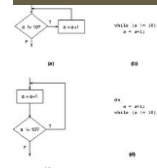


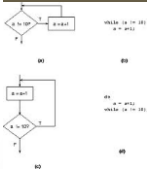
Estruturas de Dados

- Prof. Edkallenn Lima
- edkallenn@yahoo.com.br (somente para dúvidas)
- Blogs:
 - <http://professored.wordpress.com> (Computador de Papel – O conteúdo da forma)
 - <http://professored.tumblr.com/> (Pensamentos Incompletos)
 - <http://umcientistaporquinzena.tumblr.com/> (Um cientista por quinzena)
 - <http://eulinoslivros.tumblr.com/> (Eu Li nos Livros)
 - <http://linabiblia.tumblr.com/> (Eu Li na Bíblia)
- Redes Sociais:
 - <http://www.facebook.com/edkallenn>
 - <http://twitter.com/edkallenn>
 - <https://plus.google.com/u/0/113248995006035389558/posts>
 - Pinterest: <https://www.pinterest.com/edkallenn/>
 - Instagram: <http://instagram.com/edkallenn> ou [@edkallenn](#)
 - LinkedIn: br.linkedin.com/in/Edkallenn
 - Foursquare: <https://pt.foursquare.com/edkallenn>
- Telefones:
 - 68 98401-2103 (VIVO) e 68 3212-1211.
- Os exercícios devem ser enviados SEMPRE para o e-mail: edkevan@gmail.com ou para o e-mail: edkallenn.lima@uninorteac.edu.br



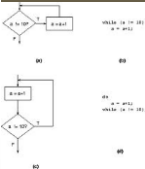
Agenda

- Ordenando e Pesquisando em arrays
- Bubblesort
- InsertionSort
- SelectionSort
- MergeSort
- Pesquisa Linear
- Pesquisa Binária
- Strings
- Funções de strings
- Biblioteca de strings
- Aritmética com endereços
- Alocação Dinâmica



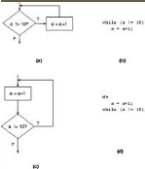
Ordenando Arrays (vetores)

- **Ordenar dados** é uma das mais importantes aplicações computacionais
- Ordenar é **colocar os dados em uma determinada ordem**, como ascendente ou descendente.
- Praticamente em **quase todas** as aplicações computacionais que lidam com dados deve-se, em algum momento, classificá-los.
- **Classificar** dados é um problema fascinante que tem atraído os esforços contínuos de pesquisadores no campo da ciência da computação
- Vamos analisar agora, o que talvez seja, o esquema **mais simples** de classificação



Bubblesort ou sinking sort

- A técnica chamada *bubblesort* ou *classificação de bolhas* ou *classificação de submersão* (*sinking sort*) é um dos algoritmos de ordenação mais simples que existem
- A ideia é **percorrer o vetor diversas vezes**, a cada passagem fazendo flutuar para o topo o maior elemento da sequência.
- Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo
- A versão que veremos hoje é a **mais simples** possível (sem a flag houvetroca)



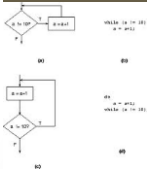
Função de ordenação (bubblesort)

```

void ordena_bolha_int(int tamanho, int a[]){
    int pass, i, aux;
    for (pass=1; pass<tamanho; pass++) //passadas
        for (i=0; i<=tamanho-2; i++) //uma passada
            if (a[i]>a[i+1]){ //uma comparacao
                aux=a[i]; //uma permuta
                a[i]=a[i+1];
                a[i+1]=aux;
            }
}

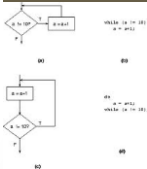
```

- A função **recebe** o **tamanho** do vetor e o **próprio vetor**, chamado internamente de **a[]**
- A técnica é **passar várias vezes pelo vetor**. Em cada passada (ou loop) são **comparados pares sucessivos** de elementos.
- Se um par **estiver na ordem crescente** (ou se os valores são iguais) são deixados como estão.
- Se um par **estiver na ordem decrescente**, seus valores são **permutados**.



Função de ordenação (bubblesort)

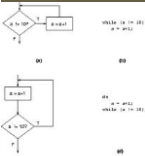
- Inicialmente o programa **compara a[0] com a[1]**, depois a[1] com a[2], depois a[2] com a[3] e assim por diante, até completar a passada comparando a[8] com a[9] (ou a[TAM-1])
- Observe que ele executa TAM-1 comparações.
- Tendo em vista como são feitas as comparações **um valor grande pode se mover para baixo várias posições**, mas um valor pequeno só pode se mover para cima uma única posição.
- Na primeira passada, garante-se que o maior valor **“submergirá”** para o elemento mais baixo (para o fundo) do vetor, a[TAM-1]. Na segunda, garante-se que o segundo maior valor submergirá para a[TAM-2]
- A principal virtude desta classificação reside em sua simplicidade de codificação.
- Mas ela é **LENTA**! Desenvolveremos versões mais eficientes para arrays grandes nas próximas aulas!



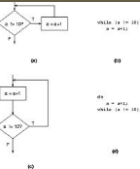
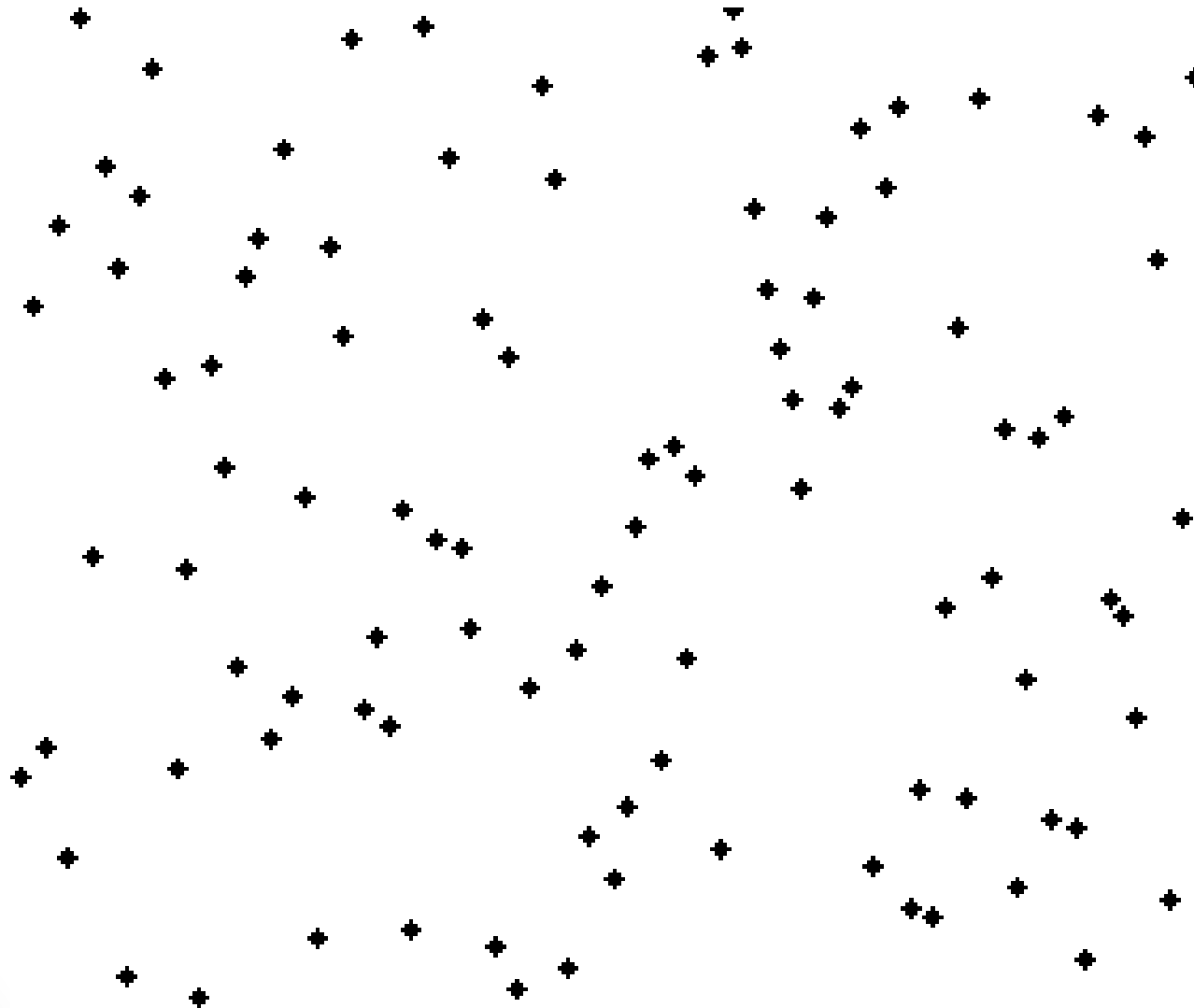
```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Classifica usando bubblesort (vetores)
4     Autor : Edkallenn - Data : 10/04/2012
5     Obs: Esta é a versão mais simples, mas com funcoes
6  */
7  #define TAM 10 //tamanho maximo do vetor
8  void ordena_bolha_int(int tamanho, int *);
9  void preenche_vetor(int n, int []);
10 void exibe_vetor(int tamanho, int *);
11
12 void ordena_bolha_int(int tamanho, int a[]){
13     int pass, i, aux;
14     for (pass=1;pass<tamanho;pass++) //passadas
15         for(i=0;i<=tamanho-2;i++) //uma passada
16             if(a[i]>a[i+1]){ //uma comparacao
17                 aux=a[i]; //uma permuta
18                 a[i]=a[i+1];
19                 a[i+1]=aux;
20             }
21 }
22
23 main(){
24     int vetor[TAM];
25     preenche_vetor(TAM, vetor);
26     printf("\nO vetor digitado eh\n");
27     exibe_vetor(TAM, vetor);
28     ordena_bolha_int(TAM, vetor);
29     printf("\nO vetor ordenado eh\n");
30     exibe_vetor(TAM, vetor);
31     printf("\n\n");
32     getchar();
33 }
34
35 void preenche_vetor(int tamanho, int vet[]){ // Preenche o vetor
36     int i;
37     for (i=0;i<tamanho;++i){
38         printf("\nDigite o elemento %d do vetor: ", i);
39         scanf("%d", &vet[i]);
40     }
41 }
42
43 void exibe_vetor(int tamanho, int v[]){ //Exibe
44     int t;
45     for (t=0;t<tamanho;t++)
46         printf("%-4d ", v[t]);
47 }

```



Animação - BubbleSort



BubbleSort em Python

```
# -*- encoding: utf-8 -*-
```

```
"""
```

```
Bubble-Sort sem Flag em Python
```

```
by Ed
```

```
"""
```

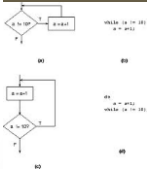
```
def bubble_sort(lista):
    for i in range(0, len(lista)-1):
        for j in range(0, len(lista)-1-i):
            if lista[ j ] < lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
            #print(lista)
    return lista

while True:
    valor_ini = input("Digite os números do seu vetor, como 10,20,30, digite:")
    valores = eval "[" + valor_ini + "]" #interpreta a sequencia como uma lista
    print (bubble_sort(valores))
```



Exercícios – Fazer em sala

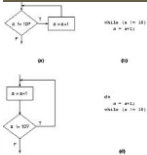
- Reescrever a função bubblesort para classificar em ordem decrescente
- Fazer um programa para receber um vetor desordenado e classificar em ordem crescente e exibir os valores, em seguida, em ordem decrescente e exibi-lo novamente.
- Gerar valores aleatórios para o vetor e em seguida, classificá-los em ordem crescente e decrescente e exibi-lo nas duas ordens
- Rodar o programa para TAM = 100, 200, 500, 1000 e 2000 com valores randômicos (gerados até num – reescrever a função preenche_vetor_random para receber o num, assim:
`void preenche_vetor_random(int tamanho, int num, int vet[]);`



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  /* Função : Classifica usando bubblesort (vetores)
5     Autor : Edkallenn - Data : 10/04/2012
6     Obs: Esta é a versão mais simples, mas com funcoes
7  */
8  #define TAM 100 //tamanho maximo do vetor
9  void ordena_bolha_desc_int(int tamanho, int *);
10 void ordena_bolha_asc_int(int tamanho, int *);
11 void preenche_vetor_random(int tamanho, int num, int vet[]) //funcao melhorada
12 void exhibe_vetor_random(int tamanho, int *);
13
14 void ordena_bolha_desc_int(int tamanho, int a[]){
15     int pass, i, aux;
16     for (pass=1;pass<tamanho;pass++) //passadas
17         for(i=0;i<=tamanho-2;i++) //uma passada
18             if(a[i]<a[i+1]){ //uma comparacao
19                 aux=a[i]; //uma permuta
20                 a[i]=a[i+1];
21                 a[i+1]=aux;
22             }
23 }
24 void ordena_bolha_asc_int(int tamanho, int a[]){
25     int pass, i, aux;
26     for (pass=1;pass<tamanho;pass++) //passadas
27         for(i=0;i<=tamanho-2;i++) //uma passada
28             if(a[i]>a[i+1]){ //uma comparacao
29                 aux=a[i]; //uma permuta
30                 a[i]=a[i+1];
31                 a[i+1]=aux;
32             }
33 }

