DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

INF 1010 – Estrutura de Dados Avançadas

Aula 02 – Revisão da Linguagem C

2020.1

Prof. Augusto Baffa
<abaffa@inf.puc-rio.br>

LES ECOA PUCRIO

ICAD GAMES visionlab visualization laboratory

Revisão Linguagem C

VETORES DE PONTEIROS

Vetor de Cadeia de Caracteres

- **Alocação Estática:**

```
char str[3][15] = {"Bom Dia.", "Boa Tarde.", "Boa Noite"};
```

  - A declaração é equivalente a matriz:

$$\begin{bmatrix} B & o & m & & D & i & a & . & \backslash 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ B & o & a & & T & a & r & d & e & . & \backslash 0 & \cdots & \cdots & \cdots & \cdots \\ B & o & a & & N & o & i & t & e & \backslash 0 & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

  - É possível alterar um caractere ou imprimir todo um elemento:

```
str[1][4] = 'X';
printf("%s\n", str[1]);
```

  - Quando passamos o vetor de caracteres como parâmetro para uma função, o protótipo da função deve ser:

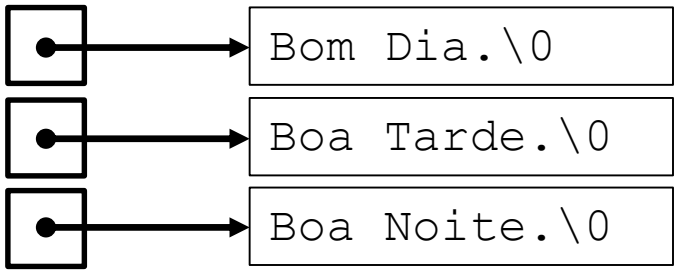
```
void teste(char str[][15], int n);
```

Vetores de Ponteiros

- **Alocação Dinâmica:**

```
char *str[] = {"Bom Dia.", "Boa Tarde.", "Boa Noite"};
```

  - Dessa forma temos um vetor de ponteiros:



A vantagem é que cada elemento aponta para strings de tamanhos diferentes.

  - É possível imprimir um elemento:

```
printf("%s\n", str[1]);
```

  - Mas nesse caso não podemos alterar os caracteres dos elementos, pois utilizamos constates de strings.

Vetores de Ponteiros

- **Alocação Dinâmica:**
  - Para podermos alterar os caracteres dos elementos, é necessário alocar as cadeias de caracteres dinamicamente:

```
char *str[3];
char *novo_elemento = "Bom Dia.";
str[0] = (char*)malloc((strlen(novo_elemento)+1) *
sizeof(char));
strcpy(str[0], novo_elemento);

...
str[0][4] = 'X';
printf("%s", str[0]);
```

  - Quando passamos o vetor de ponteiros como parâmetro para uma função, o protótipo da função deve ser:

```
void teste(char **str, int n);
```

Vetores de Ponteiros

- **Alocação Dinâmica:**
  - Também é possível alocar toda a matriz como ponteiros de ponteiro:

```
char *novo_elemento;
char **str;
str = (char**)malloc(3 * sizeof(char*));


novo_elemento = "Bom Dia.";
str[0] = (char*)malloc((strlen(novo_elemento)+1) * sizeof(char));
strcpy(str[0], novo_elemento);


novo_elemento = "Boa Tarde.";
str[1] = (char*)malloc((strlen(novo_elemento)+1) * sizeof(char));
strcpy(str[1], novo_elemento);
```

  - O protótipo de funções que recebem a estrutura deve ser:

```
void teste(char **str, int n);
```

Vetores de Ponteiros para Estruturas

- **Exemplo:** crie um programa que armazene os dados de alunos em um vetor, permitindo a preenchimento, remoção e impressão dos dados.

- **Dados de cada aluno:**
  - **matrícula:** número inteiro;
  - **nome:** cadeia com até 80 caracteres;
  - **endereço:** cadeia com até 120 caracteres;
  - **telefone:** cadeia com até 20 caracteres.

Vetores de Ponteiros para Estruturas

- **Solução 1:**

```
#define MAX 100


struct aluno
{
   int mat;
   char nome[81];
   char end[121];
   char tel[21];
};

typedef struct aluno Aluno;


Aluno tab[MAX];
```

a estrutura ocupando pelo menos 4 + 81 + 121 + 21 = 227 bytes

vetor de Aluno: representa um desperdício significativo de memória, se o número de alunos for bem inferior ao máximo estimado

Vetores de Ponteiros para Estruturas

- **Solução 2:**

```
#define MAX 100


struct aluno
{
   int mat;
   char nome[81];
   char end[121];
   char tel[21];
};

typedef struct aluno Aluno;


Aluno *tab[MAX];
```

Vetor de ponteiros para Aluno:

- um elemento do vetor ocupa espaço de um ponteiro;
- alocação dos dados de um aluno no vetor:
  - nova cópia da estrutura Aluno é alocada dinamicamente
  - endereço da cópia é armazenada no vetor de ponteiros
- posição vazia do vetor: valor é o ponteiro nulo

Vetores de Ponteiros para Estruturas

- Função `inicializa` para inicializar a tabela:
  - recebe um vetor de ponteiros (parâmetro deve ser do tipo "**ponteiro para ponteiro**");
  - atribui NULL a todos os elementos da tabela;

```
void inicializa(int n, Aluno** tab)
{
   int i;
   for (i=0; i<n; i++)
      tab[i] = NULL;
}
```

