 apresenta um exemplo de programa que utiliza ponteiros para registros ou estruturas.

Código 2.1: Exemplo de ponteiros para registros

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

struct atrAluno
{
    char nome[50];
    char curso[50];
    int idade;
};

int main()
{
    struct atrAluno aluno = {"Pedro", "Geografia", 15};
    struct atrAluno *ptrAluno;
    ptrAluno = &aluno;

    printf("Nome do aluno: %s\n", ptrAluno->nome);
    printf("Curso: %s\n", ptrAluno->curso);
    printf("Idade: %d\n", ptrAluno->idade);

    strcpy(ptrAluno->nome, "José");
    printf("\nNovo nome: %s\n", ptrAluno->nome);
    return 0;
}
```

## 2.8 Exemplo de utilização de ponteiros

O código 2.2 apresenta um exemplo que ilustra vários dos conceitos abordados nesta aula.

Código 2.2: Exemplo de utilização de ponteiros

```
#include <stdio.h>

int main(void)
{
    int *x=NULL;
    int contador = 1500;
    x = &contador;
    printf("Valor de contador = %d\n",contador);
    printf("Endereço de contador = %p\n",&contador);
    printf("Endereço armazenado em x = %p\n",x);
    printf("Valor apontado por x = %d\n\n",*x);

    int *pt=NULL;
    pt = x;
    printf("Valor apontado por pt = %d\n\n",*pt);

    int vet[5]={1,2,3,4,5};
    printf("Valor de vet[3] = %d\n",vet[3]);
    printf("Valor apontado por *(vet+3) = %d\n\n",*(vet+3));

    int *pnt=NULL;
    pnt=vet;
    printf("Valor apontado por *(pnt+3) = %d\n\n",*(pnt+3));

    pnt++;
    printf("Valor apontado por pnt++ = %d\n\n",*pnt);

    int mat[3][3] = {{11,12,13},{21,22,23},{31,32,33}};
    pnt = mat[0]+3;
    printf("Valor apontado por pnt = %d\n\n",*pnt);

    int **pp=NULL;
    pp=&pnt;
    printf("Valor apontado por pp = %d\n\n",**pp);
    return 0;
}
```

# Aula 3 Abstração procedural

## 3.1 Introdução

A abstração procedural é o processo de dividir o programa em funções ou procedimentos de forma a torná-lo mais legível e eficiente. O processo de desenvolvimento dos programas também é facilitado quando conseguimos dividi-lo em pequenas partes concisas e desacopladas.

## 3.2 A importância da abstração procedural

A abstração procedural consiste basicamente em dar um nome a um trecho de código que faz algo e depois utilizar este código, para executar a função programada, através de seu nome.

O uso de procedimentos ou funções costuma deixar os programas mais curtos e mais fáceis de entender. A abstração procedural facilita a leitura do seu código, especialmente se os procedimentos tiverem nomes correspondentes às tarefas que realizam.

Uma maior legibilidade de código torna mais fácil entender e descobrir como modificar e manter o código. Separar as funcionalidades de um programa em pequenas partes também cria muitas oportunidades para compartilhar e otimizar procedimentos individuais. Compartilhar esses procedimentos também centraliza os locais que precisam de modificação, no caso de uma alteração no código ser necessária.

Conforme as necessidades do seu código mudam, também será mais fácil reorganizar e reconfigurar o que seu código faz se as partes lógicas deste código estiverem separadas em procedimentos coesos e desacoplados.

## 3.3 Como fazer a abstração procedural

Para separar seu código em funções ou procedimentos algumas considerações devem ser feitas.

É importante que o programador entenda quando, por que e em que circunstâncias seu procedimento deve ser usado, ou seja, o que ele faz. Um procedimento bem escrito permite ser utilizado sem que se saiba como ele funciona internamente.

Cada procedimento implementado em seu programa deve ter um objetivo bem definido. Você deve poder descrever sucintamente o que cada procedimento faz em seu programa. Se você não puder, seus procedimentos estão muito grandes e deve ser dividido em unidades menores conceitualmente separáveis, ou ainda, seu procedimento está muito curto e você deve combiná-lo com outros de forma que ele execute uma tarefa completa.

É interessante que um procedimento não ultrapasse o tamanho de uma página (ou uma tela). Geralmente, um procedimento tem apenas algumas linhas. Se seus procedimentos são mais longos, você deve tentar descobrir como dividi-los em etapas menores e separáveis. Procedimentos muito grandes são difíceis de se manter na cabeça e assim fica complicado tentar descobrir o que está realmente sendo feito por este código.

A regra básica é que um procedimento deve ser conciso o suficiente para executar completamente a função para a qual foi projetado e desacoplado o suficiente para poder ser portado e compreendido sem dificuldades.

## 3.4 Quando utilizar a abstração procedural

Sempre que o mesmo código aparecer em dois ou mais lugares é um indicativo de que um procedimento pode ser escrito para desempenhar esta função. É útil dar um nome a este trecho de código e encapsulá-lo ou abstraí-lo para reutilização. Mesmo se houver pequenas diferenças no código, é possível abstrair um procedimento comum, fornecendo as informações distintas como argumentos.

Compartilhar código redundante reduz o tamanho seu programa, facilita a leitura, a compreensão, a modificação e a manutenção. Também ajuda a separar trechos de código onde cada comportamento é realizado. Esse trecho pode ser entendido, modificado e depurado uma vez, e não cada vez que aparece no programa.

## 3.5 A abstração procedural na linguagem C

A linguagem C permite a implementação de funções ou procedimentos e assim permite a abstração procedural.

Na linguagem C temos dois tipos de funções. As funções disponibilizadas pelas bibliotecas padrão da linguagem e as funções definidas pelo usuário.

### 3.5.1 Funções padrão do C

São as funções disponibilizadas pelas bibliotecas distribuídas junto com o compilador C. Para utilizar estas funções é necessário incluir suas bibliotecas através da diretiva “#include”.

A função “printf” disponibilizada na biblioteca “stdio.h” é um exemplo de função padrão da linguagem C.

Para que se possa utilizar a função “printf” em nossos programas é necessário que a biblioteca “stdio.h” seja incluída ao programa com a seguinte diretiva.

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

### 3.5.2 Funções definidas pelo usuário

As funções definidas pelo usuário são funções escritas pelo próprio programador ou importadas de outros programadores mas que não fazem parte das bibliotecas padrão do C.

É através das funções definidas pelo usuário que se faz a abstração procedural na linguagem C.

O código 3.1 apresenta um exemplo de função definida pelo usuário.

Neste exemplo temos uma função chamada “media” que recebe como parâmetros dois números e retorna sua média.

Código 3.1: Função definida pelo usuário

```
#include <stdio.h>

float media(float a, float b)
{
    return((a+b)/2);
}

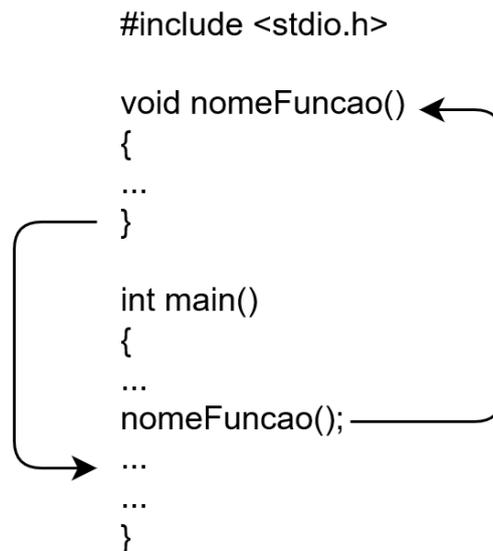
int main(void)
{
    float n1, n2;
    printf("Digite dois números\n");
    scanf("%f %f",&n1,&n2);
    printf("A média é %.3f",media(n1,n2));
    return 0;
}
```

A sessão a seguir apresenta em detalhes como funciona a implementação de funções na linguagem C.

## 3.6 Funções na linguagem C

Observando o exemplo do código 3.1 pode-se notar a estrutura de uma função na linguagem C. A figura 3.1 ilustra o mecanismo de funcionamento das funções na linguagem C.

Figura 3.1: Funções na linguagem C



### 3.6.1 Protótipo de função

Um protótipo de função é simplesmente a declaração de uma função onde se especifica o nome, os parâmetros e o tipo de retorno da função. O protótipo não contém o corpo da função. Um protótipo de função fornece informações ao compilador de como a função pode ser usada posteriormente no programa.

Sintaxe utilizada para implementar um protótipo de função é a seguinte.

```
tipoDeRetorno NomeFuncao(tipo argumento1, tipo argumento2, ...);
```

O código 3.2 apresenta o mesmo exemplo do código 3.1, só que agora utilizando protótipo para a função.

Código 3.2: Função com protótipo

```
#include <stdio.h>

float media(float a, float b);

int main(void)
{
    float n1, n2;
    printf("Digite dois números\n");
    scanf("%f %f",&n1,&n2);
    printf("A média é %.3f",media(n1,n2));
    return 0;
}

float media(float a, float b)
{
    return((a+b)/2);
}
```

### 3.6.2 Chamando uma função

Quando chamamos uma função o controle do programa é transferido para esta função. a sintaxe de uma chamada de função em C é a seguinte

```
nomeFuncao(argumento1, argumento2, ...);
```

No exemplo do código 3.2, a chamada de função é feita pela função printf através da diretiva “media(n1,n2)”.

### 3.6.3 Definição de função

A definição de função contém todo o código para executar uma tarefa específica. A sintaxe da definição de função é a seguinte.

```

tipoDeRetorno NomeFuncao(tipo argumento1, tipo argumento2, ...)
{
    // corpo da função
}

```

Quando uma função é chamada, o controle do programa é transferido para a definição da função. E o compilador começa a executar os códigos dentro do corpo de uma função. Quando a função finaliza seu processamento o controle retorna a quem chamou a função.

### 3.6.4 Passando argumentos para uma função

A passagem de argumentos é uma forma de passar o conteúdo de variáveis para a função. No exemplo do código 3.2 as variáveis *n1* e *n2* são parâmetros transmitidos para a função *media* durante sua chamada.

Os parâmetros “a” e “b” recebem os parâmetros passados para a função. Esses parâmetros ou argumentos são chamados de parâmetros formais da função.

A figura 3.2 ilustra este mecanismo.

Figura 3.2: Passagem de parâmetros

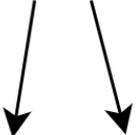
```

#include <stdio.h>

float media(float a, float b);

int main(void)
{
    ...
    resultado=media(n1,n2);
    ...
}

```



```

float media(float a, float b)
{
    ...
}

```

É importante lembrar que os parâmetros passados para a função devem ser do mesmo tipo das variáveis da definição da função.

### 3.6.5 Declaração de retorno

Uma função pode retornar um valor para a quem a chamou após sua finalização. Isto é feito através da diretiva “return”.

A figura 3.3 ilustra este mecanismo.

O tipo de dados retornado deve ser do mesmo tipo da função.

Figura 3.3: Declaração de retorno

```
#include <stdio.h>

float media(float a, float b);

int main(void)
{
  ...
  resultado=media(n1,n2);
  ...
}

float media(float a, float b)
{
  ...
  return (a+b)/2;
}
```

Com relação a passagem e ao retorno de valores uma função em C pode apresentar as seguintes condições.

- Nenhum argumento passado e nenhum valor de retorno
- Nenhum argumento passado, mas um valor de retorno
- Argumento passado mas nenhum valor de retorno
- Argumento passado e um valor de retorno

## 3.7 Passagem de parâmetros

Existem basicamente duas formas de passar parâmetros para um procedimento. Um parâmetro pode ser passado por valor, e neste caso, o procedimento recebe o valor da variável associada ao parâmetro. Um parâmetro pode também ser passado por referência, neste caso, o procedimento recebe um ponteiro que aponta para o valor do parâmetro.

### 3.7.1 Passagem de parâmetros por valor

Na passagem de parâmetros por valor a variável que foi declarada na definição da função vai receber uma cópia do valor da variável que é passada como parâmetro. A figura 3.4 ilustra este mecanismo.

No exemplo da figura 3.4 a variável “a” vai receber uma cópia do conteúdo da variável “n1”, neste caso o número 10.

É importante ressaltar que qualquer mudança realizada no conteúdo da variável “a” não vai afetar o conteúdo da variável “n1”.

Figura 3.4: Passagem de parâmetros por valor

```

#include <stdio.h>

float media(float a, float b);

int main(void)
{
  ...
  n1=10;
  n2=12;
  resultado=media(n1,n2);
  ...
}

```

```

float media(float a, float b)
{
  ...
  return (a+b)/2;
}

```

O código 3.3 apresenta um exemplo de de passagem de parâmetros por valor.

Código 3.3: Passagem de parâmetro por valor

```

#include <stdio.h>
#include <math.h>

float areaDoCirculo(float raio)
{
  float area;
  area=M_PI*pow(raio,2);
  return area;
}

int main(void)
{
  float r, a;
  printf("Digite o raio\n");
  scanf("%f",&r);
  a=areaDoCirculo(r);
  printf("A área é %.3f",a);
  return 0;
}

```

Esta forma de passagem de parâmetros possui algumas deficiências. A utilização de

memória é ineficiente, uma vez que é necessário fazer uma cópia dos valores dos parâmetros. É possível retornar apenas um valor. Para vetores ou estruturas por exemplo o processo de cópia dos dados é custoso. Veja outro exemplo no código 3.4.

Código 3.4: Passagem de parâmetros maiores