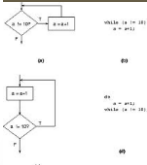
```



```

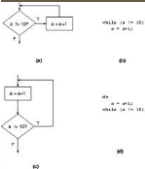
34 main() {
35     int vetor[TAM];
36     srand((unsigned)time(NULL));
37     preenche_vetor_random(TAM, 1000, vetor);
38
39     printf("\nO vetor gerado eh\n");
40     exibe_vetor(TAM, vetor);
41
42     ordena_bolha_desc_int(TAM, vetor);
43     printf("\n\nO vetor em ordem decrescente eh\n");
44     exibe_vetor(TAM, vetor);
45
46     ordena_bolha_asc_int(TAM, vetor);
47     printf("\n\nO vetor em ordem crescente eh\n");
48     exibe_vetor(TAM, vetor);
49
50     printf("\n\n");
51     getchar();
52 }
53 void preenche_vetor_random(int tamanho, int num, int vet[]){
54     int i, valor; // Preenche com valores randomicos melhor
55     for (i=0;i<tamanho;++i){
56         valor = (1 + random(num-1)); //gera ate num
57         vet[i]=valor;
58     }
59 }
60 void exibe_vetor(int tamanho, int v[]){ //Exibe
61     int t;
62     for (t=0;t<tamanho;t++)
63         printf("%-4d ", v[t]);
64 }
65 int random(int n){ //funcao para gerar aleatorios
66     return rand() % n;
67 }

```

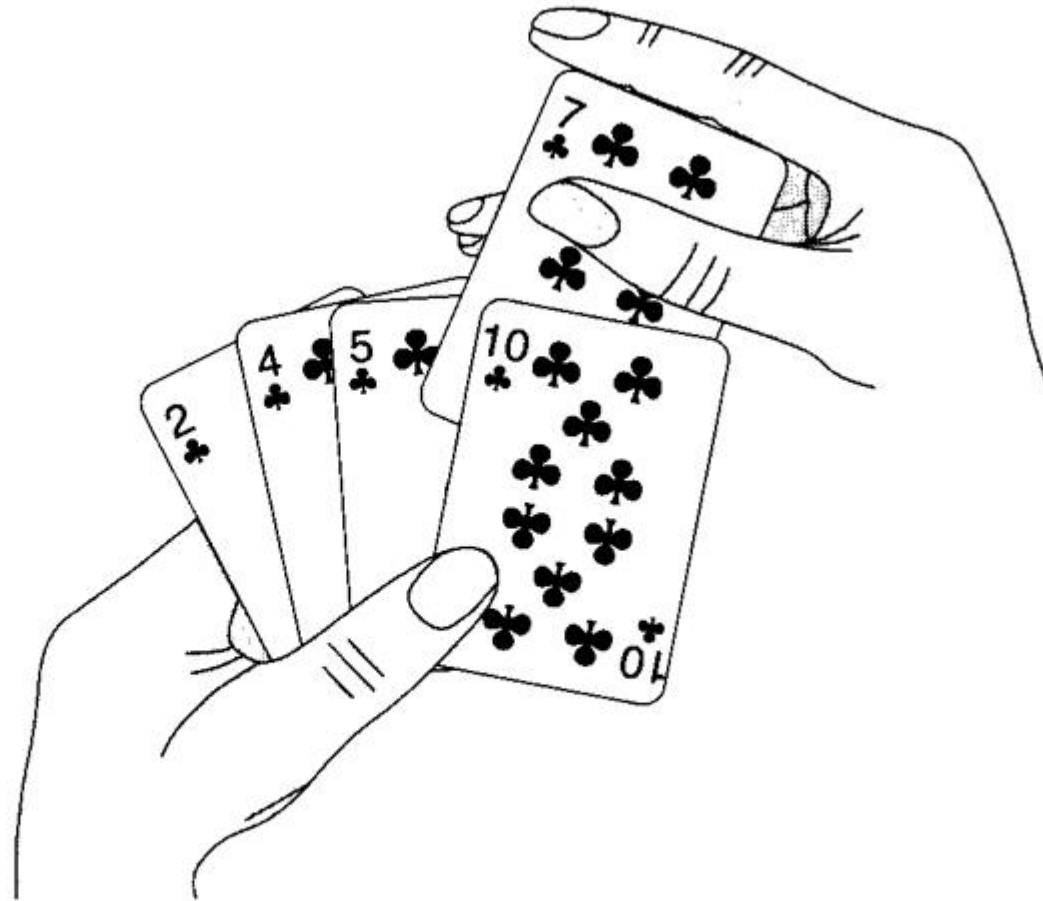


Ordenação por inserção (Insertion Sort)

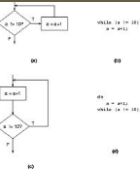
- O algoritmo de ordenação por inserção é muito **popular**
- Ele é frequentemente usado para **colocar em ordem** um **pequeno número** de elementos
- Em termos gerais ele **percorre um vetor de elementos** da **esquerda** para a **direita** e à medida que **avança** vai deixando os elementos **mais à esquerda ordenados**
- O algoritmo de inserção funciona da mesma maneira com que muitas pessoas **ordenam cartas** em um jogo de pôquer



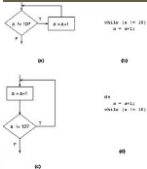
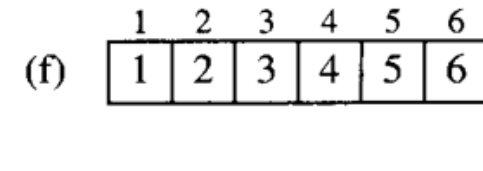
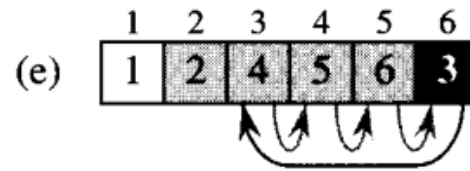
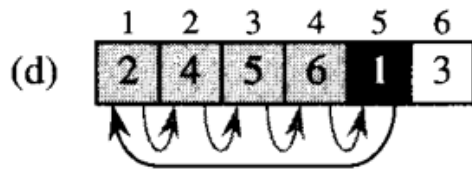
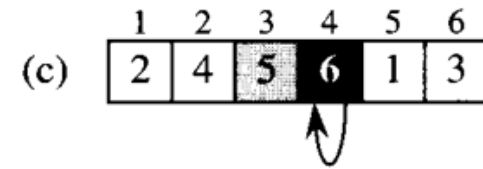
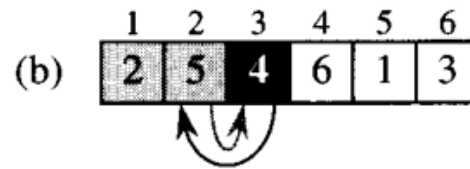
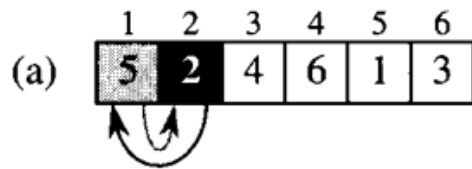
Insertion Sort



Ordenando cartas com o uso da ordenação por inserção

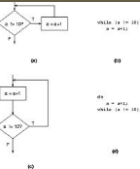


Exemplo de funcionamento



Insertion Sort (funcionamento)

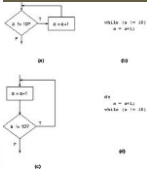
6 5 3 1 8 7 2 4



Exemplo (passo-a-passo):

- **A sequência:**
- **5 - 3 - 1 - 4 - 2**
- Inicialmente considera-se o primeiro elemento do array como se ele estivesse ordenado; ele será considerado o sub-array ordenado inicial.
- Agora o elemento imediatamente superior ao o sub-array ordenado, no o exemplo o número 3, deve se copiado para uma variável auxiliar qualquer. Após copiá-lo, devemos percorrer o sub-array a partir do último elemento para o primeiro. Assim poderemos encontrar a posição correta da nossa variável auxiliar dentro do sub-array.
- No caso verificamos que a variável auxiliar é menor que o último elemento do o sub-array ordenado (o o subarray só possui por enquanto um elemento, o número 5).
- O número 5 deve então ser copiado uma posição para a direita para que a variável auxiliar com o número 3, seja colocada em sua posição correta.

3 - 5 - 1 - 4 - 2

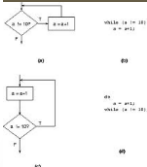


- Verifique que o sub-array ordenado possui agora dois elementos.
- Vamos repetir o processo anterior para que se continue a ordenação.
- Copiamos então mais uma vez o elemento imediatamente superior ao o sub-array ordenado para uma variável auxiliar. Logo em seguida vamos comparando nossa variável auxiliar com os elementos do subarray, sempre a partir do último elemento para o primeiro
- Neste caso verificamos que a nossa variável auxiliar é menor que o último elemento do sub-array.
- Assim, copiamos este elemento para a direita e continuamos com nossas comparações (5 permanece como cópia no lugar do 1).
- Aqui, mais uma vez a nossa variável auxiliar é menor que o elemento do sub-array que estamos comparando.
- Por isso ele deve ser copiado para a direita, abrindo espaço para que a variável auxiliar seja colocada em sua posição correta.

1 - 3 - 5 - 4 - 2

- Aplicando o algoritmo até que se chegue ao fim da sequência, poderá resultar a seguinte sequência:

- **1 - 3 - 4 - 5 - 2**
1 - 2 - 3 - 4 - 5



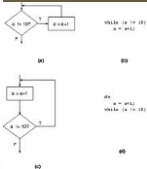
Função insertionSort_int (int, int *)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define TAM 10 //tamanho maximo do vetor
5  void insertionSort_Int(int tamanho, int *);
6  void exhibeVetor(int tamanho, int *);
7  void preencheVetorRandom_int(int tamanho, int maior, int vet[]);
8  int random(int);
9  /*
10  ** A funcao InsertionSort rearranja o vetor v[0..n-1]
11  ** em ordem crescente
12  ** Autor: Ed em 10/04/2012
13  */
14 void insertionSort_Int(int n, int vetor[]){
15     int i, j, eleito;
16     for(j=1; j<n; j++){
17         eleito = vetor[j];
18         for(i=j-1; i>=0 && vetor[i] > eleito; i--){
19             vetor[i+1]=vetor[i];
20             vetor[i+1]=eleito;
21             //exibeVetor(TAM, vetor); printf("\n");
22         }
23     }

```

• Versão 1



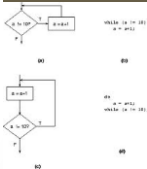
Versão alternativa

```

1 void insertionSort(int numeros[], int tam) {
2     int i, j, eleito;
3     for (i = 1; i < tam; i++){
4         eleito = numeros[i];
5         j = i - 1;
6         while ((j >= 0) && (eleito < numeros[j])) {
7             numeros[j+1] = numeros[j];
8             j--;
9         }
10        numeros[j+1] = eleito;
11    }
12 }

```

- Versão 2



InsertionSort em Python

```
# -*- encoding: utf-8 -*-
```

```
"""
```

```
Insertion Sort em Python
```

```
by Ed
```

```
"""
```

```
def insertion_sort(lista):
    for i in range(1,len(lista)):
        eleito = lista[i]
        j = i-1
        while j >= 0 and eleito < lista[j]:
            lista[j+1] = lista[j]
            j=j-1
        lista[j+1] = eleito
        print(lista)
```

```
    return lista
```

```
while True:
```

```
    valor_ini = input("Digite os números do seu vetor, como 10,20,30, digite:")
```

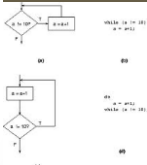
```
    valores = eval("["+valor_ini+"]") #interpreta a sequencia como uma lista
```

```
    print (insertion_sort(valores))
```

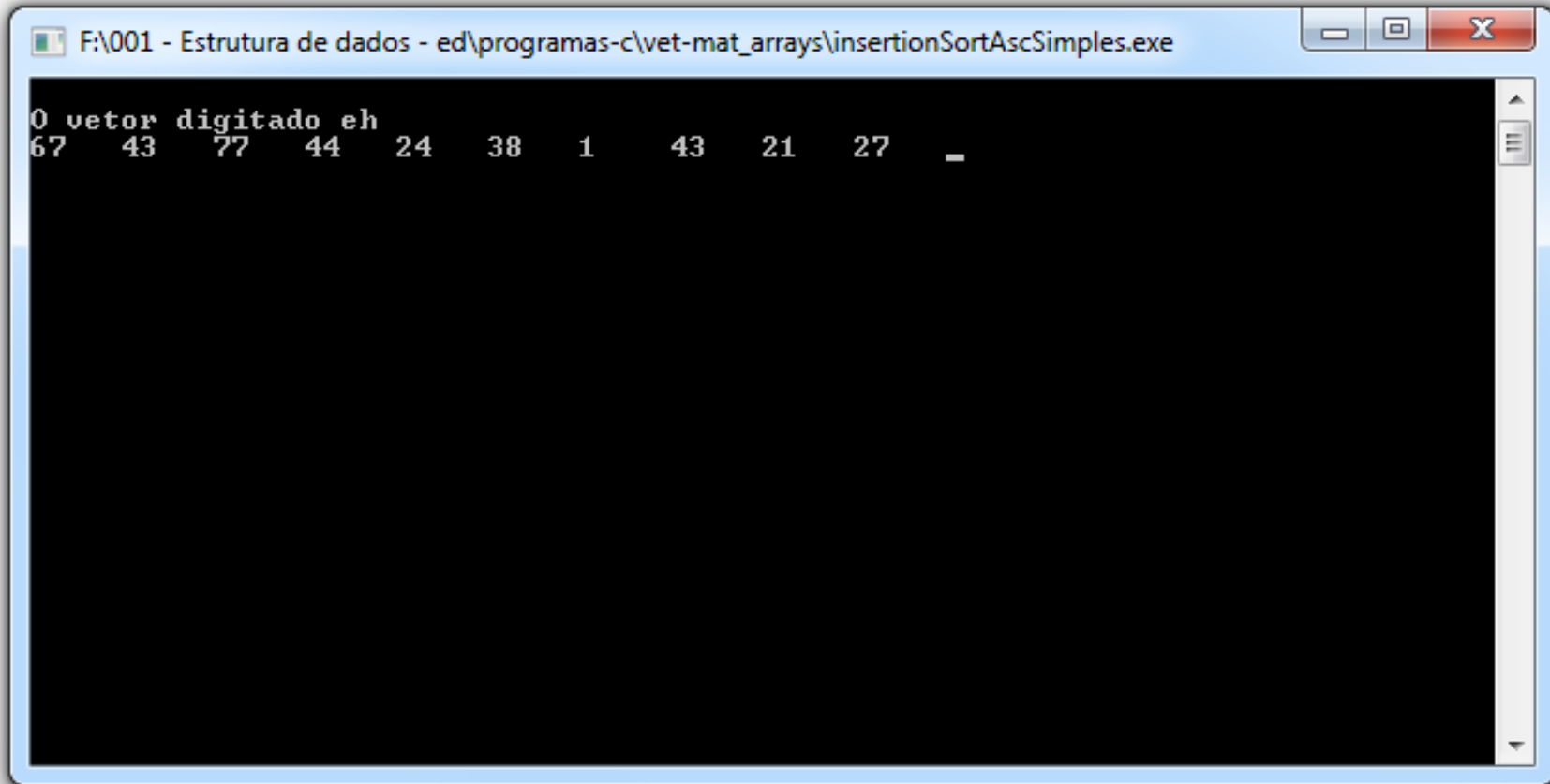


Insertion Sort

- Para entender o funcionamento do algoritmo, basta observar que no início de cada repetição do for externo, imediatamente antes da comparação de j com n ,
 - o vetor $v[0..n-1]$ **é uma permutação** do vetor original e
 - o vetor $v[0..j-1]$ **está em ordem crescente**.
- Estas condições **invariantes** são trivialmente verdadeiras no início da primeira iteração, quando j vale 1.
- No início da última iteração, j vale n e portanto o vetor $v[0..n-1]$ está em ordem, como desejado.
- (Note que a última iteração é abortada logo no início, pois a condição " $j < n$ " é falsa.)

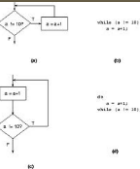


Execução da função (valores aleatórios)



```
F:\001 - Estrutura de dados - ed\programas-c\vet-mat_arrays\insertionSortAscSimples.exe

O vetor digitado eh
67 43 77 44 24 38 1 43 21 27 _
```



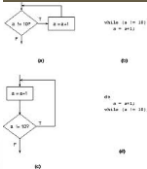
Execução da função

```

F:\001 - Estrutura de dados - ed\programas-c\vet-mat_arrays\insertionSortAscSimples.exe

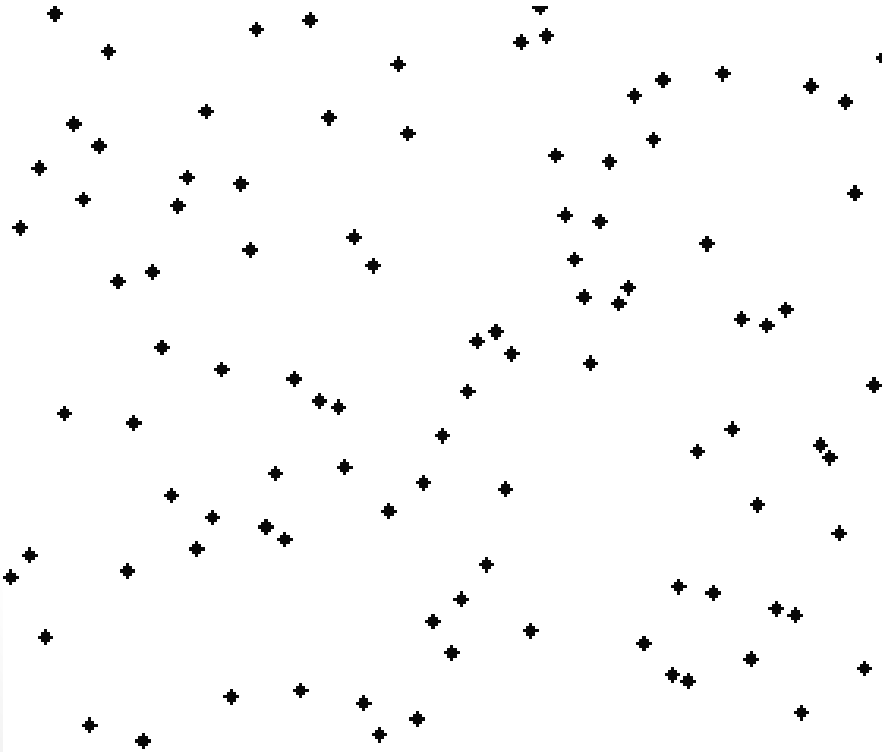
0 vetor digitado eh
67 43 77 44 24 38 1 43 21 27
43 67 77 44 24 38 1 43 21 27
43 67 77 44 24 38 1 43 21 27
43 44 67 77 24 38 1 43 21 27
24 43 44 67 77 38 1 43 21 27
24 38 43 44 67 77 1 43 21 27
1 24 38 43 44 67 77 43 21 27
1 24 38 43 43 44 67 77 21 27
1 21 24 38 43 43 44 67 77 27
1 21 24 27 38 43 43 44 67 77

0 vetor ordenado em ordem crescente eh
1 21 24 27 38 43 43 44 67 77
  
```

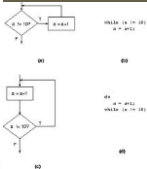


Características do Insertion Sort

- Menor número de trocas e comparações entre os algoritmos de ordenação quando o vetor está ordenado
- O pior caso (o vetor totalmente desordenado) é lento.