Vetores de Ponteiros para Estruturas

- Função `preenche` para armazenar um novo aluno na tabela:
  - recebe a posição onde os dados serão armazenados
  - se a posição da tabela estiver vazia, função aloca nova estrutura
  - caso contrário, função atualiza a estrutura já apontada pelo ponteiro

```
void preenche(int n, Aluno** tab, int i)
{
   if (i < 0 || i >= n)
   {
      printf("Indice fora do limite do vetor\n");
      exit(1); /* aborta o programa */
   }
   if (tab[i] == NULL)
      tab[i] = (Aluno*)malloc(sizeof(Aluno));
   printf("Entre com a matricula:");
   scanf("%d", &tab[i]->mat);
   ...
}
```

Vetores de Ponteiros para Estruturas

- Função `retira` para remover os dados de um aluno:
  - recebe a posição da tabela a ser liberada;
  - libera espaço de memória utilizado para os dados do aluno;

```
void retira(int n, Aluno** tab, int i)
{
   if (i<0 || i>=n)
   {
      printf("Indice fora do limite do vetor\n");
      exit(1); /* aborta o programa */
   }
   if (tab[i] != NULL)
   {
      free(tab[i]);
      tab[i] = NULL; /* indica que na posição não mais existe dado */
   }
}
```

Vetores de Ponteiros para Estruturas

- Função `imprime` para imprimir os dados de um aluno:
  - recebe a posição da tabela a ser impressa;

```
void imprime(int n, Aluno** tab, int i)
{
   if (i<0 || i>=n)
   {
      printf("Indice fora do limite do vetor\n");
      exit(1); /* aborta o programa */
   }
   if (tab[i] != NULL)
   {
      printf("Matrícula: %d\n”, tab[i]->mat);
      printf("Nome: %s\n”, tab[i]->nome);
      printf ("Endereço: %s\n”, tab[i]->end);
      printf("Telefone: %s\n”, tab[i]->tel);
   }
}
```

Vetores de Ponteiros para Estruturas

- Função `imprime_tudo` para imprimir todos os dados da tabela:
  - recebe o tamanho da tabela e a própria tabela;

```
void imprime_tudo(int n, Aluno** tab)
{
   int i;
   for (i=0; i<n; i++)
   {
      imprime(n, tab, i);
   }
}
```

Vetores de Ponteiros para Estruturas

- Função principal:

```
int main (void)
{
   Aluno* tab[10];
   inicializa(10,tab);

   preenche(10,tab,0);
   preenche(10,tab,1);
   preenche(10,tab,2);

   imprime_tudo(10,tab);

   retira(10,tab,0);
   retira(10,tab,1);
   retira(10,tab,2);

   return 0;
}
```

Vetores de Ponteiros para Estruturas

- Função principal:

```
int main (void)
{
   Aluno** tab = (Aluno**)malloc(size(Aluno*) * 10);
   inicializa(10,tab);

   preenche(10,tab,0);
   preenche(10,tab,1);
   preenche(10,tab,2);

   imprime_tudo(10,tab);

   retira(10,tab,0);
   retira(10,tab,1);
   retira(10,tab,2);

   return 0;
}
```