```
#include <stdio.h>

struct dados
{
    double valores[10];
    double pesos[10];
};

struct dados aplicaPesos(struct dados d1)
{
    int i;
    for(i=0; i<10; i++)
    {
        d1.valores[i]=d1.valores[i]*d1.pesos[i];
    }
    return d1;
}

int main(void)
{
    int j;
    struct dados meusDados={{1,2,3,4,5,6,7,8,9,10},{1,2,1,2,1,2,1,2,1,2}};
    struct dados resultado;
    resultado=aplicaPesos(meusDados);
    for(j=0; j<10; j++)
    {
        printf("%lf\n",resultado.valores[j]);
    }
    return 0;
}
```

Neste exemplo é possível observar a quantidade de dados que foram copiados para os parâmetros da função. Esse processo exige mais memória e mais processamento se comparado com a passagem de parâmetros por referência.

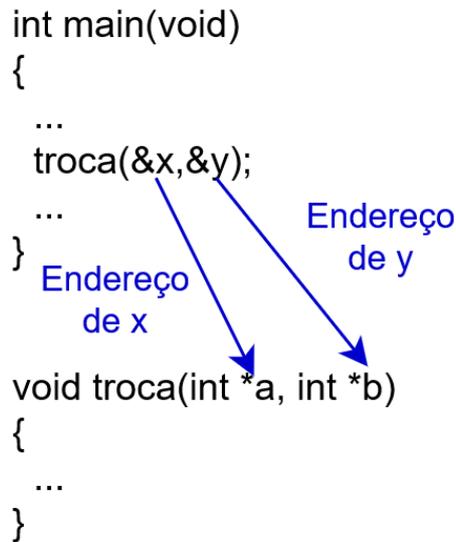
### 3.7.2 Passagem de parâmetros por referência

Na passagem de parâmetros por referência a forma de passar as informações para as funções é diferente. Ao invés de passar o valor propriamente dito o que é passado é o endereço da variável que contém o valor, ou seja, um ponteiro.

Neste o endereço de memória dos parâmetros é copiado para os parâmetros, permitindo que a função acesse a informação da variável original. Portanto, as alterações realizadas nos parâmetros da função afetam os valores dos parâmetros reais apontados pelos ponteiros. A figura 3.5 ilustra este mecanismo.

Figura 3.5: Passagem de parâmetros por referência

```
int main(void)
{
    ...
    troca(&x,&y);
    ...
}
void troca(int *a, int *b)
{
    ...
}
```



Veja um exemplo no código 3.5.

Código 3.5: Passagem de parâmetros por referência

```
#include <stdio.h>

void troca(int *a, int *b);

int main(void)
{
    int x=1, int y=2;
    troca(&x,&y);
    printf("%d %d",x,y);
    return 0;
}

void troca(int *a, int *b)
{
    int tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}
```

Observe que neste exemplo o ponteiro “a” recebe o endereço da variável “x” e o ponteiro “b” recebe o endereço da variável “y”. A função “troca” realiza a troca dos valores apontados pelos ponteiros “a” e “b”, e assim a troca dos valores das variáveis “x” e “y” é realizada.

Sem a utilização de ponteiros não seria fácil realizar a operação de troca entre duas variáveis em uma função. Como uma função pode retornar apenas um valor não seria possível retornar os dois valores envolvidos na troca.

Além de passar parâmetros para uma função através de ponteiros também é possível fazer o retorno da função como um ponteiro. Veja um exemplo no código 3.6.

Código 3.6: Retorno da função como ponteiro

```
#include <stdio.h>

int *maior(int *vet, int numRlementos)
{
    int *pt=NULL,
        int i;
    pt=&vet[0];
    for(i=1; i<numRlementos; i++)
    {
        if(vet[i]>*pt) pt=&vet[i];
    }
    return pt;
}

int main(void)
{
    int dados[10]={1,2,3,4,33,6,7,8,9,10};
    int *ptMaior=NULL;
    ptMaior=maior(dados, 10);
    printf("O maior é %d",*ptMaior);
    return 0;
}
```

Neste exemplo a função “maior” recebe o endereço de um vetor de números inteiros e recebe também o número de elementos que este vetor contém. Após descobrir qual o maior elemento do vetor a função retorna um ponteiro para este elemento.

A importância da passagem de parâmetros por referência fica ainda mais evidente quando a quantidade de dados a ser transferida como parâmetros para a função é maior. O código 3.7 apresenta o mesmo exemplo do código 3.4, só que agora é utilizada a passagem de parâmetros por referência.

Código 3.7: Passagem de parâmetros maiores por referência

```
#include <stdio.h>
struct dados
{
    double valores[10];
    double pesos[10];
};

void aplicaPesos(struct dados *d1)
{
    int i;
    for(i=0; i<10; i++)
    {
        d1->valores[i]=d1->valores[i]*d1->pesos[i];
    }
}
```

```
}  
  
int main(void)  
{  
    int j;  
    struct dados meusDados={{1,2,3,4,5,6,7,8,9,10},{1,2,1,2,1,2,1,2,1,2}};  
    aplicaPesos(&meusDados);  
    for(j=0; j<10; j++)  
    {  
        printf("%lf\n",meusDados.valores[j]);  
    }  
    return 0;  
}
```

Nesta implementação do código, a passagem de parâmetros por referência, permitiu que a função “aplicaPesos” operasse diretamente na estrutura original dos dados, eliminando a necessidade de copiar os dados para os parâmetros da função e depois para a estrutura de resultado. Assim houve uma redução do uso de memória e do uso de processamento.

### 3.8 Exercícios

1) Faça um programa em C com as seguintes características. O programa deve implementar uma estrutura chamada “strDados”, esta estrutura deve possuir três variáveis, “nome” do tipo char de 100 elementos, “idade” do tipo int e “CPF” do tipo long. O programa deve implementar 3 elementos da estrutura “strDados”. O programa deve possuir uma função chamada “leDados” que lê as informações para preencher as três estruturas do teclado e recebe o endereço da estrutura por referência. O programa deve possuir também uma função chamada “imprimeDados” que imprime as informações das três estruturas e recebe o endereço da estrutura por referência.

# Aula 4 Recursividade e iteratividade

## 4.1 Introdução

Nas ciências da computação a recursividade e a iteratividade são as principais técnicas utilizadas na criação de algoritmos que necessitam de repetições para resolver um determinado problema. Uma função iterativa é aquela que executa um laço de repetição para repetir parte do código, enquanto que uma função recursiva é aquela que chama a si mesma para repetir o código.

## 4.2 Iteratividade

Na iteratividade um processo que se repete diversas vezes para chegar a um resultado. A iteratividade acontece através das iterações, ou seja, de um processo de execução repetido de um conjunto de instruções, até que uma condição de parada seja encontrada. Um laço “for” é um exemplo de iteração ou iteratividade.

Em termos computacionais, um processo iterativo não utiliza a pilha para armazenar as variáveis. Portanto, a execução das instruções são mais rápidas se comparadas à funções recursivas. Além disso, uma função iterativa não possui a necessidade de realizar chamadas repetidas de função, o que também torna sua execução mais rápida. O ciclo de iteração é finalizado quando a condição de controle se torna falsa. A ausência de uma condição de controle pode resultar em um laço infinito ou causar um erro de compilação.

O código 4.1 apresenta um pequeno exemplo de função iterativa.

Código 4.1: Exemplo de iteratividade

```
#include <stdio.h>
int funcaoIterativa(int n)
{
    int i, soma=0;
    for(i=1; i<=n; i++)
    {
        soma=soma+i;
    }
    return soma;
}

int main(void)
{
    int num, res;
    printf("Digite um número ");
    scanf("%d",&num);
    res=funcaoIterativa(num);
    printf("O somatório dos números de 1 a %d é %d", num, res);
    return 0;
}
```

```
}
```

Neste exemplo a função “funcaoIterativa” calcula a soma dos números inteiros entre 1 e um número fornecido, pelo método iterativo.

### 4.3 Recursividade

Como já foi dito, recursão é quando uma função chama a si própria em seu corpo. Isso significa que a definição da função possui uma chamada de função para ela mesma. Sempre que isso acontece é necessário que o conjunto de variáveis e parâmetros locais usados pela função seja criado novamente na memória toda vez que a função chama a si mesma. Isto é feito armazenando as informações na parte superior da pilha. O uso de funções recursivas não reduz significativamente o tamanho do código e nem melhora a utilização da memória.

Para encerrar a recursão é necessário forçar o retorno da função sem fazer uma nova chamada a própria função. A ausência de um mecanismo de encerramento é problemática, pois a pilha de dados vai crescendo na memória até estourar seus limites.

As funções recursivas também podem ser executadas mais lentamente, pois o armazenamento dos dados na pilha leva tempo, comprometendo a performance do algoritmo se o número de repetições for muito grande. As funções que apenas iteram não exigem esse espaço na pilha e podem ser mais eficientes quando a memória é limitada.

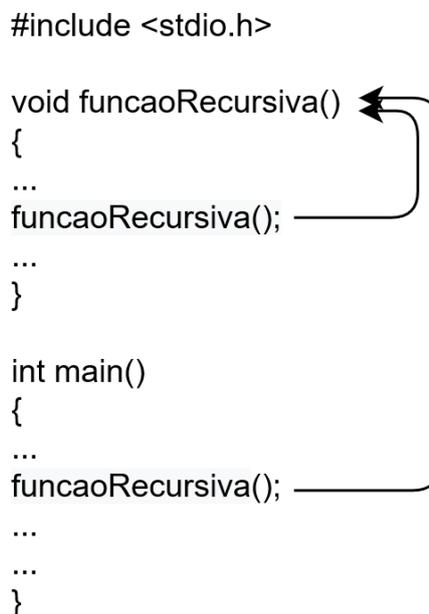
A figura 4.1 ilustra este mecanismo.

Figura 4.1: Recursividade

```
#include <stdio.h>

void funcaoRecursiva()
{
  ...
  funcaoRecursiva();
  ...
}

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



O código 4.2 apresenta um pequeno exemplo de função recursiva.

Código 4.2: Exemplo de recursividade

```
#include <stdio.h>

int funcaoRecursiva(int n)
{
    if(n>1)
    {
        return n+funcaoRecursiva(n-1);
    }
    return 1;
}

int main(void)
{
    int num, res;
    printf("Digite um número ");
    scanf("%d",&num);
    res=funcaoRecursiva(num);
    printf("O somatório dos números de 1 a %d é %d", num, res);
    return 0;
}
```

Neste exemplo a função “funcaoRecursiva” calcula a soma dos números inteiros entre 1 e um número fornecido, pelo método recursivo. Inicialmente a função “funcaoRecursiva” é chamada e recebe como parâmetro o número digitado pelo operador. Então a função verifica se este valor é maior que 1, se for a função retorna o valor da soma do número recebido com o valor retornado por uma nova chamada a si mesma. Nesta nova chamada a função recebe como argumento o número digitado pelo operador menos 1; Este processo vai se repetindo até que o valor recebido pela função seja 1. Neste momento a condição do “if” fica falsa e a última chamada da função retorna o número 1;

## 4.4 Algumas conclusões

Recursividade é quando um método utiliza recursão e chama a si mesmo repetidamente, enquanto a iteratividade ocorre quando um conjunto de instruções em um programa é executado repetidamente de forma iterativa.

A recursão é sempre aplicada ao método, enquanto a iteração é aplicada ao conjunto de instruções.

As variáveis criadas durante a recursão são armazenadas na pilha, enquanto a iteração não requer uma pilha.

A recursão pode causar a sobrecarga da chamada de função repetida, enquanto a iteração não tem uma sobrecarga de chamada de função.

A recursão reduz o tamanho do código, enquanto as iterações tornam o código mais longo.

A função recursiva é fácil de escrever e reduz o tamanho do código, mas não apresenta um bom desempenho em comparação a funções iterativas. As funções iterativas, por sua vez,

são mais difíceis de escrever, porém seu desempenho é melhor. Por outro lado, as soluções recursivas tendem a ser mais elegantes. Deve-se também considerar que funções recursivas tendem a ser mais difíceis de entender, o que dificulta o entendimento e a manutenção do programa.

## 4.5 Ordem de crescimento

Ainda com relação a algoritmos iterativos ou recursivos é importante entender como estes algoritmos se comportam em termos de complexidade computacional. Assim, nas ciências da computação a ordem de crescimento está relacionada com a complexidade de tempo de execução de algoritmos computacionais. Esta complexidade de tempo nos dá uma noção do tempo necessário para a execução de um algoritmo em função do tamanho da entrada fornecida a este algoritmo.