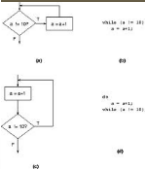


Exemplo de funcionamento do Insertion Sort em uma lista de inteiros aleatórios



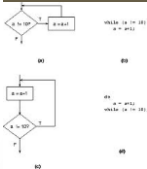
Mais sobre o Insertion Sort

- **Algoritmo de inserção: animações e vídeos**
- Veja alguns applets de animação do algoritmo de inserção:
- [Insertion Sort Algorithm](#): aula de Salman Kahn
- [Sorting Algorithms Animation](#), by David R. Martin
- [Applet de R. Sedgewick](#) na Universidade de Princeton
- [Animação de algoritmos de ordenação](#) de Nicholas André Pinho de Oliveira
- [Insert-sort with Romanian folk dance](#) created at Sapientia University (Romania)

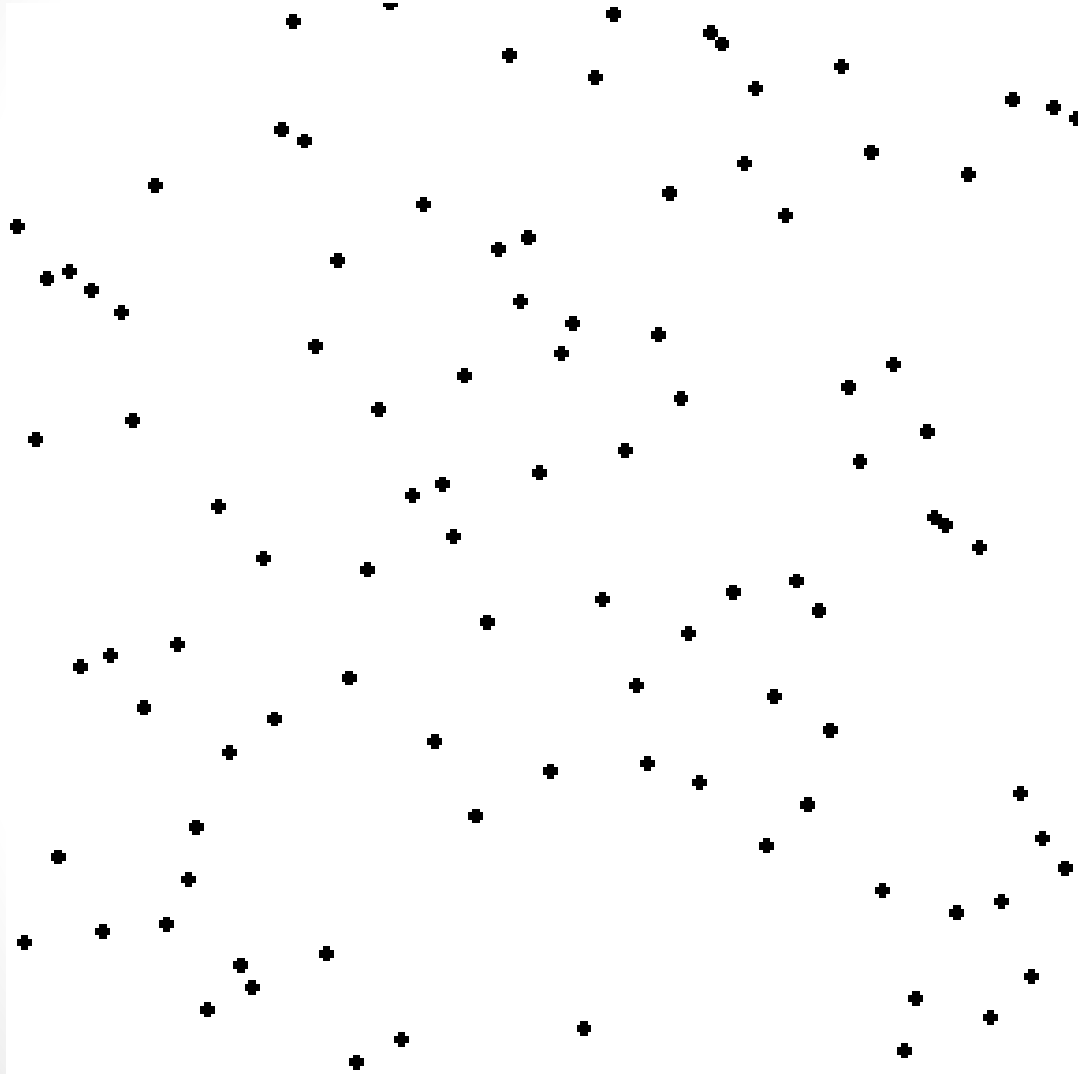


SelectionSort (Ordenação por seleção)

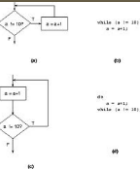
- É um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os $(n-1)$ elementos restantes, até os últimos dois elementos
- A complexidade também é quadrática no pior caso – $O(n^2)$



SelectionSort



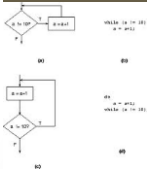
Exemplo de funcionamento do SelectionSort em uma lista de inteiros aleatórios



SelectionSort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Exemplo de funcionamento do SelectionSort em uma lista de inteiros



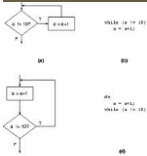
Selection Sort

● SelectionSort.c Ordenacao

```

1  /* Função : Classifica usando SelectionSort (vetores)
2      Autor : Edkallenn - Data : 10/04/2012
3      Obs: Esta é a versão mais simples, mas com funcoes
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <time.h>
8  #define INFINITO 999999
9  #define TAM_MAX  50
10
11 void SelectionSort(int x[], int n);
12 void troca(int v[], int i, int j);
13
14 void troca(int v[], int i, int j){ //funcao para trocar dois elementos
15     int aux;
16     aux = v[i];
17     v[i] = v[j];
18     v[j] = aux;
19 }

```



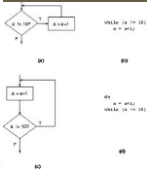
Selection Sort

• SelectionSort.c Ordenacao

```

20
21 void SelectionSort(int x[], int n){
22     int menor,pos;
23     int i,k = 0;
24
25     while (k < n){
26         menor = INFINITO;
27         for (i = k; i < n; i++)
28             if (x[i] < menor){
29                 menor = x[i];
30                 pos = i;
31             }
32         troca(x,k,pos);
33         k++;
34     }
35 }

```

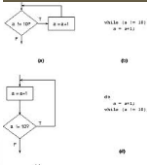


Selection Sort

```

37  int main(){
38      int i,n,a[TAM_MAX];
39
40      // Inicializar gerador de números aleatórios
41      srand((unsigned)time(NULL));
42
43      // Vetor gerado aleatoriamente
44      n = (rand() % TAM_MAX)+1;
45      printf("Vetor original: ");
46      for (i = 0; i < n; i++)
47      {
48          a[i] = (rand() % 100);
49          printf("%d ",a[i]);
50      }
51      printf("\n");
52
53      // Classificar vetor
54      SelectionSort(a,n);
55      printf("Vetor ordenado: ");
56      for (i = 0; i < n; i++)
57          printf("%d ",a[i]);
58      printf("\n");
59      system("PAUSE");
60      return 0;
61  }

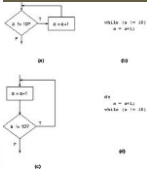
```



Execução

- Execução da função para valores gerados aleatoriamente

```
ordenacao — -bash — 129x28
edkallenn@MacBook-Ed:~/Dropbox/001 - Estrutura de dados/programas-c/ordenacao$ ./selection_sort
Vetor original: 55 63 15 29 95 2 93 86 34 85 8 71 81 31 16 14 14 18 33 78 88 70 79 92 45 91
Vetor ordenado: 2 8 14 14 15 16 18 29 31 33 34 45 55 63 70 71 78 79 81 85 86 88 91 92 93 95
edkallenn@MacBook-Ed:~/Dropbox/001 - Estrutura de dados/programas-c/ordenacao$
```



Selection Sort

```
# -*- encoding: utf-8 -*-
"""
```

```
Selection-Sort sem Flag em Python
by Ed
"""
```

```
def selection_sort(uma_lista):
```

```
    for i in range(len(uma_lista)):
```

```
        # Encontra o menor elemento restante
        posicao_menor = i
```

```
        for j in range(i+1, len(uma_lista)):
```

```
            if uma_lista[posicao_menor] > uma_lista[j]:
                posicao_menor = j
```

```
        # troca o menor elemento
```

```
        temp = uma_lista[i]
```

```
        uma_lista[i] = uma_lista[posicao_menor]
```

```
        uma_lista[posicao_menor] = temp
```

```
        print(uma_lista)
```

```
    |
```

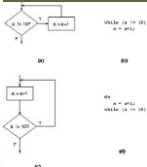
```
    return uma_lista
```

```
while True:
```

```
    valor_ini = input("Digite os números do seu vetor, como 10,20,30, digite:")
```

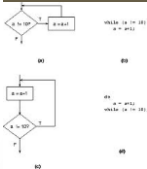
```
    valores = eval("[ "+valor_ini+" ]") #interpreta a sequencia como uma lista
```

```
    print (selection_sort(valores))
```



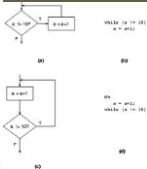
MergeSort (Ordenação por intercalação)

- O próximo algoritmo de ordenação por intercalação tem seu tempo de execução de apenas $\Theta(n \cdot \lg n)$ em todos os casos
- Quando se compara com a ordenação por **seleção** ou **inserção** (de pior caso $\Theta(n^2)$), troca-se um fator de n por um fator de apenas $\lg n$
- A maior **desvantagem** é que ela tem que fazer **cópias completas** do vetor de entrada, o que aumenta a **complexidade de espaço**.



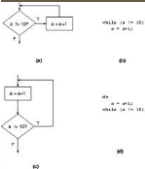
MergeSort (Ordenação por intercalação)

- Esboço geral do algoritmo **dividir-para-conquistar**:
 - **Divida** o problema em vários **subproblemas** que são instâncias **menores** do mesmo problema
 - **Conquiste** os subproblemas resolvendo-os **recursivamente**; Se eles não forem suficientemente pequenos, resolva os subproblemas como **casos-base**
 - **Combine** as soluções para os **subproblemas** na solução para o **problema original**

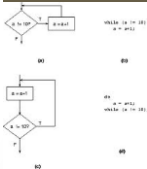
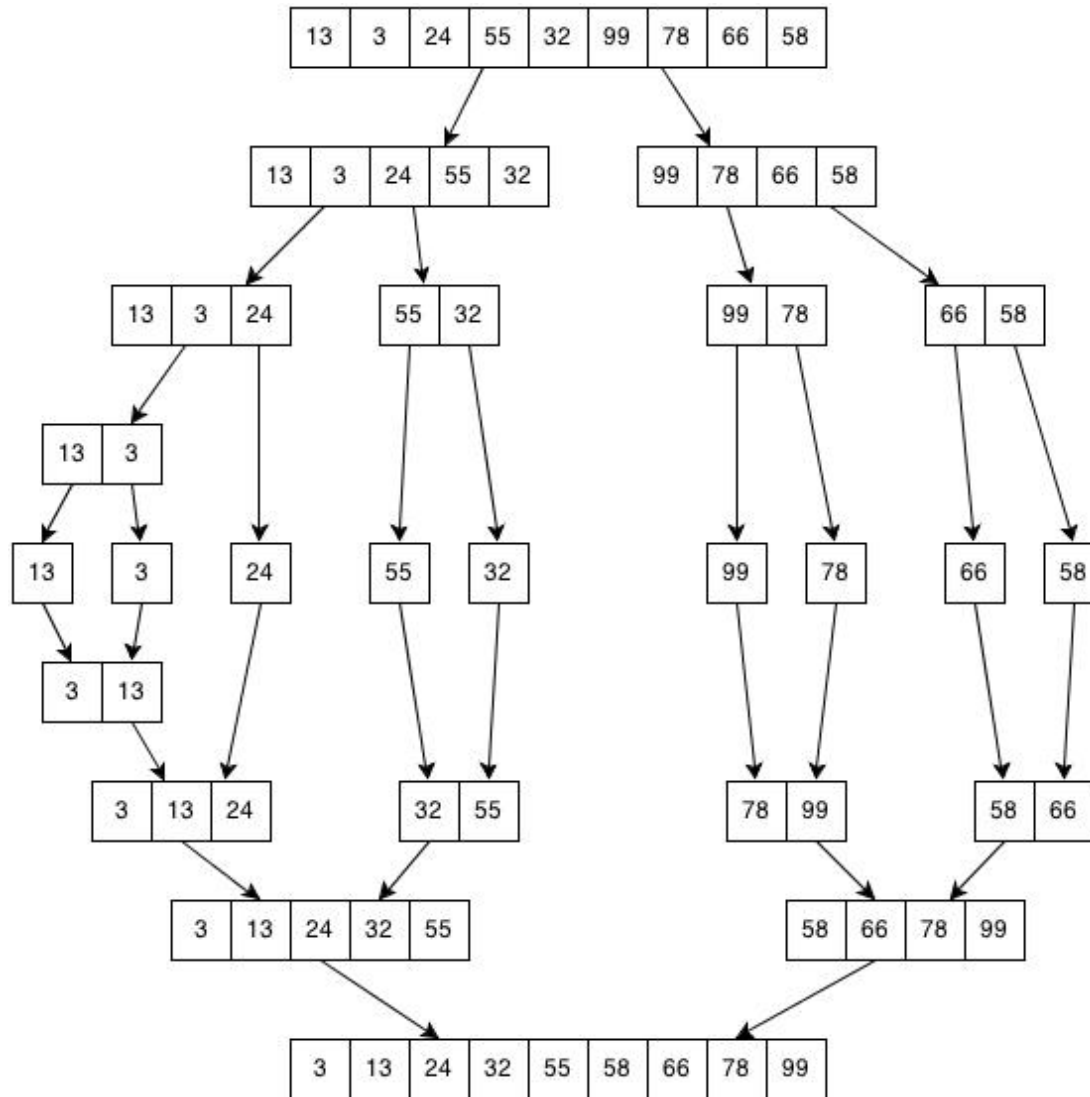


MergeSort (Ordenação por intercalação)

- Ele usa a técnica de projeto, **dividir-para-conquistar**.
- Nesse paradigma algorítmico, **desmembra-se** o problema em **subproblemas** semelhantes ao problema original;
- Em seguida **resolve-se** os **subproblemas** recursivamente
- E então **combina-se** as soluções para os **subproblemas** para resolver o problema original

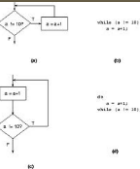


Exemplo:



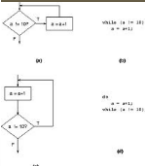
Exemplo animado

6 5 3 1 8 7 2 4



Características

- Algoritmo Criado por **Von Neumann** em 1945.
- Complexidade de **tempo**: $\Theta(n \log_2 n)$
- Complexidade de **espaço**: $\Theta(n)$
- Utiliza **funções recursivas**; e
- Gasto **extra** de **memória**.
- O algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a **($n \log n$)**



<pre> graph TD Start(()) --> Cond{a < 100} Cond -- Y --> Inc[a = a + 1] Inc --> Cond Cond -- N --> Exit(()) </pre> <p>(a)</p>	<pre> while (a < 10) a = a + 1; </pre> <p>(b)</p>
<pre> graph TD Start(()) --> Inc[a = a + 1] Inc --> Cond{a < 100} Cond -- Y --> Inc Cond -- N --> Exit(()) </pre> <p>(c)</p>	<pre> do a = a + 1; while (a < 10); </pre> <p>(d)</p>

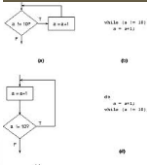
Código mergeSort

● MergeSort-Ed2-Random.cpp Ordenacao

```

87 void mergeSort(int *V, int inicio, int fim){
88     int meio;
89     if(inicio < fim){
90         meio = floor((inicio+fim)/2);
91
92         printf("Divide:   ");exibeSubVetor(V,inicio, fim);
93         printf("           ");exibeSubVetor(V,inicio, meio);
94         printf("           ");exibeSubVetor(V,meio+1, fim);
95
96         mergeSort(V,inicio,meio);
97         mergeSort(V,meio+1,fim);
98         merge(V,inicio,meio,fim);
99     }
100 }
101