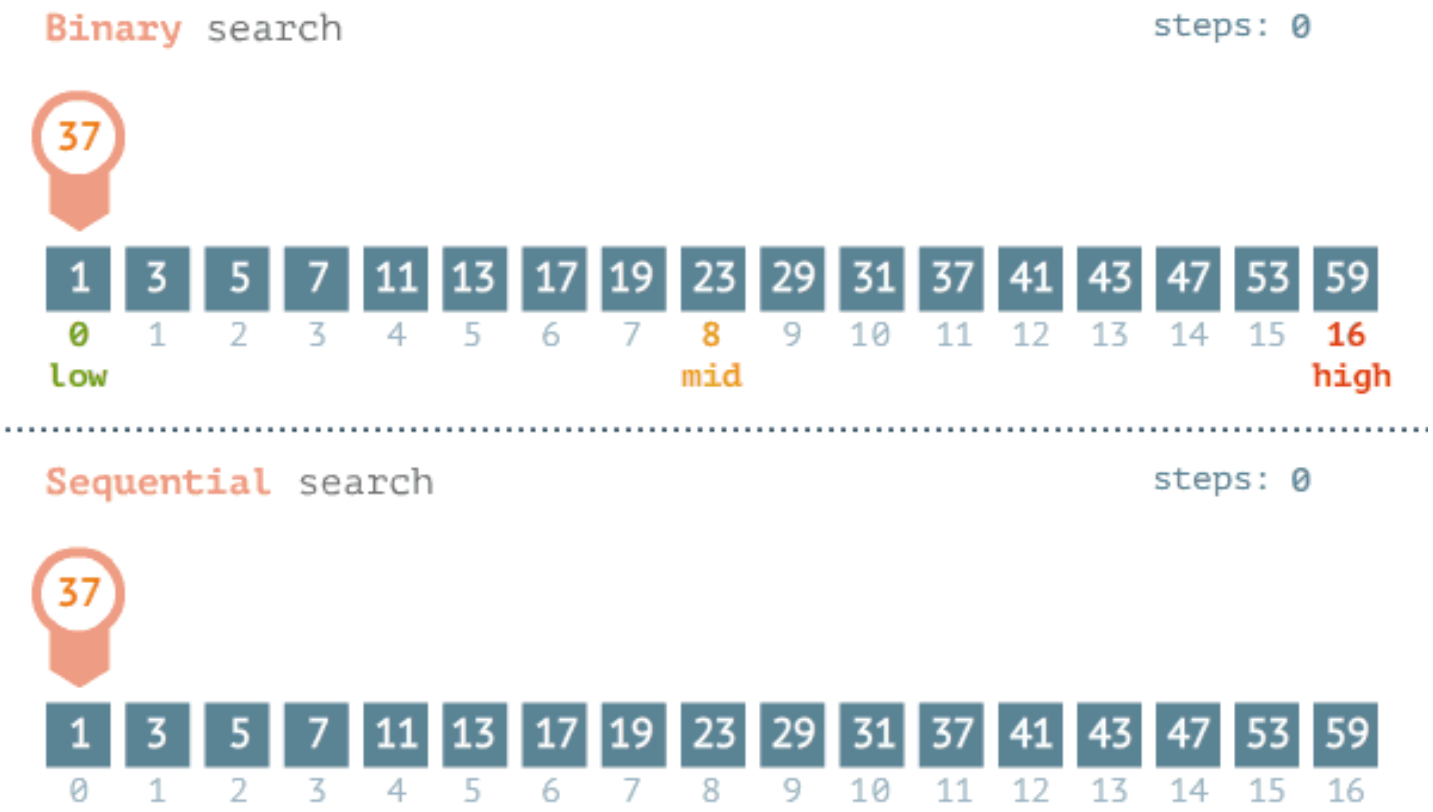
Revisão Linguagem C

BUSCA EM VETORES

Busca em Vetor

- **Problema:**
  - **Entrada:**
    - vetor **v** com **n** elementos;
    - element **d** a procurar;
  - **Saída:**
    - **m** se o elemento procurado está em **v[m]**;
    - **-1** se o elemento procurando não está no vetor;
- **Tipos de Busca em Vetor:**
  - Linear (ou sequêncial);
  - Binária;

Busca em Vetor



Busca Linear em Vetor

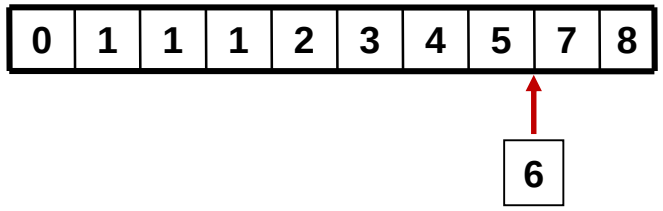
- **Algoritmo:** Percorra o vetor `vet`, elemento a elemento, verificando se `elem` é igual a um dos elementos de `vet`:

```
int busca (int n, int* vet, int elem)
{
   int i;

   for (i=0; i<n; i++) {
      if (elem == vet[i])
         return i; /* encontrou */
   }

   /* não encontrou */
   return -1;
}
```

Busca Linear em Vetor Ordenado

- **E se o vetor estiver ordenado? Como ficaria o algoritmo?**

```
int busca_ord (int n, int* vet, int elem)
{
   int i;
   for (i=0; i<n; i++) {
      if (elem == vet[i])
         return i;  /* encontrou */
      else if (elem > vet[i])
         return -1;/* interrompe busca */
   }
  return -1; /* não encontrou */
}
```



Busca Binária em Vetor Ordenado

- **Entrada:**
  - vetor **v** com **n** elementos ordenados;
  - elemento **d** a procurar;
- **Saída:**
  - **m** se o elemento elem ocorre em **v[m]**;
  - **-1** se o elemento não se encontra no vetor;

- **Procedimento:**
  - Compare o elemento **d** com o elemento do meio de **v**;
  - Se o elemento **d** for menor, pesquise na primeira metade do vetor;
  - Se o elemento **d** for maior, pesquise na segunda parte do vetor;
  - Se o elemento **d** for igual, retorne a posição;
  - Continue o procedimento, subdividindo a parte de interesse até encontrar o elemento **d** ou chegar ao fim;

Busca Binária em Vetor Ordenado

```
int busca_bin (int n, int* vet, int elem)
{
  /* no início consideramos todo o vetor */
  int ini = 0;
  int fim = n-1;
  int meio;
  /* enquanto a parte restante for maior que zero */
  while (ini <= fim) {
    meio = (ini + fim) / 2;
    if (elem < vet[meio])
      fim = meio - 1;  /* ajusta posição final */
    else if (elem > vet[meio])
      ini = meio + 1;  /* ajusta posição inicial */
    else
      return meio;     /* elemento encontrado */
  }
  return -1; /* não encontrou: restou tamanho zero */
}
```

Busca Binária em Vetor Ordenado com Função para Comparação

```
int busca_bin (int n, int* vet, int elem)
{
  int ini = 0;
  int fim = n-1;
  int meio, cmp;
  while (ini <= fim) {
    meio = (ini + fim) / 2;
    cmp = comp_int(elem, vet[meio]);
    if (cmp < 0)
      fim = meio - 1;
    else if (cmp > 0)
      ini = meio + 1;
    else
      return meio;
  }
  return -1;
}
```

```
int comp_int(int a, int b)
{
  if (a < b)
    return -1;
  else if (a > b)
    return 1;
  else
    return 0;
}
```