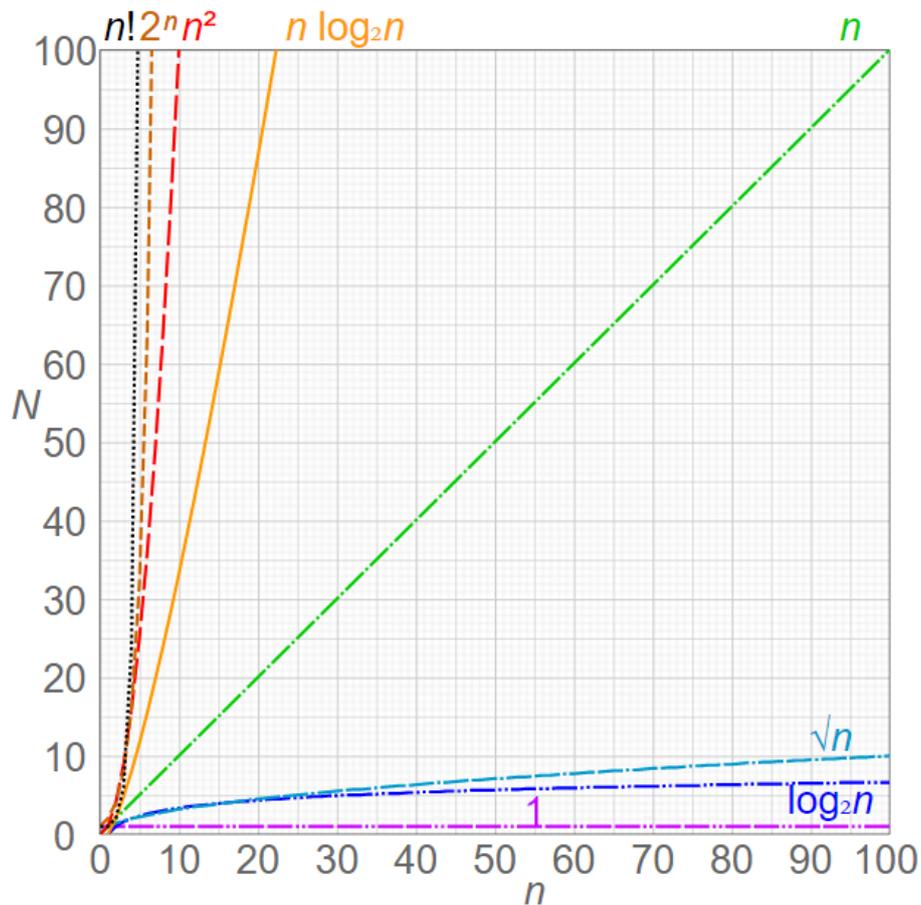
A ordem de crescimento, ou complexidade de tempo especifica como um algoritmo se comporta a medida que a ordem do tamanho da entrada é aumentada, ou seja, a ordem de crescimento é uma função do tamanho da entrada do algoritmo. A ordem de crescimento é normalmente expressada usando uma notação chamada big O. Nesta notação é utilizada uma medida assintótica da complexidade de tempo do algoritmo. Como um mesmo algoritmo pode necessitar de tempos diferentes para computar dados diferentes a notação assintótica busca considerar o pior caso. Assim, levando em consideração o número de operações necessário para executar o algoritmo, constantes multiplicativas e outros termos de menor ordem são desprezados. Por exemplo, se o tempo necessário para um algoritmo executar suas computações para um conjunto de  $n$  entradas é no máximo  $5n^3 + 3n$ , a assíntota da complexidade de tempo para este algoritmo é  $O(n^3)$ , isto considerando  $n$  tendendo ao infinito.

Existem algumas formas básicas de ordem de crescimento que representam o comportamento da maioria dos algoritmos. A seguir temos algumas delas.

- Ordem de crescimento constante:  $O(1)$ . Por exemplo, determinando se um número é par ou ímpar.
- Ordem de crescimento linear:  $O(n)$ . Por exemplo, procurar o menor valor em um vetor não ordenado.
- Ordem de crescimento logarítmico:  $O(\log n)$ . Por exemplo, busca binária.
- Ordem de crescimento loglinear ou linearitmico:  $O(n \log n)$ . Por exemplo, transformada rápida de Fourier.
- Ordem de crescimento quadrático:  $O(n^2)$ . Por exemplo, bubble sort (pior caso).
- Ordem de crescimento exponencial:  $O(2^n)$ . Por exemplo, problema do cacheiro viajante usando programação dinâmica.
- Ordem de crescimento fatorial:  $O(n!)$ . Por exemplo problema do cacheiro viajante via busca com força-bruta

A figura 4.2 apresenta o comportamento para algumas destas ordens de crescimento.

Figura 4.2: Ordens de crescimento



## 4.6 Exercícios

- 1) Faça um programa que lê um número inteiro do teclado, calcula seu fatorial e apresenta o resultado na tela. O cálculo deve ser realizado utilizando iteratividade.
- 2) Faça o mesmo algoritmo do exercício anterior só que agora utilizando recursividade.
- 3) Faça um programa que imprime na tela os 20 primeiros números da sequência de Fibonacci. O programa deve utilizar iteratividade para calcular estes números.
- 4) Faça o mesmo algoritmo do exercício anterior só que agora utilizando recursividade.

# Aula 5 Abstração de dados: pilhas e filas

## 5.1 Introdução

As abstração de dados são representações conceituais dos dados utilizadas na implementação de algoritmos. A utilização de variáveis simples não é prática para a implementação de alguns tipos de algoritmos. Existem então abstrações de dados que facilitam a manipulação das informações e as pilhas e filas são exemplos disso.

As pilhas são estruturas de dados que servem para armazenar informações de forma que a última a entrar é a primeira a sair. Do inglês LIFO (Last In First Out).

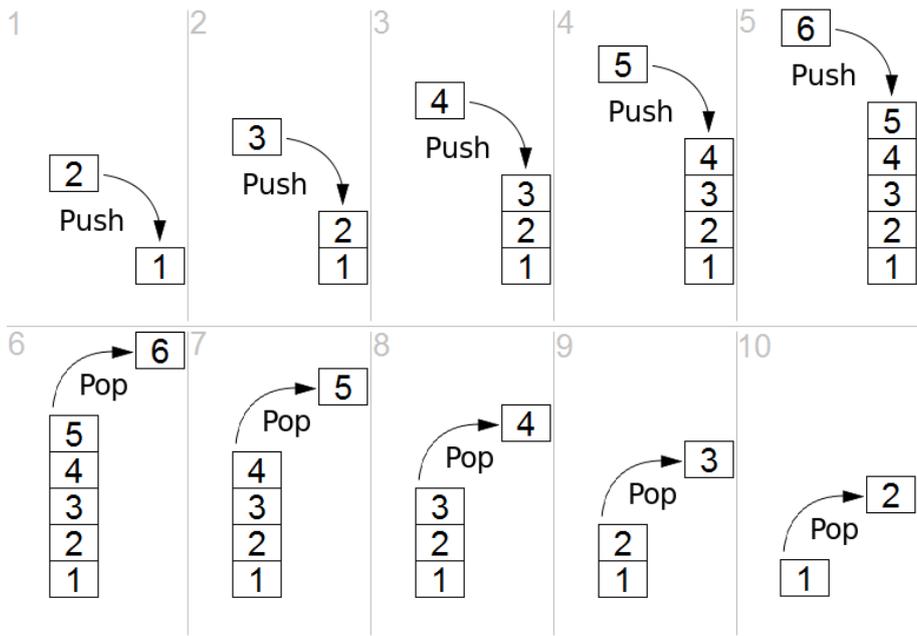
As filas por sua vez também são estruturas que servem para armazenar informações, porém, nas filas a primeira informação a entrar é a primeira informação a sair. Do inglês FIFO (First In First Out).

## 5.2 Pilhas

As pilhas são estruturas de dados que se assemelham a pilhas de pratos, o primeiro prato a ser retirado da pilha é o que está em cima, ou seja, o último a ser colocado. Este tipo de estrutura de dados é muito útil pois permite que se mantenha uma ordem inversa no acesso aos dados armazenados.

Em termos de programação, colocar um item em cima da pilha é chamado de "push" e remover um item é chamado de "pop". A figura 5.1 apresenta este conceito.

Figura 5.1: Operações push e pop



fonte: wikipedia

### 5.2.1 Implementação da pilha em C

Uma pilha é uma estrutura de dados que normalmente permite as seguintes operações:

- **Criar Pilha:** Cria a estrutura necessária para a pilha
- **Pilha Cheia:** Verifica se a pilha está cheia
- **Pilha Vazia:** Verifica se a pilha está vazia
- **Push:** Adicionar elemento para cima da pilha
- **Pop:** Remover elemento do topo da pilha

A seguir serão apresentados alguns trechos de código que implementam estas operações.

O trecho de código 5.1 apresenta a declaração da estrutura de dados e da função que cria uma pilha.

Código 5.1: Criando uma pilha

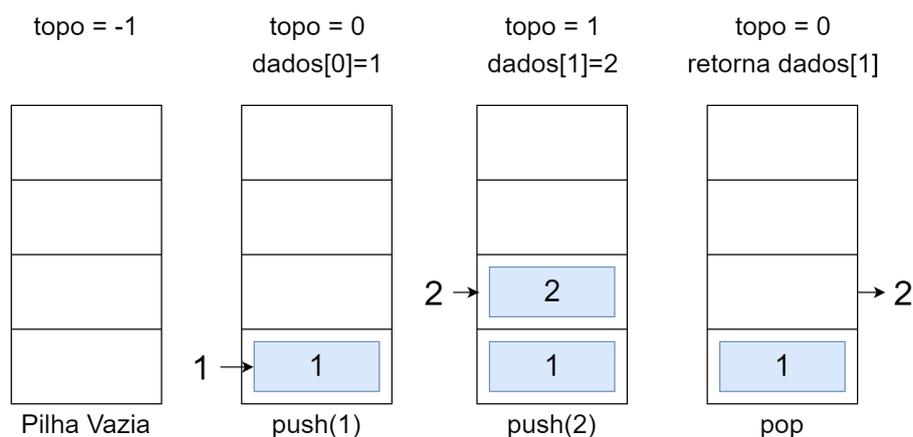
```
struct stPilha
{
    int dados[TAMANHO];
    int topo;
};

typedef struct stPilha pilha;

void criaPilha(pilha *p)
{
    p->topo=-1;
}
```

A figura 5.2 apresenta o funcionamento da pilha criada com o código anterior.

Figura 5.2: Funcionamento da pilha



Observe que a variável “topo” aponta para o elemento superior da pilha, assumindo o valor `-1` quando a pilha está vazia.

O trecho de código 5.2 verifica se a pilha está cheia.

Código 5.2: Verificando se a pilha está cheia

```
int pilhaCheia(pilha *p)
{
    if (p->topo==TAMANHO-1)
        return 1;
    else
        return 0;
}
```

O trecho de código 5.3 verifica se a pilha está vazia.

Código 5.3: Verificando se a pilha está vazia

```
int pilhaVazia(pilha *p)
{
    if (p->topo==-1)
        return 1;
    else
        return 0;
}
```

### Inserindo dados na pilha

Para inserir dados na pilha é necessário gravar a nova informação no vetor de dados e incrementar a variável que armazena o topo da pilha.

A função “push” do trecho de código 5.4 armazena dados na pilha.

Código 5.4: Armazenando dados na pilha

```
void push(pilha *p, int dado)
{
    if (pilhaCheia(p))
    {
        printf("\nErro: A pilha já esta cheia\n");
    }
    else
    {
        p->topo++;
        p->dados[p->topo]=dado;
    }
}
```

### Extraindo dados na pilha

Para extrair dados na pilha é necessário ler a informação do vetor de dados e decrementar a variável que armazena o topo da pilha.

A função “pop” do trecho de código 5.5 extrai dados na pilha.

Código 5.5: Extraindo dados na pilha

```
int pop(pilha *p)
{
    int tmp;
    if (pilhaVazia(p))
    {
        printf("\nErro: A pilha esta vazia\n");
    }
    else
    {
        tmp=p->dados[p->topo];
        p->topo--;
        return(tmp);
    }
    return(0);
}
```

### 5.2.2 Um exemplo de implementação de pilha

O código 5.6 apresenta um exemplo de programa em linguagem C que implementa uma pilha.

Código 5.6: Exemplo de pilha

```
#include<stdio.h>

#define TAMANHO 10

struct stPilha
{
    int dados[TAMANHO];
    int topo;
};

typedef struct stPilha pilha;

void criaPilha(pilha *p)
{
    p->topo=-1;
}

int pilhaCheia(pilha *p)
{
    if (p->topo==TAMANHO-1)
        return 1;
    else
        return 0;
}
```

```
int pilhaVazia(pilha *p)
{
    if (p->topo==-1)
        return 1;
    else
        return 0;
}

void push(pilha *p, int dado)
{
    if (pilhaCheia(p))
        printf("\nErro: A pilha já esta cheia\n");
    else
    {
        p->topo++;
        p->dados[p->topo]=dado;
    }
}

int pop(pilha *p)
{
    int tmp;
    if (pilhaVazia(p))
        printf("\nErro: A pilha esta vazia\n");
    else
    {
        tmp=p->dados[p->topo];
        p->topo--;
        return(tmp);
    }
    return(0);
}

int main()
{
    pilha minhaPilha;
    criaPilha(&minhaPilha);

    push(&minhaPilha, 1);
    push(&minhaPilha, 2);
    push(&minhaPilha, 3);
    push(&minhaPilha, 4);

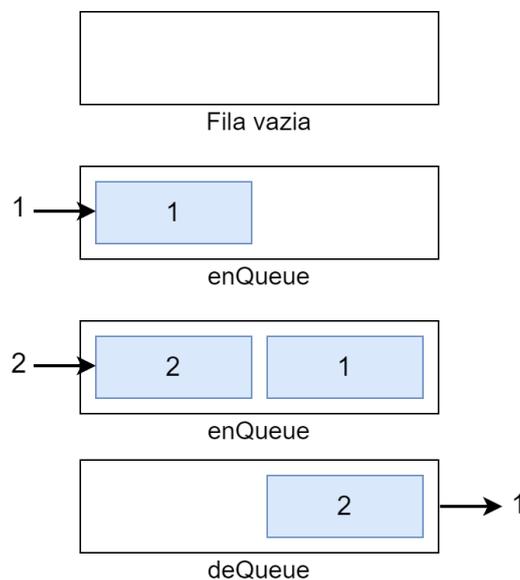
    printf("%d\n",pop(&minhaPilha));
    printf("%d\n",pop(&minhaPilha));
    printf("%d\n",pop(&minhaPilha));
    printf("%d\n",pop(&minhaPilha));
    printf("%d\n",pop(&minhaPilha));
}
```

```
return 0;
}
```

## 5.3 Filas

As filas são estruturas de dados que se assemelham a filas de pessoas em um banco por exemplo, a primeira pessoa a entrar na fila é a primeira pessoa a sair da fila e ser atendida. Este tipo de estrutura de dados é muito útil pois permite armazenar informações sem perder a sua ordem. Em termos de programação, colocar um item na fila é chamado de "enQueue". E remover um item é chamado de "deQueue". A figura 5.3 apresenta este conceito.