```



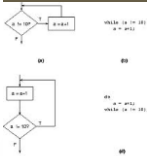
Funções auxiliares

• MergeSort-Ed2-Random.cpp Ordenacao

```

28 }
29 void geraVetorRandom(int vet[], int tamanho, int n){ // Preenche com valores randomicos
30     int i, valor;
31     for (i=0;i<tamanho;++i){
32         valor = (1 + random(n-1)); //gera ate n
33         vet[i]=valor;
34     }
35 }
36 void exibeVetorInt(int v[], int n){ //Exibe o vetor
37     int t; //n ão o tamanho
38     for (t=0;t<n;t++)
39         printf("%-3d ", v[t]);
40     printf("\n");
41 }
42 int random(int n){ //funcao para gerar aleatorios
43     return rand() % n;
44 }
45 void exibeSubVetor( int array[], int primeiro, int ultimo )
46 {
47     int i;
48     for ( i = 0; i < primeiro; i++ )
49         printf( "    " );
50     for (i = primeiro; i <=ultimo; i++ )
51         printf( " %d", array[ i ] );
52     printf("\n");
53 }

```



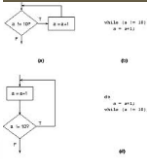
Protótipos e main()

• MergeSort-Ed2-Random.cpp Ordenacao

```

1  /*=====
2  ** Função : Ordenação MergeSort
3  ** Autor : Edkallenn
4  ** Data : 22/06/2015
5  **=====*/
6  #include <stdio.h>
7  #include <stdlib.h>
8  #define MAX 10
9  #define QL printf("\n")
10 //protótipos
11 void geraVetorRandom(int vet[], int tamanho, int n);
12 void exhibeVetorInt(int v[], int n);
13 void merge(int *V, int inicio, int meio, int fim);
14 void mergeSort(int *V, int inicio, int fim);
15 void exhibeSubVetor( int array[], int primeiro, int ultimo );
16 int random(int n);
17
18 int main(){
19     srand((unsigned)time(NULL));
20     int i, vetor[MAX];
21     geraVetorRandom(vetor,MAX,100);
22     printf("\n0 vetor gerado pelo computador eh:\n\n");
23     exhibeVetorInt(vetor,MAX); printf("\nIteracoes\n");QL;
24     mergeSort(vetor,0,MAX-1); //chamada para mergesort
25     printf("\nVetor Ordenado:\n");
26     exhibeVetorInt(vetor,MAX); //exibe vetor ordenado
27     getch();
28 }

```



```

67 79 21 60 5 67 94 31 27 75
Iteracoes
Divide:      67 79 21 60 5 67 94 31 27 75
             67 79 21 60 5
                   67 94 31 27 75
Divide:      67 79 21 60 5
             67 79 21
                   60 5
Divide:      67 79 21
             67 79
                   21
Divide:      67 79
             67
                   79
Combina:     67
             67
Combina:     67 79
             21 67
Divide:      60 5
             60
                   5
Combina:     60
             5
Combina:     21 67 79
             5
             5 21 60 67

```

```

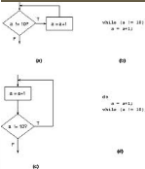
Divide:      67 94 31 27 75
             67 94 31
                   27 75
Divide:      67 94 31
             67 94
                   31
Divide:      67 94
             67
                   94
Combina:     67
             67
Combina:     67 94
             31 67
Divide:      27 75
             27
                   75
Combina:     27
             27
Combina:     31 67 94
             27
             27 31 67 75
Combina:     5 21 60 67 79
             27 31 67 75
             5 21 27 31 60 67 67 75 79

```

```

Vetor Ordenado:
5 21 27 31 60 67 67 75 79 94

```



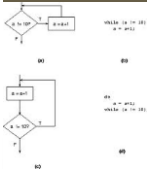
Versão “enxuta”

```

55 void merge(int p, int q, int r, int v[]){
56     /*  A função merge consiste essencialmente em movimentar
57         elementos do vetor v de um lugar para outro
58         (primeiro de v para w e depois de w para v)
59
60     */
61     int i, j, k; // *w;
62
63     int w[(r-p)];
64     i = p; j = q;
65     k = 0;
66
67     while (i < q && j < r) {
68         if (v[i] <= v[j]) w[k++] = v[i++];
69         else w[k++] = v[j++];
70     }
71     while (i < q) w[k++] = v[i++];
72     while (j < r) w[k++] = v[j++];
73     for (i = p; i < r; ++i)
74         v[i] = w[i-p];
75 }

```

A função **merge** consiste essencialmente em **movimentar** elementos do vetor **v** de um lugar para outro (primeiro de **v** para **w** e depois de **w** para **v**). A função executa **2n** movimentações, sendo **n** o tamanho do **vetor v[p..r-1]**

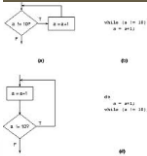


main() para versão “enxuta”

```

6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <time.h>
9  #define MAX 20
10 #define QL printf("\n")
11
12 void geraVetorRandom(int vet[], int tamanho, int n);
13 void exhibeVetorInt(int v[], int n);
14 void merge(int p, int q, int r, int v[]);
15 void mergeSort(int *v, int p, int r);
16 void exhibeSubVetor( int array[], int primeiro, int ultimo );
17 int random(int n);
18
19 □ int main(){
20     srand((unsigned)time(NULL));
21     int i, vetor[MAX];
22     geraVetorRandom(vetor,MAX,100);
23     printf("\n0 vetor gerado pelo computador eh:\n\n");
24     exhibeVetorInt(vetor,MAX); printf("\nIteracoes\n");QL;
25     mergeSort(vetor,0,MAX);
26     printf("\nVetor Ordenado:\n");
27     exhibeVetorInt(vetor,MAX);
28     getchar();
29 }

```



Execução (versão “enxuta”)

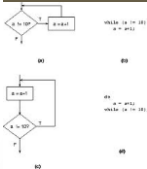
O vetor gerado pelo computador eh:

63 71 51 63 31 2 75 18 71 89 88 28 58 91 46 3 75 77 33 59

Iteracoes

Vetor Ordenado:

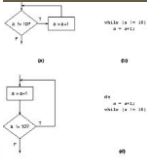
2 3 18 28 31 33 46 51 58 59 63 63 71 71 75 75 77 88 89 91



MergeSort em Python

(def merge)

```
# -*- encoding: utf-8 -*-
"""
Merge sem Flag em Python
by Ed
"""
# merge
def merge(a,b):
    """ Função para juntar os dois arrays """
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c += b
    else:
        c += a
    return c
```



MergeSort em Python

```
# Código do mergesort
def mergesort(lista):
    """ Função para ordenar um vetor usando o mergesort """
    if len(lista) == 0 or len(lista) == 1:
        return lista
    else:
        meio = int(len(lista)/2)
        esquerda = mergesort(lista[:meio])
        direita = mergesort(lista[meio:])
        return merge(esquerda,direita)

while True:
    valor_ini = input("Digite os números do seu vetor, como 10,20,30, digite:")
    valores = list(eval "["+valor_ini+"]") #interpreta a sequencia como uma lista
    #valores = list(eval(valor_ini))"""
    #valores = [9,8,7,6,5,4,3,2,1]
    print(valores)
    print (mergesort(valores))
```



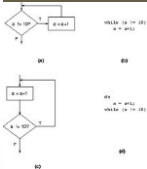
Animação dos algoritmos de ordenação

- Basta clicar no link:

<http://nicholasandre.com.br/sorting/>

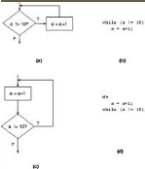
- MergeSort com dança folclórica da Transilvânia, Universidade Sapiientia (Romênia):

<https://www.youtube.com/watch?v=XaqR3GNVoo>



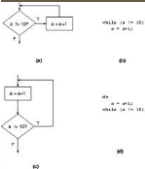
Pesquisando em matrizes (arrays)

- Frequentemente se trabalhará com grandes quantidades de dados armazenadas em matrizes
- Pode ser, então, necessário, determinar se uma matriz contém um valor que corresponde a um determinado *valor-chave*
- O processo de encontrar um determinado elemento de um array é chamado *pesquisa* (ou *busca*)
- Analisaremos aqui, com vetores, duas técnicas de pesquisa: a técnica simples de *pesquisa linear* e a técnica mais eficiente de *pesquisa binária*



Pesquisa Linear

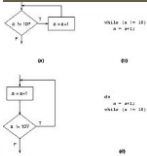
- A pesquisa linear compara cada elemento do vetor com a *chave de pesquisa* (ou *chave de busca*).
- Como o vetor não está em ordem específica, as **probabilidades** de o valor ser encontrado no primeiro ou no último elemento são as mesmas
- O método de **pesquisa linear** funciona **bem** em com **vetores pequenos** ou com vetores não-ordenados
- Para vetores (ou matrizes) grandes, a **pesquisa linear é ineficiente**. Se o vetor estiver ordenado, a técnica veloz de pesquisa binária pode ser utilizada.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Pesquisa usando busca linear (vetores)
4     Autor : Edkallenn - Data : 10/04/2012
5     Obs: Esta é a versão mais simples, mas com funcoes */
6  #define TAM 10 //tamanho maximo do vetor
7  int busca_linear(int [], int, int);
8  void preenche_vetor(int n, int []);
9  void exibe_vetor(int tamanho, int *);
10 main(){
11     int vetor[TAM], x, chave_busca, elemento; //declara o vetor
12     preenche_vetor(TAM, vetor); //preenche vetor - mesma funcao anterior
13     printf("\n\nO vetor digitado eh\n");
14     exibe_vetor(TAM, vetor); //exibe o vetor
15     //entra com a chave de pesquisa
16     printf("\n\nEntre com o inteiro para a chave de pesquisa: \n");
17     scanf("%d", &chave_busca);
18     elemento = busca_linear(vetor, chave_busca, TAM);
19     //exibe em qual índice está o elemento, se for encontrado.
20     if(elemento!=-1)
21         printf("\nValor encontrado no elemento %d\n", elemento);
22     else
23         printf("Valor nao encontrado ");
24     printf("\n\n");
25     getchar();
26 }
27 int busca_linear(int array[], int chave, int tamanho){
28     int n;
29     for(n=0;n<tamanho;n++)
30         if(array[n]==chave)
31             return n;
32     return -1;
33 }
34 }
35 void preenche_vetor(int tamanho, int vet[]){ // Preenche o vetor
36     int i;
37     for(i=0;i<tamanho;i++)
38         vet[i]=rand()%100;
39 }
40 void exibe_vetor(int tamanho, int v[]){ //Exibe
41     int i;
42     for(i=0;i<tamanho;i++)
43         printf("%d ", v[i]);
44     printf("\n");
45 }

```



Busca Linear em Python

```
# -*- encoding: utf-8 -*-
"""
Busca Sequencial em Python |
by Ed
"""

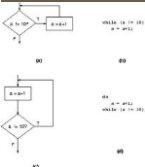
import random

def gera_lista():
    lista = random.sample(range(1,100), 20)
    return lista

def busca_linear(lista, elemento):
    for i in range(0,len(lista)):
        if elemento == lista[i]:
            return i

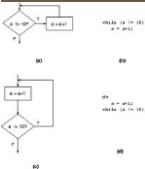
    return None

while True:
    lista = gera_lista()
    print(lista)
    valor_busca = int(input('Qual o valor que você quer buscar no vetor:'))
    encontrou=busca_linear(lista, valor_busca)
    if(encontrou!=None):
        print('Elemento encontrado na posição ', encontrou)
    else:
        print('Elemento não encontrado')
```



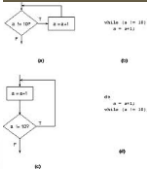
Busca recursiva

- A função pesquisa linear do slide anterior pode ser reescrita em estilo recursivo.
- A ideia do código é simples:
 - Se **tamanho=0** então o vetor é vazio e portanto **chave** não está em **array[0..tamanho-1]**;
 - se **tamanho > 0** então **chave** está em **array[0..tamanho-1]** se e somente se **chave = array[tamanho-1]** ou está **chave** no vetor **array[0..n-2]**



Busca recursiva

- Lembrando, claro que a versão **recursiva** da função busca **pode não ser** uma alternativa muito prática para a versão **iterativa**, pois a pilha de recursão consome memória adicional!



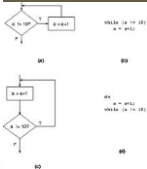
Busca Recursiva

```

27 int buscaRecursiva (int array[], int chave, int tamanho){
28     if (tamanho==0) return -1;
29     if(chave== array[tamanho-1]) return tamanho-1;
30     return buscaRecursiva(array, chave, tamanho-1);
31 }
32

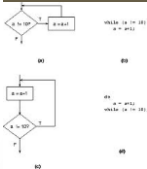
```

- Faça um novo programa (com base no anterior) e use a buscar recursiva!



Pesquisa binária

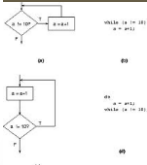
- O algoritmo de pesquisa binária elimina **metade** dos elementos do vetor que está sendo pesquisado **após cada comparação**
- O algoritmo localiza o elemento do **meio** do vetor e o **compara** com a **chave** de pesquisa
- **Se forem iguais**, a chave de pesquisa **foi encontrada** e o índice daquele elemento do vetor é retornado
- **Se não forem iguais**, o problema fica **reduzido** a pesquisar uma **metade** do vetor
- Se a **chave for menor** do que o elemento do meio do vetor, a **primeira metade** será pesquisada, caso contrário a segunda metade será pesquisada.
- Se a chave de busca não for encontrada no **subvetor**, o algoritmo é repetido na quarta parte do vetor original.
- A pesquisa **continua** até que a chave de busca seja igual ao elemento situado no meio do vetor
- Ou até que o subvetor consista de um elemento que não seja igual à chave de busca.

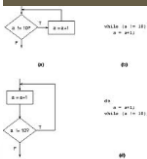


```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Pesquisa usando busca binaria (vetores)
4     Autor : Edkallenn - Data : 10/04/2012
5     Obs: Esta é a versão mais simples, mas com funcoes */
6  #define TAM 10 //tamanho maximo do vetor
7  int busca_binaria(int [], int, int, int);
8  void imprime_cabecalho(void);
9  void imprime_linha(int b[], int menor, int med, int maior);
10 void preenche_vetor(int n, int []);
11 void exibe_vetor(int tamanho, int *);
12 void ordena_bolha_asc_int(int tamanho, int *);
13 main(){
14     int vetor[TAM], chave_busca, resultado; //declara o vetor
15     preenche_vetor(TAM, vetor); //preenche vetor - mesma funcao anterior
16     printf("\n\nO vetor digitado eh\n");
17     exibe_vetor(TAM, vetor); //exibe o vetor
18     ordena_bolha_asc_int(TAM, vetor);
19     //entra com a chave de pesquisa
20     printf("\n\nEntre com o inteiro para a chave de pesquisa: \n");
21     scanf("%d", &chave_busca);
22     imprime_cabecalho();
23     resultado = busca_binaria(vetor, chave_busca, 0, TAM-1);
24     if(resultado!=-1)
25         printf("\n%d Encontrado no elemento do vetor: %d\n", chave_busca, resultado);
26     else
27         printf("%d nao encontrado ", chave_busca);
28     getchar();
29 }

```





```

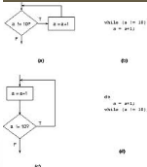
30 int busca_binaria(int b[], int chave_busca, int menor, int maior){
31     int meio;
32     while(menor<=maior){
33         meio=(menor+maior)/2;
34
35         imprime_linha(b, menor, meio, maior);
36
37         if(chave_busca==b[meio])
38             return meio;
39         else if(chave_busca<b[meio])
40             maior = meio-1;
41         else
42             menor = meio +1;
43     }
44     return -1;
45 }
46
47 void imprime_cabecalho(void){
48     int i;
49     printf("\nIndices\n");
50     for (i=0; i<TAM; i++)
51         printf("%3d ", i);
52     printf("\n");
53     for (i=0; i<4*TAM; i++)
54         printf("-");
55     printf("\n");
56 }
57 void imprime_linha(int b[], int menor, int med, int maior){

```