Busca Binária em Vetor Recursiva

- A Busca deve continuar na primeira metade do vetor:
  - Chamada recursiva com parâmetros:
    - O número de elementos da primeira parte restante
    - O mesmo ponteiro para o primeiro elemento
      - (pois a primeira parte tem o mesmo primeiro elemento do que o vetor como um todo)

- A busca deve continuar apenas na segunda parte do vetor:
  - Chamada recursiva com parâmetros:
    - Número de elementos restantes
    - Ponteiro para o primeiro elemento dessa segunda parte

  - Valor retornado deve ser corrigido

Busca Binária em Vetor Recursiva

```
int busca_bin_rec (int n, int* vet, int elem)
{
   /* testa condição de contorno: parte com tamanho zero */
   if (n <= 0)
      return -1;
   else {
      int meio = n/2;
      if (elem < vet[meio])
         return busca_bin_rec(meio,vet,elem);
      else if (elem > vet[meio]) {
         int r = busca_bin_rec(n-1-meio, &vet[meio+1],elem);
         if (r==-1)
            return -1;
         else
            return (meio+1+r);  /* correção da origem */
      }
      else  /* elem==vet[meio] */
         return meio;       /* elemento encontrado */
   }
}
```

Binary Search da `stdlib.h`

```
void * bsearch(void * info, void * v, int n, int tam,
      int (*cmp)(const void *, const void *));
```

- Parâmetros:
  - **info:** ponteiro para a informação que se deseja buscar;
  - **v:** vetor de ponteiros genéricos (ordenado);
  - **n:** número de elementos do vetor;
  - **tam:** tamanho em bytes de cada elemento (use sizeof para especificar);
  - **cmp:** ponteiro para função que compara elementos genéricos, sendo o primeiro o endereço da informação e o segundo é o ponteiro para um dos elementos do vetor. O critério de comparação deve ser o mesmo da ordenação de v. A função deve retornar <0 se a<b, >0 se a>b e 0 se a == b;

Binary Search da `stdlib.h`

```
void * bsearch(void * info, void * v, int n, int tam,
      int (*cmp)(const void *, const void *));
```

- Exemplo de função de comparação:

const é para garantir que a função não modificará os valores dos elementos

```
static int compInt(const void * a, const void * b)
{
  int * info = (int *)a; // converte o ponteiro genérico
  int * bb = (int *)b; // converte o ponteiro genérico
  if (*info < *bb) // faz as comparações
    return -1;
  else if (*info > *bb)
    return 1;
  else
    return 0;
}
```

Binary Search da `stdlib.h`

```
void * bsearch(void * info, void * v, int n, int tam,
      int (*cmp)(const void *, const void *));
```

- Exemplo de chamada:

```
int d = 23;
int *p;

p = (int *)bsearch(&d,v,N,sizeof(int),compInt); // N é o tamanho de v

i = p - v; // indice do elemento encontrado
```

Binary Search da `stdlib.h`

```
static int compInt(const void * a, const void * b)
{
  int * info = (int *)a;
  int * bb = (int *)b;
  if (*info < *bb)
    return -1;
  else if (*info > *bb)
    return 1;
  else
    return 0;
}
int main (void)
{
  int vet[6], elem = 16, *p;
  vet[0] = 8;
  vet[1] = 16;
  vet[2] = 22;
  vet[3] = 24;
  vet[4] = 25;
  vet[5] = 32;
  p = (int *)bsearch(&elem, vet, 6, sizeof(int), compInt);
  if (p != NULL)
    printf("Valor: %d \nIndice: %d\n", *p, p - vet);
  return 0;
}
```

`Bsearch` – Exemplo 2

```
struct aluno
{
  int matricula;
  char nome[41];
};
typedef struct aluno Aluno;

static int compPStructStr(const void * a, const void * b)
{
  char *info = (char *)a;
  Aluno **bb = (Aluno **)b;
  return strcmp(info, (*bb)->nome);
}

int main (void)
{
  Aluno *alunos[6];
  Aluno **p;
  char elem[] = "Maria";
  alunos[0] = (Aluno*)malloc(sizeof(Aluno)); alunos[0]->matricula = 2654951;
  strcpy(alunos[0]->nome, "Ana");
```

Bsearch – Exemplo 2

```
  alunos[1] = (Aluno*)malloc(sizeof(Aluno));
  alunos[1]->matricula = 62151578;
  strcpy(alunos[1]->nome, "Bruno");

  alunos[2] = (Aluno*)malloc(sizeof(Aluno));
  alunos[2]->matricula = 51364125;
  strcpy(alunos[2]->nome, "Joao");

  alunos[3] = (Aluno*)malloc(sizeof(Aluno));
  alunos[3]->matricula = 82135123;
  strcpy(alunos[3]->nome, "Julia");

  alunos[4] = (Aluno*)malloc(sizeof(Aluno));
  alunos[4]->matricula = 45612681;
  strcpy(alunos[4]->nome, "Maria");

  alunos[5] = (Aluno*)malloc(sizeof(Aluno));
  alunos[5]->matricula = 35641215;
  strcpy(alunos[5]->nome, "Pedro");

  p = (Aluno **) bsearch(&elem, alunos, 6, sizeof(Aluno*), compPStructStr);
  if (p != NULL)
    printf("%d\n", (*p)->matricula);
  return 0;
}
```

Revisão Linguagem C

MÓDULOS

Módulos

- Um programa em C pode ser dividido em **vários arquivos fontes**
  - Programas reais precisam ser divididos em vários arquivos;
  - Seria intratável programar e manter o código de um programa grande em único arquivo;

