

**Volume**

**1**

**FTEC**

Análise e Desenvolvimento de Sistemas

Redes de Computadores

Engenharia de Computação

# Estrutura de Dados

FTEC – FACULDADE DE TECNOLOGIA

# Estrutura de Dados

---

# Índice Analítico

---

<b>1</b>	<b>CONVENÇÕES E CARACTERÍSTICAS DESTA APOSTILA .....</b>	<b>5</b>
1.1	CONVENÇÃO.....	5
<b>2</b>	<b>APONTADORES EM C.....</b>	<b>6</b>
2.1	INTRODUÇÃO.....	6
2.2	DEFINIÇÃO .....	6
2.3	UTILIZAÇÃO DE APONTADORES.....	7
2.4	MANIPULANDO APONTADORES EM C.....	7
2.4.1	<i>Declarando variáveis do tipo apontador.....</i>	<i>8</i>
2.4.2	<i>Operadores de apontadores.....</i>	<i>8</i>
2.4.3	<i>Tipo Base.....</i>	<i>10</i>
2.4.4	<i>Expressões com apontadores .....</i>	<i>10</i>
2.4.4.1	Atribuindo valores para os apontadores .....	11
2.4.4.2	Inicialização de um ponteiro .....	11
2.4.4.3	Incremento e decremento de ponteiros .....	12
2.4.4.4	Operações Relacionais (Comparação) com apontadores .....	14
2.5	APONTADORES EM FUNÇÕES .....	14
2.6	APONTADORES PARA VETORES .....	16
2.7	APONTADOR PARA ESTRUTURA.....	18
2.8	APONTADOR PARA APONTADOR.....	19
2.9	ALOCÇÃO DINÂMICA DE MEMÓRIA .....	20
2.9.1	<i>Alocar memória.....</i>	<i>20</i>
2.9.2	<i>Liberar memória alocada.....</i>	<i>21</i>
2.9.3	<i>Alocação Dinâmica de Vetores.....</i>	<i>22</i>
2.9.4	<i>Alocação Dinâmica de Matrizes.....</i>	<i>24</i>
2.9.4.1	Implementação Versão 1 .....	24
2.9.4.2	Implementação Versão 2 .....	26
2.10	CUIDADOS COM A MANIPULAÇÃO DE APONTADORES .....	28
2.11	OS ARGUMENTOS ARGC E ARGV .....	30
<b>3</b>	<b>ESTRUTURA DE DADOS .....</b>	<b>31</b>
3.1	ABSTRAÇÃO DE DADOS.....	31
3.2	TIPOS DE DADOS .....	32
3.2.1	<i>Por que temos que informar o tipo de dados ao declarar uma variável? .....</i>	<i>33</i>
3.3	TIPOS DE DADOS ABSTRATOS (TDA).....	33
3.4	OBJETIVOS DA ESTRUTURAS DE DADOS.....	34
3.5	MÓDULOS E COMPILAÇÃO EM SEPARADO.....	34
3.6	TIPOS DE DADOS ABSTRATOS EM MÓDULOS.....	35
3.7	EXEMPLO: TAD PONTO.....	36
<b>4</b>	<b>LISTAS LINEARES .....</b>	<b>38</b>
4.1	INTRODUÇÃO.....	38
4.2	FUNDAMENTOS .....	38
4.3	CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO.....	40
4.4	ALOCÇÃO DE MEMÓRIA .....	40
4.5	ALOCÇÃO ESTATICA VERSUS DINÂMICA .....	40
4.6	ALOCÇÃO SEQUENCIAL .....	42
4.7	ALOCÇÃO ENCADEADA .....	42
4.8	LISTAS ENCADEADAS.....	43
4.8.1	<i>Fundamentos de lista encadeada .....</i>	<i>44</i>
4.8.1.1	Função de inicialização .....	45
4.8.1.2	Função de inserção.....	45
4.8.1.3	Função que percorre os elementos da lista .....	47
4.8.1.4	Função que verifica se lista está vazia .....	47

4.8.1.5	Função de busca .....	47
4.8.1.6	Função que retira um elemento da lista .....	48
4.8.1.7	Função para liberar a lista .....	49
4.8.1.8	Exemplo TAD Lista de inteiros .....	49
<b>5</b>	<b>PILHAS.....</b>	<b>51</b>
5.1	INIT, ISEEMPTY E ISFULL .....	53
5.2	EXEMPLO DE USO DE PILHAS.....	54
5.3	INTERFACE DO TIPO PILHA .....	54
5.4	IMPLEMENTAÇÃO DE PILHA COM VETOR.....	55
5.5	IMPLEMENTAÇÃO DE PILHA COM LISTA .....	56
<b>6</b>	<b>FILAS .....</b>	<b>62</b>
6.1	IMPLEMENTAÇÃO SEQUENCIAL DE FILAS .....	63
6.2	INTERFACE DO TIPO FILA.....	63
6.3	IMPLEMENTAÇÃO DE FILA COM VETOR.....	64
6.4	IMPLEMENTAÇÃO DE FILA COM LISTA .....	67
6.5	FILA DUPLA .....	70
6.6	IMPLEMENTAÇÃO DE FILA DUPLA COM LISTA .....	71

## 1 Convenções e Características desta Apostila





*Você pode economizar tempo se, antes de começar, entender como são apresentadas nesta apostila as instruções, as teclas para digitar e assim por diante. Reserve, pois, um tempo para ler a lista seguinte, que também indica características úteis da apostila que talvez você queira usar.*

### 1.1 Convenção

Abaixo estão relacionadas as convenções que são utilizadas nesta apostila, principalmente na resolução de exercícios.

- Instruções sobre os exercícios são dadas em listas numeradas passo a passo (1, 2 e assim por diante).
- Os exemplos estão no formato de *exemplo*.
- Um sinal de mais (+) entre os nomes de duas teclas, significa que você deve pressioná-las ao mesmo tempo. Por exemplo, “Digite ALT+TAB” significa que você deve manter a tecla ALT pressionada enquanto tecla TAB.

Os seguintes ícones identificam os vários tipos de informações contidas no texto:

Ícone	Legenda	Descrição
	Dica	Informação valiosa
	Exercícios	Exercícios teóricos
	Exercícios	Exercícios práticos
	Exemplos	Exemplos aplicados ao conteúdo

## 2 Apontadores em C

### 2.1 Introdução

Apontadores ou ponteiros constituem um dos recursos mais utilizados na programação em linguagem C. Eles fornecem um mecanismo poderoso, flexível e eficiente de acesso a variáveis. Há computações que só podem ser expressas através do uso de ponteiros.

Apontadores devem ser utilizados com critério e disciplina. O uso descuidado de ponteiros pode levar a situações onde um endereço inválido é acessado, levando a erros de execução de programas.

### 2.2 Definição

A memória de qualquer computador é uma sequência de bytes. Cada byte pode armazenar um número entre 0 e 255 ou um caracter. Cada objeto que reside na memória do computador ocupa um certo número de bytes. Em um computador, por exemplo, um inteiro pode ocupar 2 bytes consecutivos, um caracter ocupa 1 byte e um número ponto-flutuante ocupa 4 bytes consecutivos.

Cada objeto que reside na memória do computador tem um endereço; em um determinado computador, o endereço é do primeiro byte do objeto. Por exemplo:



```
// Programa : Exemplo 1
// Objetivo: Mostrar os endereços de variáveis
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main (void )
{
    int x;           // declara 1 variável do tipo inteiro
    char a;          // declara 1 variável do tipo caracter
    int v[4];        // declara um vetor de inteiros de 4 posições
}
```

Depois das declarações:

Os endereços das variáveis poderiam ser:

Endereço físico	Endereço lógico
1000	x
1002	a
1003	v[0]
1005	v[1]
1007	v[2]
1009	v[3]


Memória, variáveis e seus endereços de memória.

Um apontador ou ponteiro é uma variável que armazena um endereço de outra variável, ou seja, cujo valor é um endereço de memória de outra variável. Se uma variável contém o endereço de outra, então, a primeira variável é dita apontar para a segunda, como mostra na Figura abaixo.

Por exemplo: Considere a seguinte declaração de variáveis: `int x = 10, y = 20;`

Suponha que `px` seja uma variável do tipo apontador para a variável `x`, o valor de `px` é o endereço físico de `x`.

Endereço físico	Endereço lógico	Conteúdo
1000	x	10
1002	y	20
1004	px	1000



Memória, uma variável que contém o endereço de outra.

### 2.3 Utilização de Apontadores

Algumas razões para o uso de ponteiros são:

- ✚ Fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem;
- ✚ Para passar estruturas homogêneas e heterogêneas mais convenientemente de uma função para outra, possibilitando a manipulação mais facilmente através da movimentação de apontadores para elas;
- ✚ Suportam as rotinas de alocação dinâmica em C.
- ✚ Para criar estruturas de dados mais complexas, como listas encadeadas e árvores;

### 2.4 Manipulando Apontadores em C

Uma variável do tipo ponteiro diferencia-se das demais pelo conteúdo que armazena, portanto, é necessário cuidado especial ao manipular variáveis desse tipo.

### 2.4.1 Declarando variáveis do tipo apontador

A forma geral de uma variável do tipo ponteiro é a seguinte:

**SINTAXE:**

**<Tipo de Dado> \* <Nome da Variável>;**

em que:

✚ **Tipo da Dado :** é o identificador de tipo de informação, qualquer tipo válido em C, para qual a variável vai apontar.

✚ **Nome da Variável:** Em geral, pode-se dar um nome qualquer a uma variável válida em C.

**Observação:** As variáveis declaradas desta forma armazenam endereços de variáveis do tipo de dado definido antes do operador \*.

**Exemplo:**

```
// Programa : Exemplo 2
// Objetivo: Mostrar a declaração de apontadores para tipos básicos.
#include <stdio.h>
#include <stdlib.h>
main (void )
{
    int x, y;           // declara 2 variáveis do tipo inteiro
    int * p1, * p2;     // declara 2 variáveis do tipo ponteiro para
                        //inteiro
    char ch, * p3;      //declara 1 variável do tipo caracter e
                        //uma outra
                        //do tipo ponteiro para caracter
}
```

### 2.4.2 Operadores de apontadores

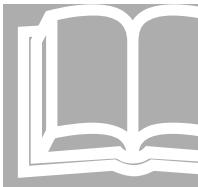
Existem dois operadores para manipular ponteiros: \* e &.

O operador & é um operador unário que retorna o endereço de memória do seu operando.

O operador \* é um operador que retorna o conteúdo da variável localizada no endereço (ponteiro) do operando, ou seja, devolve o conteúdo da variável apontada pelo operando.

**Exemplo:**

```
// Programa : Exemplo 3
// Objetivo: Mostrar os operadores de ponteiro.
#include <stdio.h>
#include <stdlib.h>
main (void )
{
    int x=10, y=20, z=0; // declara 3 variáveis do tipo inteiro
    int * p1, * p2;      // declara 2 variáveis do tipo ponteiro para inteiro
    system("cls");
    p1 = &x;             // armazena em p1 o endereço de memória da variável x
                        //ou seja, p1 aponta para x
    p2 = &y;              // armazena em p2 o endereço de memória da variável y
                        //ou seja, p2 aponta para y
    printf("\n O endereco de memoria de x: %p", &x);
    printf("\n O endereco de memoria de y: %p", &y);
}
```





```

printf("\n O endereço de memória de z: %p", &z);
printf("\n O endereço de memória de p1: %p", &p1);
printf("\n O endereço de memória de p2: %p", &p2);
printf("\n O valor de x: %d", x);
printf("\n O valor de y: %d", y);
printf("\n O valor de z: %d", z);
printf("\n O valor de p1: %p", p1); // este valor deve ser o endereço de
//memória de x
printf("\n O valor de p2: %p", p2); // este valor deve ser o endereço de
//memória de y
printf("\n O conteúdo de p1: %d", *p1); // este valor deve ser o valor de x
printf("\n O conteúdo de p2: %d", *p2); // este valor deve ser o valor de y
getch();
*p1 = *p2; //armazena no endereço apontado por p1, o conteúdo
//apontado por p2, isto é equivalente a x = y .
printf("\n-----");
printf("\n O valor de x: %d", x);
printf("\n O valor de y: %d", y);
printf("\n O valor de z: %d", z);
printf("\n O valor de p1: %p", p1); // este valor deve ser o endereço de
//memória de x
printf("\n O valor de p2: %p", p2); // este valor deve ser o endereço de
//memória de y
printf("\n O conteúdo de p1: %d", *p1); // este valor deve ser o valor de x

//alterado, deve ser o mesmo valor que y
printf("\n O conteúdo de p2: %d", *p2); // este valor deve ser o valor de y
getch();
x = 10;
p1 = p2;
printf("\n-----");
printf("\n O valor de x: %d", x);
printf("\n O valor de y: %d", y);
printf("\n O valor de z: %d", z);
printf("\n O valor de p1: %p", p1); // este valor deve ser o endereço de
//memória de y
printf("\n O valor de p2: %p", p2); // este valor deve ser o endereço de
//memória de y
printf("\n O conteúdo de p1: %d", *p1);
printf("\n O conteúdo de p2: %d", *p2);
getch();
z = *p1;
*p2 = x;
printf("\n-----");
printf("\n O valor de x: %d", x);
printf("\n O valor de y: %d", y);
printf("\n O valor de z: %d", z);
printf("\n O valor de p1: %p", p1);
printf("\n O valor de p2: %p", p2);
printf("\n O conteúdo de p1: %d", *p1);
printf("\n O conteúdo de p2: %d", *p2);
getch();
}

```

**Observação:** %p indica que um endereço de um ponteiro em hexadecimal deve ser impresso, usando notação hexadecimal.

Vejamos a execução passo a passo do exemplo anterior:

(a)  $p1 = \&x;$   
 $p2 = \&y;$

(b)  $*p1 = *p2;$

Endereço físico	Conteúdo	Endereço lógico	Endereço físico	Conteúdo	Endereço lógico
FFF4	10	X	FFF4	20	x
FFF2	20	Y	FFF2	20	y
FFF0	0	Z	FFF0	0	z
FFEE	FFF4	p1	FFEE	FFF4	p1
FFEC	FFF2	p2	FFEC	FFF2	p2

(c)  $x = 10;$   
 $p1 = p2;$

(d)  $z = *p1;$   
 $*p2 = x;$

Endereço físico	Conteúdo	Endereço lógico	Endereço físico	Conteúdo	Endereço lógico
FFF4	10	x	FFF4	10	x
FFF2	20	y	FFF2	10	y
FFF0	0	z	FFF0	20	z
FFEE	FFF2	p1	FFEE	FFF2	p1
FFEC	FFF2	p2	FFEC	FFF2	p2

### 2.4.3 Tipo Base

No exemplo anterior, ao executar a instrução  $z = *p1$ , o compilador copia uma determinada quantidade de bytes em  $z$  do endereço apontado por  $p1$ . A questão é quantos bytes o compilador deve copiar?

A resposta é que o tipo base do ponteiro determina o tipo de dado para qual o ponteiro aponta. Baseado nesse tipo, o compilador copia a quantidade de bytes correta. No exemplo anterior, uma vez que  $p1$  é um ponteiro para um dado do tipo inteiro, 2 bytes de informação são copiados em  $z$  a partir do endereço apontado por  $p1$ .

Portanto, quando você declara um ponteiro para ser do tipo `int`, o compilador assume que qualquer endereço que ele armazena apontará para uma variável do tipo inteiro.

### 2.4.4 Expressões com apontadores

Em geral, expressões envolvendo ponteiros obedecem às mesmas regras que qualquer outra expressão na linguagem C, mas alguns aspectos especiais devem enfatizados.

Ponteiros são também, antes de tudo, variáveis. Logo, pode-se realizar operações com elas, isto é, como por exemplo,  $p = p + 10$ ; Entretanto, neste caso a operação  $+$  não é a operação usual de adição. Se o valor de  $p$  é 3000, na aritmética convencional o resultado da operação seria 3010. Quando um inteiro é adicionado ou subtraído de um ponteiro, o valor do ponteiro não é aumentado ou diminuído por aquele inteiro, mas sim por aquele inteiro vezes o tamanho do objeto apontado pelo apontador.

Por exemplo, supondo que um inteiro ocupa 2 bytes, a operação  $p = p + 10$ ; quando  $p$  vale 3000, resultaria em algo como  $3000 + 2 \cdot 10$ .

### 2.4.4.1 Atribuindo valores para os pontadores

Como em qualquer variável, um ponteiro pode ser usado no lado direito de uma declaração para atribuir seu valor para outro ponteiro. Da mesma forma que ocorre nas atribuições em variáveis comum, deve-se observar o tipo de dado para qual a variável do tipo ponteiro aponta. Por exemplo, um ponteiro para inteiro, deve receber apenas endereços de variáveis do tipo inteiro.

Para atribuir o endereço de uma variável para um apontador utiliza-se o operador `&` antes da variável.

#### SINTAXE:

**<Nome do Pontoire> = & <Nome da Variável>;**

em que:

**Nome do Pontoire:** variável do tipo ponteiro.

**Nome da Variável:** variável para qual o ponteiro vai apontar.

Para atribuir o conteúdo apontado por ponteiro para uma variável utiliza-se o operador `*` antes da variável. Essa forma de obter o conteúdo de uma variável apontada por um ponteiro é dita indireta.

#### SINTAXE:

**<Nome da Variável > = \* <Nome do Pontoire >;**

Pode-se igualar dois ponteiros. Ao fazer essa atribuição, os dois ponteiros envolvidos estarão apontando para o mesmo lugar.

#### SINTAXE:

**<Nome do Pontoire 1 > = <Nome do Pontoire 2 >;**

Pode-se também atribuir o conteúdo de um apontador ao conteúdo de outro.

#### SINTAXE:

**\*<Nome do Pontoire 1 > = \*<Nome do Pontoire2 >;**

**Observação:** Depois que se aprende a usar os dois operadores (`&` e `*`) fica fácil entender operações com ponteiros.

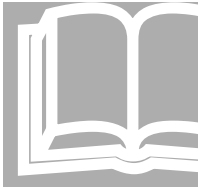
### 2.4.4.2 Inicialização de um ponteiro

Para atribuir um valor a um ponteiro recém-criado pode-se igualá-lo a um valor de memória. Entretanto, como saber a posição na memória de uma

variável de cada programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho. Para saber o endereço de uma variável usa-se o operador & antes das variáveis já declaradas.