```

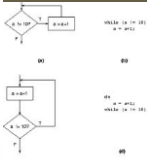
70 void ordena_bolha_asc_int(int tamanho, int a[]){
71     int pass, i, aux;
72     for (pass=1;pass<tamanho;pass++) //passadas
73         for(i=0;i<=tamanho-2;i++) //uma passada
74             if(a[i]>a[i+1]){ //uma comparacao
75                 aux=a[i]; //uma permuta
76                 a[i]=a[i+1];
77                 a[i+1]=aux;
78             }
79     }
80 void preenche_vetor(int tamanho, int vet[]){ // Preenche o vetor
81     int i;
82     for (i=0;i<tamanho;++i){
83         printf("\nDigite o elemento %d do vetor: ", i);
84         scanf("%d", &vet[i]);
85     }
86 }
87 void exibe_vetor(int tamanho, int v[]){ //Exibe
88     int t;
89     for (t=0;t<tamanho;t++)
90         printf("%-4d ", v[t]);
91 }
92

```



Pesquisa binária

- No **pior caso**, pesquisar um vetor de **1024** elementos precisará apenas de 10 comparações utilizando pesquisa binária.
- Dividir **repetidamente 1024 por 2**, resulta em 512, 256, 128, 64, 32, 16, 8, 4, 2 e 1. O número 1024 (2^{10}) é dividido por 2 10 vezes para que seja obtido o valor 1.
- **Dividir por 2** é igual ao a uma comparação no algoritmo de pesquisa binária.
- Um vetor com **1048576 (2^{20})** elementos precisa de **um máximo de 20** comparações para que a chave seja encontrada.
- Um vetor com **1 bilhão** precisa de **um máximo de 30 comparações**.
- Isso significa um **ganho aumento de performance**, comparado com a pesquisa linear que exigia, em média, a comparação da chave de pesquisa com metade dos elementos do vetor.
- Para um vetor de **1 bilhão de elementos**, isso significa uma diferença **entre 500 milhões de comparações e um máximo de 30 comparações!**



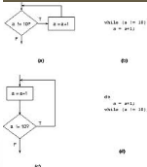
Busca binária em Python

```
# coding=utf-8
def exibe_lista(lista):
    print(lista)

def pesquisa_binaria(lista, item):
    primeiro = 0
    ultimo = len(lista) - 1
    while primeiro <= ultimo:
        meio = int((primeiro + ultimo) / 2)
        chute = lista[meio]
        if chute == item:
            return meio
        if chute > item:
            ultimo = meio - 1
        else:
            primeiro = meio + 1
    return None

import random
random.seed()
def gera_lista():
    lista = random.sample(range(1,100), 20)
    return lista
```

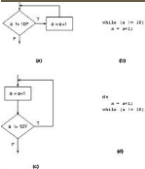
```
while True:
    outra_lista = gera_lista()
    print ('\nlista gerada pelo computador:')
    print ('Desordenada: ')
    exibe_lista(outra_lista)
    print ('')
    print ('ordenada: ')
    outra_lista.sort()
    exibe_lista(outra_lista)
    print ('Posicao: ', pesquisa_binaria(outra_lista, int(input("Digite um numero: "))))
```



Exercício (fazer em casa)-23/05

- Implemente as funções busca linear e busca binária inserindo no vetor valores aleatórios.
- Fazer versões recursivas das funções
- Apresentar um Menu para selecionar a versão de busca, binária ou linear
- Rodar os programas para TAM = 100, 200, 500, 1000 e 2000 com valores randômicos (gerados até num – reescrever a função preenche_vetor_random para receber o num, assim:

```
void preenche_vetor_random(int tamanho, int num, int vet[]);
```
- **Enviar para o email**
edkallenn.lima@uninorteac.edu.br
- **Com o título:**
- **[ED-TRAB-BUSCA] NomeESobrenome**

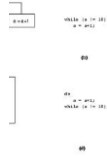


Parte 1 – Linhas 1 a 28

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Pesquisa usando busca binaria (vetores)
4   | Autor : Edkallenn - Data : 10/04/2012
5   | Obs: Esta é a versão mais simples, mas com funcoes */
6  #define TAM 20 //tamanho maximo do vetor
7  int busca_binaria(int [], int, int, int);
8  void imprime_cabecalho(void);
9  void imprime_linha(int b[], int menor, int med, int maior);
10 void preenche_vetor(int n, int []);
11 void preenche_vetor_random(int tamanho, int num, int vet[]);
12 void exhibe_vetor(int tamanho, int *);
13 void ordena_bolha_asc_int(int tamanho, int *);
14 int random(int n);
15 main() {
16     int vetor[TAM], chave_busca, resultado; //declara o vetor
17     srand((unsigned)time(NULL)); //inicializa o gerador de numeros aleatorios
18     preenche_vetor_random(TAM, 100, vetor); //preenche vetor alterada
19     printf("\n\nO vetor digitado eh\n");
20     exhibe_vetor(TAM, vetor); //exibe o vetor
21     ordena_bolha_asc_int(TAM, vetor);
22     printf("\n\nExibe o vetor ORDENADO\n\n");
23     exhibe_vetor(TAM, vetor); //exibe o vetor
24     //entra com a chave de pesquisa
25     printf("\n\nEntre com o inteiro para a chave de pesquisa: \n");
26     scanf("%d", &chave_busca);
27     imprime_cabecalho();
28     resultado = busca_binaria(vetor, chave_busca, 0, TAM-1);

```

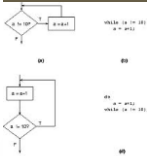


Parte 2 – linhas 29 a 61

```

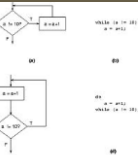
29     if(resultado!=-1)
30         printf("\n%d Encontrado no elemento do vetor: %d\n", chave_busca, resultado);
31     else
32         printf("%d nao encontrado ", chave_busca);
33     getchar();
34 }
35 int busca_binaria(int b[], int chave_busca, int menor, int maior){
36     int meio;
37     while(menor<=maior){
38         meio=(menor+maior)/2;
39
40         imprime_linha(b, menor, meio, maior);
41
42         if(chave_busca==b[meio])
43             return meio;
44         else if(chave_busca<b[meio])
45             maior = meio-1;
46         else
47             menor = meio +1;
48     }
49     return -1;
50 }
51
52 void imprime_cabecalho(void){
53     int i;
54     printf("\nIndices\n");
55     for (i=0; i<TAM; i++)
56         printf("%3d ", i);
57     printf("\n");
58     for (i=0; i<4*TAM; i++)
59         printf("-");
60     printf("\n");
61 }

```



```

62 void imprime_linha(int b[], int menor, int med, int maior){
63     int i;
64     for(i=0; i<TAM;i++)
65         if((i<menor)|| (i>maior))
66             printf("");
67         else if (i==med)
68             printf("%3d*", b[i]); //marca o valor do meio
69         else
70             printf("%3d ", b[i]);
71
72     printf("\n");
73
74 }
75 void ordena_bolha_asc_int(int tamanho, int a[]){
76     int pass, i, aux;
77     for (pass=1;pass<tamanho;pass++) //passadas
78         for(i=0;i<=tamanho-2;i++) //uma passada
79             if(a[i]>a[i+1]){ //uma comparacao
80                 aux=a[i]; //uma permuta
81                 a[i]=a[i+1];
82                 a[i+1]=aux;
83             }
84 }
85 void preenche_vetor(int tamanho, int vet[]){ // Preenche o vetor
86     int i;
87     for (i=0;i<tamanho;++i){
88         printf("\nDigite o elemento %d do vetor: ", i);
89         scanf("%d", &vet[i]);
90     }
91 }
  
```

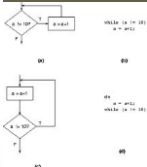


Última parte – linhas 92 a 107

```

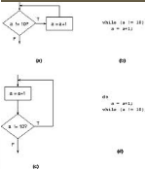
92 void preenche_vetor_random(int tamanho, int num, int vet[]){
93     int i, valor; // Preenche com valores randomicos melhor
94     for (i=0;i<tamanho;++i){
95         valor = (1 + random(num-1)); //gera ate num
96         vet[i]=valor;
97     }
98 }
99 void exhibe_vetor(int tamanho, int v[]){ //Exibe
100     int t;
101     for (t=0;t<tamanho;t++)
102         printf("%d ", v[t]); //2 espaços
103 }
104 int random(int n){ //funcao para gerar aleatorios
105     return rand() % n;
106 }
107

```



Strings (introdução)

- Os **caracteres** são representados internamente na memória do computador por códigos **numéricos**
- tipo **char** → tamanho de char = 1 byte = 8 bits = **256** valores distintos
- Os códigos associados aos caracteres estão dentro deste intervalo o char é usado para representar os caracteres.
- As **tabela** de códigos:
 - **definem** correspondência entre caracteres e códigos numéricos
 - exemplo: ASCII
 - alguns **alfabetos** precisam de maior representatividade
 - alfabeto chinês tem mais de 256 caracteres (alguns compiladores usam o tipo wchar neste caso)

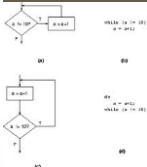


Códigos ASCII de alguns caracteres (sp representa espaço)

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

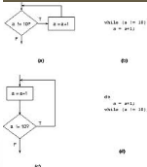
Exemplo:

82	105	110	32	100	101	32	74	97	110	101	105	114	111
R	i	o		d	e		J	a	n	e	i	r	o



Códigos ASCII de alguns caracteres de controle

0	nul	<i>null</i> : nulo
7	bel	<i>bell</i> : campainha
8	bs	<i>backspace</i> : volta e apaga um caractere
9	ht	<i>tab</i> : tabulação horizontal
10	nl	<i>newline</i> ou <i>line feed</i> : muda de linha
13	cr	<i>carriage return</i> : volta ao início da linha
127	del	<i>delete</i> : apaga um caractere

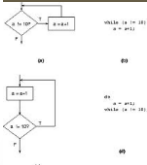


Imprimir a tabela ASCII

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /*  Função : Verifica Dígito
4      Autor : Edkallenn - Data : 07/05/2013
5      Obs:
6  */
7
8  main() {
9
10     int i;
11     printf("Tabela de Codigo ASCII\n\n");
12     for(i=0;i<=126;i++){
13         if (i%5==0) printf("\n");
14         printf("(%-3d): %c \t", i, i);
15     }
16     getchar();
17 }
18

```

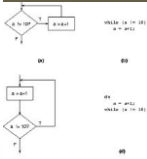


Caracteres

- Constante de caractere:
 - caractere envolvido com aspas simples
 - exemplo:
 - 'a' representa uma constante de caractere
 - 'a' resulta no valor numérico associado ao caractere **a**

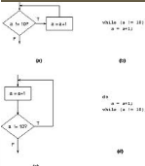
```
char c = 'a';
printf("%d %c\n", c, c);
```

- **printf** imprime o conteúdo da variável c usando dois formatos:
 - com o formato para inteiro, **%d**, imprime **97**
 - com o formato de caractere, **%c**, imprime **a** (*código 97 em ASCII*)



Verifica dígito – `int digito(char c)`

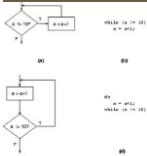
- Na tabela **ASCII** os dígitos são codificados em sequência
- Desse modo, se o dígito **zero** tem código **48**, o dígito **um** tem obrigatoriamente código **49**, e assim por diante.
- As letras minúsculas e maiúsculas também formam dois grupos sequenciais
- Mesmo desconhecendo os códigos associados aos caracteres, podemos tirar proveito desta codificação sequencial para escrever programas que usem a tabela.
- Como exemplo vamos fazer uma função que testa se um caractere **c** é um dígito (um dos caracteres entre **0** e **9**).
- Essa função retorna **1** (se verdadeiro) e **0** (falso) se **c** não for um dígito.



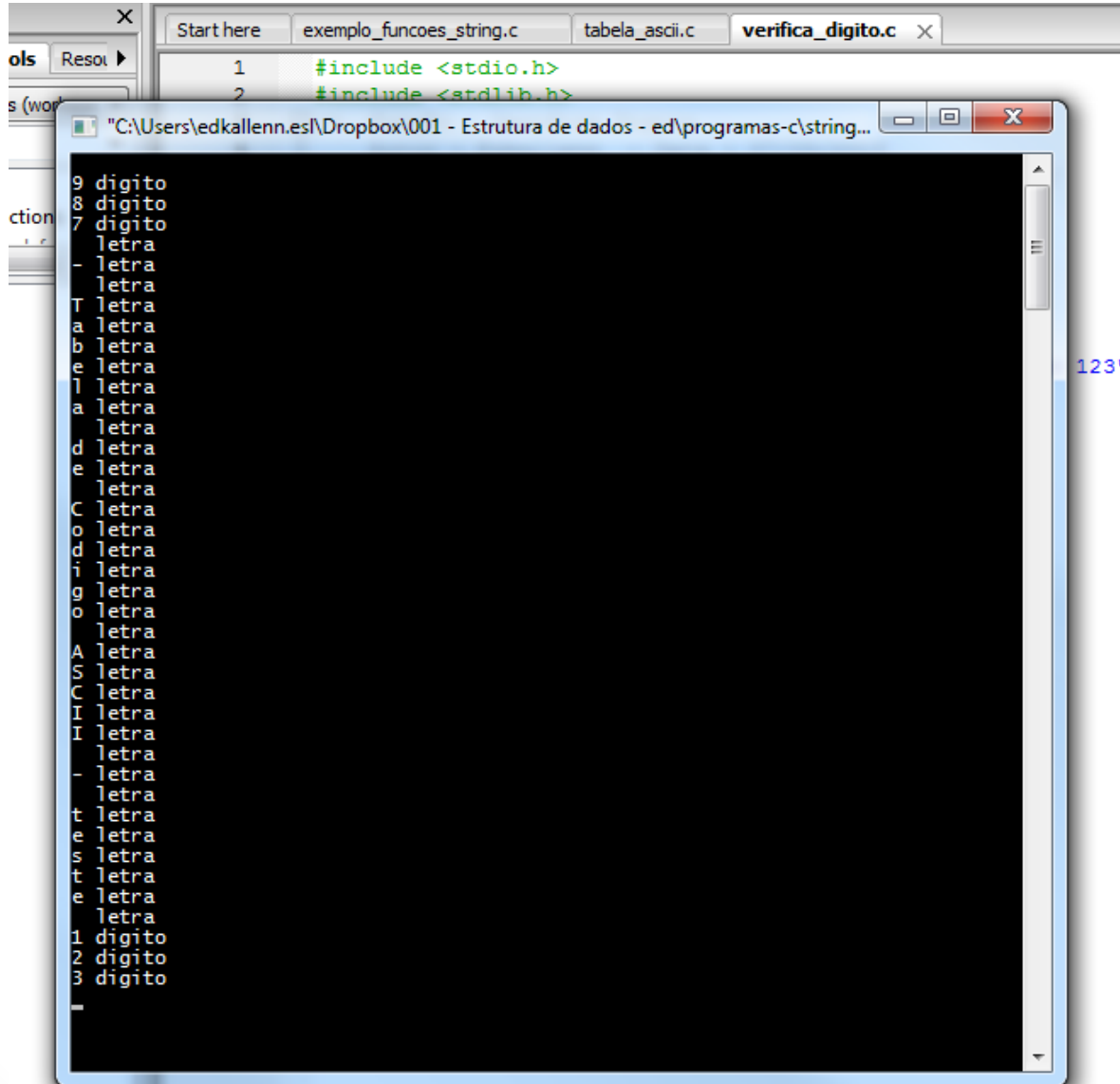
```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Verifica Dígito
4     Autor : Edkallenn - Data : 07/05/2013
5     Obs: Testa se um caractere c é um dígito ou não! */
6  #define QL printf("\n")
7
8  int digito(char c);
9
10 main() {
11     int i;
12     char texto[] = "987 - Tabela de Código ASCII - teste 123";
13     QL;
14     for(i=0; texto[i]!='\0'; i++){
15         if(digito(texto[i]))
16             printf("%c %s", texto[i], "digito\n");
17         else
18             printf("%c %s", texto[i], "letra\n");
19     }
20     getchar();
21 }
22
23 int digito(char c){
24     if((c>='0') && (c<='9'))
25         return 1; //eh digito
26     else
27         return 0; //não eh digito
28
29 }
30

```



Saída do programa anterior

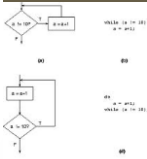


```

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

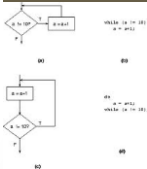
9 digito
8 digito
7 digito
  letra
- letra
  letra
T letra
a letra
b letra
e letra
l letra
a letra
  letra
d letra
e letra
  letra
C letra
o letra
d letra
i letra
g letra
o letra
  letra
A letra
S letra
C letra
I letra
I letra
  letra
- letra
  letra
t letra
e letra
s letra
t letra
e letra
  letra
1 digito
2 digito
3 digito
  
```

123"



Verifica Letra

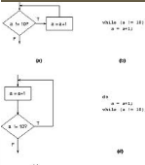
- Da mesma forma podemos **pensar** na implementação de uma função que **verifica** se um determinado caractere representa uma **letra**
- Basta **verificar** se seu código numérico representa uma letra **minúscula** ou maiúscula.
- A implementação está na próxima página



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Verifica Dígito
4   Autor : Edkallenn - Data : 07/05/2013
5   Obs: Testa se um caractere c é um letra ou não! */
6  #define QL printf("\n")
7
8  int letra(char);
9
10 main() {
11     int i;
12     char texto[] = "987 - Tabela de Codigo ASCII - teste 123";
13     QL;
14     for(i=0; texto[i] != '\0'; i++) {
15         if(letra(texto[i]))
16             printf("%c %s", texto[i], "EH LETRA\n");
17         else
18             printf("%c %s", texto[i], "nao eh letra\n");
19     }
20     getchar();
21 }
22
23 int letra(char c) {
24     if((c >= 'A' && c <= 'Z'))
25         return 1; //eh letra
26     else
27         return 0; //não eh letra
28
29 }
30