Figura 5.3: Operações enQueue e deQueue



### 5.3.1 Implementação da fila em C

Uma fila é uma estrutura de dados que normalmente permite as seguintes operações:

- **Criar fila:** Cria a estrutura necessária para a fila
- **enQueue:** Adiciona um elemento a fila
- **deQueue:** Remove um elemento da fila

A seguir serão apresentados alguns trechos de código que implementam estas operações. O trecho de código 5.7 apresenta a estrutura de dados e a função que cria uma fila.

Código 5.7: Criando uma fila

```
struct stFila
{
    int dados[TAMANHO];
};
```

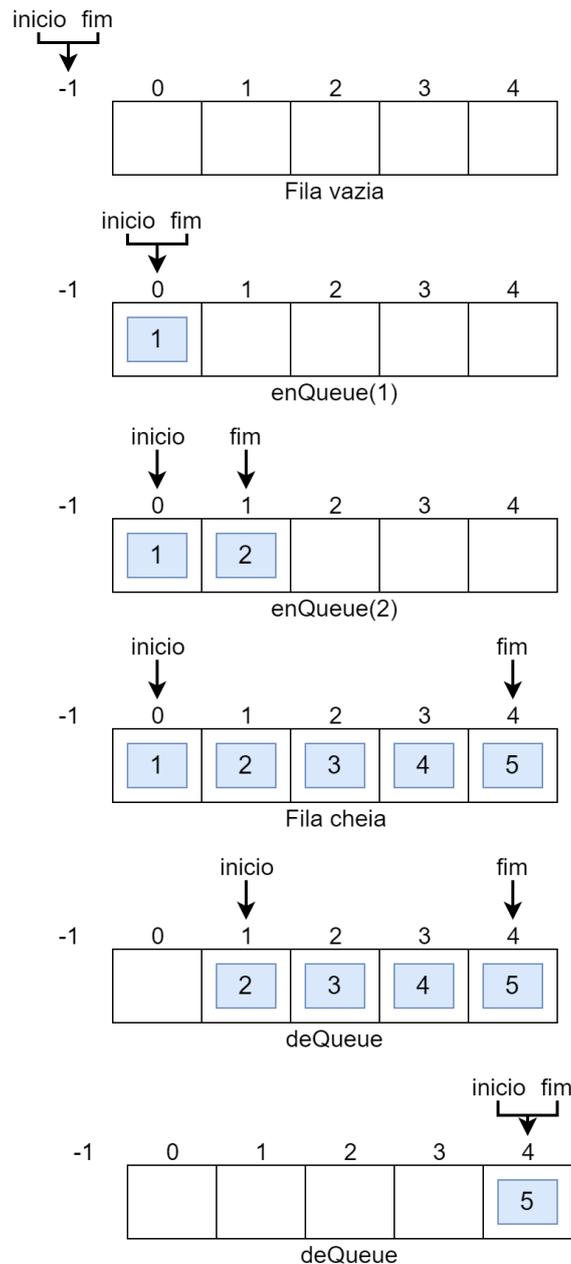
```
int inicio;
int fim;
};

typedef struct stFila fila;

void criaFila(fila *f)
{
    f->inicio=-1;
    f->fim=-1;
}
```

A figura 5.4 apresenta o funcionamento da fila.

Figura 5.4: Funcionamento da fila



### Inserindo dados na fila

Para inserir dados na fila é necessário gravar a nova informação no vetor de dados e incrementar a variável do final da fila.

A função "enQueue" do trecho de código 5.8 insere dados na fila.

Código 5.8: Inserindo dados na fila

```
void enQueue(fila *f, int dado)
{
    if(f->fim == TAMANHO-1)
        printf("\nErro: fila cheia\n");
```

```
else
{
    if(f->inicio == -1)
        f->inicio = 0;
    f->fim++;
    f->dados[f->fim] = dado;
}
}
```

### Extraindo dados na fila

Para extrair dados na fila é necessário ler a informação do vetor de dados e incrementar a variável do início da fila.

A função “deQueue” do trecho de código 5.9 extrai dados na fila.

Código 5.9: Extraindo dados na fila

```
int deQueue(fila *f)
{
    int dado;
    if(f->inicio == -1)
    {
        printf("\nErro: fila vazia\n");
        return(0);
    }
    else
    {
        dado=f->dados[f->inicio];
        f->inicio++;
        if(f->inicio > f->fim)
            f->inicio = f->fim = -1;
    }
    return dado;
}
```

Esta implementação da fila tem uma característica muito peculiar e importante. Os ponteiros que apontam o início e o fim da fila só retornam para o começo do vetor se todos os dados inseridos na fila forem lidos. Se ainda existir algum dado na última posição da fila não é mais possível inserir dados na fila, mesmo que as outras posições do vetor de dados estejam vazias.

### 5.3.2 Um exemplo de implementação de fila

O código 5.10 apresenta um exemplo de programa em linguagem C que implementa uma fila.

Código 5.10: Exemplo de fila

```
#include<stdio.h>

#define TAMANHO 5

struct stFila
{
    int dados[TAMANHO];
    int inicio;
    int fim;
};

typedef struct stFila fila;

void criaFila(fila *f)
{
    f->inicio=-1;
    f->fim=-1;
}

void enqueue(fila *f, int dado)
{
    if(f->fim == TAMANHO-1)
        printf("\nErro: fila cheia\n");
    else
    {
        if(f->inicio == -1)
            f->inicio = 0;
        f->fim++;
        f->dados[f->fim] = dado;
    }
}

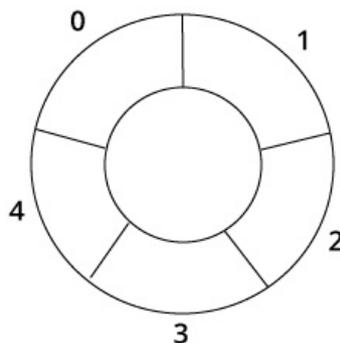
int dequeue(fila *f)
{
    int dado;
    if(f->inicio == -1)
    {
        printf("\nErro: fila vazia\n");
        return(0);
    }
    else
    {
        dado=f->dados[f->inicio];
        f->inicio++;
        if(f->inicio > f->fim)
            f->inicio = f->fim = -1;
    }
    return dado;
}
```

```
}  
  
int main()  
{  
    fila minhaFila;  
    criaFila(&minhaFila);  
    enqueue(&minhaFila,1);  
    enqueue(&minhaFila,2);  
    enqueue(&minhaFila,3);  
    enqueue(&minhaFila,4);  
    enqueue(&minhaFila,5);  
    enqueue(&minhaFila,6); // erro fila cheia  
    printf("%d\n",deQueue(&minhaFila));  
    printf("%d\n",deQueue(&minhaFila));  
    printf("%d\n",deQueue(&minhaFila));  
    printf("%d\n",deQueue(&minhaFila));  
    printf("%d\n",deQueue(&minhaFila));  
    printf("%d\n",deQueue(&minhaFila)); // erro fila vazia  
    return 0;  
}
```

## 5.4 Filas circulares

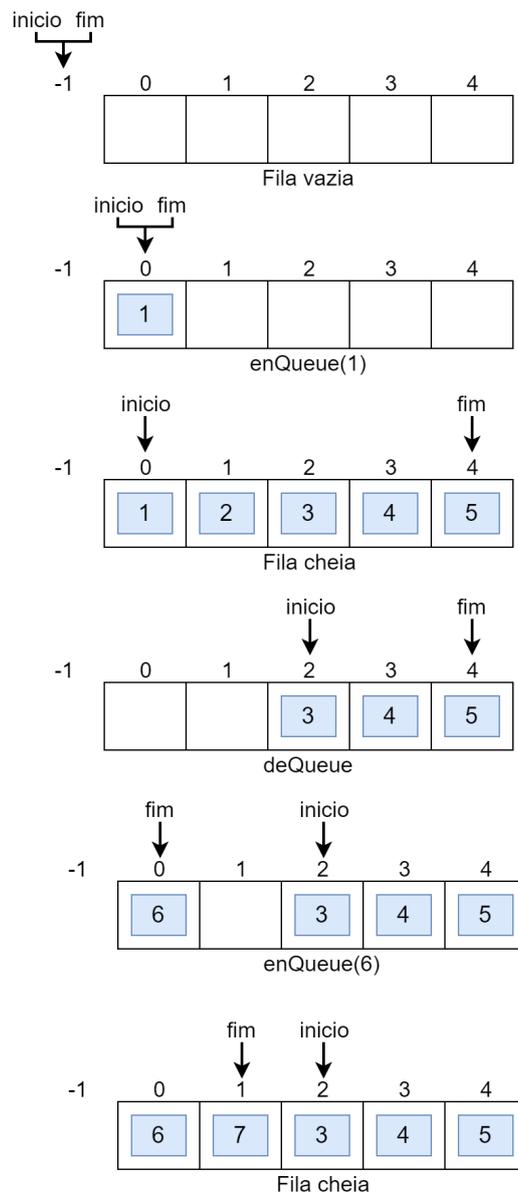
As filas circulares são estruturas semelhantes as filas normais, porém quando os ponteiros de início e fim da fila chegam ao último elemento do vetor, eles podem voltar automaticamente para o início do vetor. As figuras 5.5 e 5.6 apresentam este conceito.

Figura 5.5: Fila Circular



Este retorno ao início do vetor só pode acontecer se houver espaço no início deste vetor para a inserção de dados ou, no caso de uma leitura, se houver dados a ser lidos no início deste vetor.

Figura 5.6: Funcionamento da fila circular



### 5.4.1 Implementação da fila circular em C

Uma fila circular, assim como uma fila normal normalmente permite as seguintes operações:

- **Criar fila:** Cria a estrutura necessária para a fila
- **enQueue:** Adiciona um elemento a fila
- **deQueue:** Remove um elemento da fila

A seguir serão apresentados alguns trechos de código que implementam estas operações em uma fila circular. O trecho de código 5.11 apresenta a estrutura de dados e a função que cria uma fila circular.

Código 5.11: Criando uma fila circular

```
struct stFila
{
    int dados[TAMANHO];
    int inicio;
    int fim;
};

typedef struct stFila fila;

void criaFila(struct stFila *f)
{
    f->inicio=-1;
    f->fim=-1;
}
```

### Inserindo dados na fila circular

Para inserir dados na fila circular é necessário gravar a nova informação no vetor de dados e incrementar a variável do final da fila, se esta variável ultrapassar o tamanho do vetor, ela deve ser apontada para o início do vetor.

A função “enQueue” do trecho de código 5.12 insere dados na fila circular.

Código 5.12: Inserindo dados na fila circular

```
void enQueue(struct stFila *f, int dado)
{
    if((f->inicio == f->fim + 1) || (f->inicio == 0 && f->fim == TAMANHO-1))
        printf("\nErro: fila cheia\n");
    else
    {
        if(f->inicio == -1)
            f->inicio = 0;
        f->fim=(f->fim+1) % TAMANHO;
        f->dados[f->fim] = dado;
    }
}
```

### Extraindo dados da fila circular

Para extrair dados na fila circular é necessário ler a informação do vetor de dados e incrementar a variável do início da fila, se esta variável ultrapassar o tamanho do vetor, ela deve ser apontada para o início do vetor.

A função “deQueue” do trecho de código 5.13 extrai dados da fila circular.

Código 5.13: Extraindo dados da fila circular

```
int deQueue(struct stFila *f)
{
```

```
int dado;
if(f->inicio == -1)
{
    printf("\nErro: fila vazia\n");
    return(0);
}
else
{
    dado=f->dados[f->inicio];
    if (f->inicio == f->fim)
    {
        f->inicio = f->fim = -1;
    }
    else
    {
        f->inicio = (f->inicio + 1) % TAMANHO;
    }
}
return dado;
}
```

### 5.4.2 Um exemplo de implementação de fila circular

O código 5.14 apresenta um exemplo de programa em linguagem C que implementa uma fila circular.