A inicialização de um apontador, geralmente, é com o valor NULL, que é equivalente ao zero.

### Exemplo:



```
// Programa : Exemplo 4
// Objetivo: Mostrar as atribuições em ponteiros
#include <stdio.h>
#include <stdlib.h>
main (void )
{
    int num1, num2;
    int *pint, *pqq, *pnum1;
    char caracter;
    char *pchar;
    float teste = 0;
    system("cls");

    //Atribuições permitidas
    pint = &num1;
    pqq = &num2;
    pchar = &caracter;
    num2 = *pint;
    pqq = pint;
    *pqq = *pnum1;
    //Atribuições NÃO permitidas
    pint = &caracter;
    pchar = &pint;
    num2 = *pchar;
    pqq = pchar;
    *pqq = *pchar;
}
```

#### 2.4.4.3 Incremento e decremento de ponteiros

Quando se incrementa um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se um ponteiro para um inteiro é incrementado, ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo base de um ponteiro.

Por exemplo, ao incrementar um ponteiro char\* ele anda 1 byte na memória, ao incrementar um ponteiro double\* ele anda 8 bytes na memória. O decremento funciona da mesma forma.

### SINTAXE:

**<Nome do Ponteiro>++; //incrementa**

**OU**

**<Nome do Ponteiro> -- ; //decrementa**

### Observações:

1. O incremento e o decremento de ponteiros são utilizados principalmente em matrizes.
2. Os operadores de incremento ou decremento têm prioridade sobre o operador \*.

Ao realizar operações de incremento e decremento com ponteiros, é modificado o endereço para qual eles apontam. Para realizar operações com o conteúdo das variáveis para as quais eles apontam deve se utilizar o operador \* de acordo com a seguinte sintaxe:

### SINTAXE:

**(\*<Nome do Ponteiro>)++;**  
**//incrementa o conteúdo da variável apontada pelo ponteiro**  
**ou**  
**(\*<Nome do Ponteiro>)--;**  
**//decrementa o conteúdo da variável apontada pelo ponteiro**

Cuidado, considerando que p é um apontador, ao executar instruções como \*p++ ; não é incrementado o valor do elemento apontado por p. Nesta instrução, primeiro incrementa o valor de p (isto é, muda o elemento apontado) e depois referencia o valor apontado.

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Se quiser incrementar um ponteiro em 15. Basta fazer:

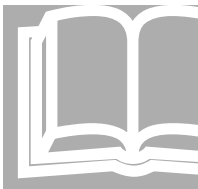
**p=p+15; ou p+=15;**

Se quiser usar o conteúdo do ponteiro 15 posições adiante:

**\*(p+15);**

Entretanto, há operações que você NÃO pode efetuar em um ponteiro. Essas operações são: dividir ou multiplicar, adicionar dois ponteiros, adicionar ou subtrair float ou double para ou de ponteiros e aplicar o deslocamento de bits e operadores de máscaras em ponteiros.

### Exemplo:



```
// Programa : Exemplo 5
// Objetivo: Mostrar as operações de incremento e decremento em ponteiros
// ESTE EXEMPLO CONSIDERA QUE UM DADO DO TIPO INT OCUPA 2
// BYTES NA MEMÓRIA
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main (void )
{
    int y=10, x=20;
    int *p;
    system("cls");
    p = &x;                // o ponteiro p aponta para x
    printf("\n O valor apontado por p: %d", *p);
    p++;                   // o ponteiro p aponta para os
                           // próximos dois bytes na memória e
                           // passa a apontar para y.
    printf("\n O valor apontado por p: %d", *p);
}
```

```

p--; // o ponteiro p aponta para dois bytes na memória anterior,
//no caso dois bytes e passa a apontar para x.
(*p)++; // incrementa o conteúdo da variável apontada por p,
//no caso, incrementa o valor de x.
printf("\n O valor de x: %d", x);
printf("\n O valor de p: %p", p);
printf("\n O valor apontado por p: %d", *p);
*p++; // incrementa o endereço de p em dois bytes,
// e referencia o valor de y
printf("\n O valor de y: %d", y);
printf("\n O valor de p: %p", p);
printf("\n O valor apontado por p: %d", *p);
getche();
}

```

Vejamos a execução passo a passo do exemplo anterior:

(b) p = &x;			(b) p++;		
Endereço físico	Conteúdo	Endereço lógico	Endereço físico	Conteúdo	Endereço lógico
FFF4	10	y	FFF4	10	y
FFF2	20	x	FFF2	20	x
FFF0	FFF2	p	FFF0	FFF4	p

(c) p--;			(d) (*p)++;		
Endereço físico	Conteúdo	Endereço lógico	Endereço físico	Conteúdo	Endereço lógico
FFF4	10	y	FFF4	10	y
FFF2	20	x	FFF2	21	x
FFF0	FFF2	p	FFEE	FFF2	p

(e) *p++;		
Endereço físico	Conteúdo	Endereço lógico
FFF4	10	y
FFF2	21	x
FFF0	FFF4	p

#### 2.4.4.4 Operações Relacionais (Comparação) com ponteiros

Uma operação, às vezes útil, é a comparação entre dois ponteiros. A operação relacional mais usuais são a de igualdade (==) e a de diferença (!=), utilizadas para verificar se o endereço de dois ponteiros são iguais ou diferentes. Utiliza-se essas operações principalmente para verificar se o valor do ponteiro é igual ou diferente de nulo.

No caso dos outros operadores relacionais do tipo >, <, >= e <=, compara-se qual ponteiro aponta para uma posição mais alta na memória. Então uma comparação entre ponteiros pode informar qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer. Essas operações não são muito utilizadas.

## 2.5 Apontadores em funções

Em muitas situações é conveniente que uma função altere o valor de variáveis declaradas em uma outra função. Lembre-se que uma função não pode fazer referência direta a uma variável declarada em uma outra função

(entre outras palavras, ela não tem conhecimento das variáveis declaradas em outra função). Portanto ela não pode alterar o valor de tal variável diretamente.

Um mecanismo que possibilita que uma função f1 altere o valor de uma variável x declarada em outra função f2 é a chamada de funções passando argumentos por referência. Nesse caso, a função f1 precisa ter um parâmetro declarado de forma especial. Esta declaração especial indica que o conteúdo da variável passada no lugar desse parâmetro, quando f1 é chamada, poderá ser alterado por f1.

A chamada por referência pode ser contraposta à chamada por valor. Na chamada de função por valor, os valores dos argumentos, passados para a função na chamada, são copiados para as variáveis dos respectivos parâmetros. Qualquer modificação no conteúdo do parâmetro não altera o valor de nenhuma variável da função que a chamou.

Na chamada por referência, para que uma função altere o valor de uma variável declarada em outra função, é necessário entender endereços de variáveis e como alterar o valor em uma certa posição de memória.

Lembre-se que variáveis correspondem a uma área da memória do computador, e essa área possui “endereço” (na verdade, um número). Ao passar o endereço de memória de uma variável para uma função, pode-se alterar o seu valor, ou seja, para alterar variáveis que não são da função deve-se passar um ponteiro como argumento.

Para indicar que um argumento da função é um ponteiro, basta utilizar o operador \* antes da variável na sua prototipação. Ao chamar a execução deve-se então passar um endereço e não o valor da variável, então se utiliza o operador &.

### Exemplo:



```
// Programa : Exemplo 6
// Objetivo: Mostrar o uso de apontadores em funções
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

//Prototipação das Funções
void testando1(int a) ;
void testando2(int *a) ;
main (void )
{
    int x = 7;
    system("cls");
    printf("\nValor utilizando a funcao testando1\n\n");

    printf("\nValor de x antes da chamada a funcao testando = %d\n", x);
    testando1(x) ;
    printf("\n Valor de x depois da chamada a funcao testando = %d\n", x);
    getch( );
    printf("\n\nValor utilizando a funcao testando2\n\n");
    printf("\n Valor de x antes da chamada a funcao testando = %d\n", x);
    testando2(&x) ;
    printf("\n Valor de x depois da chamada a funcao testando = %d\n", x);
    getch( );
}

//Especificação do corpo das funções
void testando1(int a)
```

---

```
{
    a = a * a ;
}
void testando2(int *a)
{
    (*a) = (*a) * (*a) ;
}
```

Concluindo, na passagem de variáveis “comuns” como argumento para uma função, pode-se passar o :

✚ **Valor da variável:** Neste caso, as alterações feitas dentro da função poderão alterar o valor da cópia e não da variável original.

ou

✚ **Endereço da variável:** Neste caso, se o parâmetro for um apontador (do tipo `int *p`, por exemplo), então você pode alterar o valor da variável original, alterando o conteúdo da posição de memória cujo endereço é aquele que você passou para a função (isto é, se você faz `*p = 0`, você está atribuindo 0 à variável original. Note que `*p = 0` e `p=0` são diferentes, e se `p` é um apontador, então muito provavelmente a segunda atribuição vai resultar em algum tipo de erro).

Exemplo: Um ponteiro é usado para receber "resposta" de uma função. A função calcula o quociente e o resto da divisão de `m` por `n`.

**// Programa : Exemplo 7**

**// Objetivo: Mostrar o uso de apontadores em funções**

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

**//Prototipação da Função**

```
void quocienteresto(int m, int n, int *pq, int *pr);
main (void )
```

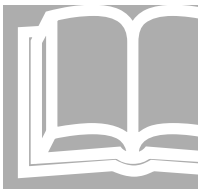
```
{
    int a, b, q, r;
    system("cls");
    printf("Digite dois numeros inteiros: ");
    scanf("%d %d", &a, &b);
    quocienteresto(a, b, &q, &r);
    printf("\nO quociente = %d e o resto = %d", q, r);
    getch( );
}
```

**//Especificação do corpo das funções**

```
void quocienteresto(int m, int n, int *pq, int *pr)
{
    *pq = m / n;
    *pr = m % n;
}
```

## 2.6 Apontadores para Vetores

Ao declarar um vetor também estamos manipulando ponteiros. Por exemplo, ao declarar um vetor, além de reservar uma determinada quantidade





de espaço para esse vetor, estamos definindo um ponteiro indiretamente. Por exemplo:

**int Num[10];**

Na verdade Num é um apontador para inteiro. O valor da variável Num é igual ao endereço do primeiro elemento do vetor, ou seja, igual a &Num[0]. Como Num é um ponteiro, referencia-se os valores do vetor indexando através de operações com ponteiros.

Por exemplo, para saber o valor da primeira posição utiliza-se Num[0], da mesma forma pode-se manipular os dados de um vetor utilizando as operações com apontadores, ou seja pode utilizar \*(Num+0).

Assim, qualquer operação que possa ser realizada com índices de um vetor ou uma matriz pode ser feita com apontadores.

**vetor [índice] <=> \*(vetor + índice)**

Para referenciar o endereço de um elemento do vetor:

**&vetor [índice] <=> (vetor + índice)**

Portanto, quando vetores ou matrizes são passadas para uma função, o que é passado são os respectivos endereços. Vetores e matrizes são tratados como ponteiros. Logo, para vetores e matrizes só é possível realizar uma passagem por referência. Isto significa que qualquer alteração no valor do vetor ou da matriz dentro da função alterará o vetor ou a matriz original.

### Exemplo:



```
// Programa : Exemplo 8
// Objetivo: Mostrar a manipulação de vetores através de ponteiros
// Dados de entrada: Um vetor com 10 elementos
// Dados de saída: Um vetor com 10 elementos com valores iguais a zero
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define MAX 10
void zera_vetor(int *p, int tamanho);
//void zera_vetor(int p[0], int tamanho);
main (void )
{
    int i,
        vetor[MAX],           // declara um vetor com 10 de inteiros
        *pvetor=NULL;         // declara 1 variável do tipo ponteiro para int
    system("cls");
    pvetor = vetor;           // pvetor = &vetor[0];
    for(i=0;i<10;i++)
        *(pvetor+i) = i+1;    // pvetor[i] = i + 1;
    // vetor[i] = i + 1;
    // *(vetor + i) = i + 1;
    for(i=0;i<MAX;i++)
        printf("\n O conteudo de pvetor[%d] = %d" , i, *(pvetor+i) );
    zera_vetor (pvetor, MAX);
    getch();
}//fim da função main()
```

```
//Especificação do corpo das funções
void zera_vetor(int *p, int tamanho)
{
    int i;
    for(i=0;i<tamanho;i++)
        *(p+i) =0;           //p[i] = 0;
                               // *p = 0; p++;
}
```

## 2.7 Apontador para Estrutura

A linguagem C permite ponteiros para estruturas da mesma forma que aceita ponteiros para qualquer outro tipo de variável. Contudo, existem alguns aspectos especiais:

### SINTAXE DA DECLARAÇÃO:

**struct Nome\_Estrutura \* NomePonteiro;**

OU

Se a estrutura estiver definida como tipo:

**Nome\_Estrutura \* NomePonteiro;**

em que:

✚ **Nome\_Estrutura:** nome da estrutura

✚ **NomePonteiro:** ponteiro que irá receber endereços de estruturas definidas em Nome\_Estrutura

Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C. Para usá-lo, há duas possibilidades. A primeira é apontá-lo para uma variável struct já existente.

Para acessar uma informação em um membro da estrutura segue a seguinte sintaxe:

**( \*NomePonteiro) . membro**

**Observação:** Os parênteses são necessários, porque o operador '.'(ponto) tem precedência maior que o operador \*. Porém, este formato não é muito usado. O que é comum de se fazer é acessar o membro de uma estrutura através do operador seta, que é formado por um sinal de "menos" (-) seguido por um sinal de "maior que" (>), isto é: ->

**NomePonteiro -> membro**

**Exemplo:**

```
// Programa : Exemplo 9
// Objetivo: Mostrar a manipulação de estruturas através de ponteiros
// Dados de entrada: Informações de um aluno
```



```
// Dados de saída: As notas com valores iguais a zero
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct alunos {
    char nome[40];
    float nota1;
    float nota2;
    float media;
} NOTAS_ALUNOS;

//Protótipo da função
void zera_estrutura(NOTAS_ALUNOS *p);

main (void)
{
    NOTAS_ALUNOS aluno;
    NOTAS_ALUNOS *pestrutura;
    pestrutura = &aluno;
    system("cls");
    printf("Cadastro de aluno\n");
    printf("Informe o nome .....: "); gets (pestrutura->nome)
    printf("Informe a 1a. nota .....: "); scanf ("%f", &pestrutura->nota1);
    printf("Informe a 2a. nota .....: "); scanf ("%f", &pestrutura->nota2);
    pestrutura->media = (aluno.nota1+ aluno.nota2 )/2.0;
    printf("Nome.....: %s \n" , pestrutura->nome);
    printf("Nota 1.....: %5.2f \n" , pestrutura->nota1);
    printf("Nota 2.....: %5.2f \n" , pestrutura->nota2);
    printf("Media.....: %5.2f \n" , pestrutura->media);
    getch( );
    zera_estrutura(pestrutura); //zera_estrutura(&aluno);
    printf("-----Após a chamada da função-----\n");
    printf("Nome.....: %s \n" , pestrutura->nome);
    printf("Nota 1.....: %5.2f \n" , pestrutura->nota1);
    printf("Nota 2.....: %5.2f \n" , pestrutura->nota2);
    printf("Media.....: %5.2f \n" , pestrutura->media);
    getch( );
} // fim main

void zera_estrutura(NOTAS_ALUNOS *p)
{
    strcpy(p->nome,""); //strcpy((*p).nome,"");
    p->nota1 = 0.0; //(*p).nota1 = 0.0;
    p->nota2 = 0.0; //(*p).nota2 = 0.0;
    p->media = 0.0; // (*p).media = 0.0;
}
```

## 2.8 Apontador para Apontador

Um ponteiro pode guardar o endereço de qualquer tipo de dado existente no programa, inclusive o endereço de outro ponteiro. Um ponteiro para ponteiro é como se anotasse o endereço de um papel que tem o endereço da informação que precisamos.

A forma geral para declarar um ponteiro para um ponteiro é:

**SINTAXE:**

**<Tipo Base> \*\* <NomePonteiro>;**

Algumas considerações:

**\*\*NomePonteiro** é o conteúdo final da variável apontada;

**\*NomePonteiro** é o conteúdo do ponteiro intermediário.

Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes.

Na linguagem C pode-se declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros, e assim por diante. Para fazer isto basta aumentar o número de asteriscos na declaração. A lógica é a mesma.

A utilização de ponteiro para ponteiro é mais comum quando deseja-se passar o endereço de um ponteiro para uma função.

### Exemplo



```
// Programa : Exemplo 10
// Objetivo: Mostrar a utilização de ponteiro para ponteiro
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

main (void )
{
    float fpi = 3.1415, *pf, **ppf;
    system("cls");
    pf = &fpi;           // pf armazena o endereço de fpi
    ppf = &pf;           // ppf armazena o endereço de pf
    printf("%f", **ppf); // Imprime o valor de fpi
    printf("\n%f", *pf); // Também imprime o valor de fpi
    getch( );
}
```

## 2.9 Alocação Dinâmica de Memória

A alocação dinâmica permite ao programador alocar memória para variáveis quando o programa está sendo executado. Assim, poderemos definir, por exemplo, um vetor ou uma matriz cujo tamanho descobriremos em tempo de execução. O padrão C ANSI define 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca `stdlib.h`.


### 2.9.1 Alocar memória

Para alocar memória pode-se utilizar a função `calloc`.

**SINTAXE:**

**void \* calloc (Num\_Elementos, Tamanho\_Elemento);**

em que:

 **Num\_Elementos** : é um número inteiro sem sinal que define quantidade de elementos para qual serão alocados memória.

- + **Tamanho\_Elemento:** é um número inteiro sem sinal que define o número de bytes de cada elemento. Para saber o número de bytes de um tipo de dado, utiliza-se a função **sizeof(tipo\_de\_dado)**. Esta função retorna o número de bytes do tipo\_de\_dado passado como argumento.
- + **void \*:** este é o tipo de retorno da função **calloc**, um ponteiro do tipo **void**. Este ponteiro pode ser atribuído a qualquer tipo ponteiro.