```



Saída do programa anterior

Algoritmo Verifica_Digito

* Função : Verifica Dígito

#de

nt

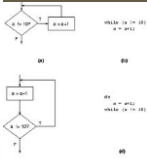
ain

nt

```

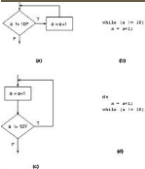
"C:\Users\edkallenn.es\Dropbox\001 - Estrutura de dados - ed\programas-c\string...
9 nao eh letra
8 nao eh letra
7 nao eh letra
nao eh letra
- nao eh letra
nao eh letra
T EH LETRA
a EH LETRA
b EH LETRA
e EH LETRA
l EH LETRA
a EH LETRA
nao eh letra
d EH LETRA
e EH LETRA
nao eh letra
C EH LETRA
o EH LETRA
d EH LETRA
i EH LETRA
g EH LETRA
o EH LETRA
nao eh letra
A EH LETRA
S EH LETRA
C EH LETRA
I EH LETRA
I EH LETRA
nao eh letra
- nao eh letra
nao eh letra
t EH LETRA
e EH LETRA
s EH LETRA
t EH LETRA
e EH LETRA
nao eh letra
1 nao eh letra
2 nao eh letra
3 nao eh letra

```



Converte maiúscula

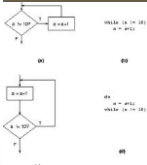
- Podemos também considerar uma função para converter um caractere para **maiúscula**
- Se o caractere dado representar uma letra **minúscula**, devemos ter como valor de retorno a letra **maiúscula** correspondente
- Se o **caractere** dado não for uma letra minúscula, devemos ter como valor de retorno o mesmo caractere, sem alterações.
- A implementação é mostrada a seguir:



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Verifica Dígito
4   Autor : Edkallenn - Data : 07/05/2013
5   Obs: Testa se um caractere c é um letra ou não! */
6  #define QL printf("\n")
7
8  char maiuscula(char c);
9
10 main() {
11     int i;
12     char texto[] = "Tabela de Codigo ASCII - by Edk";
13     QL;
14     for(i=0; texto[i]!='\0'; i++) {
15         printf("%c ", texto[i]);
16     } QL; QL;
17     for(i=0; texto[i]!='\0'; i++) {
18         printf("%c ", maiuscula(texto[i]));
19     }
20     getchar();
21 }
22
23 char maiuscula(char c) {
24     //verifica se é letra minúscula
25     if(c>='a' && c<='z')
26         c = c-'a'+'A'; //converte para maiúscula
27
28     return c;
29 }
30

```



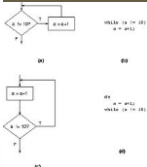
Saída do programa anterior

```
#include <stdio.h>

T a b e l a   d e   C o d i g o   A S C I I   -   b y   E d k
T A B E L A   D E   C O D I G O   A S C I I   -   B Y   E D K _

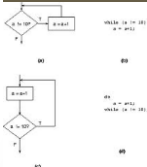
char maiuscula(char c){
    //verifica se é letra minúscula
    if(c>='a' && c<='z')
        c = c-'a'+'A'; //converte para maiuscula

    return c;
}
```



Strings (introdução)

- C não oferece um tipo de dado **string** (cadeia de caracteres)
- O uso mais comum de matrizes unidimensionais (vetores) é como **strings**
- Em C uma string é definida como um vetor de caracteres (**vetor** de **char**) que é terminada por um nulo.
- Um **nulo** é especificado como '**\0**' e geralmente é zero
- Deve-se, portanto, **declarar** o vetor de caracteres como sendo um caractere mais longo que a maior string que eles devem guardar

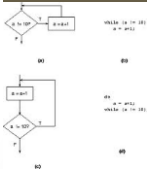


Strings (introdução)

- Por exemplo, para declarar uma matriz **str** que guarda uma string de 10 caracteres, você escreveria:

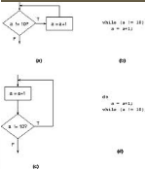
char str[11];

- Isso reserva espaço no final da string para o nulo
- Embora C não tenha o tipo de dado string, ela permite constantes string
- Uma constante string é uma lista de caracteres entre aspas duplas



Strings

- Exemplo de uma **constante string**:
“olá mundo”
- Não há a necessidade de se adicionar o nulo no final de constantes string manualmente – o compilador C faz isso por você automaticamente



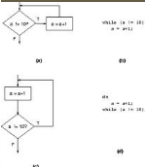
Strings

- Inicialização de cadeias de caracteres:
 - caracteres entre **aspas duplas**
 - caractere nulo é representado **implicitamente**
- Exemplo:
 - variável cidade dimensionada e inicializada com 4 elementos:

```
int main ( void )
{
    char cidade[ ] = "Rio";
    printf("%s \n", cidade);
    return 0;
}
```

≡

```
int main ( void )
{
    char cidade[ ] = {'R', 'i', 'o', '\0'};
    printf("%s \n", cidade);
    return 0;
}
```

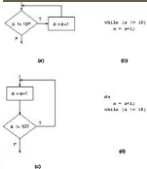


Strings

- Exemplos:

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

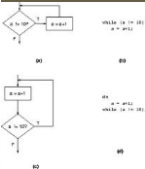
- s1 cadeia de caracteres vazia (armazena o caractere '\0')
- s2 armazena cadeia de 14 caracteres (em vetor com 15 elementos)
- s3 armazena cadeia com até 80 caracteres (dimensionada com 81 elementos, mas não inicializada)
- s4 armazena cadeias com até 80 caracteres primeiros quatro elementos atribuídos na declaração
{ 'R', 'i', 'o', '\0' };



Strings

- Leitura de **caracteres** e **strings**
- Pode se usar **scanf** com o especificador de formato **%c** (lê o valor de um único caractere fornecido via teclado)
- Exemplo:


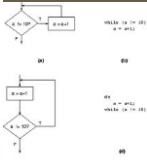
```
char a;
...
scanf("%c", &a);
...
```



Strings

- scanf com o especificador de formato %c (cont.):
 - não pula os “**caracteres brancos**”
 - “caractere branco” = **espaço (' '), tabulação ('\t') ou nova linha ('\n')**
 - se o usuário teclar um espaço antes da letra:
 - o código do espaço será capturado
 - a letra será capturada apenas na próxima chamada de scanf
 - para pular todos os “caracteres brancos” antes do caractere:
 - basta incluir um espaço em branco no formato, antes do especificador

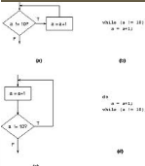
```
char a;
...
scanf(" %c", %a); /* o branco no formato pula brancos da entrada */
...
```

Strings

- **scanf** com o especificador de formato **%s**
- lê uma cadeia de caracteres não brancos
- pula os eventuais caracteres brancos antes da cadeia
- Exemplo:


```
char cidade[81];
...
scanf("%s", cidade);
...
```
- **&cidade** não é usada pois a cadeia é um vetor
- o código acima funciona apenas para capturar nomes simples
 - se o usuário digitar **Rio de Janeiro**, apenas **Rio** será capturada, pois **%s** lê somente uma sequência de caracteres não brancos

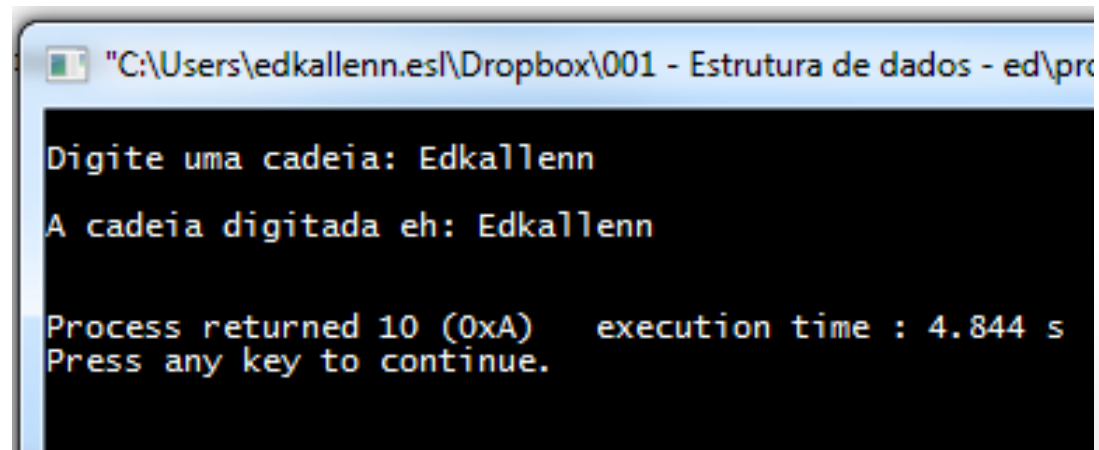
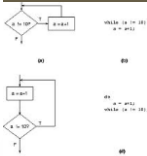


Exemplo

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Strings
4   | Autor : Edkallenn - Data : 07/05/2013
5   | Obs: testa strings */
6  #define QL printf("\n")
7
8  main() {
9
10     char texto[81];
11     QL;
12     printf("Digite uma cadeia: ");
13     scanf("%s", texto); QL;
14     printf("A cadeia digitada eh: %s", texto);
15     QL; QL;
16     getchar();
17 }

```

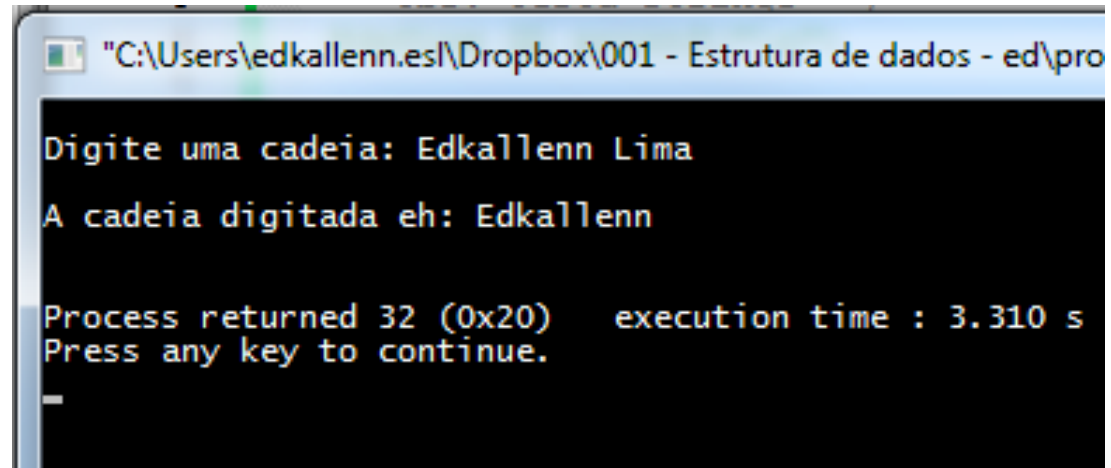
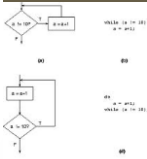



Problema!

```

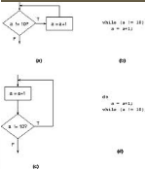
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Strings
4   | Autor : Edkallenn - Data : 07/05/2013
5   | Obs: testa strings */
6  #define QL printf("\n")
7
8  main() {
9
10     char texto[81];
11     QL;
12     printf("Digite uma cadeia: ");
13     scanf("%s", texto); QL;
14     printf("A cadeia digitada eh: %s", texto);
15     QL; QL;
16     getchar();
17 }

```

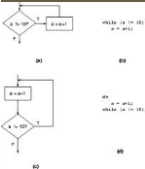
Strings

- **scanf** com o especificador de formato **%[...]**
- **%[...]** lista entre os colchetes todos os caracteres aceitos na leitura
- **%[^...]** lista entre os colchetes todos os caracteres não aceitos na leitura



Strings

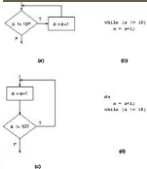
- Exemplos:
 - %[aeiou]
 - lê sequências de **vogais**
 - leitura prossegue até encontrar um caractere que **não** seja uma vogal
 - %[!aeiou]
 - lê sequências de caracteres que **não** são vogais
 - leitura prossegue até encontrar um caractere que **seja** uma vogal



Strings

- Exemplo:
 - lê uma sequência de **caracteres** até que seja encontrado o caractere de **mudança** de linha ('\n')
 - captura linha fornecida pelo usuário até que ele tecle “Enter”
 - inclusão do espaço no **formato** garante que eventuais caracteres brancos que precedam a cadeia de caracteres sejam **descartados**

```
char cidade[81];
...
scanf(" %[^\n]", cidade);
...
```



Strings

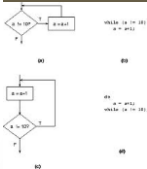
- Pode-se limitar a quantidade de caracteres que serão lidos, por precaução (para que os limites do vetor não sejam atingidos)
- Assim:

```
char cidade[81];
```

• • •

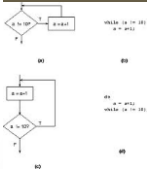
```
scanf ("%80[^\n]", cidade);
```

- Lê, no máximo, 80 caracteres.



Cadeias de caracteres (strings)

- Exemplos de funções para manipular cadeias de caracteres:
 - **imprime**
 - **comprimento**
 - **copia**
 - **concatena**
 - **compara**

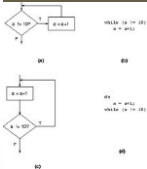


Strings (função imprime)

- Imprime uma cadeia de caracteres, caractere por caractere.

```
void imprime(char *s) {
    int i;
    for (i=0; s[i] != '\0'; i++)
        printf("%c", s[i]);
    printf("\n");
}
```

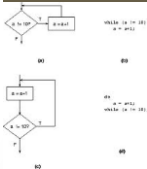
- Interessante notar como cada caractere da cadeia é acessado, até que o caractere **'\0'** seja encontrado



Strings (função exhibe)

- O código anterior teria uma funcionalidade análoga à utilização do especificador de formato **%s**.