Código 5.14: Exemplo de fila circular

```
#include<stdio.h>
#define TAMANHO 5

struct stFila
{
    int dados[TAMANHO];
    int inicio;
    int fim;
};

typedef struct stFila fila;

void criaFila(struct stFila *f)
{
    f->inicio=-1;
    f->fim=-1;
}

void enqueue(struct stFila *f, int dado)
{
    if((f->inicio == f->fim + 1) || (f->inicio == 0 && f->fim == TAMANHO-1))
```

```
printf("\nErro: fila cheia\n");
else
{
    if(f->inicio == -1)
        f->inicio = 0;
    f->fim=(f->fim+1) % TAMANHO;
    f->dados[f->fim] = dado;
}
}

int deQueue(struct stFila *f)
{
    int dado;
    if(f->inicio == -1)
    {
        printf("\nErro: fila vazia\n");
        return(0);
    }
    else
    {
        dado=f->dados[f->inicio];
        if (f->inicio == f->fim)
        {
            f->inicio = f->fim = -1;
        }
        else
        {
            f->inicio = (f->inicio + 1) % TAMANHO;
        }
    }
    return dado;
}

void imprimeDados(struct stFila *f)
{
    printf("\n[");
    for(int cont=0; cont<TAMANHO; cont++)
        printf("%d ",f->dados[cont]);
    printf("]");
}

int main()
{
    fila minhaFila;

    criaFila(&minhaFila);
    enQueue(&minhaFila,1);
    enQueue(&minhaFila,2);
```

```
enQueue(&minhaFila,3);
enQueue(&minhaFila,4);
enQueue(&minhaFila,5);
enQueue(&minhaFila,6); // erro fila cheia
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila));
enQueue(&minhaFila,7);
enQueue(&minhaFila,8);
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila));
printf("%d\n",deQueue(&minhaFila)); // erro fila vazia
return 0;
}
```

## 5.5 Exercícios

- 1) Crie um programa que implementa uma pilha de livros. Cada livro deve possuir as seguintes informações: título com 200 caracteres e ano.
- 2) Construa agora um programa que implementa uma fila para os livros do exercício 1.
- 3) Altere o programa do exercício 2 para implementar uma fila circular para os livros do exercício 1.

# Aula 6 Abstração de dados: listas

## 6.1 Introdução

As abstrações de dados são representações conceituais dos dados utilizados na implementação de algoritmos de forma a permitir uma abordagem mais prática dos problemas computacionais. Diferente dos tipos básicos de dados, as abstrações de dados possuem além dos próprios dados informações sobre as relações entre estes dados.

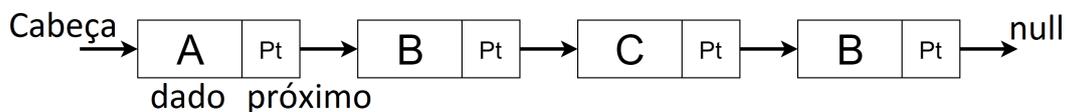
Existem diversas abstrações de dados, nesta aula iniciaremos como estudos das listas, ou listas ligadas.

## 6.2 Listas

Assim como um vetor, uma lista serve para armazenar um conjunto de dados, mas diferentemente dos vetores, as listas utilizam ponteiros para interconectar os elementos de dados.

A figura 6.1 apresenta o conceito de lista.

Figura 6.1: Listas



O uso de tipos de dados tradicionais têm algumas limitações.

1. Seu tamanho é fixo: portanto, precisamos saber o limite superior do número de elementos com antecedência. Além disso, geralmente, a memória alocada é igual ao limite superior, independentemente do seu uso.
2. Inserir um novo elemento em uma determinada posição de um vetor ou uma matriz, deslocando os demais elementos é muito custoso, isso porque um espaço precisa ser criada para os novos elementos e para criar este espaço, os elementos existentes precisam ser deslocados.
3. A exclusão de elementos neste contexto também é custosa, pois os elementos devem ser deslocados para ocupar o espaço deixado.

Assim as listas tem as vantagens de possuir um tamanho dinâmico e facilitar a inserção e exclusão de dados.

As listas também tem algumas desvantagens a ser consideradas.

1. Acesso aleatório não é permitido. Temos que acessar os elementos sequencialmente começando no primeiro nó.

2. É necessário um espaço de memória extra para um ponteiro com cada elemento da lista.
3. Não é compatível com o sistema de cache do computador. A localização dos dados da lista não é sequencial na memória.

### 6.2.1 Implementação de uma lista

Uma lista é implementada por um ponteiro que aponta para o primeiro nó da lista. O primeiro nó é chamado de cabeça (Head). Se a lista vinculada estiver vazia, o valor do ponteiro será NULL.

Para a construção da lista utiliza-se estruturas que contém pelo menos as seguintes partes.

1. Dados
2. Ponteiro (ou referência) para o próximo nó

O trecho de código 6.1 mostra a estrutura de um nó básico de uma lista.

Código 6.1: Exemplo de nó de lista

```
// Nó de uma lista
struct No {
    int dado;
    struct No* proximo;
};
```

O código 6.2 mostra a implementação de uma lista simples.

Código 6.2: Exemplo de lista simples

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

struct No {
    int dado;
    struct No* proximo;
};

int main()
{
    struct No* cabeca = NULL;
    struct No* segundo = NULL;
    struct No* terceiro = NULL;

    cabeca = (struct No*)malloc(sizeof(struct No));
    segundo = (struct No*)malloc(sizeof(struct No));
    terceiro = (struct No*)malloc(sizeof(struct No));
```

```
cabeca->dado = 1;
cabeca->proximo = segundo;
segundo->dado = 2;
segundo->proximo = terceiro;

terceiro->dado = 3;
terceiro->proximo = NULL;

return 0;
}
```

### 6.2.2 Percorrendo uma lista

No exemplo anterior, foi criada uma lista simples com três nós. Para acessar os dados de uma lista é necessário percorrê-la. O código 6.3 apresenta uma função que percorre a lista para imprimir seus elementos.

Código 6.3: Percorrendo uma lista

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

struct No {
    int dado;
    struct No* proximo;
};

void imprimeLista(struct No *ptNo)
{
    while(ptNo!=NULL)
    {
        printf("%d ",ptNo->dado);
        ptNo=ptNo->proximo;
    }
}

int main()
{
    struct No* cabeca = NULL;
    struct No* segundo = NULL;
    struct No* terceiro = NULL;

    cabeca = (struct No*)malloc(sizeof(struct No));
    segundo = (struct No*)malloc(sizeof(struct No));
    terceiro = (struct No*)malloc(sizeof(struct No));

    cabeca->dado = 1;
    cabeca->proximo = segundo;
```

```
segundo->dado = 2;
segundo->proximo = terceiro;
terceiro->dado = 3;
terceiro->proximo = NULL;

imprimeLista(cabeca);

return 0;
}
```

### 6.2.3 Inserindo nós na lista

Uma operação comum quando estamos trabalhando com listas é a inserção de novos dados na lista. É possível inserir dados no início da lista, após um determinado nó e no final da lista.

### 6.2.4 Inserindo um nó no início da lista

O trecho de código 6.4 apresenta uma função `insere` um nó no início da lista.

Código 6.4: Inserindo dados no início da lista

```
void empura(struct No** noCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));
    novoNo->dado = novoDado;
    novoNo->proximo = (*noCabeca);
    (*noCabeca) = novoNo;
}
```

Esta função recebe como parâmetros um ponteiro para o ponteiro da cabeça da lista e o novo dado a ser inserido na lista. Este dado é inserido no começo da lista e os endereços da lista são atualizados.

### 6.2.5 Inserindo um nó após uma posição da lista

O trecho de código 6.5 apresenta uma função que insere um nó após uma determinada posição da lista.

Código 6.5: Inserindo dados no após de um nó

```
void insereDado(struct No *noAnterior, int novoDado)
{
    if (noAnterior == NULL)
    {
        printf("O nó anterior não pode ser NULL");
        return;
    }
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));
```

```
novoNo->dado = novoDado;
novoNo->proximo = noAnterior->proximo;
noAnterior->proximo = novoNo;
}
```

Esta função insere um novo nó após o nó fornecido como referência e ajusta os endereços.

### 6.2.6 Anexando um nó no final da lista

O trecho de código 6.6 apresenta uma função que anexa um nó no final da lista.

Código 6.6: Anexando dados no final da lista

```
void anexa(struct No** ptCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));

    struct No *ultimo = *ptCabeca;
    novoNo->dado = novoDado;
    novoNo->proximo = NULL;

    if (*ptCabeca == NULL)
    {
        *ptCabeca = novoNo;
        return;
    }

    while (ultimo->proximo != NULL)
    {
        ultimo = ultimo->proximo;
    }

    ultimo->proximo = novoNo;
}
```

Esta função insere um novo nó final da lista e ajusta os endereços.

### 6.2.7 Um exemplo mais completo

O código 6.7 apresenta um exemplo de programa que demonstra a utilização das funções de inserção de dados em uma lista.

Código 6.7: Exemplo de inserção de dados na lista

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

struct No {
    int dado;
    struct No* proximo;
}
```

```
};

void imprimeLista(struct No *ptNo)
{
    while(ptNo!=NULL)
    {
        printf("%d ",ptNo->dado);
        ptNo=ptNo->proximo;
    }
}

void empura(struct No** noCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));
    novoNo->dado = novoDado;
    novoNo->proximo = (*noCabeca);
    (*noCabeca) = novoNo;
}

void insereDado(struct No *noAnterior, int novoDado)
{
    if (noAnterior == NULL)
    {
        printf("O nó anterior não pode ser NULL");
        return;
    }

    struct No* novoNo = (struct No*) malloc(sizeof(struct No));
    novoNo->dado = novoDado;
    novoNo->proximo = noAnterior->proximo;
    noAnterior->proximo = novoNo;
}

void anexa(struct No** ptCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));

    struct No *ultimo = *ptCabeca;
    novoNo->dado = novoDado;
    novoNo->proximo = NULL;

    if (*ptCabeca == NULL)
    {
        *ptCabeca= novoNo;
        return;
    }

    while (ultimo->proximo != NULL)
```

```
{
    ultimo = ultimo->proximo;
}

ultimo->proximo = novoNo;
}

int main()
{
    struct No* cabeca = NULL;
    struct No* segundo = NULL;
    struct No* terceiro = NULL;

    cabeca = (struct No*)malloc(sizeof(struct No));
    segundo = (struct No*)malloc(sizeof(struct No));
    terceiro = (struct No*)malloc(sizeof(struct No));

    cabeca->dado = 1;
    cabeca->proximo = segundo;
    segundo->dado = 2;
    segundo->proximo = terceiro;
    terceiro->dado = 3;
    terceiro->proximo = NULL;

    empura(&cabeca, 0);
    insereDado(segundo, 23);
    anexa(&cabeca, 34);

    imprimeLista(cabeca);
    return 0;
}
```

### 6.2.8 Apagando um nó de uma lista

Para apagarmos um nó de uma lista é necessário desalocar a memória alocada para este nó. Na linguagem C a função utilizada para este fim é a função `free()`.

O trecho de código 6.8 apresenta uma função que exclui um nó de uma lista. Esta função vai apagar o primeiro nó que contenha o elemento fornecido.

Código 6.8: Apagando um nó de uma lista

```
void apagaNo(struct No **ptCabeca, int elemento)
{
    struct No* temp = *ptCabeca, *previo;
    if (temp != NULL && temp->dado == elemento)
    {
        *ptCabeca = temp->proximo;
        free(temp);
        return;
    }
}
```

```
}
while (temp != NULL && temp->dado != elemento)
{
    previo = temp;
    temp = temp->proximo;
}

if (temp == NULL) return;

previo->proximo = temp->proximo;

free(temp);
}
```

### 6.2.9 Apagando um nó em uma posição da lista

O trecho de código 6.9 apresenta uma função que exclui um nó em uma posição de uma lista.

Código 6.9: Apagando um nó em uma posição da lista

```
void apagaNoPosicao(struct No **ptCabeca, int Posicao)
{
    if (*ptCabeca == NULL)
        return;

    struct No* temp = *ptCabeca;

    if (Posicao == 0)
    {
        *ptCabeca = temp->proximo;
        free(temp);
        return;
    }

    for (int i=0; temp!=NULL && i<Posicao-1; i++)
        temp = temp->proximo;

    if (temp == NULL || temp->proximo == NULL)
        return;

    struct No *proximoNo = temp->proximo->proximo;

    free(temp->proximo);

    temp->proximo = proximoNo;
}
```

### 6.2.10 Exemplo de exclusão de nós de uma lista

O código 6.10 apresenta um programa que utiliza as funções para apagar elementos de uma lista.

Código 6.10: Apagando elementos da lista

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

struct No {
    int dado;
    struct No* proximo;
};

void imprimeLista(struct No *ptNo)
{
    while(ptNo!=NULL)
    {
        printf("%d ",ptNo->dado);
        ptNo=ptNo->proximo;
    }
}

void anexa(struct No** ptCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));

    struct No *ultimo = *ptCabeca;
    novoNo->dado = novoDado;
    novoNo->proximo = NULL;

    if (*ptCabeca == NULL)
    {
        *ptCabeca= novoNo;
        return;
    }

    while (ultimo->proximo != NULL)
    {
        ultimo = ultimo->proximo;
    }

    ultimo->proximo = novoNo;
}

void apagaNo(struct No **ptCabeca, int elemento)
{
    struct No* temp = *ptCabeca, *previo;
    if (temp != NULL && temp->dado == elemento)
```

```
{
    *ptCabeca = temp->proximo;
    free(temp);
    return;
}

while (temp != NULL && temp->dado != elemento)
{
    previo = temp;
    temp = temp->proximo;
}

if (temp == NULL) return;

previo->proximo = temp->proximo;

free(temp);
}

void apagaNoPosicao(struct No **ptCabeca, int Posicao)
{
    if (*ptCabeca == NULL)
        return;

    struct No* temp = *ptCabeca;

    if (Posicao == 0)
    {
        *ptCabeca = temp->proximo;
        free(temp);
        return;
    }

    for (int i=0; temp!=NULL && i<Posicao-1; i++)
        temp = temp->proximo;

    if (temp == NULL || temp->proximo == NULL)
        return;

    struct No *proximoNo = temp->proximo->proximo;

    free(temp->proximo);

    temp->proximo = proximoNo;
}

int main()
{
```

```
struct No* cabeca = NULL;
anexa(&cabeca, 10);
anexa(&cabeca, 11);
anexa(&cabeca, 12);
anexa(&cabeca, 13);
anexa(&cabeca, 14);
anexa(&cabeca, 15);

imprimeLista(cabeca);
apagaNo(&cabeca, 14);
printf("\n");
imprimeLista(cabeca);
apagaNoPosicao(&cabeca, 2);
printf("\n");
imprimeLista(cabeca);

return 0;
}
```

### 6.2.11 Apagando uma lista

O trecho de código 6.11 apresenta uma função para apagar uma lista.