### A função **calloc** retorna:

- + o **ENDEREÇO** para o primeiro byte alocado.
- ou
- + o valor **NULL**, quando não foi possível alocar memória.

Importante:

- + O retorno da função deve ser armazenado em uma variável do tipo ponteiro.
- + A função **calloc** aloca um bloco de bytes consecutivos e inicializa-os com valor nulo.

### CHAMADA DA FUNÇÃO:

**Ponteiro = ( tipo \*) calloc (Num\_Elementos, Tamanho\_Elemento);**

Como a função **calloc** devolve um ponteiro do tipo **void \***, é preciso moldar o resultado (**tipo \***), ou seja, este é o papel do molde ( = cast) que precede **calloc**.

### 2.9.2 Liberar memória alocada

Ao alocar memória dinamicamente é necessário liberá-la quando ela não for mais necessária. Para isto existe a função **free()** cujo protótipo é:

#### SINTAXE:

**void free (void \* ponteiro);**

em que:

- + **Ponteiro :** é o endereço do primeiro byte alocado.

### CHAMADA DA FUNÇÃO:

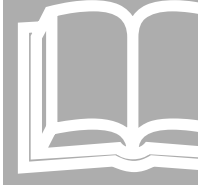
**free( ponteiro );**

Na chamada da função para **free( )**, basta passar o ponteiro que aponta para o início da memória alocada. Uma questão comum é como que o programa vai saber quantos bytes devem ser liberados? Ele sabe, pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna.

### 2.9.3 Alocação Dinâmica de Vetores

A alocação dinâmica de vetores utiliza os conceitos aprendidos sobre ponteiros e as funções de alocação dinâmica apresentados. Um exemplo de implementação para vetor de inteiros é fornecido no exemplo seguinte.

#### Exemplo:



```
// Programa : Exemplo 11
// Objetivo: Mostrar a alocação de memória de vetores, fazendo uso de
// chamada
// de função
// Dados de entrada: O número de elementos de um vetor
// Dados de saída: Os elementos do vetor inicializados
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // Para usar calloc( ), free( ) e exit( )
int * Aloca_Vetor(int tam);
main (void )
{
    int n, *pveter, i;
    system("cls");
    printf("Digite o número de elementos do vetor:");
    scanf("%d", &n);
    pveter = Aloca_Vetor (n);

    // pveter pode ser tratado como um vetor com n posições

    for (i=0; i < n; i++) {
        pveter[i] = i*i;                // *(pveter + i) = i*i ;
        printf("pveter[%d] = %d\n", i, pveter[i]);
    }
    free (pveter);
    getch();
}

int * Aloca_Vetor(int tam)
{
    int * v;                // ponteiro para o vetor
    v = (int *) calloc(tam, sizeof(int) );

    //Aloca tam números inteiros; v pode agora ser tratado como um vetor
    //com tam posições.

    if (!v)                //(v == NULL)
    {
        printf ("** Erro: Memória Insuficiente **");
        exit(1);
    }
    return v;                // retorna o ponteiro para o vetor
}
```

No exemplo anterior, é alocada memória suficiente para se colocar **n** números inteiros.

A função **sizeof( )** retorna o número de bytes de um inteiro. Ela é útil para saber o tamanho de tipos. O ponteiro **void \*** que **calloc( )** retorna é convertido para um **int \*** pelo *cast* e é atribuído a **pveter**.

A declaração seguinte testa se a operação foi bem sucedida.

Se não tiver sido, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o

vetor de inteiros alocados normalmente, por exemplo, indexando-o de **pvetor[0]** a **pvetor[(n-1)]** ou através da indexação de ponteiros equivalente.

A chamada a **free (pvvetor)**, avisa ao sistema que o bloco de bytes apontado por **pvetor** está livre.

**Exemplo:** Ler um vetor de inteiros, fazer a alocação dinâmica e informar o valor máximo do vetor, usando função.



```
// Programa : Exemplo 12
// Objetivo: Ler um vetor de inteiros e calcular seu valor máximo.
// Dados de entrada: O tamanho e os elementos do vetor
// Dados de saída: O valor máximo do vetor de entrada
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // Para usar calloc ( ), free ( ) e exit ( )
int max (int *v, int n);
int * Aloca_Vetor(int tam);
main (void )
{
    int tam,
    *pvvetor, // sem espaço alocado
    i;
    system("cls");
    printf("Digite o número de elementos do vetor:");
    scanf("%d", &tam);
    pvvetor = Aloca_Vetor (tam);

    // pvvetor pode ser tratado como um vetor com tam posições

    for (i=0; i < tam; i++) {
        printf("pvvetor[%d] = ", i);
        scanf ("%d", &pvvetor[i]); // scanf ("%d", (pvvetor + i));
    }

    //Informa o resultado
    printf(" O valor maximo do vetor eh: %d\n", max(pvvetor, tam));
    free (pvvetor);
    getch();
}

int max (int *v , int n)
{
    int i, maior;
    maior = v[0];
    for (i= 0; i < n; i++)
        if ( v[i] >maior )
            maior = v[i];

    return maior;
}

int * Aloca_Vetor(int tam)
{
    int * v; // ponteiro para o vetor
    v = (int *) calloc(tam, sizeof(int) );

    //Aloca tam números inteiros; v pode agora ser tratado como um vetor
    //com tam posições.

    if (!v) // (v == NULL)
    {
        printf ("** Erro: Memória Insuficiente **");
        exit(1);
    }
}
```

```

        return v;        // retorna o ponteiro para o vetor
    }

```

#### 2.9.4 Alocação Dinâmica de Matrizes

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, ou seja, é um ponteiro para ponteiro, o que é denominada indireção múltipla.

Conforme já vimos, a indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro.

Assim, qualquer operação que possa ser realizada com índices de uma matriz pode ser feita com apontadores.

**matriz [índice1][índice2] ⇔ (\*(matriz + índice1) + índice2)**

Para referenciar o endereço de um elemento da matriz:

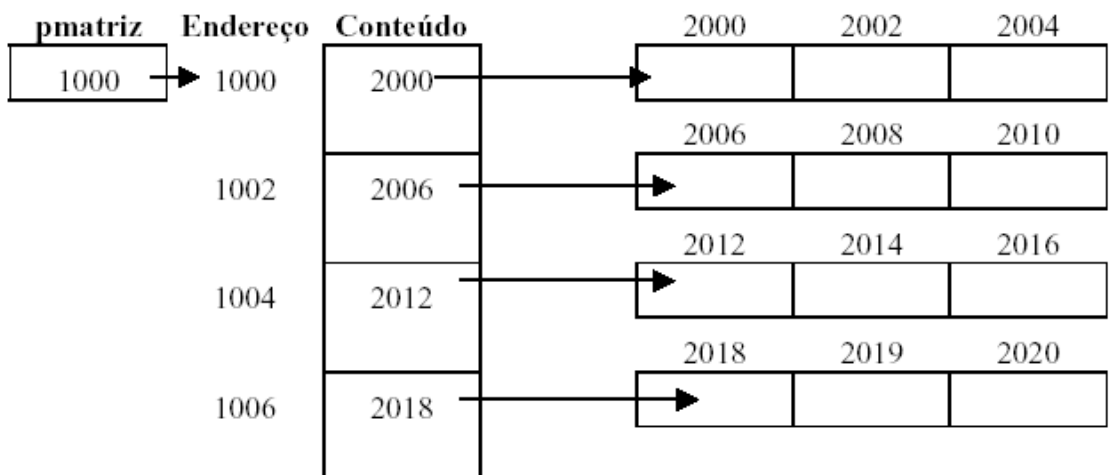
**&matriz [índice1][índice2] ⇔ \*(matriz + índice1) + índice2)**

##### 2.9.4.1 Implementação Versão 1

A alocação dinâmica de matriz bidimensional é uma aplicação de ponteiro para ponteiro. Existem maneiras diferentes de se realizar esta implementação. Apresentaremos agora, o que chamamos de **versão 1**.

A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz, ou seja, as linhas da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, compondo o segundo índice ou a coluna da matriz.

**Exemplo:** Memória alocada para uma matriz de 4 linhas e 3 colunas.







// Programa : **Exemplo 13 – Versão 1**

// Objetivo: Mostra a alocação de memória para uma matriz bidimensional  
 // Dados de entrada: Dimensões da matriz: número de linhas e de colunas  
 // Dados de saída: A matriz inicializada

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // Para usar calloc( ), free( ) e exit( )
int ** Aloca_Matriz (int lin, int col);
void Libera_Matriz (int **mat, int l);
void LeMatriz (int **mat, int lin, int col, char *nome_mat);
void MostraMatriz (int **mat, int lin, int col, char *nome_mat);
main(void)
{
    int **pmatriz,          // Matriz de inteiros
        linha,
        coluna;
    system("cls");
    printf("Digite o número de linhas dimensoes da matriz: ");
    scanf("%d", &linha);
    printf("Digite o número de colunas dimensoes da matriz: ");
    scanf("%d", &coluna);
    pmatriz = Aloca_Matriz (linha, coluna);
    LeMatriz (pmatriz, linha, coluna, "matriza");
    MostraMatriz (pmatriz, linha, coluna, "PMATRIZ");
    Libera_Matriz (pmatriz, linha);
    getch();
}

// fim main()

int ** Aloca_Matriz (int lin, int col)
{
    int ** mat;
    int i, k;
    mat = (int **) calloc(lin, sizeof(int *)); // aloca um vetor com linha
    // posições para armazenar
    // ponteiros inteiros
    if (mat == NULL) {
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    for (i=0; i<lin; i++)
    {
        mat[i]= (int *)calloc(col, sizeof (int)); // aloca um vetor com
        // coluna
        // posições para inteiros, em que o endereço é atribuído a
        // cada linha da matriz
        if (mat[i] == NULL) {
            printf("Memoria insuficiente.\n");
            for (k = i-1; k >= 0; k--) // devolve os blocos alocados
                free(mat[k]);
            free(mat);
            exit (-1); // abandona a tarefa
        } // fim if
    } // fim for
    return mat;
}

void Libera_Matriz(int **mat, int lin)
{
    int i;
    for (i=0; i<lin; i++)
```

```

    free(mat[i]); // libera o vetor de inteiros com coluna posições para
                  // cada linha
    free(mat); // libera o vetor de ponteiros com linha posições
}

```

```

void LeMatriz (int **mat, int lin, int col, char *nome_mat)
{
    int i, j;
    printf("\n Digite os elementos da matriz %s \n", nome_mat);
    for (i=0; i<lin; i++)
        for (j=0; j<col; j++)
        {
            printf("%s[%d][%d]:", nome_mat, i, j );
            scanf("%d", &mat[i][j]);
        }
}

```

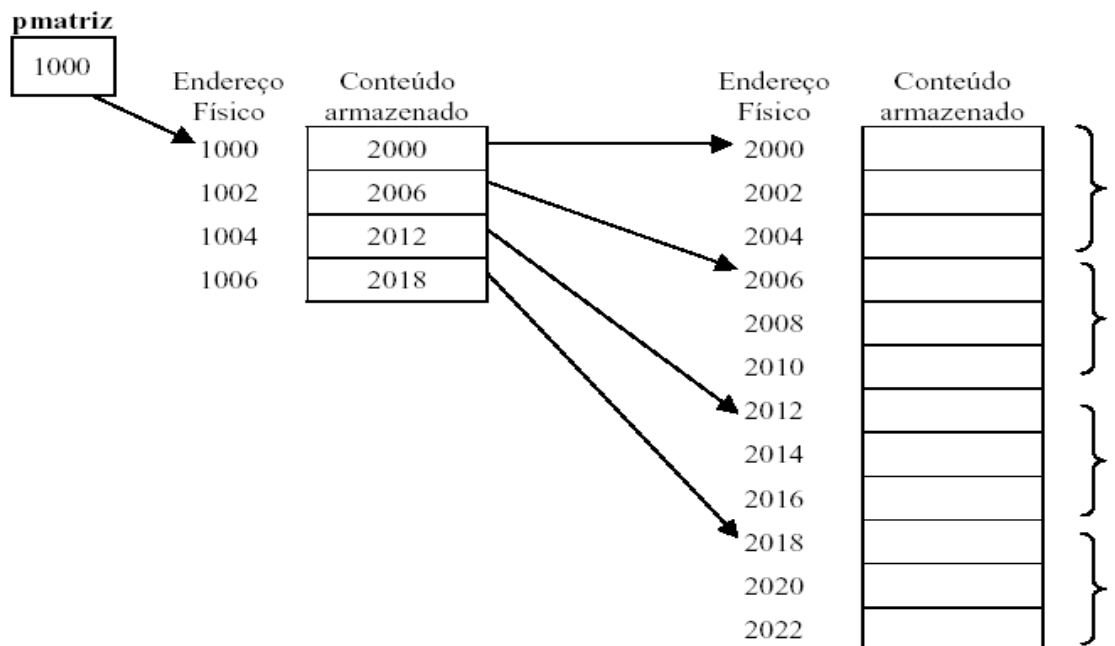
```

void MostraMatriz (int **mat, int lin, int col, char *nome_mat)
{
    int i, j;
    printf("\n Os elementos da matriz %s sao: \n", nome_mat);
    for (i=0; i<lin; i++)
    {
        for (j=0; j<col; j++)
            printf("%d ", mat[i][j]); //printf("%d ", *(mat+i+j) );
        printf("\n");
    }
}

```

#### 2.9.4.2 Implementação Versão 2

A estrutura de dados utilizada nesta versão 2 é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz, ou seja, as linhas da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz, da mesma forma da versão 1. A diferença está na composição dos elementos, aqui se aloca dinamicamente um único vetor, com dimensões iguais ao primeiro índice vezes o segundo índice da matriz, ou seja, linha\* coluna da matriz.





// Programa : **Exemplo 14 – Versão 2**

// Objetivo: Mostra a alocação de memória para uma matriz bidimensional,  
// Dados de entrada: Dimensões da matriz: número de linhas e de colunas  
// Dados de saída: A matriz inicializada

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // Para usar calloc( ), free( ) e exit( )
int ** Aloca_Matriz (int lin, int col);
void Libera_Matriz (int **mat);
void LeMatriz (int **mat, int lin, int col, char *nome_mat);
void MostraMatriz (int **mat, int lin, int col, char *nome_mat);

main (void)
{
    int **pmatriz, linha, coluna;
    system("cls");
    printf("Digite o número de linhas dimensoes da matriz: ");
    scanf("%d", &linha);
    printf("Digite o número de linhas dimensoes da matriz: ");
    scanf("%d", &coluna);
    pmatriz = Aloca_Matriz (linha, coluna);
    LeMatriz (pmatriz, linha, coluna, "PMATRIZ");
    MostraMatriz (pmatriz, linha, coluna, "PMATRIZ");
    Libera_Matriz (pmatriz);
    getch();
} // fim main

int ** Aloca_Matriz (int lin, int col)
{
    int *posicao, **mat, i;
    mat = (int **) calloc ( lin, sizeof(int *) ); /* um vetor de lin ponteiros para
                                                    inteiro */
    if (mat == NULL) {
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    posicao = (int *) calloc ( lin*col, sizeof(int) ); /* um vetor de
                                                         lin*col para inteiros */
    if (posicao == NULL) {
        free(mat);
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    for (i=0; i<lin; i++) {
        mat[i]= &posicao[i*col]; //faz com que cada linha aponte para
                                // o
    } //inicio dos elementos da coluna
    return mat;
}

void LeMatriz (int **mat, int lin, int col, char *nome_mat)
{
    int i, j;
    printf("\n Digite os elementos da matriz %s \n", nome_mat);
    for (i=0; i<lin; i++)
        for (j=0; j<col; j++)
        {
            printf("%s[%d][%d]:", nome_mat, i, j );
            scanf("%d", &mat[i][j]);
        }
}
```

```
void MostraMatriz (int **mat, int lin, int col, char *nome_mat)
{
    int i, j;
    printf("\n Os elementos da matriz %s sao: \n", nome_mat);
    for (i=0; i<lin; i++)
    {
        for (j=0; j<col; j++)
            printf("%d ", mat[i][j]); //printf("%d ", *(mat+i+j) );
        printf("\n");
    }
}


void Libera_Matriz (int **mat)
{
    free(mat[0]); //libera o vetor "posição" de inteiros
    free(mat); //libera o vetor de ponteiros p/inteiros
}
```

## 2.10 Cuidados com a manipulação de apontadores

Para fazer programas utilizando-se apontadores é necessário precaver-se, pois os apontadores estão manipulando endereços de memória e assim um erro simples pode invadir uma área da memória que é desconhecida.

Antes de usar um ponteiro, deve-se sempre saber *para onde* o ponteiro está apontando. Isto significa: **nunca** use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como **não** usar um ponteiro:

**Exemplo:**

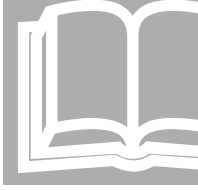


```
// Programa: Exemplo 15a
// Objetivo: Mostrar como NÃO utilizar ponteiros
// Dados de entrada: Nenhum
// Dados de saída: Nenhum
#include <stdio.h>
#include <stdlib.h> // Para usar calloc( ) e free( )
main (void) //Errado ! – Não execute
{
    int x, *p;
    x = 13;
    *p = x;
}
```

Este programa compilará e executará. O que acontecerá? Ninguém sabe. O ponteiro *p* pode estar apontando para qualquer lugar. **Você estará gravando o número 13 em um lugar desconhecido.** Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

Ao manipular matriz através de apontadores, **cuidar para não invadir espaço de outras variáveis:**

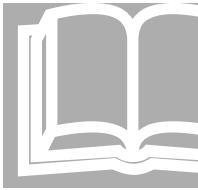
**Exemplo:**



```
// Programa : Exemplo 15b
// Objetivo: Mostrar como NÃO utilizar ponteiros
#include <stdio.h>
#include <stdlib.h>           // Para usar calloc( ) e free( )
main (void) //Errado ! – Não execute
{
    int i,
    vetor[10],                // declara um vetor com 10 de inteiros
    *pvetor = NULL;          // declara 1 variável do tipo ponteiro para int
    system("cls");
    pvetor = vetor;
    for(i=0; i<100; i++) // até 100????
        *(pvetor+i) = i+1; //pvetor[i]=i+1;
}
```

Outro cuidado é com **estruturas que possuem membros do tipo apontador**. Considerando o exemplo seguinte. Ao atribuir p1 a p2, os valores de cada membro de p1 é copiado para os respectivos membros de p2. Quanto ao membro idade, não há problemas, porém o membro nome do apontador p2 possui o mesmo endereço do membro nome apontador p1. Isso significa que o membro nome de p2 está apontando para o membro nome de p1.