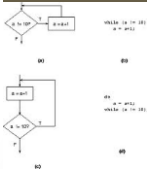
```
void exhibe(char* s) {
    printf("%s\n", s);
}
```



Função comprimento

- retorna o comprimento de uma cadeia de entrada **s**
- conta o número de caracteres até encontrar o caractere nulo (o caractere nulo em si não é contado)

```
int comprimento(char *s) {
    int i, n;
    n = 0; /* contador */
    for (i=0; s[i] != '\0'; i++)
        n++;
    return n;
}
```

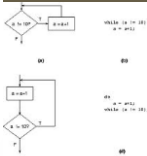


Teste da função comprimento

```
#include <stdio.h>

int comprimento (char* s)
{
    int i;
    int n = 0; /* contador */
    for (i=0; s[i] != '\0'; i++)
        n++;
    return n;
}

int main (void)
{
    int tam;
    char cidade[] = "Rio de Janeiro";
    tam = comprimento(cidade);
    printf("A string \"%s\" tem %d caracteres\n", cidade, tam);
    return 0;
}
```



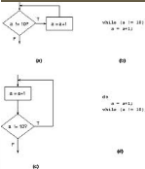
Função “copia”

- copia os elementos de uma cadeia de origem (orig) para uma cadeia de destino (dest)
- cadeia de destino deverá ter espaço suficiente

```

- void copia(char* dest, char* orig) {
    int i;
    for (i=0; orig[i] != '\0'; i++)
        dest[i] = orig[i];
    /* fecha a cadeia copiada */
    dest[i] = '\0';
}

```



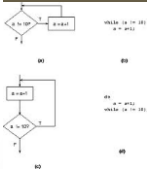
Função “concatena”

- copia os elementos de uma cadeia de origem (orig) para o final da cadeia de destino (dest)

```

void concatena (char* dest, char* orig){
    int i = 0; /* índice usado na cadeia dest, inicial c/ zero */
    int j; /* índice usado na cadeia origem */
    /* acha o final da cadeia destino */
    i = 0;
    while (dest[i] != '\0')
        i++;
    /* copia elementos da origem para o final do destino */
    for (j=0; orig[j] != '\0'; j++){
        dest[i] = orig[j]; i++;
    }
    /* fecha cadeia destino */
    dest[i] = '\0';
}

```



Função “compara”

```
int compara (char* s1, char* s2) {
    int i;
    /* compara caractere por caractere */
    for (i=0; s1[i]!='\0' && s2[i]!='\0'; i++) {
        if (s1[i] < s2[i])
            return -1;
        else if (s1[i] > s2[i])
            return 1;
    }
    /* compara se cadeias têm o mesmo comprimento */
    if (s1[i]==s2[i])
        return 0; /* cadeias iguais */
    else if (s2[i]!='\0')
        return -1; /* s1 é menor, pois tem menos caracteres */
    else
        return 1; /* s2 é menor, pois tem menos caracteres */
}
```



Programa para testar as funções

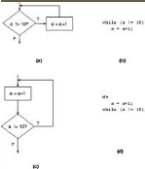
```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Strings
4     Autor : Edkallenn - Data : 07/05/2013
5     Obs: testa strings */
6  #define QL printf("\n")
7
8  int comprimento (char *s);
9  void imprime(char *);
10 void exhibe(char []);
11 void copia (char* dest, char* orig);
12 void concatena (char* dest, char* orig);
65 main() {
66
67     char texto[81], texto2[81];
68     printf ("Digite uma string: ");
69     scanf(" %80[^\n]s", texto); QL;
70     printf("O tamanho da cadeia: %s eh %d", texto, comprimento(texto));
71     QL;
72     imprime(texto);QL; // exhibe(texto);
73     copia(texto2, texto); QL;
74     imprime(texto2);QL; imprime(texto);QL;QL;
75     printf ("Digite uma string: ");
76     scanf(" %80[^\n]s", texto2); QL;
77     printf("O tamanho da cadeia: %s eh %d", texto2, comprimento(texto2));
78     concatena(texto, texto2);QL;
79     imprime(texto); QL;
80     printf("O tamanho da cadeia: %s eh %d", texto, comprimento(texto));QL;
81     getchar();
82 }

```

Strings - Função gets()

- A função **gets()**, com protótipo no arquivo stdio.h é mais conveniente para leitura de textos
- Seu propósito é **unicamente** ler uma cadeia de caracteres do teclado enquanto não for pressionada a tecla [ENTER]
- Todos os caracteres são armazenados na string e é **incluído** o caractere NULL ao final.
- Caracteres brancos, como espaços e tabulações são perfeitamente **aceitáveis** como parte da cadeia.

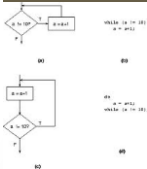


Strings – função gets()

```

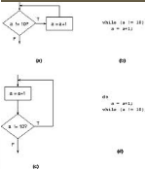
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  /*  Função : Mostra leitura de string com gets
5      Autor : Edkallenn - Data : 10/04/2012
6      Obs:
7      */
8
9  main() {
10     char nome[80];
11     printf("Digite o seu nome: ");
12     gets(nome);
13     printf("Saudacoes, %s \n", nome);
14     getchar();
15 }

```



Strings – Função puts()

- A função **puts()** é o complemento da função **gets()** e é bem fácil de usar
- O propósito de **puts()** é imprimir uma única string por vez
- O **endereço** da string deve ser enviado a **puts()** como argumento

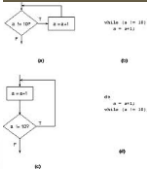


String – Função puts()

```

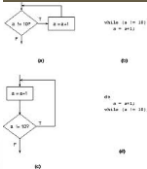
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  /* Função : Mostra leitrua de string com gets
5     Autor : Edkallenn - Data : 10/04/2012
6     Obs:
7  */
8
9  main() {
10     char nome[80];
11     printf("Digite o seu nome: ");
12     gets(nome);
13     puts("Saudacoes");
14     puts(nome);
15     puts("puts() pula de linha sozinha");
16     puts(&nome[4]);
17     getchar();
18 }
19

```



Função fgets()

- Uma alternativa mais eficiente para a leitura de **strings** é a função **fgets()**
- Na prática, **fgets()** não lê exatamente **strings**, mas usada de forma adequada também permite a leitura de **strings**

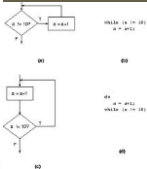


Função fgets()

- O protótipo desta função é:

```
char *fgets (char *str, int tamanho, FILE *fp)
```

- Ela recebe 3 parâmetros de entrada:
 - str: a string a ser lida
 - tamanho: o limite máximo de caracteres a serem lidos
 - fp: a variável que está associada ao arquivo de onde a string será lida.
- E retorna:
 - NULL: no caso de erro ou fim de arquivo
 - O ponteiro para o primeiro caractere da string recuperada em str



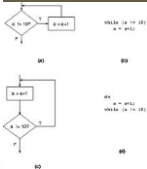
fgets()

- A função fgets() utiliza uma variável FILE *fp que está associada ao arquivo de onde a string será lida
- Para ler do teclado, basta substituir FILE *fp por stdin que representa o dispositivo de entrada-padrão (geralmente o teclado). Assim:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  /* Função : Mostra leitura de string com fgets
5   Autor : Edkallenn - Data : 10/04/2012
6   Obs:
7   */
8
9  main() {
10     char nome[30];
11     printf("Digite o seu nome: ");
12     fgets(nome, 30, stdin);
13     puts("Saudacoes");
14     puts(nome);
15
16     getchar();
17 }
18

```

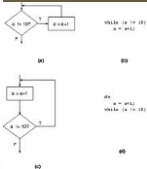


fgets() – considerações (limpando buffer)

- Na função `fgets()`, o caractere nova linha (“\n”) fará parte da string
- A função `fgets()` especifica o tamanho máximo da string de entrada
- Para limpar o buffer de entrada do teclado (entrada-padrão) podemos fazer:

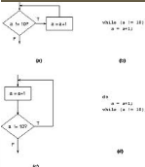
```
setbuf(stdin, NULL);
```

- Basicamente a função `setbuf()` preenche um buffer (primeiro parâmetro) com determinado valor (segundo parâmetro)



Cadeias de Caracteres (strings)

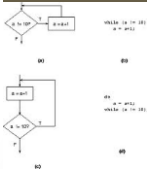
- Biblioteca de cadeias de caracteres **string.h**
 - “comprimento” **strlen**
 - “copia” **strcpy**
 - “concatena” **strcat**
 - “compara” **strcmp**



Strings

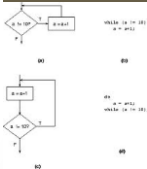
- C suporta uma ampla gama de funções de manipulação de strings.
- As mais comuns são:

Nome	Função
<code>strcpy(s1, s2)</code>	Copia s2 em s1
<code>strcat(s1, s2)</code>	Concatena s2 ao final de s1
<code>strlen(s1)</code>	Retorna o tamanho de s1
<code>strcmp(s1, s2)</code>	Retorna 0 se s1 e s2 são iguais; menor que 0 se $s1 < s2$; maior que 0 se $s1 > s2$.
<code>strchr(s1, ch)</code>	Retorna um ponteiro para a primeira ocorrência de ch em s1
<code>strstr(s1, s2)</code>	Retorna um ponteiro para a primeira ocorrência de s2 em s1



Strings

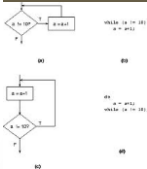
- Estas funções do slide anterior usam o cabeçalho padrão STRING.H
- Exemplo de uso das funções no próximo slide



```

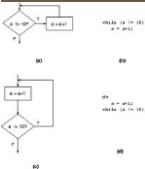
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  /*  Função : Teste de funcoes de manipulacao de string
5      Autor : Edkallenn - Data : 10/04/2012
6      Obs:
7  */
8
9  main() {
10     char s1[80], s2[80];
11     printf("Digite uma string: ");
12     gets(s1);
13     printf("Digite outra string: ");
14     gets(s2);
15
16     printf("Comprimentos: %d e %d\n", strlen(s1), strlen(s2));
17
18     if(!strcmp(s1, s2)) printf ("As strings sao iguais\n");
19     strcat(s1, s2);
20     printf("%s\n", s1);
21
22     strcpy(s1, "Isso eh um teste.\n");
23     printf(s1);
24     if(strchr("alo", 'o')) printf("o estah em alo\n");
25     if(strstr("ola aqui", "ola")) printf ("ola encontrado");
26
27     getchar();
28 }
29

```



Operações inválidas com strings

- A string **não** é um tipo **primitivo** da Linguagem C
- Ou seja, Existem operações que **não** são **possíveis** de serem realizadas nativamente
- Como exemplo, não é possível **atribuir** vários valores (cadeias de caracteres) ao mesmo tempo para os elementos de um vetor
- Isso só pode ser feito na **declaração** do vetor, informando o valor correspondente.



Operações inválidas com strings

- Exemplos:

```
char str1[10]; //Correto
```

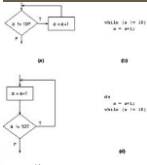
```
char str2[10]={\"Palavra 2\"}; //Correto. Declaração e atribuição de valores
```

```
str1 = str2; //ERRO! Não é possível copiar diretamente str2 para str1
```

```
if (str2 == str1) //ERRO! Não é possível comparar diretamente str1 com str2
```

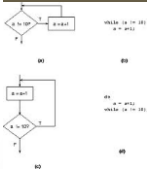
```
char linguagem[20]; //Correto
```

```
Linguagem = \"Linguagem C\"; //ERRO! Não é possível atribuir desta forma!
```



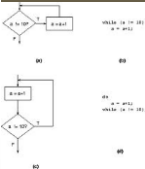
Operações

- Para realizar algumas das operações anteriores devemos recorrer à funções de manipulação de strings
- Estas funções usam o cabeçalho padrão `STRING.H`



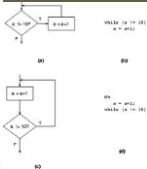
Exercício – fazer em sala

- Fazer um programa para ler **4 strings**
- Mostrar o **tamanho** de **todas** as strings
- **Concatenar** as **duas primeiras** em uma string e as **duas últimas** em outra string
- **Comparar** se alguma das **strings** é igual a outra (de duas a duas)
- **Concatenar** todas em uma **única string** e exibir o tamanho da mesma
- Usar **gets()**, **puts()**, **printf()**, **fgets()** e **scanf()**



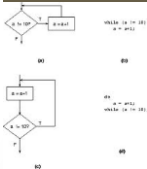
Strings – Aritmética com Endereços

- O nome de um array é um endereço, que corresponde ao endereço do primeiro elemento do array
- Ou seja: **string** é o endereço de **string[0]**
- Se **string** é do tipo char, **string** é, portanto, o endereço de uma variável do tipo char
- O que significa string + 1? O número 1 aqui corresponde a 1 byte (Já que char tem 1 byte)
- Se **string** fosse do tipo int, seu tamanho seria 4 bytes (ou o tamanho da palavra de sua máquina)



Strings – Aritmética com Endereços

- Ou seja, se somarmos 1 ao endereço de uma matriz de elementos do tipo int, estaremos obtendo o endereço da próxima variável int da memória, ou o próximo elemento da matriz.
- Em regra geral, se M é o nome de uma matriz e i é uma variável do tipo int, então:
- $M + i$
- É equivalente a $\&M[i]$
- $M + i \Leftrightarrow M[i]$
- Utilizando este conceito, pode se extrair uma substring de uma string somando ao seu endereço um número inteiro que indique o byte onde começará nossa substring

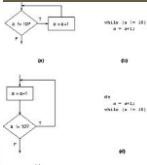




```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Mostra exemplo de aritmética de ponteiros com
4      strings
5      Autor : Edkallenn - Data : 10/04/2012
6      Obs:
7  */
8  #define TAM 80 //tamanho maximo do vetor
9
10 main() {
11     char saudacao[] = "Saudacoes, ";
12     char nome[TAM];
13     int i;
14     printf("Digite seu nome: ");
15     gets(nome);
16     puts(saudacao);
17     for (i = 0; i < strlen(nome); i++)
18         printf("%s\n", nome + i);
19
20     getchar();
21 }
22

```

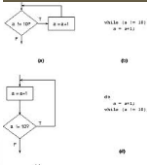


Qual a diferença?

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /*  Função : Mostra exemplo de aritmética de ponteiros com
4          strings
5          Autor : Edkallenn - Data : 10/04/2012
6          Obs:
7          */
8  #define TAM 80 //tamanho maximo do vetor
9
10  main() {
11      char saudacao[] = "Saudacoes, ";
12      char nome[TAM];
13      int i;
14      printf("Digite seu nome: ");
15      gets(nome);
16      puts(saudacao);
17      for (i = 0; i < (strlen(nome)); i++)
18          printf("%s\n", &nome[i]);
19
20      getchar();
21  }
22

```

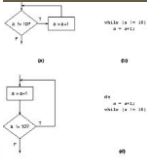


Como imprimir invertido? Tentem fazer!

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Função : Mostra exemplo de aritmética de ponteiros com
4      strings
5      Autor : Edkallenn - Data : 10/04/2012
6      Obs:
7  */
8  #define TAM 80 //tamanho maximo do vetor
9
10 main() {
11     char saudacao[] = "Saudacoes, ";
12     char nome[TAM];
13     int i, len;
14     printf("Digite seu nome: ");
15     gets(nome);
16     puts(saudacao);
17     for (i=strlen(nome); i>=0; i--)
18         printf("%s\n", nome+i);
19
20     getchar();
21 }
22

```

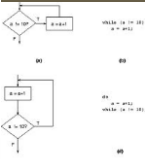


Imprime invertido recursiva

```

1  #include <stdio.h>
2  /** Função : exibe string Invertida recursivamente
3   * Autor : Edkallenn
4   * Data : 22/06/2014
5   * Obs: Usa a função putchar --> exibe um único caractere
6   */
7  void exibeInvertido (char *string);
8
9  main() {
10     char frase[80];
11     printf("\nDigite uma frase: ");
12     gets(frase);
13     printf("\nA frase invertida e':\n ");
14     exibeInvertido(frase);
15     getchar();
16 }
17
18 void exibeInvertido (char *string){
19     if (*string){
20         exibeInvertido (string +1);
21         putchar(*string);
22     }
23 }
24

```

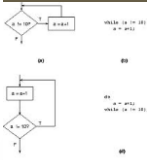


Outra forma de percorrer a string

```

1  #include <stdio.h>
2  /** Função : caminhar String
3   * Autor : Edkallenn
4   * Data : 22/06/2015
5   * Obs: Usa a função putchar --> exibe um único caractere
6   */
7  main() {
8      char frase[80];
9      int i;
10
11     printf("\nDigite uma frase: ");
12     gets(frase);
13     //outra forma de "caminhar" na string, usando NULL
14     for(i=0; frase[i] != NULL; i++){ //outra forma de "caminhar" na
15         putchar(frase[i]);          //string
16     }
17     printf("\n\nO numero de caracteres de %s = %d", frase,i);
18
19     getchar();
20 }
21