Código 6.11: Apagando uma lista

```
void apagarLista(struct No **ptCabeca)
{
    struct No *currente = *ptCabeca;
    struct No *ptProximo;

    while (currente != NULL)
    {
        ptProximo = currente->proximo;
        free(currente);
        currente = ptProximo;
    }

    *ptCabeca = NULL;
}
```

### 6.2.12 Exemplo de programa que apaga uma lista

O código 6.12 apresenta um programa que utiliza a função para apagar uma lista.

Código 6.12: Exemplo de função para apagar uma lista

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

```
struct No {
    int dado;
    struct No* proximo;
};

void imprimeLista(struct No *ptNo)
{
    while(ptNo!=NULL)
    {
        printf("%d ",ptNo->dado);
        ptNo=ptNo->proximo;
    }
}

void anexa(struct No** ptCabeca, int novoDado)
{
    struct No* novoNo = (struct No*) malloc(sizeof(struct No));

    struct No *ultimo = *ptCabeca;
    novoNo->dado = novoDado;
    novoNo->proximo = NULL;

    if (*ptCabeca == NULL)
    {
        *ptCabeca= novoNo;
        return;
    }

    while (ultimo->proximo != NULL)
    {
        ultimo = ultimo->proximo;
    }

    ultimo->proximo = novoNo;
}

void apagarLista(struct No **ptCabeca)
{
    struct No *currente = *ptCabeca;
    struct No *ptProximo;

    while (currente != NULL)
    {
        ptProximo = currente->proximo;
        free(currente);
        currente = ptProximo;
    }
    *ptCabeca = NULL;
}
```

```
}  
  
int main()  
{  
    struct No* cabeca = NULL;  
    anexa(&cabeca, 10);  
    anexa(&cabeca, 11);  
    anexa(&cabeca, 12);  
    anexa(&cabeca, 13);  
    anexa(&cabeca, 14);  
    anexa(&cabeca, 15);  
  
    imprimeLista(cabeca);  
    apagarLista(&cabeca);  
    imprimeLista(cabeca);  
  
    return 0;  
}
```

### 6.2.13 Encontrados nós na lista

Encontrar uma informação na lista é uma operação bastante comum. O trecho de código 6.13 apresenta uma função para encontrar o a posição na lista onde um determinado valor aparece pela primeira vez.

Código 6.13: Encontrando um nó na lista

```
int encontraNo(struct No **ptCabeca, int valor)  
{  
    int posicao=0;  
    struct No* temp = *ptCabeca;  
  
    if (*ptCabeca == NULL)  
    {  
        printf("Erro: Lista vazia\n");  
        return -1;  
    }  
  
    if (temp != NULL && temp->dado == valor)  
    {  
        return posicao;  
    }  
  
    while (temp != NULL && temp->dado != valor)  
    {  
        temp = temp->proximo;  
        posicao++;  
    }  
}
```

```
if (temp == NULL)
{
    printf("Erro: Valor não encontrado\n");
    return -1;
}

return posicao;
}
```

### 6.2.14 Lendo nós da lista

A leitura das informação é uma operação das mais importantes quando se trabalha com listas, permitindo o acesso as informações armazenadas. A busca pelas informações é realizada pela sua posição na lista.

O trecho de código 6.14 apresenta uma função que retorna o valor de uma determinada posição da lista.

Código 6.14: Lendo um nó da lista

```
int leNo(struct No **ptCabeca, int Posicao)
{
    if (*ptCabeca == NULL)
    {
        printf("Erro: Lista vazia\n");
        return 0;
    }

    struct No* temp = *ptCabeca;

    for (int i=0; temp!=NULL && i<Posicao; i++)
    {
        temp = temp->proximo;
    }

    if (temp == NULL)
    {
        printf("Erro: Posição inválida\n");
        return 0;
    }

    return temp->dado;
}
```

### 6.2.15 Alterando nós da lista

Outra operação muito comum é a alteração dos dados armazenados em um determinado nó da lista. O trecho de código 6.15 apresenta uma função que altera o dado armazenado em uma determinada posição da lista.

Código 6.15: Alterando um nó da lista

```
void alteraNo(struct No **ptCabeca, int Posicao, int novoValor)
{
    if (*ptCabeca == NULL)
    {
        printf("Erro: Lista vazia\n");
        return;
    }

    struct No* temp = *ptCabeca;

    for (int i=0; temp!=NULL && i<Posicao; i++)
        temp = temp->proximo;

    if (temp == NULL)
    {
        printf("Erro: Posição inválida\n");
        return;
    }

    temp->dado=novoValor;
    return;
}
```

### 6.2.16 Exemplo com as três últimas funções

O código 6.16 apresenta um programa que encontra, lê e altera nós de uma lista.

Código 6.16: Encontrando, lendo e alterando nós da lista

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

struct No {
    int dado;
    struct No* proximo;
};

void imprimeLista(struct No *ptNo)
{
    while(ptNo!=NULL)
    {
        printf("%d ",ptNo->dado);
        ptNo=ptNo->proximo;
    }
}

void anexa(struct No** ptCabeca, int novoDado)
{
```

```
struct No* novoNo = (struct No*) malloc(sizeof(struct No));

struct No *ultimo = *ptCabeca;
novoNo->dado = novoDado;
novoNo->proximo = NULL;

if (*ptCabeca == NULL)
{
    *ptCabeca= novoNo;
    return;
}

while (ultimo->proximo != NULL)
{
    ultimo = ultimo->proximo;
}

ultimo->proximo = novoNo;
}

int leNo(struct No **ptCabeca, int Posicao)
{
    if (*ptCabeca == NULL)
    {
        printf("Erro: Lista vazia\n");
        return 0;
    }

    struct No* temp = *ptCabeca;

    for (int i=0; temp!=NULL && i<Posicao; i++)
    {
        temp = temp->proximo;
    }
    if (temp == NULL)
    {
        printf("Erro: Posição inválida\n");
        return 0;
    }

    return temp->dado;
}

int encontraNo(struct No **ptCabeca, int valor)
{
    int posicao=0;
    struct No* temp = *ptCabeca;
```

```
if (*ptCabeca == NULL)
{
    printf("Erro: Lista vazia\n");
    return -1;
}

if (temp != NULL && temp->dado == valor)
{
    return posicao;
}

while (temp != NULL && temp->dado != valor)
{
    temp = temp->proximo;
    posicao++;
}

if (temp == NULL)
{
    printf("Erro: Valor não encontrado\n");
    return -1;
}

return posicao;
}

void alteraNo(struct No **ptCabeca, int Posicao, int novoValor)
{
    if (*ptCabeca == NULL)
    {
        printf("Erro: Lista vazia\n");
        return;
    }

    struct No* temp = *ptCabeca;

    for (int i=0; temp!=NULL && i<Posicao; i++)
        temp = temp->proximo;

    if (temp == NULL)
    {
        printf("Erro: Posição inválida\n");
        return;
    }
    temp->dado=novoValor;
    return;
}
```

```
int main()
{
    int valor, posicao;
    struct No* cabeca = NULL;
    for(int i=10; i<=20; i++)
    {
        anexa(&cabeca, i);
    }

    imprimeLista(cabeca);
    printf("\n");
    valor=leNo(&cabeca,2);
    printf("%d",valor);
    printf("\n");
    posicao=encontraNo(&cabeca, 14);
    printf("%d\n",posicao);
    alteraNo(&cabeca, posicao, 123);
    imprimeLista(cabeca);
    return 0;
}
```

## 6.3 Exercícios

1) Crie um programa que implementa uma lista ligada de alunos. Cada nó da lista de alunos deve armazenar o nome com até 200 caracteres, a idade e a média. O programa também deve implementar funções para imprimir a lista de alunos e para anexar alunos a esta lista.

2) Para a lista do exercício anterior implemente uma função para apagar um nó da lista em uma posição qualquer determinada pelo usuário.

3) Também para a lista do exercício anterior implemente uma função para ler os valores de um determinado nó da lista. Os valores devem ser retornados através de ponteiros.

4) Ainda para a lista do exercício anterior, crie uma função que altera os dados de um nó especificado pelo usuário.

# Aula 7 Abstração de dados: Árvores binárias

## 7.1 Introdução

As estruturas de dados estudadas anteriormente são estruturas de dados lineares. As árvores binárias por sua vez são estruturas de dados hierárquicas. As árvores binárias assim como outras estruturas de dados em forma de árvore costumam ser utilizadas para armazenar e recuperar rapidamente dados classificados ou ordenados.

## 7.2 As árvores binárias

Diferentemente das árvores reais, as árvores binárias utilizadas na computação não crescem para cima, elas crescem para baixo. Assim o nó superior é chamado raiz da árvore. Os elementos diretamente abaixo de um elemento são chamados de filhos. O elemento diretamente acima de algo é chamado de pai. No final dos ramos da árvore, os elementos sem filhos são chamados de folhas.

As árvores binárias são assim chamado porque cada nó possui apenas dois filhos. Os filhos são normalmente chamados de filho da direita e filho da esquerda. É este relacionamento entre os nós que torna a árvore binária uma estrutura de dados tão eficiente, sua ordem de crescimento é  $O(\log_n)$  pra operações de inserção e procura. Os dados costumam ser armazenados de forma ordenada, assim o nó da esquerda costuma conter um valor menor ou igual ao nó da direita.

A estrutura da uma árvore binária é tal que cada nó que possui filhos é por si uma árvore binária. Devido a esta estrutura, é possível acessar e inserir dados facilmente em uma árvore binária usando as funções de pesquisa e inserção recursivamente.

De forma resumida, uma árvore binária é basicamente uma árvore na qual cada nó pode ter dois nós filhos e cada nó filho pode ser uma árvore binária menor. A figura 7.1 apresenta um exemplo de árvore binária.

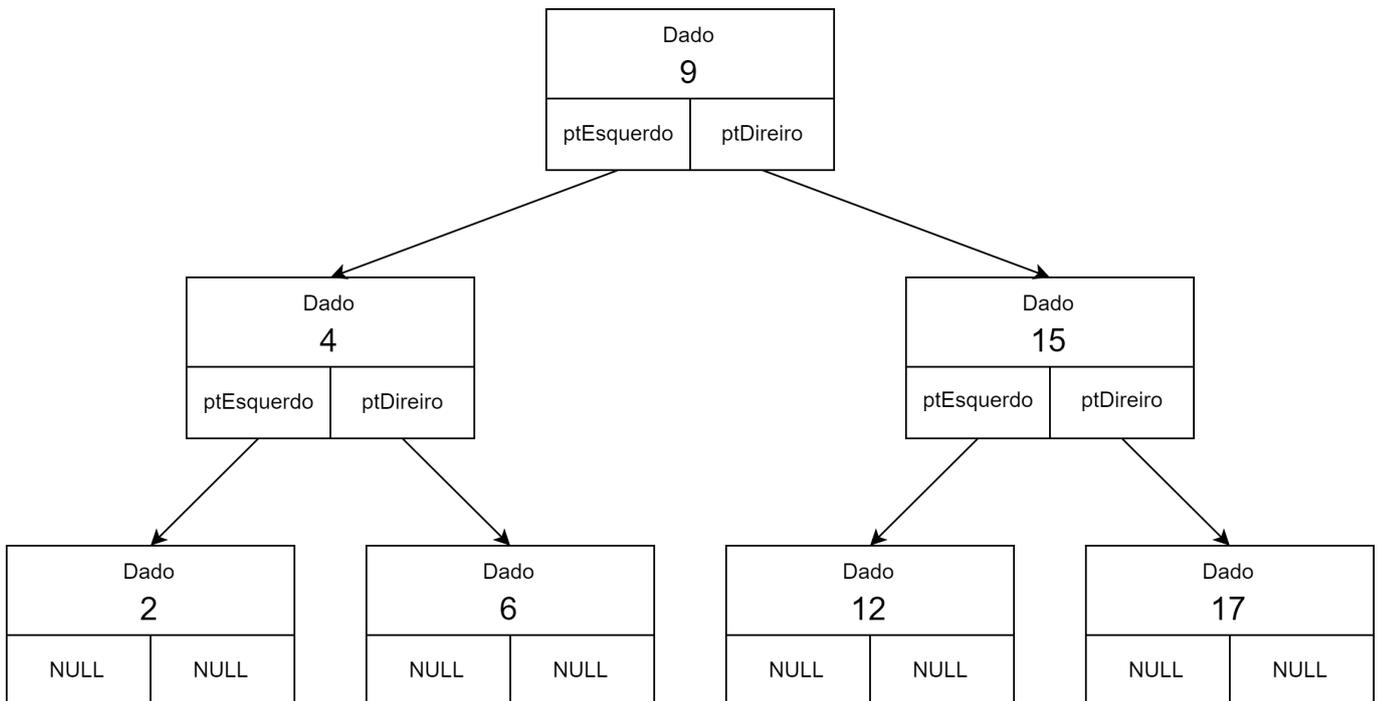
É possível observar que a árvore binária trabalha com a regra de que os nós filhos menores que o nó raiz são armazenados no lado esquerdo e os nós filhos maiores que o nó raiz são armazenados no lado direito. Esta regra se repete também nos nós filhos. No exemplo da figura 7.1, os nós (2, 4, 6) estão no lado esquerdo do nó raiz (9) e os nós (12, 15, 17) estão no lado direito do nó raiz (9).

O trecho de código 7.1 apresenta um exemplo de estrutura para armazenar os dados de um nó de uma árvore binária.

Código 7.1: Exemplo de estrutura para árvore binária