### Exemplo:



```
// Programa : Exemplo 15c
// Objetivo: Mostrar como NÃO utilizar ponteiros
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct {
    char nome[40];
    int idade;
}PESSOA;
main (void)
{
    PESSOA aluno1, aluno2,
    *p1, *p2;
    //inicializar valores para aluno1 e aluno2
    p1 = &aluno1;
    p2 = p1;
    printf ("Aluno1:");
    gets(p1->nome);
    printf ("Nome.....: %s \n" , p1->nome);
    printf ("Idade :");
    scanf("%d", &p1->idade);
    printf ("Idade.....: %d \n" , p1->idade);
    printf ("Aluno2:");
    printf ("Nome.....: %s \n" , p2->nome);
    printf ("Idade.....: %d \n" , p2->idade);
    //atualizar os dados do aluno2
    strcpy(p2->nome, "XXXXXXX");
    p2->idade = 20;
    printf ("Aluno1:");
    printf ("Nome.....: %s \n" , p1->nome);
    printf ("Idade.....: %d \n" , p1->idade);
    printf ("Aluno2:");
    printf ("Nome.....: %s \n" , p2->nome);
    printf ("Idade.....: %d \n" , p2->idade);
}
```

```
system("pause");  
} // fim main
```

## 2.11 Os argumentos *argc* e *argv*

A função **main()** pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função **main()** é:

```
int main ( int argc, char *argv[ ] );
```

Os parâmetros *argc* e *argv* dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O **argc (argument count)** é um inteiro e possui o número de argumentos com os quais a função **main()** foi chamada na linha de comando. Ele é, no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

O **argv (argument values)** é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O *argv[0]* sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em *argv* que temos *argc*.

**Exemplo:** Escreva um programa que faça uso dos parâmetros *argv* e *argc*. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame *exemplo*:

*exemplo 19 04 99*

O programa deverá imprimir:

*19 de abril de 1999*

// Programa: **Exemplo16**

### 3 Estrutura de Dados

*Durante a execução do programa os dados são armazenados na memória do computador. As localizações de memória em que esses dados são armazenados se denominam estruturas de dados.*

As estruturas de dados são organizadas conforme os tipos de dados disponíveis na linguagem de programação, de forma a representar o mais fielmente possível os tipos abstratos de dados em consideração.

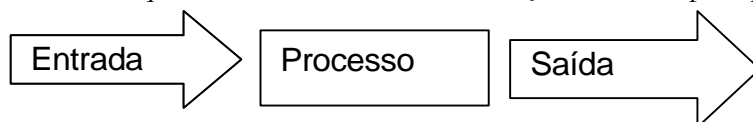
Por exemplo, a criação da estrutura item adiciona um novo tipo em seu vocabulário de programação. Em vez de pensar a respeito de um id, uma descrição e um preço de produto como entidades separadas, provavelmente regiões desconectadas da memória ou variáveis. Você pensa simplesmente em termos de Item. Assim, os tipos permitem representar estruturas complexas em um nível mais simples e mais conceitual. Eles o protegem dos detalhes desnecessários. Isso o libera para trabalhar no nível do problema, em vez de trabalhar no nível da implementação.

#### 3.1 Abstração de Dados

**Processo:** é uma seqüência ordenada de passos que visa promover transformações definidas sobre uma determinada matéria-prima.

**Entrada:** é a matéria-prima no seu estado inicial.

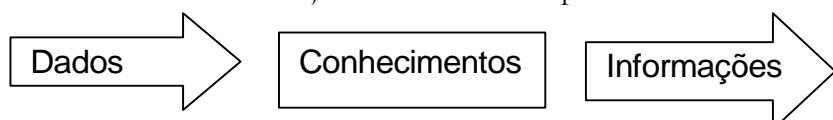
**Saída:** é o que obtemos com a transformação efetuada pelo processo.



**Processamento de Dados:** é quando a matéria-prima utilizada no processo é **abstrata**, ou seja, apresenta-se sob a forma de valores, quantidade ou símbolos. Quando este processo é realizado por computador a **entrada** refere-se aos dados do mundo real externo ao computador, e **processo** refere-se a uma série finita de operações realizadas a partir destes dados, para transforma-los em alguma informação desejada (**saída**).

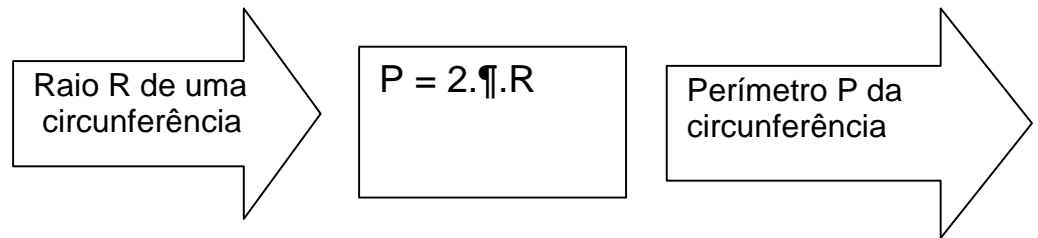
**Abstração:** é o dado retirado de parte da realidade; ou seja, representam algumas características selecionadas das entidades do mundo real, necessárias para a solução de um determinado problema, utilizando o computador.

Sempre que utilizamos computador para resolver problemas, a abstração de dados se faz necessária, visto que o computador não pode trabalhar diretamente com os objetos envolvidos no problema.



Podemos verificar que os dados são manipulados a partir de um conhecimento, com o objetivo de produzir informação.

**Dado + Conhecimento = Informação.**



Devemos observar que Dado e Informação são conceitos relativos.

- **Dados:** é o que entra no processo, é a representação da abstração de objetos do mundo real, valores, quantidades ou símbolos.
- **Informação:** é o dado processado ou alterado pelo processo de acordo com o conhecimento que se tem de um determinado problema. Uma informação gerada por um processo pode passar a ser dado para outro processo.

Devido a grande importância da abstração para o processamento de dados em computador, os programadores deverão estar atentos para duas questões básicas:

1. Como representar a abstração da realidade dentro do computador?
2. Como representar o conhecimento necessário para manipular esta abstração?



### 3.2 Tipos de Dados

Um programa de computador é descrito em uma linguagem de programação. Geralmente, cada linguagem de programação tem seus próprios tipos de dados, isto é, conjunto de valores, operações e relações já implementadas (disponíveis para uso). Na implementação surge o conceito de domínio, isto é, da limitação do conjunto de valores dos elementos representados.

Os tipos de dados existentes na linguagem de programação podem representar fielmente ou não os tipos de dados idealizados (tipos abstratos de dados).

A unidade básica de informação dentro da memória do computador é o **bit (binary digit)**; os dígitos 0 e 1 (Desligado e Ligado) são usados para representar os dois possíveis estados de um bit particular. Obviamente com um único bit não podemos representar muita informação. Mas na memória do computador podemos armazenar uma quantidade muito grande de bits que, uma vez agrupados adequadamente são capazes de representar uma enorme quantidade e variedade de informação.

Por exemplo, em um programa em C para calcular e imprimir a média final de um aluno, num curso onde foram aplicadas duas provas.



```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
    float m, p1, p2;
    printf("Nota da primeira prova: ");
    scanf("%f", &p1);
```



```

    printf("Nota da primeira prova: ");
    scanf("%f", &p2);
    m = (p1 + p2) / 2;
    printf("Media = %f ", m);
}

```

Como podemos observar a forma segundo a qual os bits são agrupados, interpretados e manipulados pelo computador normalmente não nos interessa quando a nossa preocupação maior é resolver o problema específico.

As maiorias das linguagens de programação modernas oferecem um conjunto básico de tipos de dados primitivos (**inteiro, real, caractere, lógico**) que podemos usar na solução de problemas em geral, bem como alguns mecanismos que permitem criar agrupamentos complexos de dados destes tipos primitivos (**vetor, registro, união, ponteiro**).

No programa **Media**, para representar os valores da média e das duas notas de provas, precisamos usar três localizações de memória (m, p1, p2) cujo tipo de dados é real.

### 3.2.1 Por que temos que informar o tipo de dados ao declarar uma variável?

É a partir do tipo de dado indicado na declaração que o computador “sabe” como agrupar os bits, como interpretá-los e quais são as operações que pode efetuar com eles.

## 3.3 Tipos de Dados Abstratos (TDA)

Um **tipo de dados abstrato** é formado por um conjunto de valores e por uma série de funções que podem ser aplicados sobre estes valores. Funções e valores no computador servem para especificar as características relevantes dos objetos envolvidos no problema, de que forma eles se relacionam e como podem ser manipulados. *Em conjunto, constituem um modelo matemático que pode ser empregado para “modelar” e solucionar problemas do mundo real.*

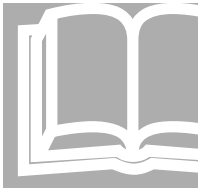
Para realmente aplicar um modelo matemático na resolução de problemas por computador, é preciso primeiro transformar os dados abstratos em **dados concretos**, pois os dados abstratos em sua definição não levam em consideração como os valores serão representados na memória do computador (organização dos bits), nem se preocupa com o “tempo” que será gasto para aplicar as funções (rotinas) sobre tais valores.

**Implementação:** é a transformação de um tipo de dados abstrato em um tipo de dados concreto.

É durante o processo de implementação que a estrutura de armazenamento dos valores é especificada, e que os algoritmos que desempenharão o papel das funções são projetados.

Tipo de Dados Abstrato --> Implementação --->	Tipo de Dados Concreto
(modelo matemático)	(Padrão de Bits/Rotinas)

É importante saber que nem todo o tipo de dados abstrato pode ser implementado em toda sua generalidade.



*Exemplo:*

*Tipo de dados abstrato dos números inteiros,  $Z = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$*

*Funções  $\{+, -, *, /\}$*

Este exemplo é um tipo de dados abstrato que não pode ser implementado universalmente, pois qualquer que seja a estrutura escolhida para armazenar os elementos de **Z**, **nunca conseguiremos mapear num espaço limitado de memória um conjunto infinito de valores.**

Desta forma devemos observar as limitações de uma implementação particular. Devemos ter em mente que podemos chegar a diversas implementações para um mesmo tipo de dados abstrato, cada uma delas apresentando vantagens e desvantagens em relação à outra. Para isso o programador deve ser capaz de escolher aquela mais adequada para resolver o problema específico proposto, tomando como medidas de eficiência da implementação, sobretudo, as suas necessidades de espaço de armazenamento e tempo de execução.



### 3.4 Objetivos da Estruturas de Dados

- **Teórico:** identificar e desenvolver modelos matemáticos, determinando que classe de problemas pode ser resolvida com o uso deles; Este objetivo considera o tipo de dados abstrato como um recurso a ser empregado durante a resolução de problemas em geral.
- **Prático:** criar representações concretas dos objetos e desenvolver rotinas capazes de atuar sobre estas representações, de acordo com o modelo considerado. Este é considerado como um problema em si, que pode ser resolvido através do uso de outro tipo de dados já disponíveis, possivelmente predefinidos na linguagem escolhida para a implementação.

### 3.5 Módulos e compilação em separado

Neste capítulo, discutiremos uma importante técnica de programação baseada na definição de tipos estruturados, conhecidos como tipos abstratos de dados (TAD). A idéia central é encapsular (esconder) de quem usa um determinado tipo a forma concreta com que ele foi implementado. Por exemplo, se criarmos um tipo para representar um ponto no espaço, um cliente desse tipo usa-o de forma abstrata, com base apenas nas funcionalidades oferecidas pelo tipo. A forma com que ele foi efetivamente implementado (armazenado cada coordenada num campo ou agrupamento todas num vetor) passa a ser um detalhe de implementação, que não deve afetar o uso de tipo nos mais diversos contextos. Com isso, desacoplamos a implementação do uso, facilitamos a manutenção e aumentaremos o potencial de reutilização do tipo criado. Por exemplo, a implementação do tipo pode ser alterada sem afetar seu uso em outros exemplos.

Veremos como a linguagem C pode ajudar na implementação de um TAD, com alguns de seus mecanismos básicos de modularização, isto é, divisão de um programa em vários arquivos-fontes.

Os arquivos de protótipos das funções da biblioteca padrão de C são incluídas da forma **#include <arquivo.h>**, enquanto os arquivos de protótipos de nossos módulos são geralmente incluídas da forma **#include "arquivo.h"**. A diferença é o local onde o compilador procura pelas bibliotecas, quando o nome da biblioteca for escrito entre símbolos < e >, o compilador irá procurar pela biblioteca no diretório de arquivos de cabeçalho (por exemplo, "C:\Dev-Cpp\lib" ou "C:\Dev-Cpp\include") e, caso não a encontre, procura na pasta local. Quando o nome da biblioteca for escrito entre aspas, o compilador irá procurar pela biblioteca apenas na pasta local.

Um programa em C pode ser dividido em vários arquivos-fontes. Quando desenvolvemos programas, procuramos identificar funções afins e agrupá-las por arquivo. Quando temos um arquivo com funções que representam apenas parte da implementação de um arquivo completo, denominamos esse arquivo de módulo. Assim a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um deles deve ser compilado separadamente, gerando um arquivo objeto para cada módulo. Após a compilação de todos os módulos, uma outra ferramenta, denominada ligador, é usada para juntar todos os arquivos objeto em um único arquivo executável.

Para programas pequenos, o uso de vários módulos pode não se justificar. No entanto, para programas de médio e grande porte, a sua divisão em vários módulos é uma técnica fundamental, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores e, provavelmente, mais fáceis de implementar e de testar. Além disso, um módulo com funções C pode ser utilizado para compor vários programas e, assim, poupar muito tempo de programação.

### **3.6 Tipos de Dados Abstratos em Módulos**

Geralmente, um módulo agrupa vários tipos e funções como funcionalidades relacionadas, o que caracteriza uma finalidade bem definida. Nos casos em que o módulo define um novo tipo de dado e o conjunto de operações para manipular dados desse tipo, dizemos que o módulo representa um tipo abstrato de dados (TAD). Nesse contexto, abstrato significa "esquecida a forma de implementação", ou seja, um TAD é descrito pela finalidade do tipo e de suas operações, e não pela forma como está implementada.

Como exemplo, podemos criar um TAD para representar matrizes alocadas dinamicamente. Para isso, criamos um tipo "matriz" e uma série de funções que o manipulam, podemos pensar, por exemplo, em funções que acessem e manipulam os valores dos elementos da matriz. Se criarmos um tipo abstrato, podemos "esconder" a estratégia de implementação. Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como é implementado, o que facilita a manutenção e a reutilização de códigos.

A interface de um TAD consiste, basicamente, na definição do nome do tipo e do conjunto de funções exportadas para sua criação e manipulação.

A divisão de programas em módulos e a criação de TADs são técnicas de programação muito importantes.

### 3.7 Exemplo: TAD Ponto

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no  $R^2$ . para isso, devemos definir um tipo abstrato, denominado *ponto*, e o conjunto de funções que operam sobre esse tipo. Nesse exemplo, vamos considerar as seguintes operações:

- + Cria: operação que cria um ponto com coordenadas  $x$  e  $y$ ;
- + Libera: operação que libera memória alocada por um ponto;
- + Acessa: operação que retorna as coordenadas de um ponto;
- + Atribui: operação que atribui novos valores às coordenadas de um ponto;
- + Distancia: operação que calcula a distância entre dois pontos

Como só precisamos guardar as coordenadas de um ponto podemos definir a estrutura ponto da seguinte forma: Salve esta estrutura com nome de *ponto.h*.



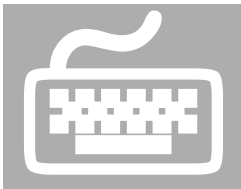
```
struct ponto {  
    float x;  
    float y;  
};  
typedef struct ponto ponto;
```

A função que cria um ponto dinamicamente deve alocar a estrutura que representa o ponto e inicializar os seus campos:

```
ponto * cria_ponto(float f, float g)  
{  
    ponto *p;  
    p=(ponto *)malloc(sizeof(ponto));  
    if (p==NULL)  
    {  
        printf("Memória insuficiente!\n");  
        exit(1);  
    }  
    p->x=f;  
    p->y=g;  
    return p;  
}  
  
void libera_ponto( ponto * p)  
{  
    free(p);  
}  
  
void escreve_ponto(ponto * p1)  
{  
    printf(" \nvalores dos pontos atribuídos x= %f\n",p1->x);
```

```
    printf(" \nvalores dos pontos atribuídos y= %f\n",p1->y);
}
A função principal
#include<stdio.h>
#include<stdlib.h>
#include "ponto.h" // arquivo com a estrutura
main (void)
{
    ponto *p, *q;
    p=cria_ponto(2.0,1.0);
    q=cria_ponto(3.4,2.1);
    escreve_ponto(p);
    escreve_ponto(q);
    libera_ponto(p);
    libera_ponto(q);
    system("pause");
}
```

### Exercício



1. Crie uma estrutura utilizando um vetor inteiro onde será realizada as operações de leitura, escrita, encontrar o maior valor, encontrar o menor valor, a média. Imprimir os resultados encontrados.(utilizar módulos e alocação dinâmica). Utilizar os seguinte módulos: alocar memória, leitura do vetor, escrita do vetor, encontrar elementos maior e menor, desalocar memória.
- 2.
3. Aloque uma matriz dinamicamente e encontre o elemento minimax. Imprima no main o elemento encontrado. Utilizar os seguinte módulos: alocar memória, leitura da matriz, escrita da matriz, encontrar minimxz e desalocar memória.