- É possível criar um arquivo separado para implementar funções do programa. Esse arquivo é chamado de **módulo**;

- A implementação de um programa pode ser composta por um ou mais módulos;

Módulos

- **Interface de um módulo de funções (Arquivo .h):**
  - Contem somente os protótipos das funções oferecidas pelo módulo;
  - Em geral possui o mesmo nome do módulo ao qual está associado;
  - Possui extensão .h

- **Módulo (Arquivo .c):**
  - Contem a implementação das funções que foram definidas na interface .h;

- **Programa Principal (Arquivo .c):**
  - Contem a função main do programa;
  - Inclui os módulos que foram criados e utiliza a suas funções;

Exemplo

- ***"Crie um programa para calcular a área e o volume de um cilindro"***
  - Podemos criar um módulo especifico para implementar as funções relacionadas ao cilindro;
  - Dessa forma, o nosso programa será dividido em 3 arquivos:
    - **Geometria.h** – interface com a definição dos protótipos das funções `volume_cilindro` e `area_cilindro`;
    - **Geometria.c** – implementação das funções `volume_cilindro` e `area_cilindro` de acordo com o protótipo definido na interface;
    - **Principal.c** – implementação da função principal do programa;

Exemplo

- **Arquivo Geometria.h:**

```
#define PI 3.14159

float volume_cilindro(float raio, float altura);

float area_cilindro(float raio, float altura);
```

Exemplo

- **Arquivo Geometria.c:**

```
#include <math.h>
#include "Geometria.h"

float volume_cilindro(float raio, float altura)
{
  float volume = PI * pow(raio,2) * altura;
  return volume;
}

float area_cilindro(float raio, float altura)
{
  float area = 2 * PI * raio * (altura + raio);
  return area;
}
```

Exemplo

- **Arquivo Principal.c:**

```c
#include <stdio.h>
#include <math.h>
#include "Geometria.h"

int main(void)
{
  float raio, altura, volume, area;

  printf("Entre com o valor do raio e da altura: ");
  scanf("%f %f", &raio, &altura);

  volume = volume_cilindro(raio, altura);
  area = area_cilindro(raio, altura);

  printf("Volume do cilindro: %f\n", volume);
  printf("Area do cilindro: %f\n", area);

  return 0;
}
```

Compilando Módulos com GNU C

- **Compilador GNU C**
  - O compilador GNU C (ou GCC) é um compilador **open-source**, distribuído pela Free Software Foundation (FSF) sob licença GNU GPL e é disponível em sistemas Linux, Mac OS X e Microsoft Windows.
  - O GNU C pode ser utilizado manualmente ou através de uma IDE como Dev-C++. Os **arquivos .c** devem ser primeiramente compilados para o **arquivo objeto** para que seja possível gerar o **arquivo executável**.
- **Arquivo objeto**
  - resultado de compilar um módulo
  - geralmente com extensão .o ou .obj
- **Ligador (Linker)**
  - junta todos os arquivos objeto em um único arquivo executável

Compilando Módulos com GNU C

- Exemplo:
  - str.c:
    - arquivo com a implementação das funções de manipulação de strings: “comprimento”, “copia” e “concatena”
    - usado para compor outros módulos que utilizem estas funções
      - módulos precisam conhecer os protótipos das funções em str.c

Compilando Módulos com GNU C

- Exemplo:
  - prog1.c: arquivo com o seguinte código

```
#include <stdio.h>

int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void) {
  char str[101], str1[51], str2[51];
  printf("Digite uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Digite outra seqüência de caracteres: ");
  ...
  return 0;
}
```

Compilando Módulos com GNU C

- Exemplo:
  - prog1.exe:
    - arquivo executável gerado em 2 passos:
      - compilando os arquivos str.c e prog1.c separadamente
      - ligando os arquivos resultantes em um único arquivo executável
    - sequência de comandos para o compilador Gnu C (gcc):

```
> gcc -c str.c -o str.o
> gcc -c prog1.c -o prog1.o
> gcc -o prog1.exe str.o prog1.o
```

Compilando Módulos com GNU C

- Interface de um módulo de funções:
  - arquivo contendo apenas:
    - os protótipos das funções oferecidas pelo módulo
    - os tipos de dados exportados pelo módulo (typedef's, struct's, etc)
  - em geral possui:
    - nome: o mesmo do módulo ao qual está associado
    - extensão: .h

Compilando Módulos com GNU C

- Inclusão de arquivos de interface no código:
  - #include <arquivo.h>
    - protótipos das funções da biblioteca padrão de C
  - #include "arquivo.h"
    - protótipos de módulos do usuário

Compilando Módulos com GNU C

- Exemplo – arquivos str.h e prog1.c

```
/* Funções oferecidas pelo modulo str.c */
/* Função comprimento
** Retorna o número de caracteres da string passada como parâmetro
*/
int comprimento (char* str);
/* Função copia
** Copia os caracteres da string orig (origem) para dest (destino)
*/
void copia (char* dest, char* orig);
/* Função concatena
** Concatena a string orig (origem) na string dest (destino)
*/
void concatena (char* dest, char* orig);
```