```
struct stNoArvore
{
    int dado;
    struct stNoArvore *direita;
    struct stNoArvore *esquerda;
};
```

Figura 7.1: Exemplo de árvore binária



As operações realizadas sobre as árvores binárias são diversas, neste material serão apresentadas as seguintes.

- Criando uma árvore binária
- Pesquisa em árvores binárias
- Exclusão de árvore binária
- Impressão de árvore binária

### 7.2.1 Criando uma árvore binária

A criação de uma árvore binárias é realizada alocando-se dinamicamente na memória cada um dos nós. O trecho de código 7.2 apresenta um exemplo de função para a criação dos nós de uma árvore binária.

O funcionamento desta função é o seguinte. Se o ponteiro de árvore fornecido é nulo, um novo nó é alocado na memória e seu endereço é atribuído ao ponteiro. O valor a ser inserido na árvore é armazenado neste nó.

Se o ponteiro não é nulo, é verificado se o valor a ser inserido na árvore é menor do que o valor do nó apontado pelo ponteiro. Se for a função é chamada novamente de forma recursiva e o endereço do ramo da esquerda é utilizado. Se o valor a ser inserido não for menor do que o valor do nó apontado pelo ponteiro, a função é chamada novamente de forma recursiva e o endereço do ramo da direita é utilizado.

Observe que a recursividade é profundamente explorada nesta função.

Código 7.2: Inserindo nó na árvore

```

void insereNo(noArvore ** arvore, int val)
{
    noArvore *temp = NULL;
    if(!(*arvore))
    {
        temp = (noArvore *)malloc(sizeof(noArvore));
        temp->esquerda = NULL;
        temp->direita = NULL;
        temp->dado = val;
        *arvore = temp;
        return;
    }

    if(val < (*arvore)->dado)
    {
        insereNo(&(*arvore)->esquerda, val);
    }
    else
    {
        insereNo(&(*arvore)->direita, val);
    }
}

```

### 7.2.2 Pesquisa em árvores binárias

Uma vez que as informações foram inseridas em uma árvore binária é possível pesquisar esta árvore para encontrar um determinado valor. A função do trecho de código 7.3 apresenta um exemplo pesquisa em árvore binária.

Código 7.3: Pesquisando nó na árvore

```

noArvore* procuraNo(noArvore ** arvore, int val)
{
    if(!(*arvore))
    {
        return NULL;
    }

    if(val == (*arvore)->dado)
    {
        return *arvore;
    }
    if(val < (*arvore)->dado)
    {
        return procuraNo(&(*arvore)->esquerda, val);
    }
    return procuraNo(&(*arvore)->direita, val);
}

```

Esta função utiliza recursividade para pesquisar os ramos da árvore e retornar um ponteiro para o nó que contem o valor desejado

A maior vantagem das árvores binárias consiste no fato de não ser necessário percorrer todos os nós da árvore para tentar encontrar o valor desejado, como a árvore é ordenada por natureza, apenas os ramos necessários são percorridos.

A pesquisa em uma árvore binária também utiliza recursividade para facilitar seu trabalho.

### 7.2.3 Exclusão de árvore binária

A exclusão de uma árvore é tarefa simples. Basta liberar o memória alocada para cada um dos nós. A função do trecho de código 7.4 realiza esta tarefa de forma recursiva.

Código 7.4: Excluindo uma árvore

```
void excluiArvore(noArvore * arvore)
{
    if (arvore)
    {
        excluiArvore(arvore->esquerda);
        excluiArvore(arvore->direita);
        free(arvore);
    }
}
```

### 7.2.4 Impressão de árvore binária

Para imprimir os dados armazenados em uma árvore binária é preciso percorre-la. Diferente das listas que só tinham uma forma de ser percorridas, as árvores possuem normalmente três formas que são:

- **Pre-Order**, exibe o nó raiz, o esquerdo e o direito.
- **In-Order**, exibe o nó esquerdo, o nó raiz e o nó direito.
- **Post-Order**, exibe o nó esquerdo, o direito e o nó raiz.

O trecho de código 7.5 apresenta três funções que implementam recursivamente estas três formas de imprimir uma árvore.

Código 7.5: Imprimindo uma árvore

```
void imprimePreorder(noArvore *arvore)
{
    if (arvore)
    {
        printf("%d\n", arvore->dado);
        imprimePreorder(arvore->esquerda);
        imprimePreorder(arvore->direita);
    }
}
```

```

    }
}

void imprimeInorder(noArvore *arvore)
{
    if(arvore)
    {
        imprimeInorder(arvore->esquerda);
        printf("%d\n", arvore->dado);
        imprimeInorder(arvore->direita);
    }
}

void ImprimePostorder(noArvore *arvore)
{
    if (arvore)
    {
        ImprimePostorder(arvore->esquerda);
        ImprimePostorder(arvore->direita);
        printf("%d\n", arvore->dado);
    }
}

```

### 7.2.5 Um exemplo de árvore binária

O código 7.6 apresenta uma implementação de árvore binária em C.

Código 7.6: Exemplo de árvore binária

```

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

struct stNoArvore
{
    int dado;
    struct stNoArvore *direita;
    struct stNoArvore *esquerda;
};
typedef struct stNoArvore noArvore;

void insereNo(noArvore ** arvore, int val)
{
    noArvore *temp = NULL;
    if(!(*arvore))
    {
        temp = (noArvore *)malloc(sizeof(noArvore));
        temp->esquerda = NULL;
        temp->direita = NULL;
        temp->dado = val;
    }
}

```

```
    *arvore = temp;
    return;
}
if(val < (*arvore)->dado)
{
    insereNo(&(*arvore)->esquerda, val);
}
else
{
    insereNo(&(*arvore)->direita, val);
}
}

void imprimePreorder(noArvore *arvore)
{
    if (arvore)
    {
        printf("%d\n", arvore->dado);
        imprimePreorder(arvore->esquerda);
        imprimePreorder(arvore->direita);
    }
}

void imprimeInorder(noArvore *arvore)
{
    if(arvore)
    {
        imprimeInorder(arvore->esquerda);
        printf("%d\n", arvore->dado);
        imprimeInorder(arvore->direita);
    }
}

void ImprimePostorder(noArvore *arvore)
{
    if (arvore)
    {
        ImprimePostorder(arvore->esquerda);
        ImprimePostorder(arvore->direita);
        printf("%d\n", arvore->dado);
    }
}

void excluiArvore(noArvore * arvore)
{
    if (arvore)
    {
        excluiArvore(arvore->esquerda);
        excluiArvore(arvore->direita);
    }
}
```

```
    free(arvore);
}
}

noArvore* procuraNo(noArvore ** arvore, int val)
{
    if(!(*arvore))
    {
        return NULL;
    }

    if(val == (*arvore)->dado)
    {
        return *arvore;
    }
    if(val < (*arvore)->dado)
    {
        return procuraNo(&((*arvore)->esquerda), val);
    }
    return procuraNo(&((*arvore)->direita), val);
}

int main()
{
    noArvore *raiz = NULL;
    noArvore *tmp;

    insereNo(&raiz, 9);
    insereNo(&raiz, 4);
    insereNo(&raiz, 15);
    insereNo(&raiz, 6);
    insereNo(&raiz, 12);
    insereNo(&raiz, 17);
    insereNo(&raiz, 2);

    printf("\n Imprime Pre Order\n");
    imprimePreorder(raiz);
    printf("\n Imprime In Order\n");
    imprimeInorder(raiz);
    printf("\n Imprime Post Order\n");
    ImprimePostorder(raiz);

    tmp = procuraNo(&raiz, 12);
    if (tmp)
    {
        printf("\nNó com valor %d encontrado\n", tmp->dado);
    }
    else
```

```
{  
    printf("\nNó não encontrado.\n");  
}  
excluiArvore(raiz);  
}
```

## 7.3 Exercícios

1) Crie um programa que implementa uma árvore binária para armazenar nomes de alunos. Os nomes devem ser inseridos em ordem alfabética. Utilize a função `strcmp(...)` para verificar a ordem dos nomes.

# Aula 8 Algoritmos de ordenação

## 8.1 Introdução

Algoritmos de ordenação são utilizados para reorganizar um determinado conjunto de dados de acordo com algum critério. Estes critérios podem ser por exemplo, ordem alfabética, ordem numérica crescente ou decrescente, etc. Existem diversos algoritmos utilizados na ordenação de dados, cada um com suas vantagens e desvantagens. Neste material serão apresentados três destes algoritmos, o BubbleSort, o InsertSort e o QuickSort.

## 8.2 BubbleSort

O BubbleSort é um dos algoritmos de ordenação mais simples. É ideia é comparar os elementos adjacentes e trocar suas posições se eles não estiverem na ordem pretendida, que pode ser crescente ou decrescente.

### 8.2.1 Funcionamento do BubbleSort

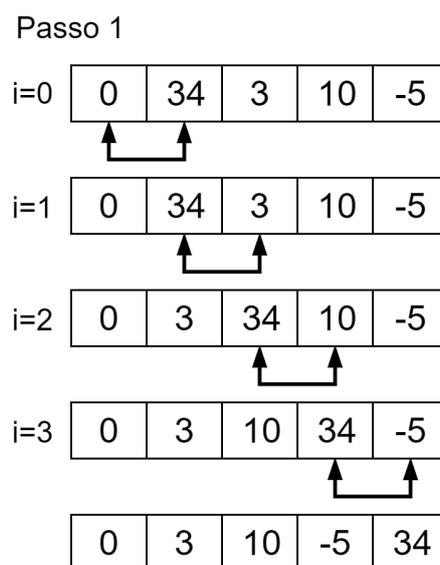
A primeiro passo do funcionamento do algoritmo é a seguinte. A partir do primeiro índice, o algoritmo compara o primeiro e o segundo elementos. Se o primeiro elemento for maior que o segundo, eles serão trocados.

Na sequência são comparados o segundo e o terceiro elementos. Se não estiverem em ordem também são trocados.

Este processo continua até o último elemento.

A figura 8.1 ilustra o primeiro passo deste processo para um vetor de números inteiros.

Figura 8.1: Primeiro passo do BubbleSort



No segundo passo, o mesmo processo continua nas iterações restantes. Após cada iteração, o maior elemento entre os elementos não classificados é colocado no final.

Em cada iteração, a comparação ocorre até o último elemento não classificado.

O conjunto de dados é ordenado quando todos os elementos são colocados em suas posições corretas.

As figuras 8.2, 8.3 e 8.4 ilustram estes passos.

Figura 8.2: Segundo passo do BubbleSort

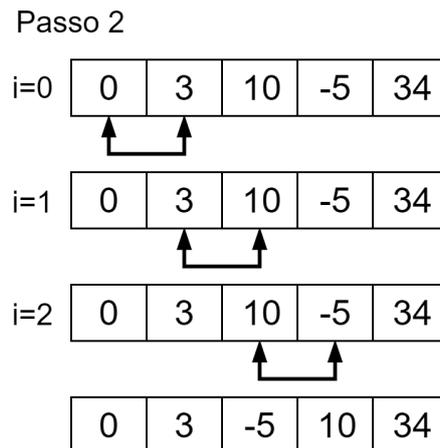


Figura 8.3: Terceiro passo do BubbleSort

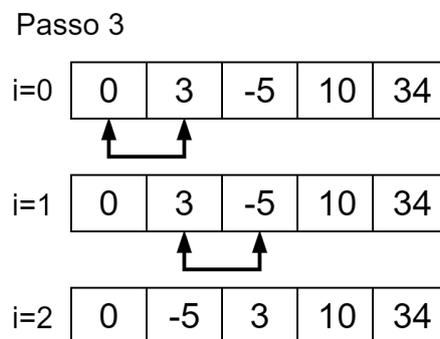
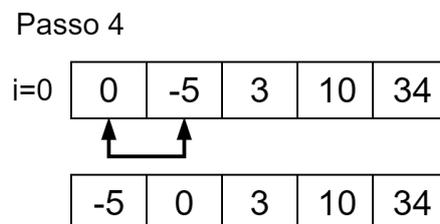


Figura 8.4: Quarto passo do BubbleSort



O trecho de código 8.1 apresenta o algoritmo do BubbleSort em linguagem C.

Código 8.1: O algoritmo do BubbleSort