## 4 Listas Lineares

### 4.1 Introdução

No dia a dia das pessoas é comum elas manterem dados agrupados utilizando-se de listas.

Exemplo :

- *Listas de compras do mercado;*
- *Listas de amigos para festa;*
- *Listas de equipamentos e outras tantas listas.*

Desta forma as listas têm se mostrado um recurso bastante útil e eficiente como recurso de manter dados agrupados, em computação, não tem sido diferente: a lista é uma das estruturas de dados mais empregadas no desenvolvimento de programas.

É comum nos depararmos com problema onde o conjunto de dados que se relacionam entre si de alguma forma, refletem algumas propriedades que eles apresentam no problema real. É natural que estes relacionamentos sejam preservados com objetivo de se fazer uso do mesmo, quando de sua representação no computador.

Supondo que desejamos encontrar a precipitação pluviométrica média do período, visualizando cronologicamente o comportamento dos indicadores diários de chuva. Neste caso, é preciso que mantenhamos um relacionamento entre elementos de dados anteriores e posteriores de tal modo a obter um visão do todo e do comportamento médio.

### 4.2 Fundamentos

Uma **Lista Linear** é uma estrutura que permite representar um conjunto de dados de forma a preservar a relação de ordem linear (ou total) entre eles. Uma lista é composta de **nós**, os quais podem conter, cada um deles, um registro composto de um ou mais tipos de dados.

Ou seja, uma lista é formada por um conjunto ou coleção de  $n \geq 0$  de **nós** tal que **L**:  $[x_1, x_2, \dots, x_n]$ , onde  $x_k$  é precedido por  $x_{k-1}$  e seguido de  $x_{k+1}$ , para  $1 < k < n$ .

Se  $n=0$  diz-se que a lista é vazia.

- $x_1$  é o primeiro elemento de **L**;
- $x_n$  é o último elemento de **L**;
- $x_k$ ,  $1 < k < n$ , é precedido pelo elemento  $x_{k-1}$  e seguido por  $x_{k+1}$  em **L**.

Em outras palavras, a característica fundamental de uma lista linear é o sentido de ordem unidimensional dos elementos que a compõem. Uma ordem

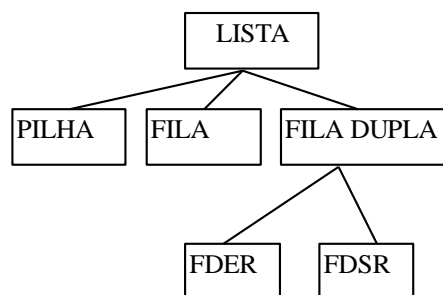
que nos permite dizer com precisão onde a coleção inicia-se e onde termina, sem possibilidade de dúvida.

Em listas podemos realizar diversas operações entre elas :

- acessar um elemento qualquer da lista;
- inserir um elemento numa posição específica da lista;
- remover um elemento de uma posição específica da lista;
- combinar duas lista em uma única;
- particionar uma lista em duas;
- obter cópia de uma lista;
- determinar o total de elementos na lista;
- ordenar os elementos da lista;
- procurar um determinado elemento na lista;
- apagar uma lista;
- outras...

Para determinadas aplicações é imposto um critério que restringe as operações de acesso, inserção e remoção, restritas aos extremos da lista, temos casos especiais que aparecem muito freqüentemente na modelagem de problemas a serem resolvidos por computador. Estes critério de ordem em geral não depende da ordem natural dos valores dos dados.

1. **PILHA**: Lista linear onde todas as inserções, remoções e acessos são realizados em um único extremos. Lista com esta característica são também denominadas listas **LIFO** (Last-In/Frist-Out, ou em português: último que entra/primeiro que sai).Dentre os elementos que permanecem no conjunto, o primeiro elemento a ser retirado é o último que foi inserido;
2. **FILA**: Lista linear onde todas as inserções são feitas num certo extremo e todas as remoções e acessos são realizadas no outro. Filas são também denominadas listas **FIFO** (Frist-In/Frist-Out, ou em português: primeiro que entra/primeiro que sai). Dentre os elementos que ainda permanecem no conjunto, o primeiro elemento a ser retirado é o primeiro que foi inserido.
3. **FILA DUPLA**: Lista Linear onde as inserções, remoções ou acessos são realizados em qualquer extremo. Filas Duplas são também denominadas **DEQUE** (Double-Ended QUEue, ou em português: fila de extremidade dupla). Uma Fila Dupla pode ainda gerar dois casos especiais: Fila Dupla de Entrada Restrita (se a inserção for restrita a um único extremo) e Fila Dupla de Saída Restrita (se a remoção for restrita a um único extremo).



### 4.3 Considerações sobre a implementação.

Quando programamos utilizando listas para armazenar dados, raramente usamos todas as operações que podem ser feitas com listas lineares. Normalmente um número reduzido destas operações será necessário para o funcionamento correto do programa.

É extremamente difícil de se obter uma implementação onde todas as operações sejam realizadas com a mesma eficiência. Por exemplo, uma implementação que facilite o acesso a qualquer elemento da lista, certamente dificultará a inserção e a remoção de elementos no meio da mesma. Existem algumas alternativas de como implementar listas lineares na memória do computador, assim **dependendo das operações que realizaremos com maior frequência, pode escolher qual o tipo que melhor se adapte para nosso programa ser o mais eficiente possível.**

### 4.4 Alocação de Memória

Ao desenvolver uma implementação para listas lineares, o primeiro problema que surge é: como podemos armazenar os elementos da lista, dentro do computador?

Para executar um programa, primeiro o computador precisa carregar seu código executável para a memória. Ocupando parte da memória total disponível no sistema, esta fica reservada para o uso do programa e o restante fica livre. Raramente um programa irá ocupar toda a memória instalada no computador.

Dá memória reservada para o programa, uma parte é usada para armazenar as instruções a serem executadas e a outra é destinada ao armazenamento dos dados. Quem determina quanto de memória será usado para as instruções é o compilador. Alojar área para armazenamento de dados é responsabilidade do **programador**.

Existem quatro categorias para alocação de memória, sendo que cabe ao programador escolher quais usar em seus programas:

	Seqüencial	Encadeada
Estática	Estática Seqüencial	Estática Encadeada
Dinâmica	Dinâmica Seqüencial	Dinâmica Encadeada

### 4.5 Alocação Estática versus Dinâmica

**Alocação Estática:** é quando a quantidade total de memória utilizada pelos dados é previamente conhecida e definida no código fonte do programa e durante toda a sua execução, a quantidade de memória utilizada pelo programa não varia.

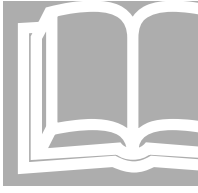
**Alocação Dinâmica:** é quando o programa é capaz de criar novas variáveis enquanto executa, isto é, se áreas de memória que não foram declaradas no programa passam a existir durante a sua execução.

Então considerando que uma variável nada mais é que uma área de memória, dizemos que sua alocação é estática se sua existência já era prevista no código do programa, e quando ela é criada durante a execução é dinâmica.

{-----}







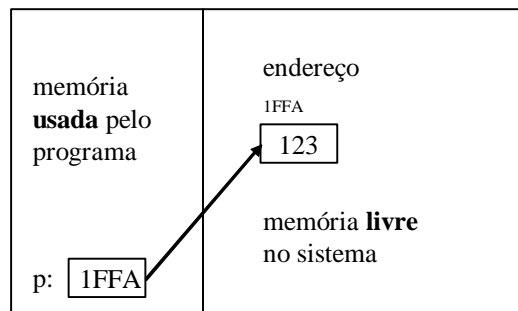
*{ Alocação: mostra uso de variáveis estáticas X dinâmicas }*  
 { ----- }

//Este programa em C possui o nome de **alocação**.

//Diferença de **calloc** - inicializa o espaço alocado com 0

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
main (void )
{
    int *p;           // alocação estática
    system("cls");
    p = (int *) malloc(sizeof(int) ); // alocação dinâmica
    *p=123;
    printf("O valor de p = %d\n", p);
    printf("O valor apontado por p = %d\n", *p);

    free (p);
    getch();
}
```



Observando o programa **Alocação**, identificamos a presença de uma variável **p**. Dizemos, portanto, que esta é uma variável alocada estaticamente.

Entretanto, observando a figura acima, que representa a memória no momento da execução, verificamos que mais uma variável foi criada na posição de memória 1FFFA. Esta é uma variável alocada dinamicamente (às vezes denominada variável **anônima**, pois não conhecemos seu nome, temos apenas seu endereço!).

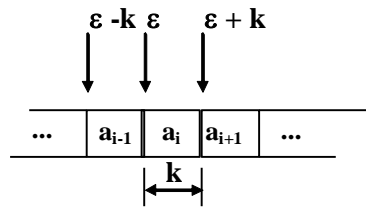
Em C, como já vimos no capítulo 2 sobre ponteiros, as variáveis dinâmicas são alocadas com o comando **malloc**, **calloc** (**calloc** inicializa o espaço alocado com 0). Este comando recebe como argumento uma variável ponteiro **p** (que é capaz de armazenar endereços), verifica se existe uma área de memória disponível e, se existir, aloca esta área e armazena seu endereço na variável **p**. Uma vez que um ponteiro **p** armazena um endereço, o operador posfixo **\*** permite acessar o conteúdo da área cujo endereço esta em **p**. Para liberar uma área previamente alocada pelo comando **malloc**, **calloc**, usamos o comando **free(p)**.

Os atributos sequenciais e encadeados só fazem sentido se a memória considerada for armazenar uma coleção de objetos (vetores), como é o caso da memória necessária para armazenar uma lista linear.

## 4.6 Alocação Seqüencial

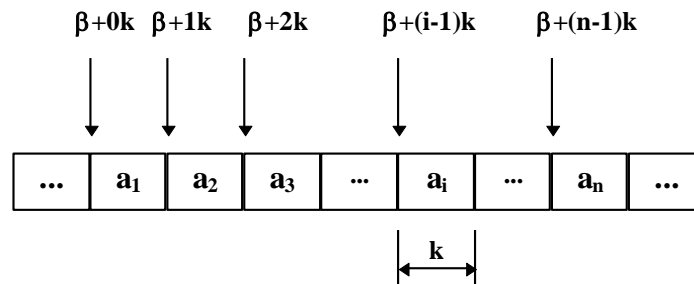
Armazenar os elementos de uma lista em células consecutivas de memória, um após o outro, chamamos de alocação seqüencial.

Assim, se cada célula tem um endereço único  $\epsilon$  e utiliza  $k$  bytes, temos:



- $\text{Endereço}(a_i) = \epsilon$
- $\text{Endereço}(a_{i-1}) = \epsilon - k$
- $\text{Endereço}(a_{i+1}) = \epsilon + k$

De forma um pouco mais genérica, assumindo que o elemento  $a_i$  encontra-se na célula de endereço  $\beta$ , temos a equação:  $\text{Endereço}(a_i) = \beta + (i - 1)k$



A maior vantagem no uso de uma área seqüencial de memória para armazenar uma lista linear é que, dado o endereço inicial  $\beta$  da área alocada e o índice  $i$  de um elemento qualquer da lista, podemos acessá-lo imediatamente, com um simples e rápido cálculo. O ponto fraco desta forma de armazenamento aparece quando precisamos inserir ou suprimir elementos do meio da lista, quando então um certo esforço será necessário para movimentar os elementos, de modo a abrir espaço para inserção, ou de modo a ocupar o espaço liberado por um elemento que foi removido.

## 4.7 Alocação Encadeada

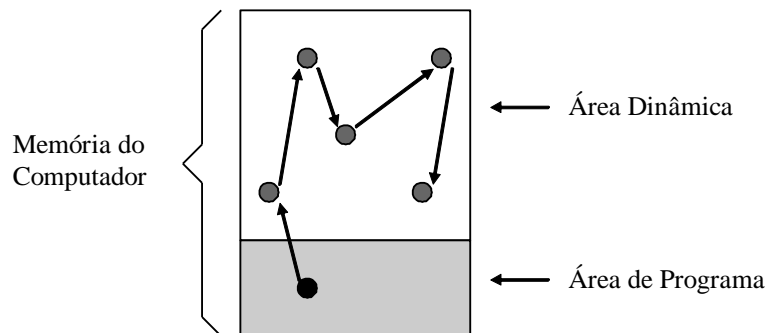
Ao invés de manter os elementos agrupados numa área contínua de memória isto é, ocupando células consecutivas, na alocação encadeada os elementos podem ocupar quaisquer células (não necessariamente consecutivas) e, para manter a relação da ordem linear, juntamente com cada elemento é armazenado o endereço do próximo elemento da lista.

Desta forma, na alocação encadeada, os elementos são armazenados em blocos de memória denominados **nodos**, sendo que cada **nodo** é composto por dois campos: um para armazenar dados e outro para armazenar endereço.

Dois endereços especiais devem ser destacados:

- o endereço do primeiro elemento da lista ( $L$ );
- o endereço do elemento fictício que segue o último elemento da lista ( $\phi$ ).

Endereço	Conteúdo				
$L = 3FFA$	<table border="1"><tr><td><math>a_1</math></td><td><b>1C34</b></td></tr></table>	$a_1$	<b>1C34</b>	←	Primeiro elemento, acessível a partir de $L$
$a_1$	<b>1C34</b>				
1C34	<table border="1"><tr><td><math>a_2</math></td><td><b>BD2F</b></td></tr></table>	$a_2$	<b>BD2F</b>	←	Note que o segundo elemento não ocupa um endereço consecutivo àquele ocupado por $a_1$
$a_2$	<b>BD2F</b>				
BD2F	<table border="1"><tr><td><math>a_3</math></td><td><b>AC12</b></td></tr></table>	$a_3$	<b>AC12</b>		
$a_3$	<b>AC12</b>				
.....					
1000	<table border="1"><tr><td><math>a_i</math></td><td><b>3A7B</b></td></tr></table>	$a_i$	<b>3A7B</b>	←	Cada nodo armazena um elemento e o endereço do próximo elemento da lista
$a_i$	<b>3A7B</b>				
.....					
5670	<table border="1"><tr><td><math>a_{n-1}</math></td><td><b>14F6</b></td></tr></table>	$a_{n-1}$	<b>14F6</b>		
$a_{n-1}$	<b>14F6</b>				
14F6	<table border="1"><tr><td><math>a_{n-1}</math></td><td><b>5D4A</b></td></tr></table>	$a_{n-1}$	<b>5D4A</b>		
$a_{n-1}$	<b>5D4A</b>				
5D4A	<table border="1"><tr><td><math>a_n</math></td><td><math>\phi</math></td></tr></table>	$a_n$	$\phi$	←	Último elemento da cadeia, o endereço nulo $\phi$ indica que o elemento $a_n$ não tem um sucessor
$a_n$	$\phi$				



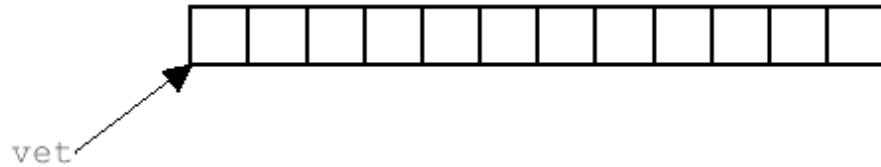
A alocação encadeada apresenta como maior vantagem a facilidade de inserir ou remover elementos do meio da lista. Como os elementos não precisam estar armazenados em posições consecutivas de memória, nenhum dado precisa ser movimentado, bastando atualizar o campo de ligação do elemento que precede aquele inserido ou removido. Por exemplo, para remover o elemento  $a_2$  da lista representada anteriormente, basta mudar o nodo no endereço 3FFA de  $(a_1, 1C34)$  para  $(a_1, BD2F)$ . Como apenas o primeiro elemento é acessível diretamente através do endereço L, a grande desvantagem da alocação encadeada surge quando desejamos acessar uma posição específica dentro da lista. Neste caso devemos partir do primeiro elemento e ir seguindo os campos de ligação, um a um, até atingir a posição desejada. Obviamente, para listas extensas, esta operação pode ter um alto custo em relação a tempo.

## 4.8 Listas Encadeadas

Para representarmos um grupo de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000
int vet[MAX];
```

Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.



Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo `vet`, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de `vet` é o endereço da memória onde o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação `vet[i]`.

Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas dinâmicas e armazenam cada um dos seus elementos usando alocação dinâmica.

Nas seções a seguir, discutiremos a estrutura de dados conhecida como lista encadeada. As listas encadeadas são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos capítulos seguintes.

### 4.8.1 Fundamentos de lista encadeada

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A figura abaixo ilustra o arranjo da memória de uma lista encadeada.

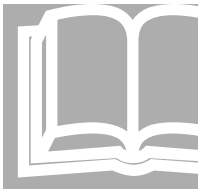


Arranjo da memória de uma lista encadeada.

A estrutura consiste numa seqüência encadeada de elementos, em geral chamados de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó).

Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros numa lista encadeada. O nó da lista pode ser representado pela estrutura abaixo:



```

struct lista {
    int info;
    struct lista* prox;
};
typedef struct lista Lista;

```

Devemos notar que trata-se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definirmos o tipo Lista como sinônimo de struct lista, conforme ilustrado acima. O tipo Lista representa um nó da lista e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo Lista\*).

Considerando a definição de Lista, podemos definir as principais funções necessárias para implementarmos uma lista encadeada.

#### 4.8.1.1 Função de inicialização

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro NULL, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é NULL. Uma possível implementação da função de inicialização é mostrada a seguir:



```

/* função de inicialização: retorna uma lista vazia */

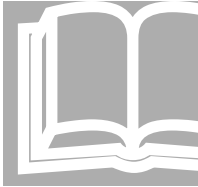
Lista* inicializa (void)
{
    return NULL;
}

```

#### 4.8.1.2 Função de inserção

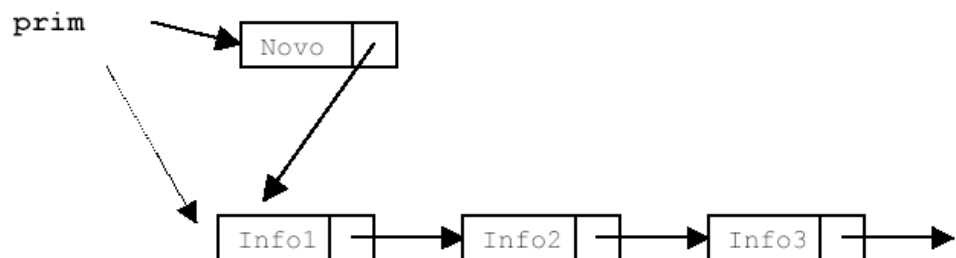
Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.



```
/* inserção no início: retorna a lista atualizada */  
  
Lista* insere (Lista* l, int i)  
{  
    Lista* novo = (Lista*) malloc(sizeof(Lista));  
    novo->info = i;  
    novo->prox = l;  
    return novo;  
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A Figura a seguir ilustra a operação de inserção de um novo elemento no início da lista.



Inserção de um novo elemento no início da lista.

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos.