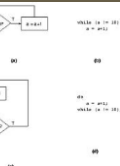
```



```

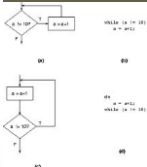
1  #include <stdio.h>
2  /** Função : conta o numero de ocorrencias de um caractere em string
3   * Autor : Edkallenn
4   * Data : 22/06/2015
5   * Obs:
6   */
7  int contachar (const char *string, char letra);
8  main() {
9      char frase[80], caractere;
10     int quantidade;
11
12     printf("\nDigite uma frase: ");
13     gets(frase);
14     printf("\nQual caractere procurar? ");
15     caractere = getchar();
16     quantidade = contachar(frase,caractere);
17
18     printf("\n\nO numero de ocorrencias de %c em \"%s\"", caractere,frase);
19     printf(" = %d\n\n", quantidade);
20     getchar();
21 }
22 int contachar (const char string[], char letra){
23     int i, quant = 0;
24     for(i=0;string[i]!='\0';i++)
25         if (string[i] == letra)
26             quant++;
27     return quant;
28 }
29

```



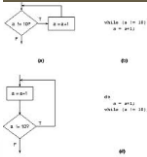
Alocação dinâmica

- Alocação dinâmica é o meio pelo qual um programa pode **obter memória** enquanto está em execução
- **Variáveis globais** têm o armazenamento alocado em **tempo de compilação**
- Variáveis **locais** usam a **pilha**
- Nem variáveis globais nem locais podem ser **acrescentadas** durante o **tempo de execução**
- Há momentos em que o programa precisará **utilizar quantidades de armazenamento variáveis**



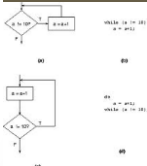
Alocação dinâmica

- Para usar vetores tivemos que, até agora, **dimensioná-lo**, o que nos obriga a saber, **de antemão**, quanto espaço (ou **qual o tamanho**) seria necessário
- Ou seja, tínhamos que **prever o número máximo de elementos** no vetor durante a codificação
- Esse pré-dimensionamento é um fator **limitante**
- Exemplo: para fazer um programa que calcule a **média e a variância** das notas de uma prova, teremos que **prever** o número máximo de alunos
- Ou dimensionar o vetor com um número **absurdamente alto**, para não termos limitações no momento da utilização do programa
- **Se o número for pequeno, não atende à turmas maiores**



Alocação dinâmica

- A linguagem C tem **meios de requisitar espaços de memória** em tempo de **execução**.
- Com esse recurso qualquer programa pode, em tempo de execução, consultar o número de alunos da turma e então **fazer a alocação do vetor** dinamicamente, sem desperdício de memória.
- O espaço **permanece reservado** até que seja **explicitamente liberado**
- Depois de liberado, **espaço estará disponibilizado para outros usos** e não pode mais ser acessado
- Será automaticamente liberado quando ao final da execução

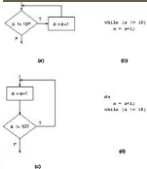


Alocação dinâmica

Uso da memória:

- **memória estática:**
 - código do programa
 - variáveis globais
 - variáveis estáticas
- **memória dinâmica:**
 - variáveis alocadas dinamicamente
 - **memória livre**
 - variáveis locais

memória estática	Código do programa
	Variáveis globais e Variáveis estáticas
memória dinâmica	Variáveis alocadas dinamicamente
	Memória livre
	Variáveis locais (Pilha de execução)

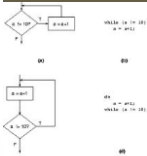


Alocação dinâmica

Uso da memória:

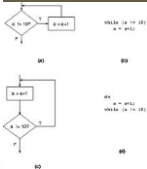
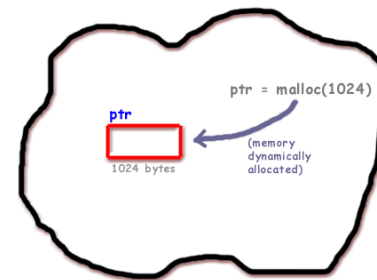
- alocação dinâmica de memória:
 - usa a memória livre
 - se o espaço de memória livre for menor que o espaço requisitado, a alocação não é feita e o programa pode prever tratamento de erro
- pilha de execução:
 - utilizada para alocar memória quando ocorre chamada de função:
 - sistema reserva o espaço para as variáveis locais da função
 - quando a função termina, espaço é liberado (desempilhado)
 - se a pilha tentar crescer mais do que o espaço disponível existente, programa é abortado com erro

memória estática	Código do programa
	Variáveis globais e Variáveis estáticas
memória dinâmica	Variáveis alocadas dinamicamente
	Memória livre
	Variáveis locais (Pilha de execução)



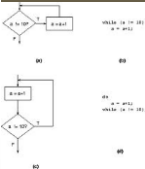
Alocação dinâmica

- A biblioteca **stdlib.h** contém uma série de funções pré-definidas:
- Entre elas funções para tratar **alocação dinâmica** de memória
- E constantes pré-definidas
- O operador **sizeof** retorna o número de bytes ocupado por um tipo



Função **malloc()**

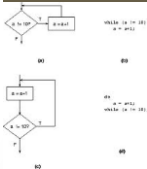
- Recebe como parâmetro o **número de bytes** que se deseja alocar
- **Retorna um ponteiro genérico** para o endereço inicial da área de memória alocada, **se houver espaço livre**:
 - **ponteiro genérico** é representado por **void***
 - ponteiro é **convertido automaticamente** para o tipo apropriado
 - ponteiro **pode ser convertido explicitamente** (casting)
- Retorna **um endereço nulo**, **se não houver espaço livre**:
 - representado pelo símbolo **NULL**



Alocação dinâmica

- Exemplo:
- **Alocação dinâmica** de um vetor de **inteiros** com **10 elementos**
- **malloc** retorna o **endereço da área alocada** para armazenar valores inteiros
- ponteiro de inteiro recebe endereço inicial do espaço alocado

```
int *v;
v = (int *) malloc(10*sizeof(int));
```



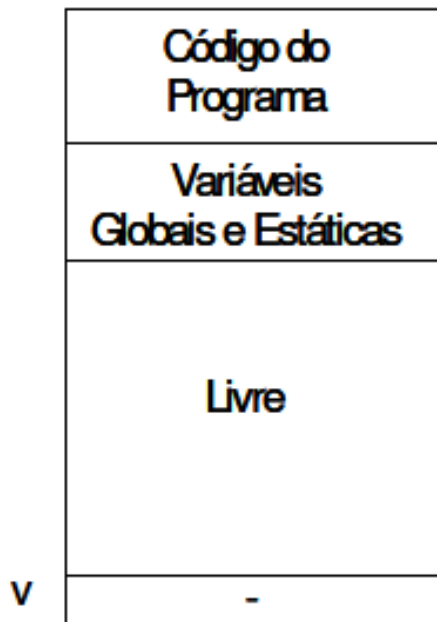
Alocação dinâmica

- Exemplo:

```
v = (int *) malloc(10*sizeof(int));
```

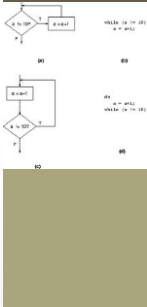
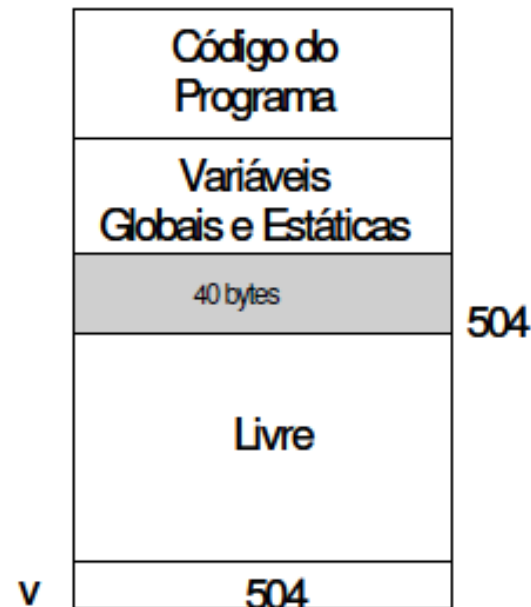
1 - Declaração: `int *v`

Abre-se espaço na pilha para o ponteiro (variável local)



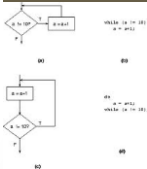
2 - Comando: `v = (int *) malloc (10*sizeof(int))`

Reserva espaço de memória da área livre e atribui endereço à variável



Alocação dinâmica

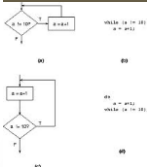
- Exemplo:
- **v** armazena **endereço inicial** de uma área contínua de memória suficiente para armazenar 10 valores **inteiros**
- **v** pode ser tratado como um vetor declarado estaticamente
- **v** aponta para o início da área alocada
- **v[0]** acessa o espaço para o primeiro elemento
- **v[1]** acessa o segundo
- até **v[9]**
- Ou usando **aritmética de ponteiros**
- **v**
- **v + 1**
- **v + 2** etc...



Alocação Dinâmica

- **Tratamento de erro** (exceção) após chamada a **malloc**
 - Imprime mensagem de erro
 - Aborta o programa (com a função **exit**)

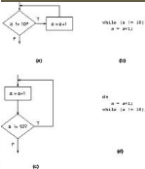
```
...
v = (int*) malloc(10*sizeof(int));
if (v==NULL)
{
    printf("Memoria insuficiente.\n");
    exit(1); /* aborta o programa e retorna 1 para o sist. operacional */
}
...
```



Função free()

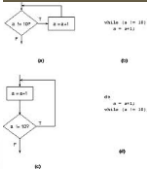
- Recebe como parâmetro o ponteiro da memória a ser liberada
- A função **free** deve receber um **endereço de memória** que tenha sido alocado dinamicamente

```
free (v);
```



Exemplo

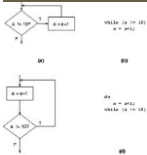
- Calcular a média e a variância de N números reais (usando funções anteriores) com alocação dinâmica.



```

1  /* Cálculo da média e da variância de n reais */
2  #include <stdio.h>
3  #include <stdlib.h>
4  float media(int n, float *v);
5  float variancia(int n, float *v, float med);
6
7  int main ( void )
8  {
9      int i, n;
10     float *v;
11     float med, var;
12     /* leitura do número de valores */
13     printf("Digite a quantidade de alunos:");
14     scanf("%d", &n);
15     /* alocação dinâmica */
16     v = (float*) malloc(n*sizeof(float));
17     if (v==NULL) {
18         printf("Memoria insuficiente.\n");
19         return 1;
20     }
21     /* leitura dos valores */
22     for (i = 0; i < n; i++){
23         printf("Digite a media do %do. aluno: ", i+1);
24         scanf("%f", &v[i]);
25     }
26     med = media(n,v);
27     var = variancia(n,v,med);
28     printf("Media = %f Variancia = %f \n", med, var);
29     /* libera memória */
30     free(v);
31     return 0;
32 }

```

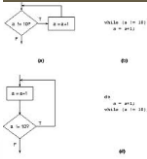


Continuação do programa:

```

33  float media(int n, float *v){    // calcula a media de um vetor
34      int i;
35      float soma=0.0f;             //de n elementos
36      for (i=0;i<n;i++)
37          soma+=v[i];
38      return soma/n;
39  }
40  float variancia(int n, float *v, float med){ //calcula a variancia
41      int i;                       //dada a media
42      float s = 0.0f;              //que eh passada como parametro
43      for (i = 0;i<n;i++)          //alem do tamanho n do vetor
44          s+=((v[i]-med)*(v[i]-med));
45      return s/n;
46  }
47

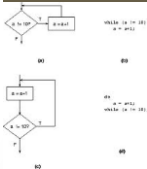
```



Função calloc

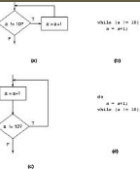
- A função **calloc** requer dois parâmetros: o número de posições de memória e o tamanho em bytes de cada posição.
- **calloc** também retorna um ponteiro do tipo **void** para o início do espaço alocado.
- Este ponteiro deve ser convertido para o tipo de dado desejado (explicitamente)
- Para fazer com que **vetor** aponte para um espaço capaz de acomodar 10 elementos do tipo **float**, pode se escrever:

```
vetor = (float *)calloc(10, sizeof(float))
```



Exercício

- Alterar o programa anterior para usar **calloc** em vez de **malloc**

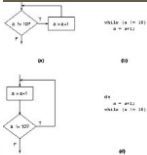


- O mesmo programa usando funções
- Funções que retornam ponteiros
- A função aloca deste programa retorna um ponteiro

```

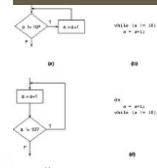
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* Funcao: Cálculo da média e da variância de n reais
4     Autor: Edkallenn - Data: 20/05/2012
5     Obs: Usando funcoes para TUDO */
6  int le_quant_int();
7  float *aloca(int n, float *);    //funcao para ALOCAR n floats
8  void preenche_vetor(int n, float []);
9  void exhibe_vetor(int tamanho, float v[]);
10 float media(int n, float *v);
11 float variancia(int n, float *v, float med);
12
13 int main(){
14     int i, n;
15     float *vetor, *p_vetor; //declara o vetor (como ponteiro)
16     float med, var;         //delcara as variaveis
17     n = le_quant_int(); // leitura do tamanho do vetor
18     if (p_vetor = aloca(n, vetor)){
19         preenche_vetor(n, p_vetor);
20         exhibe_vetor(n, p_vetor);
21         med = media(n, p_vetor);
22         var = variancia(n, p_vetor, med);
23         printf("\n\nMedia = %g Variancia = %g \n", med, var);
24         free(p_vetor);
25     }else
26         exit(1);    //sai do programa
27     getchar();
28     return 0;
29 }
30 float *aloca(int n, float *vetor){
31     //funcao que aloca dinamicamente n floats
32     vetor = (float*) calloc(n, sizeof(float));
33     if (vetor==NULL){
34         printf("Memoria insuficiente.\n");
35         return 0;
36     }
37     return vetor;
38 }

```



Continuação do programa

```
39 int le_quant_int(){
40     int valor;
41     printf("Digite a quantidade de alunos:");
42     scanf("%d", &valor);
43     return valor;
44 }
45 void preenche_vetor(int n, float vet[]){ // Preenche o vetor
46     int i;
47     for (i=0;i<n;i++){
48         do{
49             printf("\nDigite a media do %do. aluno: ", i+1);
50             scanf("%g", &vet[i]);
51             }while(vet[i]>10);
52         }
53     }
54 void exibe_vetor(int tamanho, float v[]){ //Exibe o vetor do tipo float
55     int t;
56     printf("\nO vetor digitado eh:\n");
57     for (t=0;t<tamanho;t++)
58         printf("%-4.2f ", v[t]);
59     }
60 float media(int n, float *v){ // calcula a media de um vetor
61     int i; //de n elementos
62     float soma=0.0f;
63     for (i=0;i<n;i++)
64         soma+=v[i];
65     return soma/n;
66 }
67 float variancia(int n, float *v, float med){ //calcula a variancia
68     int i; //dada a media
69     float s = 0.0f; //que eh passada como parametro
70     for (i = 0;i<n;i++) //alem do tamanho n do vetor
71         s+=((v[i]-med)*(v[i]-med));
72     return s/n;
73 }
74 }
```

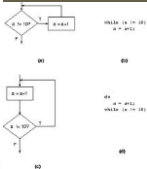


Exercícios (fazer em casa)

- Ler as **N** notas obtidas pelos alunos de uma classe e determinar: **a maior nota** obtida, **a menor** nota obtida, a **média** das notas, a **variância** e número de alunos com nota **abaixo e acima** da média usando alocação dinâmica
- Fazer uma função para preencher o vetor com valores randômicos.
- Apresentar um menu ao usuário dando as opções de digitação das **N** notas ou da geração randômica das notas
- Usar todas as funções anteriores (para alocação, pode ser calloc ou malloc ou criar uma própria)
- Enviar até o dia 04/06/2017 (Domingo) para o e-mail: edkallenn.lima@uninorteac.edu.br ou edkevan@gmail.com com o título:

[TRAB-ED-ALOCAÇÃO-N2]NomesSobrenome

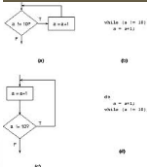
- Enviar **SOMENTE** o arquivo fonte



Alocação estática x dinâmica

- Alocação **estática** de vetor:
- é necessário **saber de antemão** a dimensão máxima do vetor
- variável que representa o vetor armazena o endereço ocupado pelo primeiro elemento do vetor
- vetor declarado dentro do corpo de uma função não pode ser usado fora do corpo da função (**escopo local**)

```
#define N 10
int v[N];
```



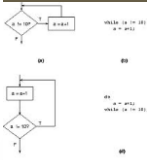
Alocação Estática x Dinâmica

- Alocação **dinâmica** de vetor:
 - dimensão do vetor pode ser definida em tempo de execução
 - variável do tipo ponteiro recebe o valor do endereço do primeiro elemento do vetor
 - área de memória ocupada pelo vetor **permanece válida** até que seja explicitamente **liberada** (através da função free)
 - vetor alocado dentro do corpo de uma função pode ser usado fora do corpo da função, enquanto estiver alocado

```
int* v;

...

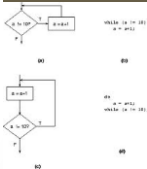
v = (int*) malloc(n * sizeof(int));
```



Alocação Estática x Dinâmica

- Função “**realloc**”:
- permite **realocar** um vetor preservando o conteúdo dos elementos, que permanecem válidos após a realocação
- Exemplo:
 - m representa a nova dimensão do vetor

```
v = (int*) realloc(v, m*sizeof(int));
```



- VER A LISTA DE EXERCÍCIOS QUE ESTARÁ DISPONÍVEL NO BLOG E NO DROPBOX.
- VERIFIQUE TAMBÉM A LISTA DE EXERCÍCIOS NO URI