```
void BubbleSort(int *vet, int n_elementos)
{
    int temp;
    for(int i=0; i<n_elementos-1; i++)
    {
        for(int j=0; j<n_elementos-i-1; j++)
        {
            if(vet[j]>vet[j+1])
            {
                temp=vet[j];
                vet[j]=vet[j+1];
                vet[j+1]=temp;
            }
        }
    }
}
```

Esta implementação do algoritmo BubbleSort possui uma deficiência. Na forma como ele foi implementado, todas as comparações possíveis são feitas mesmo que o conjunto de dados já esteja ordenado, aumentando assim o tempo de execução. É possível otimizar este algoritmo através da introdução de uma variável de controle que finaliza o laço de repetição quando o conjunto de dados estiver ordenado. Para isso, basta verificar se após cada iteração houve alguma troca de valores. Se não houver troca os dados já estão ordenados e o algoritmo pode ser finalizado.

O código 8.2 apresenta um exemplo de programa que utiliza o algoritmo do BubbleSort otimizado para ordenar números inteiros.

Código 8.2: O algoritmo do BubbleSort otimizado

```
#include <stdio.h>

void BubbleSort(int *vet, int n_elementos)
{
    int temp, troca;
    for(int i=0; i<n_elementos-1; i++)
    {
        troca=0;
        for(int j=0; j<n_elementos-i-1; j++)
        {
            if(vet[j]>vet[j+1])
            {
                temp=vet[j];
                vet[j]=vet[j+1];
                vet[j+1]=temp;
                troca=1;
            }
        }
        if(troca==0) break;
    }
}
```

```
    }  
  }  
  
  int main(void)  
  {  
    int c;  
    printf("Digite 10 Valores\n");  
    int vetor[10];  
    for(c=0;c<10;c++)  
    {  
      scanf("%d",&vetor[c]);  
    }  
  
    BubbleSort(vetor, 10);  
  
    for(c=0;c<10;c++)  
    {  
      printf("\n%d",vetor[c]);  
    }  
  
    return 0;  
  }
```

### 8.2.2 Ordem de crescimento do BubbleSort

O algoritmo do BubbleSort tem as seguintes características de ordem de crescimento ou complexidade.

- Pior caso  $O(n^2)$
- Melhor caso  $O(n)$
- performance média  $O(n^2)$

Embora o BubbleSort seja um dos algoritmos de ordenação mais simples de entender e implementar, sua complexidade de  $O(n^2)$  significa que sua eficiência diminui drasticamente conforme o conjunto de dados aumenta. Mesmo entre algoritmos simples de ordenação o BubbleSort é um dos mais ineficientes, porém ele continua sendo uma ferramenta didática importante no ensino das ciências da computação.

## 8.3 InsertSort

O InsertSort ou ordenação por inserção é outro algoritmo simples de ordenação que classifica um elemento do conjunto de dados por vez. Suas vantagens são, a implementação simples e sua eficiências para pequenos conjuntos de dados ou conjuntos de dados parcialmente ordenados.

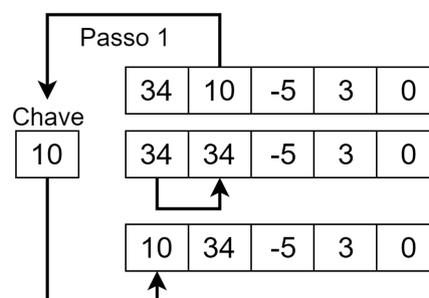
### 8.3.1 Funcionamento do InsertSort

A lógica de funcionamento do InsertSort é a seguinte. Inicialmente assume-se que o primeiro elemento do conjunto de dados já está ordenado. Em seguida, seleciona-se um elemento não ordenado. Se este elemento não ordenado for maior que o primeiro elemento, ele é inserido à direita, caso contrário, à esquerda. Repetindo este processo os outros elementos não ordenados são colocado no lugar correto. Assim a ordenação por inserção um elemento não ordenado em seu local correto em cada iteração.

O algoritmo funciona assim. No primeiro passo, o primeiro elemento do conjunto de dados é assumido como ordenado. Então o segundo elemento é definido como chave. Então a chave é comparada com o primeiro elemento e se o primeiro elemento for maior que a chave esta chave é colocada a esquerda do primeiro elemento.

A figura 8.5 ilustra este passo.

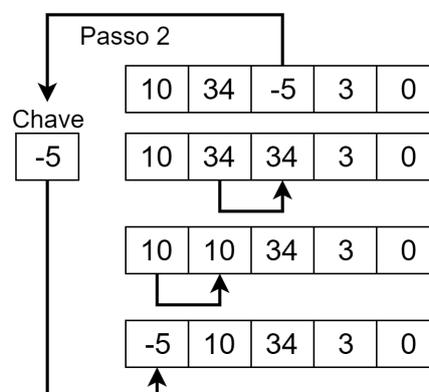
Figura 8.5: Primeiro passo do InsertSort



Este processo garante que os dois primeiros elementos do conjunto de dados estejam ordenados. O segundo passo consiste em selecionar o terceiro elemento do conjunto de dados como a chave. A chave é então comparada aos elementos a sua esquerda. A chave deve ser inserida logo a direita do primeiro elemento encontrado que seja menor que ela. Se não houver um elemento menor que a chave a chave deve ser colocada na primeira posição do conjunto. Os elementos entre a posição inicial da chave e a posição onde a chave vai ser inserida devem ser deslocados para a direita.

A figura 8.6 ilustra este passo.

Figura 8.6: Segundo passo do InsertSort



Nos passos seguintes o algoritmo vai selecionando como chave os elementos a direita da chave anterior e seguindo a mesma lógica para posicioná-la no local correto.

As figuras 8.7 e 8.8 ilustram este processo.

Figura 8.7: Terceiro passo do InsertSort

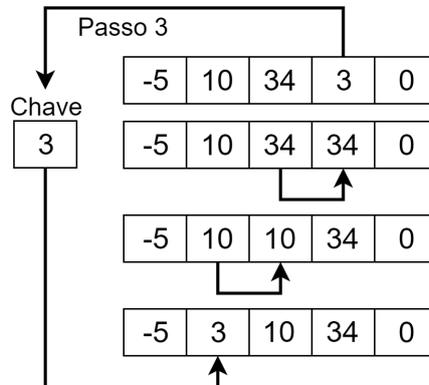
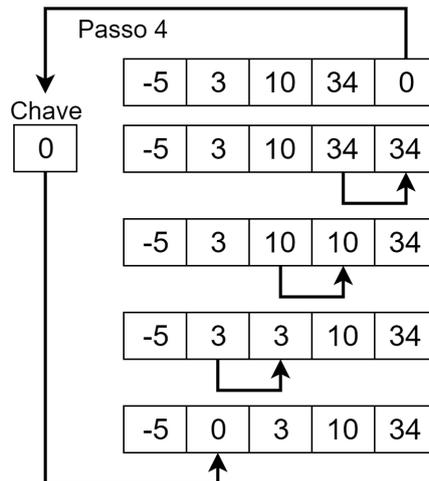


Figura 8.8: Quarto passo do InsertSort



O código 8.3 apresenta um exemplo de programa que utiliza o algoritmo do InsertSort para ordenar números inteiros.

Código 8.3: O algoritmo InsertSort

```
#include <stdio.h>

void InsertSort(int *vet, int n_elementos)
{
    int chave, j;
    for(int i=1; i<n_elementos; i++)
    {
        chave=vet[i];
        j=i-1;
        while(chave<vet[j] && j>=0)
        {
            vet[j+1]=vet[j];

```

```
        j--;
    }
    vet[j+1]=chave;
}
}

int main(void)
{
    int c;
    printf("Digite 10 Valores\n");
    int vetor[10];
    for(c=0;c<10;c++)
    {
        scanf("%d",&vetor[c]);
    }

    InsertSort(vetor, 10);

    for(c=0;c<10;c++)
    {
        printf("\n%d",vetor[c]);
    }

    return 0;
}
```

### 8.3.2 Ordem de crescimento do InsertSort

O algoritmo do InsertSort tem as seguintes características de ordem de crescimento ou complexidade.

- Pior caso  $O(n^2)$
- Melhor caso  $O(n)$
- performance média  $O(n^2)$

## 8.4 QuickSort

O Quicksort é um algoritmo de ordenação eficiente e por isso muito utilizado. Sua principal vantagem é sua eficiência, sendo muito mais rápido que a maioria dos algoritmos de ordenação. A lógica de funcionamento do Quicksort é baseada na abordagem de dividir e conquistar, no qual o conjunto de dados é dividido em sub-conjuntos e esses sub-conjuntos são ordenados recursivamente.

### 8.4.1 Funcionamento do QuickSort

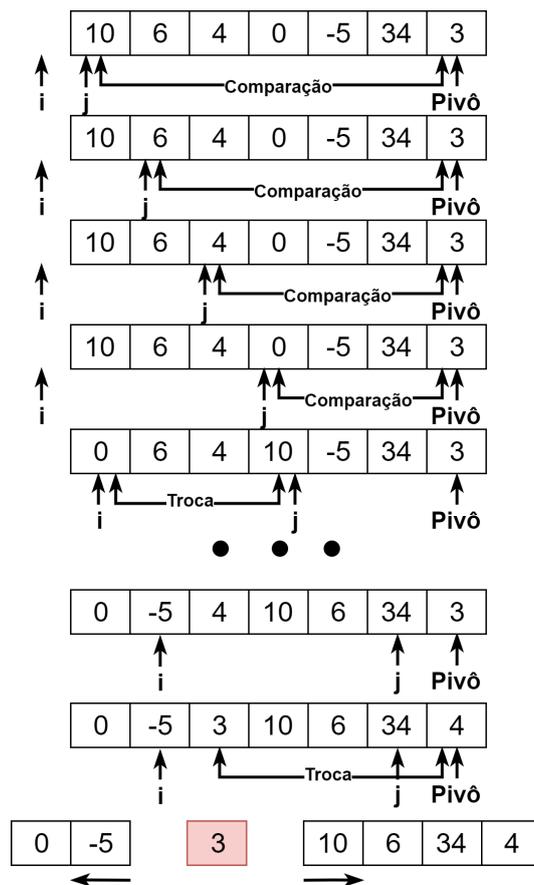
O funcionamento do QuickSort é o seguinte. Inicialmente escolhe-se um pivô. O pivô pode ser qualquer elemento do conjunto de dados. Em seguida, o conjunto de dados é arranjado de forma que todos os dados menores que o pivô fiquem a sua esquerda e todos os dados maiores que o pivô fiquem a sua direita.

A implementação computacional deste processo é a seguinte. Após a escolha do pivô, suponhamos que seja o elemento mais a direita do conjunto de dados, um ponteiro é fixado neste elemento pivô. Dois ponteiros auxiliares são utilizados, um ponteiro ( $i$ ) inicializado a esquerda do primeiro elemento e um ponteiro ( $j$ ) que aponta para o elemento a ser comparado, e é incrementado a cada comparação. O pivô é então comparado aos outros elementos, começando com o elemento mais a esquerda do conjunto de dados, apontado por ( $j$ ). Se este elemento for maior que o pivô o ponteiro ( $j$ ) é incrementado e a comparação se repete. Porém, se nesta comparação o valor apontado por ( $j$ ) for menor que o pivô, o ponteiro ( $i$ ) é incrementado e os elementos apontados por ( $i$ ) e ( $j$ ) são trocados. Este processo finaliza quando o ponteiro ( $j$ ) chegar na posição imediatamente a esquerda do pivô.

Para finalizar a etapa, o pivô é trocado com o elemento apontado por ( $i$ )+1.

As figuras 8.9 ilustram este processo.

Figura 8.9: Primeiros passos do QuickSort



Ao final deste processo o elemento pivô está inserido em sua posição correta, e não necessita mais ser movido. A esquerda deste elemento estão somente valores menores que ele, e a direita somente valores maiores.

Para dar continuidade ao processo, o algoritmo QuickSort é chamado recursivamente para os elementos da esquerda e da direita do pivô, separadamente

Este processo recursivo é finalizado quando cada sub-parte do conjunto de dados possuir apenas um elemento.

O código 8.4 apresenta um exemplo de programa que utiliza o algoritmo QuickSort para ordenar números inteiros.

Código 8.4: O algoritmo QuickSort

```
#include <stdio.h>

void troca(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int ordena(int *vet, int inicio, int fim)
{
    int pivo = vet[fim];
    int i = (inicio - 1);
    for (int j = inicio; j < fim; j++)
    {
        if (vet[j] <= pivo)
        {
            i++;
            troca(&vet[i], &vet[j]);
        }
    }
    troca(&vet[i + 1], &vet[fim]);
    return (i + 1);
}

void QuickSort(int *vet, int inicio, int fim)
{
    if (inicio < fim)
    {
        int corte = ordena(vet, inicio, fim);
        QuickSort(vet, inicio, corte - 1);
        QuickSort(vet, corte + 1, fim);
    }
}

int main(void)
{
    int c, vetor[10];
    printf("Digite 10 Valores\n");
    for(c=0;c<10;c++)
```

```
{
    scanf("%d",&vetor[c]);
}
QuickSort(vetor, 0, 9);
for(c=0; c<10; c++)
{
    printf("\n%d",vetor[c]);
}
return 0;
}
```

### 8.4.2 Ordem de crescimento do QuickSort

O algoritmo do QuickSort tem as seguintes características de ordem de crescimento ou complexidade.

- Pior caso  $O(n^2)$
- Melhor caso  $O(n * \log(n))$
- Performance média  $O(n * \log(n))$

## 8.5 Exercícios

1) Faça um programa que implementa e utiliza o algoritmo de ordenação BubbleSort para ordenar uma lista de 10 nomes de pessoas.

2) Faça um programa que implementa e utiliza o algoritmo de ordenação InsertSort para ordenar uma lista de 10 nomes de pessoas.