```
int main (void)  
{  
    Lista* l;          /* declara uma lista não inicializada */  
    l = inicializa();  /* inicializa lista como vazia */  
    l = insere(l, 23); /* insere na lista o elemento 23 */  
    l = insere(l, 45); /* insere na lista o elemento 45 */  
    ...  
    return 0;  
}
```

Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção de um novo elemento. Pode-se utilizar uma outra estratégia para atualizar a lista utilizando ponteiro para ponteiro, mas é preferível utilizar a função mais simples que é o retorno.

#### 4.8.1.3 Função que percorre os elementos da lista

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.



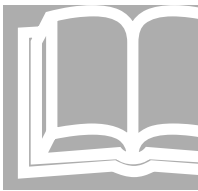
```
/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
    Lista* p;      /* variável auxiliar para percorrer a lista */
    for (p = l; p != NULL; p = p->prox)
        printf("info = %d\n", p->info);
}
```

//p aponta para cada próximo da lista, do 1º ao último.

#### 4.8.1.4 Função que verifica se lista está vazia

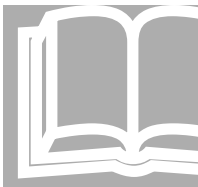
Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não.

A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é NULL. Uma implementação dessa função é mostrada a seguir:



```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
    if (l == NULL)
        return 1;
    else
        return 0;
}
```

Essa função pode ser re-escrita de forma mais compacta, conforme mostrado abaixo:



```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
    return (l == NULL);
}
```

#### 4.8.1.5 Função de busca

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

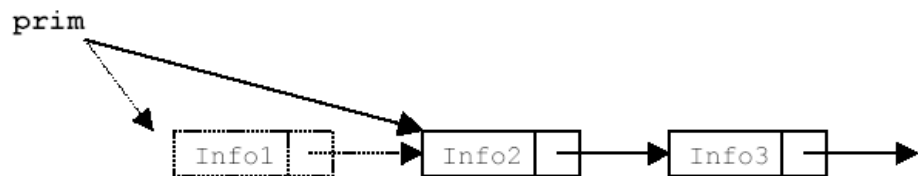


```
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{
    Lista* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL; /* não achou o elemento */
}
```

#### 4.8.1.6 Função que retira um elemento da lista

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As Figuras abaixo ilustram as operações de remoção.



Remoção do primeiro elemento da lista.



Remoção de um elemento no meio da lista.

Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, guardando uma referência para o elemento anterior.

```
/* função retira: retira elemento da lista */

Lista* retira (Lista* l, int v)
{
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    Lista* p = l; /* ponteiro para percorrer a lista */

    /* procura elemento na lista, guardando anterior */
    while (p != NULL && p->info != v)
    {
        ant = p;
        p = p->prox;
    }

    /* verifica se achou elemento */
    if (p == NULL)
        return l; /* não achou: retorna lista original */

    /* retira elemento */
```





```
if (ant == NULL)
{
    /* retira elemento do inicio */
    l = p->prox;
}
else
{
    /* retira elemento do meio da lista */
    ant->prox = p->prox;
}
free(p);
return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação acima. Mais adiante, estudaremos a implementação de filas com listas encadeadas. Numa fila, devemos armazenar, além do ponteiro para o primeiro elemento, um ponteiro para o último elemento. Nesse caso, se for removido o último elemento, veremos que será necessário atualizar a fila.

#### 4.8.1.7 Função para liberar a lista

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).



```
void libera (Lista* l)
{
    Lista* p = l;
    while (p != NULL)
    {
        Lista* t = p->prox; /* guarda referência para o próximo
                             elemento */
        free(p);           /* libera a memória apontada por p */
        p = t;             /* faz p apontar para o próximo */
    }
}
```

#### 4.8.1.8 Exemplo TAD Lista de inteiros

Com base na implementação exemplificada, podemos criar um tipo abstrato de dados para representar uma lista encadeada de valores inteiros. A interface do módulo pode ser dada pelo arquivo lista.h mostrado anteriormente.

Este arquivo deve conter a estrutura do tipo definir o tipo  
typedef struct lista Lista;

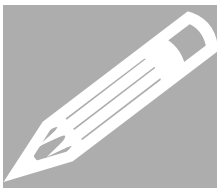
Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>
#include "lista.cpp"
```



```
int main (void)
{
    Lista* l;          /* declara uma lista não iniciada */
    l = inicializa();   /* inicia lista vazia */
    l = insere(l, 23);  /* insere na lista o elemento 23 */
    l = insere(l, 45);  /* insere na lista o elemento 45 */
    l = insere(l, 56);  /* insere na lista o elemento 56 */
    l = insere(l, 78);  /* insere na lista o elemento 78 */
    imprime(l);         /* imprimirá: 78 56 45 23 */
    l = retira(l, 78);
    printf(" retirado elemento 78\n");
    imprime(l);         /* imprimirá: 56 45 23 */
    l = retira(l, 45);
    printf(" retirado elemento 45\n");
    imprime(l);         /* imprimirá: 56 23 */
    libera(l);
    system("pause");
    return 0;
}
```

Mais uma vez, observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como alternativa, poderíamos fazer com que as funções insere e retira recebessem o endereço da variável que representa a lista. Nesse caso, os parâmetros das funções seriam do tipo ponteiro para lista (`Lista** l`) e seu conteúdo poderia ser acessado/atualizado de dentro da função usando o operador conteúdo (`*l`).



### Exercícios

1. Por que motivo as listas devem ser implementadas na forma seqüencial quando acessar um elemento arbitrário for uma operação muito requisitada?
2. Usando C, como você faria uma alocação estática seqüencial de uma área de memória suficiente para armazenar 10 elementos do tipo real?
3. Usando C, como você definiria uma variável capaz de representar um nodo de uma lista linear implementada com alocação encadeada?
4. Por que motivo os valores  $L$  e  $\phi$  são considerados especiais no esquema de alocação encadeada apresentada no material?
5. 2 listas com valores int, não ordenados, simplesmente encadeados, solicite o numero de elementos das listas.

Apresente um algoritmo que intercale as duas listas, de forma que a lista resultante contenha todos os elementos das duas listas.

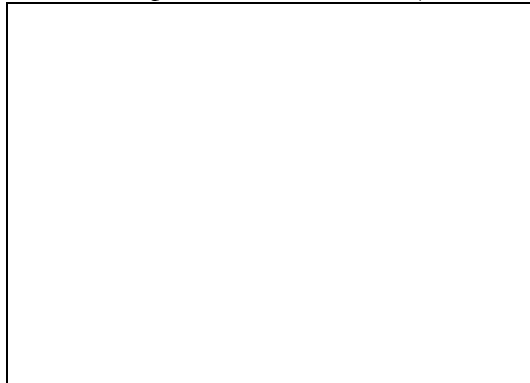
## 5 Pilhas

Uma das estruturas de dados mais simples é a pilha. Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação, sendo inclusive implementada diretamente pelo hardware da maioria das máquinas modernas.

Pilha é um tipo especial de lista linear em que todas as operações de inserção e remoção são realizadas numa mesma extremidade, denominada **topo**.

A inserção na pilha de um novo elemento, sempre este é colocado no topo da mesma; e sempre que removemos um elemento este é retirado do topo da pilha. Devido a esta regra de acesso, os elementos são sempre removidos numa ordem inversa àquela em que foram inseridos, de modo que o último elemento que entra é exatamente o primeiro que sai. Daí o fato de estas listas serem também denominadas de listas **LIFO** (Last-In/First-Out).

Uma lista **LIFO** é uma **estrutura dinâmica**, ou seja, é uma coleção que pode aumentar e diminuir durante a sua existência. Por causa desta dinamicidade que as listas LIFO possuem capacidade de aumentar e diminuir e devida a sua semelhança de funcionamento com as pilhas no mundo real. O exemplo mais comum do cotidiano é uma pilha de pratos, onde o último prato colocado é o primeiro a ser usado (removido).



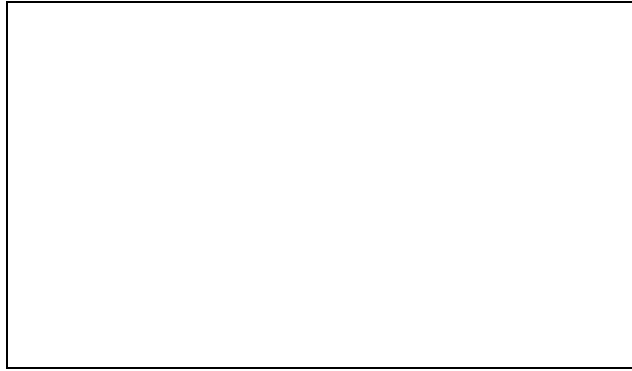
Uma pilha suporta três operações básicas, tradicionalmente denominadas como:

- **Top** : acessa o elemento posicionado no topo da pilha;
- **Push** : insere um novo elemento no topo da pilha;
- **Pop** : remove um elemento do topo da pilha.

Estes nomes poderiam ser qualquer outro, pois temos que criar os algoritmos para que possamos manipular a pilha. (Insere, Retira, MostraTopo,...).

Um exemplo bastante ilustrativo, que nos permite entender a origem dos termos push-down e pop-up, é um porta-guardanapos daqueles normalmente encontrados em lanchonetes.

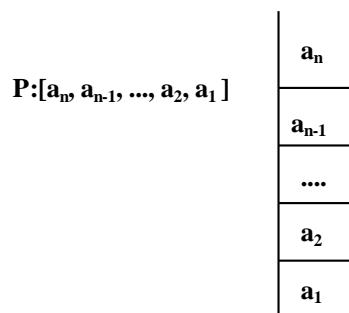




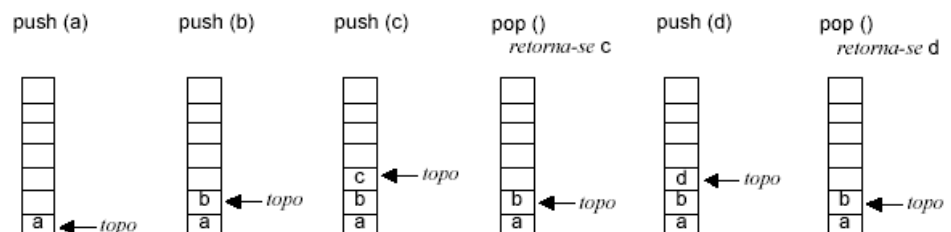
Para entendermos o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha.

Existem duas operações básicas que devem ser implementadas numa estrutura de pilha: a operação para empilhar um novo elemento, inserindo-o no topo, e a operação para desempilhar um elemento, removendo-o do topo.

Para melhor visualização, ao invés de utilizar a notação de lista linear, geralmente as pilhas são representadas na forma de um gráfico, crescendo na vertical, de baixo para cima, conforme o esquema a seguir:



Temos a seguir, representados na forma gráfica, os sucessivos estados que uma pilha assume quando novos elementos são nela colocados ou dela retirados:



Funcionamento da pilha.

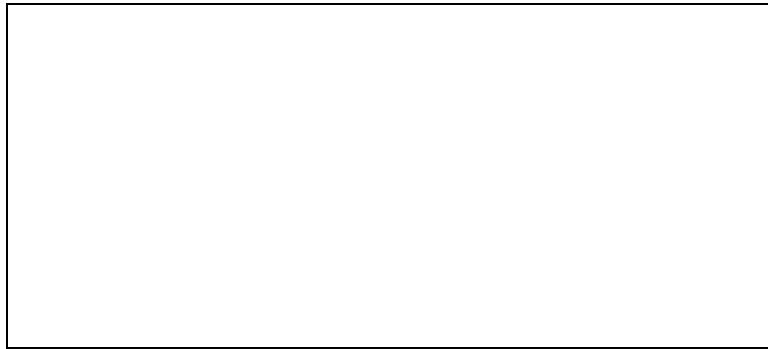
O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C. As variáveis locais das funções são dispostas numa

pilha e uma função só tem acesso às variáveis que estão no topo (não é possível acessar as variáveis da função locais às outras funções).

Há várias implementações possíveis de uma pilha, que se distinguem pela natureza dos seus elementos, pela maneira como os elementos são armazenados e pelas operações disponíveis para o tratamento da pilha.

### 5.1 *Init, IsEmpty e IsFull*

Observe uma pilha de pratos sobre uma mesa, dentro de uma sala. Seria possível colocar novos pratos indefinidamente sobre ela? Obviamente, não! Em algum momento o prato do topo da pilha tocaria o teto da sala. E o contrário: seria possível remover pratos do topo da pilha indefinidamente? Também não! Em algum momento a pilha tornar-se-ia vazia. A mesa e o teto são limites físicos que impedem que a pilha cresça ou diminua indefinidamente.



Neste exemplo antes de começarmos a empilhar os pratos sobre a mesa, devemos garantir que a mesa estará limpa, isto é, que não existem outros objetos no local onde colocaremos os pratos. Para adicionar um prato à pilha, primeiro verificamos se ainda existe espaço entre o topo da pilha e o teto da sala. Finalmente, para remover, precisamos nos certificar de que ainda existem pratos sobre a mesa.

Da mesma forma, ao implementarmos uma pilha dentro do computador, a quantidade de memória alocada para representar seus elementos funciona como fator limitante. Como no exemplo da pilha de pratos na mesa, no computador precisamos de mais três operações para manipular pilhas:

- **Init** : inicializa a pilha no estado “vazia”;
- **IsEmpty** : verifica se a pilha está vazia;
- **IsFull** : verifica se a pilha está cheia.

Observação: Sempre que criamos uma variável, ela permanece com conteúdo indefinido, até que um determinado valor seja a ela atribuído. Normalmente na criação de uma variável isto se restringe apenas à alocação de área de memória necessária para representá-la; nenhum valor inicial é armazenado nesta área. No caso de uma variável do tipo **pilha**, isto não é diferente.

A operação **Init(P)** tem como objetivo definir um estado inicial para a pilha **P**. Por questões práticas, uma pilha normalmente é inicializada no estado “vazia”. Toda vez que criamos uma variável pilha, antes de qualquer coisa, devemos inicializá-la para garantir que não haverá nenhuma “sujeira” no local onde ela será “montada”.

Para verificarmos se uma pilha **P** está vazia, podemos usar a função lógica **IsEmpty(P)**, que toma como argumento a pilha em que estamos



interessados e retorna somente se ela estiver vazia, sem nenhum elemento armazenado. Já a função **IsFull(P)** é usada para verificar se uma pilha está cheia, isto é, ela retorna verdadeiro somente quando não há mais espaço para armazenar nenhum elemento na pilha.

Note que **IsFull(P)** não é o inverso de **IsEmpty(P)**. Quando **IsEmpty(P)** retorna falso, indicando que a pilha **P** não está vazia, não significa que ela se encontra cheia. Da mesma forma, quando **IsFull(P)** retorna falso, informando que a pilha **P** não está cheia, não quer dizer que ela esteja vazia. Uma pilha pode estar nem vazia, nem cheia.



## 5.2 Exemplo de uso de Pilhas.

Programa para mostrar o valor binário de um número inteiro positivo em decimal.

Para entender a lógica do programa, lembre-se de que para converter um número inteiro da base decimal (10) para a binária (2), devemos dividi-lo sucessivamente por 2, até obtermos um quociente igual a 0. Neste momento, os restos obtidos nas divisões devem ser tomados em ordem inversa.



Assim, o valor 13 decimal fica 1101 em binário. No programa a seguir, a pilha é utilizada para armazenar os restos obtidos, de modo que depois eles possam ser recuperados na ordem inversa em que foram gerados.

Depois de ler o valor **n** a ser convertido e inicializar a pilha, o programa inicia um laço que se repete até que seja obtido um quociente nulo. Este laço (**do**) executa dois comandos: o primeiro deles calcula o resto da divisão inteira (%) de **n** por 2, este resto é passado à rotina **push** para ser armazenado na pilha **P**; o segundo comando altera a variável **n**, de modo que ela passe a armazenar o quociente da divisão inteira (**div**) do valor de **n** por 2.

Em algum momento, este quociente será 0 e a repetição terminará.

Após a impressão da frase 'Correspondente binário: ', o segundo laço (**while**) entra em execução, finalizando somente quando a pilha estiver vazia. Enquanto a pilha não esvaziar, a função **Pop** retira um valor do topo da pilha que é então impresso. Quando todos os restos armazenados na pilha já tiverem sido impressos, a pilha ficará vazia e a repetição terminará.

## 5.3 Interface do tipo pilha

Neste capítulo, consideraremos duas implementações de pilha: usando vetor e usando lista encadeada. Para simplificar a exposição, consideraremos uma pilha que armazena valores reais. Independente da estratégia de implementação, podemos definir a interface do tipo abstrato que representa uma estrutura de pilha. A interface é composta pelas operações que estarão disponibilizadas para manipular e acessar as informações da pilha. Neste exemplo, vamos considerar a implementação de cinco operações:

- ✚ criar uma estrutura de pilha;
- ✚ inserir um elemento no topo (*push*);
- ✚ remover o elemento do topo (*pop*);

- ✚ verificar se a pilha está vazia;
- ✚ liberar a estrutura de pilha.

O arquivo `pilha.cpp`, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct pilha Pilha;  
Pilha* cria (void);  
void push (Pilha* p, float v);  
float pop (Pilha* p);  
int vazia (Pilha* p);  
void libera (Pilha* p);
```

A função *cria* aloca dinamicamente a estrutura da pilha, inicializa seus campos e retorna seu ponteiro; as funções *push* e *pop* inserem e retiram, respectivamente, um valor real na pilha; a função *vazia* informa se a pilha está ou não vazia; e a função *libera* destrói a pilha, liberando toda a memória usada pela estrutura.