Compilando Módulos com GNU C

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

int main (void) {
  char str[101], str1[51], str2[51];
  printf("Digite uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Digite outra seqüência de caracteres: ");
  scanf(" %50[^\n]", str2);
  copia(str, str1);
  concatena(str, str2);
  printf("Comprimento da concatenação: %d\n",comprimento(str));
  return 0;
}
```

Revisão Linguagem C

TIPO DE DADOS ABSTRATOS

Tipo de Dados Abstratos

- **Um TAD define:**
  - Um novo tipo de dado;
  - O conjunto de operações para manipular dados desse tipo;

- **Um TAD facilita:**
  - A manutenção e a reutilização de código;
  - A implementação das operações de TAD não precisa ser conhecida;
  - Para utilizar um TAD é necessário conhecer a sua funcionalidades, mas não a sua implementação;

- **Um TAD possui:**
  - Uma interface definindo o nome e as funcionalidades do TAD;
  - Uma implementação contendo a real implementação das funcionalidades do TAD;

Tipo de Dados Abstratos

- **Exemplo: TAD Ponto** - Tipo de dado para representar um ponto no R2 com as seguintes operações:
  - **cria** - cria um ponto com coordenadas x e y;
  - **libera** - libera a memória alocada por um ponto;
  - **acessa** - retorna as coordenadas de um ponto;
  - **atribui** - atribui novos valores às coordenadas de um ponto;
  - **distancia** - calcula a distância entre dois pontos;

Tipo de Dados Abstratos

- **Interface do TAD Ponto:**
  - Define o nome do tipo e os nomes das funções exportadas;
  - A composição da estrutura Ponto não faz parte da interface:
    - Não é exportada pelo módulo;
    - Não faz parte da interface do módulo;
    - Não é visível para outros módulos;
  - Os módulos que utilizarem o TAD Ponto:
    - Não poderão acessar diretamente os campos da estrutura Ponto;
    - Só terão acesso aos dados obtidos através das funções exportadas.

ponto.h - arquivo com a interface de *Ponto*

```
/* TAD: Ponto (x,y) */
/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */
/* Função cria - Aloca e retorna um ponto com coordenadas (x,y) */
Ponto* pto_cria (float x, float y);

/* Função libera - Libera a memória de um ponto previamente criado */
void pto_libera (Ponto* p);

/* Função acessa - Retorna os valores das coordenadas de um ponto */
void pto_acessa (Ponto* p, float* x, float* y);

/* Função atribui - Atribui novos valores às coordenadas de um ponto
*/
void pto_atribui (Ponto* p, float x, float y);

/* Função distancia - Retorna a distância entre dois pontos */
float pto_distancia (Ponto* p1, Ponto* p2);
```

Tipo de Dados Abstratos

- **Implementação do TAD Ponto:**
  - Inclui o arquivo de interface de Ponto;
  - Define a composição da estrutura Ponto;
  - Inclui a implementação das funções externas;

Arquivo: ponto.c (implementação do TAD Ponto)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "ponto.h"

struct ponto {
   float x;
   float y;
};

Ponto* pto_cria(float x, float y)
{
   Ponto* p = (Ponto*) malloc(sizeof(Ponto));
   if (p == NULL) {
      printf("Memória insuficiente!\n");
      exit(1);
   }
   p->x = x;
   p->y = y;
   return p;
}
```

Tipo de Dados Abstratos

```
void pto_libera(Ponto* p){
   free(p);
}
void pto_acessa(Ponto* p, float* x, float* y){
   *x = p->x;
   *y = p->y;
}
void pto_atribui(Ponto* p, float x, float y) {
   p->x = x;
   p->y = y;
}
float pto_distancia(Ponto* p1, Ponto* p2) {
   float dx = p2->x - p1->x;
   float dy = p2->y - p1->y;
   return sqrt(dx*dx + dy*dy);
}
```

Tipo de Dados Abstratos

- **Exemplo de utilização do TAD Ponto:**

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

int main(void)
{
   float x, y;
   Ponto* p = pto_cria(2.0,1.0);
   Ponto* q = pto_cria(3.4,2.1);
   float d = pto_distancia(p,q);
   printf("Distancia entre pontos: %f\n",d);
   pto_libera(q);
   pto_libera(p);
   return 0;
}
```

Tipo de Dados Abstratos

- **Exemplo: TAD Circulo** - Tipo de dado para representar um circulo com as seguintes operações:
  - **cria** - cria um círculo com centro (x,y) e raio r;
  - **libera** - libera a memória alocada por um círculo;
  - **area** - calcula a área do círculo;
  - **interior** - verifica se um dado ponto está dentro do círculo;

circulo.h - arquivo com a interface do TAD

interface *ponto.h* incluída na interface pois a operação *interior* faz uso do tipo Ponto

```
/* TAD: Círculo */
/* Dependência de módulos */
#include "ponto.h"

/* Tipo exportado */
typedef struct circulo Circulo;

/* Funções exportadas */
/* cria - Aloca e retorna um círculo com centro (x,y) e raio r */
Circulo* circ_cria (float x, float y, float r);

/* libera - Libera a memória de um círculo previamente criado */
void circ_libera (Circulo* c);

/* area - Retorna o valor da área do círculo */
float circ_area (Circulo* c);

/* interior - Verifica se um dado ponto p está dentro do círculo */
int circ_interior (Circulo* c, Ponto* p);
```

circulo.c - arquivo com o TAD Circulo

```
#include <stdlib.h>
#include "circulo.h"

#define PI 3.14159

struct circulo {
   Ponto* p;
   float r;
};

Circulo* circ_cria (float x, float y, float r)
{
   Circulo* c = (Circulo*)malloc(sizeof(Circulo));
   c->p = pto_cria(x,y);
   c->r = r;
}