### 5.4 Implementação de pilha com vetor

Em aplicações computacionais que precisam de uma estrutura de pilha, é comum sabermos de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nestes casos, a implementação da pilha pode ser feita usando um vetor. A implementação com vetor é bastante simples. Devemos ter um vetor (*vet*) para armazenar os elementos da pilha. Os elementos inseridos ocupam as primeiras posições do vetor. Desta forma, se temos *n* elementos armazenados na pilha, o elemento `vet[n-1]` representa o elemento do topo.

A estrutura que representa o tipo pilha deve, portanto, ser composta pelo vetor e pelo número de elementos armazenados.

```
#define MAX 50  
  
struct pilha  
{  
    int n;  
    float vet[MAX];  
};
```

A função para criar a pilha aloca dinamicamente essa estrutura e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
Pilha* cria (void)  
{  
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));  
    p->n = 0; /* inicializa com zero elementos */  
    return p;  
}
```

Para inserir um elemento na pilha, usamos a próxima posição livre do vetor. Devemos ainda assegurar que exista espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com dimensão fixa.

```
Pilha* push (Pilha* p, float v)  
{  
    if (p->n == MAX)
```

```
    { /* capacidade esgotada */
        printf("Capacidade da pilha estourou.\n");
        exit(1); /* aborta programa */
    }

    /* insere elemento na próxima posição livre */
    p->vet[p->n] = v;
    p->n++;
    return p;
}
```

A função pop retira o elemento do topo da pilha, fornecendo seu valor como retorno. Podemos também verificar se a pilha está ou não vazia.

```
Pilha* pop (Pilha* p)
{
    if (vazia(p))
    {
        printf("Pilha vazia.\n");
        exit(1); /* aborta programa */
    }
    /* retira elemento do topo */
    p->vet[p->n-1]=0.0;
    p->n--;
    return p;
}
```

A função que verifica se a pilha está vazia pode ser dada por:

```
int vazia (Pilha* p)
{
    return (p->n == 0);
}
```

Finalmente, a função para liberar a memória alocada pela pilha pode ser:

```
void libera (Pilha* p)
{
    free(p);
}
```

### 5.5 Implementação de pilha com lista

Quando o **número máximo de elementos** que serão armazenados na pilha não é conhecido, devemos implementar a pilha usando uma estrutura de dados dinâmica, no caso, empregando uma lista encadeada. Os elementos são armazenados na lista e a pilha pode ser representada simplesmente por um ponteiro para o primeiro nó da lista.

O nó da lista para armazenar valores reais pode ser dado por:

```
struct no
{
    float info;
    struct no* prox;
};

typedef struct no No;
```

A estrutura da pilha é então simplesmente:



```
struct pilha
{
    No* prim;
};
```

A função cria aloca a estrutura da pilha e inicializa a lista como sendo vazia.

```
Pilha* cria (void)
{
    Pilha* p = (Pilha*) malloc(sizeof(Pilha));
    p->prim = NULL;
    return p;
}
```

O primeiro elemento da lista representa o topo da pilha. Cada novo elemento é inserido no início da lista e, conseqüentemente, sempre que solicitado, retiramos o elemento também do início da lista. Desta forma, precisamos de duas funções auxiliares da lista: para inserir no início e para remover do início. Ambas as funções retornam o novo primeiro nó da lista.

```
/* função auxiliar: insere no início */
No* ins_ini (No* l, float v)
{
    No* p = (No*) malloc(sizeof(No));
    p->info = v;
    p->prox = l;
    return p;
}

/* função auxiliar: retira do início */
No* ret_ini (No* l)
{
    No* p = l->prox;
    free(l);
    return p;
}
```

As funções que manipulam a pilha fazem uso dessas funções de lista:

```
void push (Pilha* p, float v)
{
    p->prim = ins_ini(p->prim,v);
}

float pop (Pilha* p)
{
    float v;
    if (vazia(p))
    {
        printf("Pilha vazia.\n");
        exit(1); /* aborta programa */
    }
    v = p->prim->info;
    p->prim = ret_ini(p->prim);
    return v;
}
```

A pilha estará vazia se a lista estiver vazia:

```
int vazia (Pilha* p)
{
    return (p->prim==NULL);
}
```

```
}
```

Por fim, a função que libera a pilha deve antes liberar todos os elementos da lista.

```
void libera (Pilha* p)
{
    No* q = p->prim;
    while (q!=NULL)
    {
        No* t = q->prox;
        free(q);
        q = t;
    }
    free(p);
}
```

A rigor, pela definição da estrutura de pilha, só temos acesso ao elemento do topo. No entanto, para testar o código, pode ser útil implementarmos uma função que imprima os valores armazenados na pilha. Os códigos abaixo ilustram a implementação dessa função nas duas versões de pilha (vetor e lista). A ordem de impressão adotada é do topo para a base.

```
/* imprime: versão com vetor */
void imprime (Pilha* p)
{
    int i;
    for (i=p->n-1; i>=0; i--)
        printf("%f\n",p->vet[i]);
}

/* imprime: versão com lista */
void imprime (Pilha* p)
{
    No* q;
    for (q=p->prim; q!=NULL; q=q->prox)
        printf("%f\n",q->info);
}
```

### 11.4. Exemplo de uso: calculadora pós-fixada

Ver link teoria <http://www.icmc.usp.br/~sce182/paplic.html>

Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras da HP (Hewlett-Packard). Elas trabalham com expressões pós-fixadas, então para avaliarmos uma expressão como  $(1-2)*(4+5)$  podemos digitar `1 2 - 4 5 + *`. O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado numa pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Deste modo, na expressão acima, são empilhados os valores 1 e 2. Quando aparece o operador `-`, 1 e 2 são desempilhados e o resultado da operação, no caso  $-1$  ( $= 1 - 2$ ), é colocado no topo da pilha. A seguir, 4 e 5 são empilhados. O operador seguinte, `+`, desempilha o 4 e o 5 e empilha o resultado da soma, 9. Nesta hora, estão na pilha os dois resultados parciais,  $-1$  na base e 9 no topo. O operador `*`, então, desempilha os dois e coloca  $-9$  ( $= -1 * 9$ ) no topo da pilha.

Como exemplo de aplicação de uma estrutura de pilha, vamos implementar uma calculadora pós-fixada. Ela deve ter uma pilha de valores reais para representar os operandos. Para enriquecer a implementação, vamos

considerar que o formato com que os valores da pilha são impressos seja um dado adicional associado à calculadora. Esse formato pode, por exemplo, ser passado quando da criação da calculadora. Para representar a interface exportada pela calculadora, podemos criar o arquivo `calc.cpp`:

```
/* Arquivo que define a interface da calculadora */
typedef struct calc Calc;
/* funções exportadas */
Calc* cria_calc (char* f);
void operando (Calc* c, float v);
void operador (Calc* c, char op);
void libera_calc (Calc* c);
```

Essas funções utilizam as funções mostradas acima, independente da implementação usada na pilha (vetor ou lista). O tipo que representa a calculadora pode ser dado por:

```
struct calc {
char f[21]; /* formato para impressão */
Pilha* p; /* pilha de operandos */
};
```

A função `cria` recebe como parâmetro de entrada uma cadeia de caracteres com o formato que será utilizado pela calculadora para imprimir os valores. Essa função cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* cria_calc (char* formato)
{
    Calc* c = (Calc*) malloc(sizeof(Calc));
    strcpy(c->f, formato);
    c->p = cria(); /* cria pilha vazia */
    return c;
}
```

A função `operando` coloca no topo da pilha o valor passado como parâmetro. A função `operador` retira os dois valores do topo da pilha (só consideraremos operadores binários), efetua a operação correspondente e coloca o resultado no topo da pilha. As operações válidas são: '+' para somar, '-' para subtrair, '\*' para multiplicar e '/' para dividir. Se não existirem operandos na pilha, consideraremos que seus valores são zero. Tanto a função `operando` quanto a função `operador` imprimem, utilizando o formato especificado na função `cria`, o novo valor do topo da pilha.

```
void operando (Calc* c, float v)
{
    /* empilha operando */
    push(c->p, v);
    /* imprime topo da pilha */
    printf(c->f, v);
}
```

```
void operador (Calc* c, char op)
{
    float v1, v2, v;
    /* desempilha operandos */
    if (vazia(c->p))
        v2 = 0.0;
    else
        v2 = pop(c->p);
    if (vazia(c->p))
```

```
        v1 = 0.0;
    else
        v1 = pop(c->p);
    /* faz operação */
    switch (op)
    {
        case '+': v = v1+v2; break;
        case '-': v = v1-v2; break;
        case '*': v = v1*v2; break;
        case '/': v = v1/v2; break;
    }
    /* empilha resultado */
    push(c->p,v);
    /* imprime topo da pilha */
    printf(c->f,v);
}
```

Por fim, a função para liberar a memória usada pela calculadora libera a pilha de operandos e a estrutura da calculadora.

```
void libera_calc (Calc* c)
{
    libera(c->p);
    free(c);
}
```

Um programa cliente que faça uso da calculadora é mostrado abaixo:

```
/* Programa para ler expressão e chamar funções da calculadora */
#include <stdio.h>
#include "calc.h"
int main (void)
{
    char c;
    float v;
    Calc* calc;
    /* cria calculadora com precisão de impressão de duas casas decimais */
    calc = cria_calc("%.2f\n");
    do
    {
        /* le proximo caractere nao branco */
        scanf("%c",&c);
        /* verifica se e' operador valido */
        if (c=='+' || c=='-' || c=='*' || c=='/')
        {
            operador(calc,c);
        }
        /* devolve caractere lido e tenta ler número */
        else
        {
            ungetc(c,stdin);
            if (scanf("%f",&v) == 1)
                operando(calc,v);
        }
    } while (c!='q');
    libera_calc(calc);
    return 0;
}
```

Esse programa cliente lê os dados fornecidos pelo usuário e opera a calculadora. Para tanto, o programa lê um caractere e verifica se é um operador válido. Em caso negativo, o programa “devolve” o caractere lido para o *buffer* de leitura, através da função *ungetc*, e tenta ler um operando. O usuário finaliza a execução do programa digitando q.

Se executado, e considerando-se as expressões digitadas pelo usuário mostradas abaixo, esse programa teria como saída:

3 5 8 * +	-> digitado pelo usuário
3.00	
5.00	
8.00	
40.00	
43.00	
7 /	-> digitado pelo usuário
7.00	
6.14	
q	-> digitado pelo usuário

### Exercícios: TRABALHO FINAL DO GA

1. Mostre a situação da pilha P, inicialmente vazia, após a execução de cada uma das operações a seguir:  
Push(P,1);  
Push(P,b);  
Push(P,c);  
Push(P,Top(P));  
Push(P,Pop(P));  
Pop(P);  
Push(P,e);  
Pop(p);
2. Escreva um programa para ler uma frase e imprima-la com a palavra invertidas. Exemplo: a frase “A pilha do gato” deve sair “A ahlip od otag”.
3. Estenda a funcionalidade da calculadora incluindo novos operadores (sugestão: # como raiz quadrada, ^ como exponenciação).
4. Implementar utilizando pilha a conversão de um número decimal para binário e imprimir na tela os dois valores. A operação deve ser realizada para calcular vários números decimais, até digitar o valor zero (0).

## 6 Filas

Um **fila** é um tipo especial de lista linear em que as inserções são realizadas num extremo, ficando as remoções restritas ao outro. O extremo onde os elementos são inseridos é denominado **final** da fila, e aquele de onde são removidos é denominado **começo** da fila.

Outra estrutura de dados bastante usada em computação é a *fila*. Na estrutura de fila, os acessos aos elementos também seguem uma regra. O que diferencia a *fila* da *pilha* é a ordem de saída dos elementos: enquanto na pilha “o último que entra é o primeiro que sai”, na fila “o primeiro que entra é o primeiro que sai” (a sigla FIFO – *first in, first out* – é usada para descrever essa estratégia). A idéia fundamental da fila é que só podemos inserir um novo elemento no final da fila e só podemos retirar o elemento do início.

Cada vez que uma operação de inserção é executada, um novo elemento é colocado no final da fila. Na remoção, é sempre retornado o elemento que aguarda há mais tempo na fila, ou seja, aquele posicionado no começo. A ordem de saída corresponde diretamente à ordem de entrada dos elementos na fila, de modo que os primeiros elementos que entram são os primeiros a sair. Em vista disto, as filas são denominadas lista **FIFO** (Frist-In/Frist-Out).

Um exemplo bastante comum de filas verifica-se num balcão de atendimento. Naturalmente, devemos desconsiderar os casos de pessoas que “furam” a fila ou que desistem de aguardar! Diferentemente das filas no mundo real, o tipo de dados abstrato não suporta inserção nem remoção no meio da lista.



Um exemplo de utilização em computação é a implementação de uma fila de impressão. Se uma impressora é compartilhada por várias máquinas, deve-se adotar uma estratégia para determinar que documento será impresso primeiro. A estratégia mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

A palavra **queue**, da língua inglesa, significa fila. Por tradição, as duas operações básicas que uma fila suporta são denominadas como a seguir:

- **Enqueue:** insere um elemento no final da fila;

- **Dequeue:** remove um elemento do começo da fila.

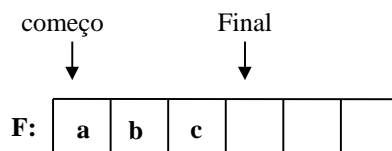
Sendo **F** um fila de **x** um elemento qualquer, a operação **Enqueue(F,x)** aumenta o tamanho da fila **F**, acrescentando o elemento **x** no seu final. A operação **Dequeue(F,x)** faz a fila diminuir, já que remove e retorna o elemento posicionado no seu começo.



Operação	Estado da Fila	Resultado
-----	$F: []$	---
$Enqueue(F, a)$	$F: [a]$	---
$Enqueue(F, b)$	$F: [a, b]$	---
$Enqueue(F, c)$	$F: [a, b, c]$	---
$Enqueue(F, d)$	$F: [a, b, c, d]$	---
$Dequeue(F)$	$F: [b, c, d]$	$a$
$Dequeue(F)$	$F: [c, d]$	$b$
$Enqueue(F, e)$	$F: [c, d, e]$	---
$Enqueue(F, Dequeue(F))$	$F: [d, e]$	$c$
	$F: [d, e, c]$	---
$Dequeue(F)$	$F: [e, c]$	$d$
$Dequeue(F)$	$F: [c]$	$e$

## 6.1 Implementação Sequencial de Filas

Graficamente, representamos uma fila como uma coleção de objetos que cresce da esquerda para a direita, como dois extremos bem-definidos: começo e final:



Intuitivamente, a partir da representação gráfica, percebemos que é possível implementar uma fila tendo três recursos básicos:

- espaço de memória para armazenar os elementos;
- uma referência ao primeiro elemento da coleção;
- uma referência à primeira posição livre, após o último elemento da fila.

De modo análogo ao que fizemos com a estrutura de pilha, neste capítulo discutiremos duas estratégias para a implementação de uma estrutura de fila: usando vetor e usando lista encadeada. Para implementar uma fila, devemos ser capazes de inserir novos elementos em uma extremidade, o *fim*, e retirar elementos da outra extremidade, o *início*.

## 6.2 Interface do tipo fila

Antes de discutirmos as duas estratégias de implementação, podemos definir a interface disponibilizada pela estrutura, isto é, definir quais operações serão implementadas para manipular a fila. Mais uma vez, para simplificar a exposição, consideraremos uma estrutura que armazena valores reais.

Independente da estratégia de implementação, a interface do tipo abstrato que representa uma estrutura de fila pode ser composta pelas seguintes operações:

- ✚ criar uma estrutura de fila;
- ✚ inserir um elemento no fim;
- ✚ retirar o elemento do início;
- ✚ verificar se a fila está vazia;
- ✚ liberar a fila.