…
```

```
…

void circ_libera (Circulo* c)
{
   pto_libera(c->p);
   free(c);
}

float circ_area (Circulo* c)
{
   return PI*c->r*c->r;
}

int circ_interior (Circulo* c, Ponto* p)
{
   float d = pto_distancia(c->p,p);
   return (d<c->r);
}
```

Tipo de Dados Abstratos

- Podemos utilizar um TAD sem precisarmos conhecer a sua implementação.

- Exemplo: **TAD Matriz** - Tipo de dado para representar matrizes com as seguintes operações:
  - **cria**: operação que cria uma matriz de dimensão m por n;
  - **libera**: operação que libera a memória alocada para a matriz;
  - **acessa**: operação que acessa o elemento da linha i e da coluna j da matriz;
  - **atribui**: operação que atribui o elemento da linha i e da coluna j da matriz;
  - **linhas**: operação que devolve o número de linhas da matriz;
  - **colunas**: operação que devolve o número de colunas da matriz.

matriz.h - arquivo com a interface do TAD

```
/* TAD: matriz m por n */
/* Tipo exportado */
typedef struct matriz Matriz;
/* Funções exportadas */
/* cria - Aloca e retorna uma matriz de dimensão m por n */
Matriz* mat_cria(int m, int n);

/* libera - Libera a memória de uma matriz previamente criada. */
void mat_libera(Matriz* mat);

/* acessa - Retorna o valor do elemento da linha i e coluna j */
float mat_acessa(Matriz* mat, int i, int j);

/* atribui - atribui o valor ao elemento da linha i e coluna j */
void mat_atribui(Matriz* mat, int i, int j, float v);

/* linhas - Retorna o número de linhas da matriz */
int mat_linhas(Matriz* mat);

/* colunas - Retorna o número de colunas da matriz */
int mat_colunas(Matriz* mat);
```

Tipo de Dados Abstratos

- A estrutura representando uma matriz na implementação como vetores simples:

```
struct matriz {
   int lin;
   int col;
   float* v;
};
```

- A estrutura representando uma matriz na implementação como vetores de ponteiros:

```
struct matriz {
   int lin;
   int col;
   float** v;
};
```

Tipo de Dados Abstratos

- Verificar se uma matriz é **simétrica**: retorna 1 se verdadeiro e 0 se falso:
  - Uma matriz é **simétrica** se ela for igual a sua transposta;
  - Uma matriz **transposta** é o resultado da troca de linhas por colunas da matriz original;

```
int simetrica(Matriz *mat)
{
   int i, j;
   for (i=0; i<mat_linhas(mat); i++)
   {
      for (j=0; j<mat_colunas(mat); j++)
      {
         if (mat_acessa(mat,i,j) != mat_acessa(mat,j,i))
           return 0;
      }
   }
   return 1;
}
```

Tipo de Dados Abstratos

- **Multiplicar** uma matriz por um **escalar**:

```
Matriz *mult_matriz_escalar(Matriz *mat_a, float s) {
   int i, j;
   Matriz *mat_b = mat_cria(mat_linhas(mat_a), mat_colunas(mat_a));
   for (i=0; i<mat_linhas(mat_a); i++)
   {
      for (j=0; j<mat_colunas(mat_a); j++)
      {
         mat_atribui(mat_b, i, j, s * mat_acessa(mat_a,i,j));
      }
   }
   return mat_b;
}
```

Resumo

Módulo	arquivo com funções que representam apenas parte da implementação de um programa completo
Arquivo objeto	resultado de compilar um módulo geralmente com extensão .o ou .obj
Interface de módulo	arquivo contendo apenas os protótipos das funções oferecidas pelo módulo, e os tipos de dados exportados pelo módulo
TAD	define um novo tipo de dado e o conjunto de operações para manipular dados do tipo
Interface de TAD	define o nome do tipo e das funções exportadas

Revisão Linguagem C

VAMOS EXERCITAR?

Exercício Aula 1
Vetores de Ponteiros para Estruturas

- **Exemplo:** Crie um programa para registrar uma tabela com dados de alunos.
  - Deverá ser organizada em um vetor de ponteiros.
  - Utilizando a estrutura de dados abaixo.
  - Faça todas as alocações de memória dinamicamente.
  - Escreva as funções de manipulação de strings de forma recursiva:
    - Imprime String
    - Comprimento de String
  - Estrutura de dados de cada aluno:
    - matrícula: número inteiro
    - nome: cadeia com até 80 caracteres
    - endereço: cadeia com até 120 caracteres
    - telefone: cadeia com até 20 caracteres
- Obs: Não é necessário criar uma interface para coletar os dados que serão inseridos para testar a tabela com os dados. Podem ser inseridos programaticamente (hard-coded).

Exercício Aula 2
Tipo de Dados Abstrato c/ Busca

- **Exemplo:** Criar um Tipo de Dados Abstrato para a estrutura de dados de alunos.
  - Deverá ser organizada em um vetor de ponteiros.
  - Utilizando a estrutura de dados abaixo.
  - Criar as funções para inserção, deleção, atualização e busca p/ nome
    - A busca pode ser linear ou binária (indique qual foi a escolhida)

  - Estrutura de dados de cada aluno:
    - matrícula: número inteiro
    - nome: cadeia com até 80 caracteres
    - endereço: cadeia com até 120 caracteres
    - telefone: cadeia com até 20 caracteres

- Obs: Não é necessário criar uma interface para coletar os dados que serão inseridos para testar a tabela com os dados. Podem ser inseridos programaticamente (hard-coded).

Exemplo 1 – Busca String

- **Escreva um programa que permita buscar um nome em um vetor de strings ordenado alfabeticamente.**
  - A função de busca deve seguir o seguinte protótipo:

```
int busca_bin(int n, char vet[][20], char *elem)
```

  - Dica: lembre-se que a função `strcmp(a, b)` já faz o trabalho de retornar:
    - -1 se a < b;
    - +1 se a > b;
    - 0 se a == b;

Exemplo 1 - Solução

```
int busca_bin(int n, char vet[][20], char *elem)
{
  int ini = 0;
  int fim = n-1;
  int meio, cmp;
  while(ini <= fim)
  {
     meio = (ini + fim) / 2;
     cmp = strcmp(elem, vet[meio]);
     if (cmp < 0)
       fim = meio - 1;
     else if (cmp > 0)
       ini = meio + 1;
    else
      return meio;
  }
  return -1;
}
```

Exemplo 1 - Solução

```
int main (void)
{
  char nomes[][20] = {{"Ana"},{"Joao"},{"Maria"},
                      {"Pedro"},{"Silvio"}};
  char elem[] = "Silvio";

  int res = busca_bin(5, nomes, elem);

  printf("%d\n", res);
  return 0;
}
```

Exemplo 2 – Busca Estrutura

- Escreva um programa que crie um vetor de ponteiros para a estrutura Aluno (ordenado crescentemente por nome e matricula) e permita realizar buscas por nomes nesse vetor.

```
struct aluno
{
  char *nome;
  int matricula;
};
typedef struct aluno Aluno;
```

  - A função de busca binária deve receber como parâmetros o número de alunos, o vetor e o nome do aluno que se deseja buscar, e deve ter como valor de retorno um ponteiro para o registro do aluno procurado. Se não houver um aluno com o nome procurado, a função deve retornar NULL.

Exemplo 2 - Solução

```
Aluno* busca_bin(int n, Aluno *vet, char *elem)
{
  int ini = 0;
  int fim = n-1;
  int meio, cmp;
  while(ini <= fim)
  {
    meio = (ini + fim) / 2;
    cmp = strcmp(elem, vet[meio].nome);
    if (cmp < 0)
      fim = meio - 1;
    else if (cmp > 0)
      ini = meio + 1;
    else
      return &vet[meio];
    }
  return NULL;
}
```

Exemplo 2 - Solução

```
int main (void)
{
  Aluno alunos[] = {{"Ana", 1},{"Joao", 2},{"Maria", 3},
                    {"Pedro", 4},{"Silvio", 5}};
  char elem[] = "Silvio";

  Aluno *res = busca_bin(5, alunos, elem);

  printf("Nome: %s\nMatricula: %d \n", res->nome, res->matricula);
  return 0;
}
```

Exemplo 3 – Busca c/ Critério de Ordenação

- Considere a seguinte estrutura representando um registro de um calendário de provas:

```
struct prova
{
  char *disciplina;
  Data dt_prova;
  Data dt_seg_chamada;
};
typedef struct prova Prova;

struct data
{
  int dia, mes, ano;
};
typedef struct data Data;
```

Exemplo 3 – Busca c/ Critério de Ordenação

- Escreva uma função que faça uma busca binária em um vetor de ponteiros para o tipo Prova, cujos elementos estão em ordem cronológica, de acordo com a data da prova (dt_prova), com desempate pela ordem alfabética de acordo com o nome da disciplina.
  - Se existir mais de uma prova na data procurada, a função deve retornar o índice da primeira delas;
  - Se não houver uma prova com a data procurada, a função deve retornar -1;
  - Sua função deve ter o seguinte cabeçalho:

```
int busca(Prova **v, int n, Data d);
```

Exemplo 3 - Solução

```
int datacmp(Data d1, Data d2)
{
  if(d1.ano<d2.ano)
    return -1;
  if(d1.ano>d2.ano)
    return 1;
  if(d1.mes<d2.mes)
    return -1;
  if(d1.mes>d2.mes)
    return 1;
  if(d1.dia<d2.dia)
    return -1;
  if(d1.dia>d2.dia)
    return 1;

  return 0;
}
```

Exemplo 3 - Solução

```
int busca_bin(Prova **v, int n, Data d)
{
  int ini = 0;
  int fim = n-1;
  int meio, cmp;
  while(ini <= fim)
  {
    meio = (ini + fim) / 2;
    cmp = datacmp(d, v[meio]->dt_prova);
    if (cmp == -1)
      fim = meio - 1;
    else if (cmp == 1)
      ini = meio + 1;
    else
    {
      while((meio > 0) && (datacmp(d, v[meio-1]->dt_prova)==0))
        meio--;
      return meio;
    }
  }
  return -1;
}
```

Leitura Complementar

- Celes, W., Cerqueira, R., Rangel, J.L., **Introdução a Estruturas de Dados – Uma introdução com técnicas de programação em C**, Ed. Campus, 2004

  - **Vetores de Ponteiros: Capítulo 6 - Cadeia de caracteres**
  - **Busca em Vetores: Capítulo 17 – Busca**
  - **Módulo: Capítulo 9 – Módulos e compilação em separado**
  - **TDA: Capítulo 9 – Tipos de dados abstratos**