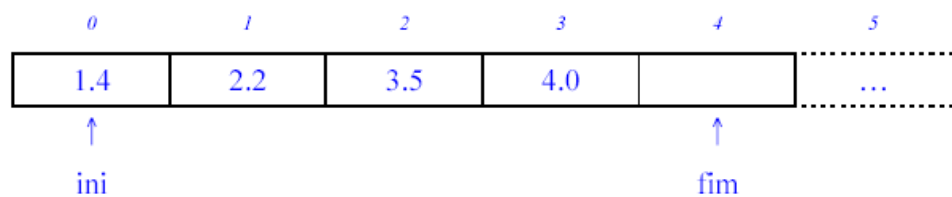
O arquivo fila.h, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila Fila;  
Fila* cria (void);  
void insere (Fila* f, float v);  
float retira (Fila* f);  
int vazia (Fila* f);  
void libera (Fila* f);
```

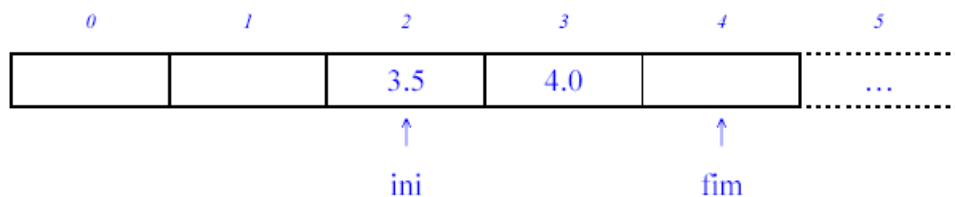
A função *cria* aloca dinamicamente a estrutura da fila, inicializa seus campos e retorna seu ponteiro; a função *insere* adiciona um novo elemento no final da fila e a função *retira* remove o elemento do início; a função *vazia* informa se a fila está ou não vazia; e a função *libera* destrói a estrutura, liberando toda a memória alocada.

### 6.3 Implementação de fila com vetor

Como no caso da pilha, nossa primeira implementação de fila será feita usando um vetor para armazenar os elementos. Para isso, devemos fixar o número máximo *N* de elementos na fila. Podemos observar que o processo de inserção e remoção em extremidades opostas fará com que a fila “ande” no vetor. Por exemplo, se inserirmos os elementos 1.4, 2.2, 3.5, 4.0 e depois retirarmos dois elementos, a fila não estará mais nas posições iniciais do vetor. A Figura (a) abaixo ilustra a configuração da fila após a inserção dos primeiros quatro elementos e a Figura (b) após a remoção de dois elementos.



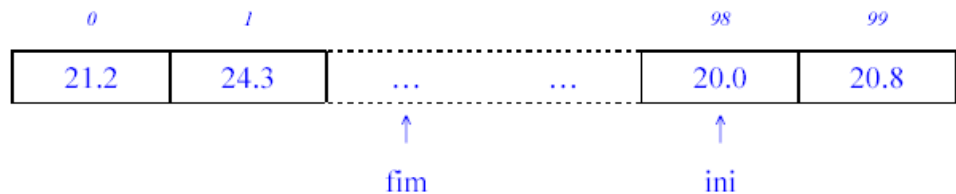
(a) Fila após inserção de quatro novos elementos.



(b) Fila após retirar dois elementos.



Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada do vetor pode chegar à última posição. Para reaproveitar as primeiras posições livres do vetor sem implementarmos uma re-arrumação trabalhosa dos elementos, podemos incrementar as posições do vetor de forma “circular”: se o último elemento da fila ocupa a última posição do vetor, inserimos os novos elementos a partir do início do vetor. Desta forma, em um dado momento, poderíamos ter quatro elementos, 20.0, 20.8, 21.2 e 24.3, distribuídos dois no fim do vetor e dois no início.

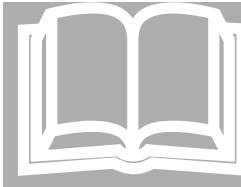


Fila com incremento circular.

Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam “circularmente”. Desta forma, se temos 100 posições no vetor, os valores dos índices assumem os seguintes valores:

0, 1, 2, 3, ..., 98, 99, 0, 1, 2, 3, ..., 98, 99, 0, 1, ...

Podemos definir uma função auxiliar responsável por incrementar o valor de um índice. Essa função recebe o valor do índice atual e fornece com valor de retorno do índice incrementado, usando o incremento circular. Uma possível implementação dessa função é:



```
int incr (int i)
{
    if (i == N-1)
        return 0;
    else
        return i+1;
}
```

Essa mesma função pode ser implementada de uma forma mais compacta, usando o operador módulo:

```
int incr(int i)
{
    return (i+1)%N;
}
```

Com o uso do operador módulo, muitas vezes optamos inclusive por dispensar a função auxiliar e escrever diretamente o incremento circular:

```
...
i=(i+1)%N;
...
```

Podemos declarar o tipo fila como sendo uma estrutura com três componentes: um vetor vet de tamanho N, um índice ini para o início da fila e um índice fim para o fim da fila.

Conforme ilustrado nas figuras acima, usamos as seguintes convenções para a identificação da fila:

- ✚ `ini` marca a posição do próximo elemento a ser retirado da fila;
- ✚ `fim` marca a posição (vazia), onde será inserido o próximo elemento.

Desta forma, a fila vazia se caracteriza por ter `ini == fim` e a fila cheia (quando não é possível inserir mais elementos) se caracteriza por ter `fim` e `ini` em posições consecutivas (circularmente): `incr(fim) == ini`. Note que, com essas convenções, a posição indicada por `fim` permanece sempre vazia, de forma que o número máximo de elementos na fila é `N-1`. Isto é necessário porque a inserção de mais um elemento faria `ini == fim`, e haveria uma ambigüidade entre fila cheia e fila vazia. Outra estratégia possível consiste em armazenar uma informação adicional, `n`, que indicaria explicitamente o número de elementos armazenados na fila. Assim, a fila estaria vazia se `n == 0` e cheia se `n == N-1`. Nos exemplos que se seguem, optamos por não armazenar `n` explicitamente.

A estrutura de fila pode então ser dada por:

```
#define N 100
struct fila
{
    int ini, fim;
    float vet[N];
};
```

A função para criar a fila aloca dinamicamente essa estrutura e inicializa a fila como sendo vazia, isto é, com os índices `ini` e `fim` iguais entre si (no caso, usamos o valor zero).

```
Fila* cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = 0; /* inicializa fila vazia */
    return f;
}
```

Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por `fim`. Devemos ainda assegurar que há espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com capacidade limitada. Consideraremos que a função auxiliar que faz o incremento circular está disponível.

```
void insere (Fila* f, float v)
{
    if (incr(f->fim) == f->ini) /* fila cheia: capacidade
    {
        esgotada */
        printf("Capacidade da fila estourou.\n");
        exit(1); /* aborta programa */
    }

    /* insere elemento na próxima posição livre */
    f->vet[f->fim] = v;
    f->fim = incr(f->fim);
}
```

A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno. Podemos também verificar se a fila está ou não vazia.

```
float retira (Fila* f)
{
    float v;
    if (vazia(f))
    {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    /* retira elemento do início */
    v = f->vet[f->ini];
    f->ini = incr(f->ini);
    return v;
}
```

A função que verifica se a fila está vazia pode ser dada por:

```
int vazia (Fila* f)
{
    return (f->ini == f->fim);
}
```

Finalmente, a função para liberar a memória alocada pela fila pode ser:

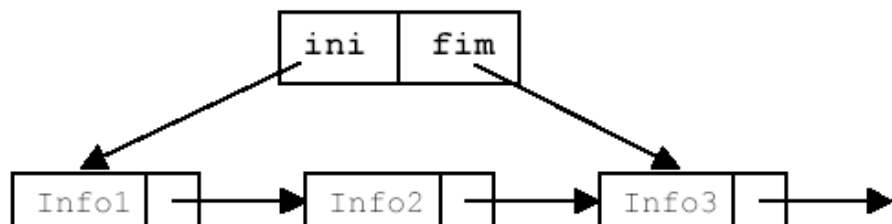
```
void libera (Fila* f)
{
    free(f);
}
```

A função que imprime um fila utilizando vetor pode ser:

```
/* imprime: versão com vetor */
void imprime (Fila* f)
{
    int i;
    for (i=f->ini; i!=f->fim; i=incr(i))
        printf("%f\n", f->vet[i]);
}
```

#### **6.4 Implementação de fila com lista**

Vamos agora ver como implementar uma fila através de uma lista encadeada, que será, como nos exemplos anteriores, uma lista simplesmente encadeada, em que cada nó guarda um ponteiro para o próximo nó da lista. Como teremos que inserir e retirar elementos das extremidades opostas da lista, que representarão o início e o fim da fila, teremos que usar dois ponteiros, *ini* e *fim*, que apontam respectivamente para o primeiro e para o último elemento da fila. Essa situação é ilustrada na figura abaixo:

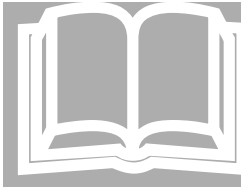


Estrutura de fila com lista encadeada.

A operação para retirar um elemento se dá no início da lista (fila) e consiste essencialmente em fazer com que, após a remoção, *ini* aponte para o

sucessor do nó retirado. (Observe que seria mais complicado remover um nó do fim da lista, porque o antecessor de um nó não é encontrado com a mesma facilidade que seu sucessor.) A inserção também é simples, pois basta acrescentar à lista um sucessor para o último nó, apontado por fim, e fazer com que fim aponte para este novo nó.

O nó da lista para armazenar valores reais, como já vimos, pode ser dado por:



```
struct no
{
    float info;
    struct no* prox;
};
typedef struct no No;
```

A estrutura da fila agrupa os ponteiros para o início e o fim da lista:

```
struct fila
{
    No* ini;
    No* fim;
};
```

A função cria aloca a estrutura da fila e inicializa a lista como sendo vazia.

```
Fila* cria (void)
{
    Fila* f = (Fila*) malloc(sizeof(Fila));
    f->ini = f->fim = NULL;
    return f;
}
```

Cada novo elemento é inserido no fim da lista e, sempre que solicitado, retiramos o elemento do início da lista. Desta forma, precisamos de duas funções auxiliares de lista: para inserir no fim e para remover do início. A função para inserir no fim ainda não foi discutida, mas é simples, uma vez que temos explicitamente armazenado o ponteiro para o último elemento. Essa função deve ter como valor de retorno o novo “fim” da lista. A função para retirar do início é idêntica à função usada na implementação de pilha.

```
/* função auxiliar: insere no fim */
No* ins_fim (No* fim, float v)
{
    No* p = (No*) malloc(sizeof(No));
    p->info = v;
    p->prox = NULL;
    if (fim != NULL) /* verifica se lista não estava vazia */
        fim->prox = p;
    return p;
}

/* função auxiliar: retira do início */
No* ret_ini (No* ini)
{
    No* p = ini->prox;
    free(ini);
    return p;
}
```

As funções que manipulam a fila fazem uso dessas funções de lista. Devemos salientar que a função de inserção deve atualizar ambos os ponteiros, ini e fim, quando da inserção do primeiro elemento. Analogamente, a função para retirar deve atualizar ambos se a fila tornar-se vazia após a remoção do elemento:

```
void insere (Fila* f, float v)
{
    f->fim = ins_fim(f->fim,v);
    if (f->ini==NULL) /* fila antes vazia? */
        f->ini = f->fim;
}

float retira (Fila* f)
{
    float v;
    if (vazia(f))
    {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->ini->info;
    f->ini = ret_ini(f->ini);
    if (f->ini == NULL) /* fila ficou vazia? */
        f->fim = NULL;
    return v;
}
```

A fila estará vazia se a lista estiver vazia:

```
int vazia (Fila* f)
{
    return (f->ini==NULL);
}
```

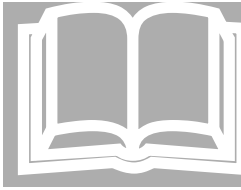
Por fim, a função que libera a fila deve antes liberar todos os elementos da lista.

```
void libera (Fila* f)
{
    No* q = f->ini;
    while (q!=NULL)
    {
        No* t = q->prox;
        free(q);
        q = t;
    }
    free(f);
}
```

Analogamente à pilha, para testar o código, pode ser útil implementarmos uma função que imprima os valores armazenados na fila. Os códigos abaixo ilustram a implementação dessa função nas duas versões de fila (vetor e lista). A ordem de impressão adotada é do início para o fim.

```
/* imprime: versão com lista */
void imprime (Fila* f)
{
    No* q;
    for (q=f->ini; q!=NULL; q=q->prox)
        printf("%f\n",q->info);
}
```

Um exemplo simples de utilização da estrutura de fila é apresentado a seguir:



```
/* Módulo para ilustrar utilização da fila */
#include <stdio.h>
#include "fila.h"
int main (void)
{
    Fila* f = cria();
    insere(f,20.0);
    insere(f,20.8);
    insere(f,21.2);
    insere(f,24.3);
    printf("Primeiro elemento: %f\n", retira(f));
    printf("Segundo elemento: %f\n", retira(f));
    printf("Configuracao da fila:\n");
    imprime(f);
    libera(f);
    return 0;
}
```

### 6.5 Fila dupla

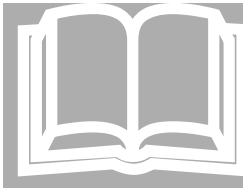
A estrutura de dados que chamamos de *fila dupla* consiste numa fila na qual é possível inserir novos elementos em ambas as extremidades, no início e no fim. Conseqüentemente, permite-se também retirar elementos de ambos os extremos. É como se, dentro de uma mesma estrutura de fila, tivéssemos duas filas, com os elementos dispostos em ordem inversa uma da outra.

A interface do tipo abstrato que representa uma fila dupla acrescenta novas funções para inserir e retirar elementos. Podemos enumerar as seguintes operações:

- + criar uma estrutura de fila dupla;
- + inserir um elemento no início;
- + inserir um elemento no fim;
- + retirar o elemento do início;
- + retirar o elemento do fim;
- + verificar se a fila está vazia;
- + liberar a fila.

O arquivo `fila2.h`, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct fila2 Fila2;
Fila2* cria (void);
void insere_ini (Fila2* f, float v);
void insere_fim (Fila2* f, float v);
float retira_ini (Fila2* f);
float retira_fim (Fila2* f);
int vazia (Fila2* f);
void libera (Fila2* f);
```



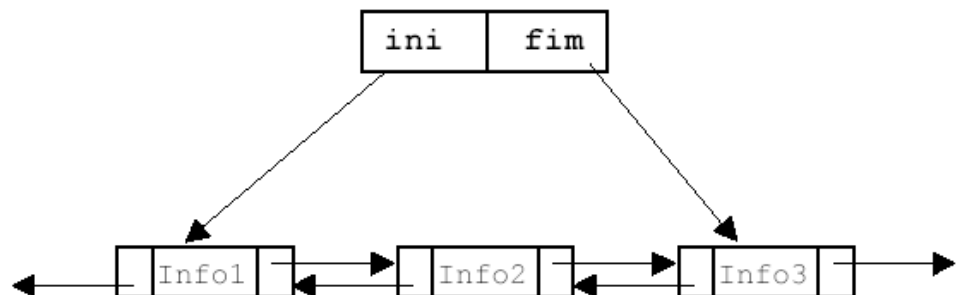
A implementação dessa estrutura usando um vetor para armazenar os elementos não traz grandes dificuldades, pois o vetor permite acesso randômico aos elementos, e fica como exercício.

## 6.6 Implementação de fila dupla com lista

A implementação de uma fila dupla com lista encadeada merece uma discussão mais detalhada. A dificuldade que encontramos reside na implementação da função para retirar um elemento do final da lista. Todas as outras funções já foram discutidas e poderiam ser facilmente implementadas usando uma lista simplesmente encadeada. No entanto, na lista simplesmente encadeada, a função para retirar do fim não pode ser implementada de forma eficiente, pois, dado o ponteiro para o último elemento da lista, não temos como acessar o anterior, que passaria a ser o então último elemento.

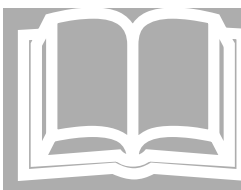
Para solucionar esse problema, temos que lançar mão da estrutura de lista duplamente encadeada. Nessa lista, cada nó guarda, além da referência para o próximo elemento, uma referência para o elemento anterior: dado o ponteiro de um nó, podemos acessar ambos os elementos adjacentes. Este arranjo resolve o problema de acessarmos o elemento anterior ao último. Devemos salientar que o uso de uma lista duplamente encadeada para implementar a fila é bastante simples, pois só manipulamos os elementos das extremidades da lista.

O arranjo de memória para implementarmos a fila dupla com lista é lustrado na figura abaixo:



Arranjo da estrutura de fila dupla com lista.

O nó da lista duplamente encadeada para armazenar valores reais pode ser dado por:



```
struct no2
{
    float info;
    struct no2* ant;
    struct no2* prox;
};
typedef struct no2 No2;
```

A estrutura da fila dupla agrupa os ponteiros para o início e o fim da lista:

```
struct fila2
{
```

```
No2* ini;  
No2* fim;  
};
```

Interessa-nos discutir as funções para inserir e retirar elementos. As demais são praticamente idênticas às de fila simples. Podemos inserir um novo elemento em qualquer extremidade da fila. Portanto, precisamos de duas funções auxiliares de lista: para inserir no início e para inserir no fim. Ambas as funções são simples e já foram exaustivamente discutidas para o caso da lista simples. No caso da lista duplamente encadeada, a diferença consiste em termos que atualizar também o encadeamento para o elemento anterior. Uma possível implementação dessas funções é mostrada a seguir.

Essas funções retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: insere no início */  
No2* ins2_ini (No2* ini, float v)  
{  
    No2* p = (No2*) malloc(sizeof(No2));  
    p->info = v;  
    p->prox = ini;  
    p->ant = NULL;  
    if (ini != NULL) /* verifica se lista não estava vazia */  
        ini->ant = p;  
    return p;  
}  
  
/* função auxiliar: insere no fim */  
No2* ins2_fim (No2* fim, float v)  
{  
    No2* p = (No2*) malloc(sizeof(No2));  
    p->info = v;  
    p->prox = NULL;  
    p->ant = fim;  
    if (fim != NULL) /* verifica se lista não estava vazia */  
        fim->prox = p;  
    return p;  
}
```

Uma possível implementação das funções para remover o elemento do início ou do fim é mostrada a seguir. Essas funções também retornam, respectivamente, o novo nó inicial e final.

```
/* função auxiliar: retira do início */  
No2* ret2_ini (No2* ini)  
{  
    No2* p = ini->prox;  
    if (p != NULL) /* verifica se lista não ficou vazia */  
        p->ant = NULL;  
    free(ini);  
    return p;  
}  
  
/* função auxiliar: retira do fim */  
No2* ret2_fim (No2* fim)  
{  
    No2* p = fim->ant;  
    if (p != NULL) /* verifica se lista não ficou vazia */  
        p->prox = NULL;  
    free(fim);  
    return p;  
}
```



As funções que manipulam a fila fazem uso dessas funções de lista, atualizando os ponteiros ini e fim quando necessário.

```
void insere_ini (Fila2* f, float v)
{
    f->ini = ins2_ini(f->ini,v);
    if (f->fim==NULL) /* fila antes vazia? */
        f->fim = f->ini;
}

void insere_fim (Fila2* f, float v)
{
    f->fim = ins2_fim(f->fim,v);
    if (f->ini==NULL) /* fila antes vazia? */
        f->ini = f->fim;
}

float retira_ini (Fila2* f)
{
    float v;
    if (vazia(f))
    {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->ini->info;
    f->ini = ret2_ini(f->ini);
    if (f->ini == NULL) /* fila ficou vazia? */
        f->fim = NULL;
    return v;
}

float retira_fim (Fila2* f)
{
    float v;
    if (vazia(f))
    {
        printf("Fila vazia.\n");
        exit(1); /* aborta programa */
    }
    v = f->fim->info;
    f->fim = ret2_fim(f->fim);
    if (f->fim == NULL) /* fila ficou vazia? */
        f->ini = NULL;
    return v;
}
```