- Que faz o programa?

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

int main(){
   int num, i=10;

   printf("Digite o numero: ");
   scanf("%d", &num);
   while(num){
      printf("%d", num%i);
      num /= i;
   }
   printf("\n");

   system("PAUSE");
   return 0;
}
```

Linguagens de Programação II

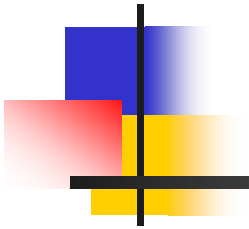
Ponteiros

Alocação Dinâmica de Memória

Dany Sanchez Dominguez

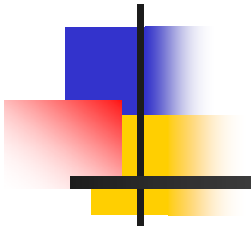
dsdominguez@gmail.com

Sala 1 – NBCGIB



Roteiro

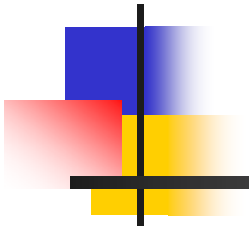
- Introdução
- Definição de ponteiros
- Declaração de ponteiros
- Inicialização de ponteiros
- Manipulando ponteiros
- Gerenciamento de memória
- Operações com ponteiros
- Vetores de ponteiros
- Ponteiros para ponteiros
- Alocação dinâmica
- Erros comuns com ponteiros.



Introdução

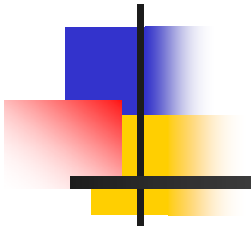
- Ponteiros?

- Os ponteiros são um dos recursos mais poderosos da linguagem C,
- Os ponteiros estão entre os aspectos de C mais difíceis de dominar,
- O correto entendimento de ponteiros é crítico para uma programação bem sucedida em C.



Introdução

- Os ponteiros:
  - Permitem modificar os argumentos de uma função (passagem de parâmetros por referencia),
  - Suportam as rotinas de alocação dinâmica de memória (gerenciamento dinâmico de memória),
  - podem aumentar a eficiência de certas rotinas.



Introdução

Dados estáticos VS Dados dinâmicos

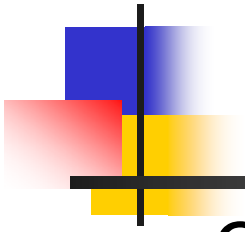
- Memória é alocada em tempo de compilação
- A quantidade de memória reservada é invariante
- Memória é gerenciada pelo compilador e o SO
- Todas as variáveis declaradas tradicionalmente

- Memória é alocada em tempo de execução
- A quantidade de memória reservada é variável
- O programador gerencia a memória
- Variáveis alocadas dinamicamente

Exemplo: Dados estáticos

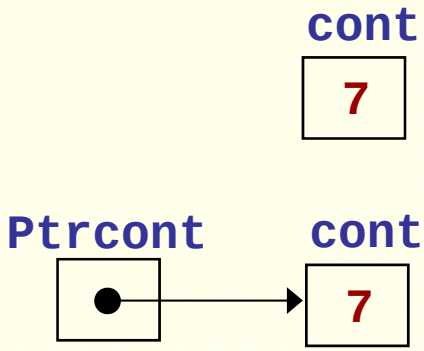
```
/* Calcula a media de tres valores */
#include <stdio.h>
#include <stdlib.h>
main(){
   float A, B, C, MEDIA;
   scanf (“%f %f %f” , &A, &B, &C);
   MEDIA = (A + B + C)/3;
   printf (“A media eh: %f” , MEDIA);
   system("PAUSE");
   return 0;
}
```

- tipo e quantidade de variáveis imutável, para modificar a memória usada é necessário modificar o código fonte e recompilar.

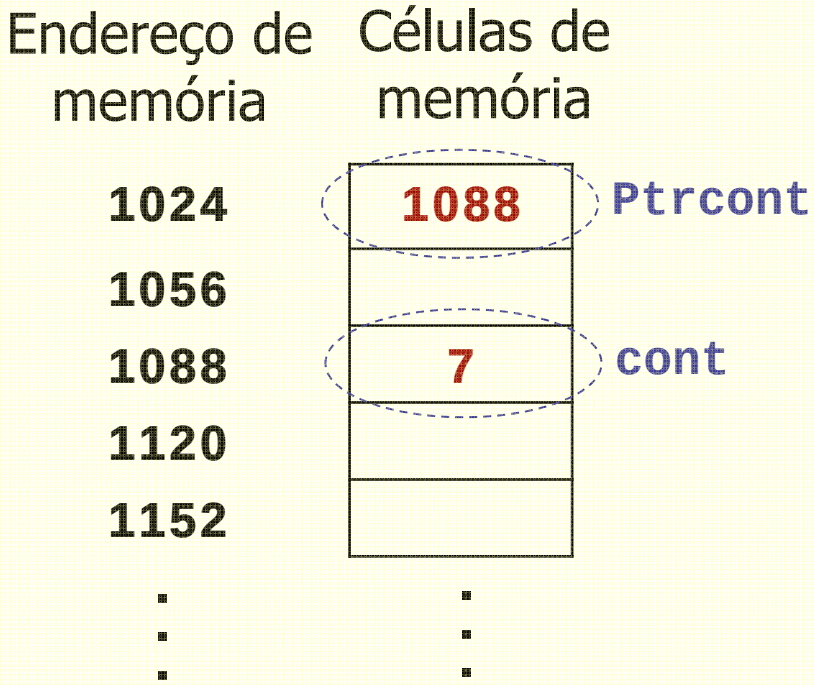


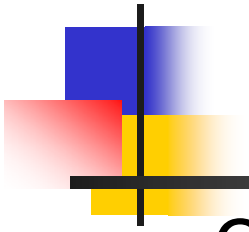
Ponteiros

- Os ponteiros são variáveis que contém endereços de memória como valores,
- Variável: faz referência direta a um valor
- Ponteiro: faz referência indireta a um valor (usando o endereço de memória da variável que contêm o valor)
- Se uma variável **`Ptrcont`** contém o endereço de uma outra variável **`cont`**, então a primeira variável é dita para apontar para a segunda.



- **cont** faz uma referencia direta a uma variável cujo valor é 7
- **Ptrcont** faz uma referencia indireta a uma variável cujo valor é 7





Declaração de Ponteiros

- Os ponteiros como quaisquer outras variáveis devem ser declaradas antes de serem usadas,
- Podemos declarar ponteiros para quaisquer tipo de dado básico ou agregado,
- Sintaxe: *tipo_de_dados* * *nome_ponteiro*;
- * é chamado de operador de referencia indireta,
- ao usarmos o operador de referencia indireta em uma declaração declaramos uma variável ponteiro.

Declaração de Ponteiros

- Exemplo:

```
int *Ptrcont, cont;
float *Ptrsom, som, div;
```

- **`Ptrcont`** é um ponteiro para um valor inteiro,
- **`Ptrsom`** é um ponteiro para um valor de ponto flutuante,
- **`cont`**, **`som`** e **`div`** não são ponteiros.

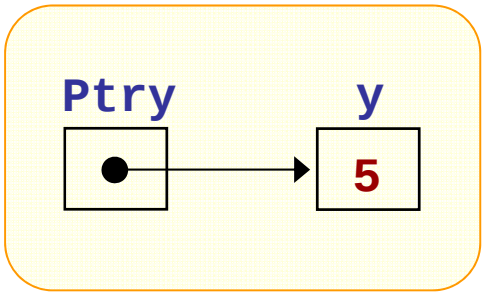
Inicialização de Ponteiros

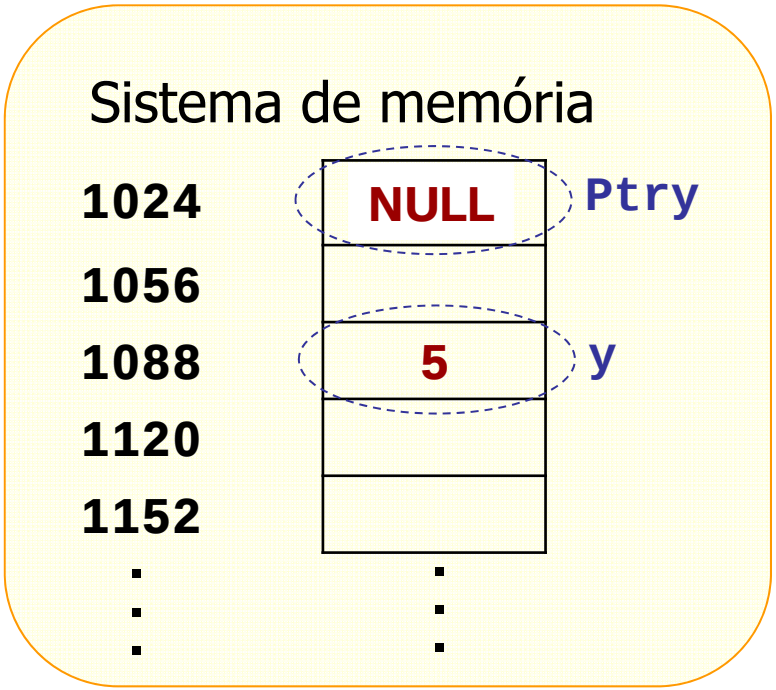
- Ponteiros devem ser inicializados ao serem declarados ou em uma instrução de atribuição,
- Um ponteiro pode ser inicializado com **`NULL`** ou com um endereço de variável,
- Um ponteiro com valor **`NULL`** não aponta para lugar algum,
- **`NULL`** é uma constante simbólica definida no arquivo de cabeçalho **`<stdio.h>`**
- Para inicializar um ponteiro com um endereço de variável usamos o operador de endereço **`&`**,
- O operador **`&`**, é um operador unário que retorna o endereço de seu operando.

Inicialização de Ponteiros

- Exemplo:

```
int y=5;
int *Ptry = NULL;
  . . .
Ptry = &y;
  . . .
```





- **Sempre** inicialize seus ponteiros para evitar resultados inesperados.

Manipulando Ponteiros

- O operador de referencia indireta (*) pode ser usado para acessar o conteúdo de uma variável apontada,
- *Ptry permite o acesso ao valor do objeto apontado por Ptry,
- `printf("%d", *Ptry);` imprime o valor 5 na tela
- `*Ptry = 10;` modifica o valor de y para 10
- Usar o operador * para manipular o conteúdo da variável apontada é chamado desreferenciar um ponteiro.

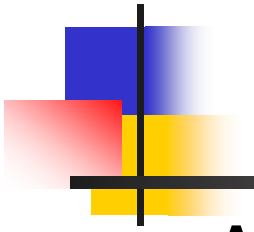
Manipulando Ponteiros

- * (*asterisco*), acessa o conteúdo da variável que está sendo apontada
- & (*e comercial*), acessa o endereço da variável
- **DECORAR!!!:**
  - * "conteúdo do endereço armazenado em ..."
  - & "endereço de ..."
- Um programador iniciante pode confundir
  - o operador de referencia com operador de multiplicação
  - o operador de endereço com o operador lógico AND
- Fique de olho são muito diferentes!!!

```c
/* Usando os operadores & e * */
#include <stdio.h>
#include <stdlib.h>

main()
{
   int a;      /* a é um enteiro */
   int *aPtr;  /* aPtr é um ponteiro para um inteiro */
   a = 7;
   aPtr = &a;  /* aPtr aponta para a */

   printf("O endereco de a e %p\n"
          "O valor de aPtr e %p\n\n", &a, aPtr);
   printf("O valor de a e %d\n"
          "O valor de *aPtr e %d\n\n", a, *aPtr);

   system("PAUSE");
   return 0;
}
```

Envia para a tela um enteiro hexadecimal

```
O endereco de a e 0022FF74
O valor de aPtr e 0022FF74

O valor de a e 7
O valor de *aPtr e 7

Pressione qualquer tecla para continuar. . .
```



Gerenciamento de memória

- Ao trabalharmos com ponteiros implicitamente manipulamos células de memória,
- Na declaração de um ponteiro o tipo indica a quantidade de bytes que serão manipulados,
- Conhecer a quantidade de bytes associada ao ponteiro permite ao compilador realizar diferentes operações com o ponteiro,
- Em ocasiões o programador precisa conhecer a quantidade de bytes associada a um tipo de dados básico ou agregado,

Gerenciamento de memória

- Para conhecer o tamanho em bytes de um tipo básico ou de uma estrutura de dados usamos o operador **`sizeof()`**,
- O operador **`sizeof()`** pode ser aplicado a qualquer nome de variável, tipo de dados ou constante,

- Sintaxe:

  sizeof(nome_ou_tipo)

Gerenciamento de memória

- Exemplo:

```
/* Demonstrando o operador sizeof */
#include <stdio.h>
#include <stdlib.h>

main(){
   printf("Tamanho em bytes dos tipos de dados:\n"
          "       sizeof(char) = %d\n"
          "        sizeof(int) = %d\n"
          "      sizeof(float) = %d\n"
          "     sizeof(double) = %d\n",
          sizeof(char), sizeof(int),
          sizeof(float), sizeof(double));
   system("PAUSE");
   return 0;
}
```

Gerenciamento de memória

- Exemplo ...

```
Tamanho em bytes dos tipos de dados:
       sizeof(char) = 1
        sizeof(int) = 4
      sizeof(float) = 4
     sizeof(double) = 8
Pressione qualquer tecla para continuar. . .
```

- Um ponteiro pode apenas apontar para variáveis do mesmo tipo,
- Se um ponteiro `float` apontar para uma variável `int` ao desreferenciar o ponteiro teremos resultados inesperados.

Gerenciamento de memória

- Em ocasiões é desejável termos um ponteiro que possa apontar a variáveis de qualquer tipo,
- Como podemos declarar um ponteiro que possa apontar a uma variável de qualquer tipo?

```
void *Ptr;
```

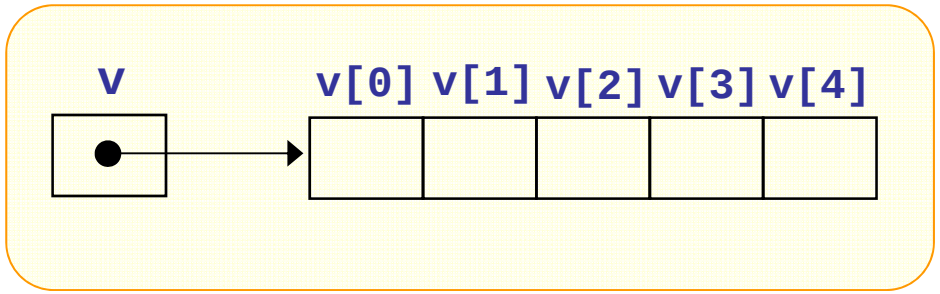
- Um ponteiro de tipo void **NÃO** pode ser desreferenciado, pois não existe informação sobre a quantidade de bytes que o compilador deve desreferenciar.

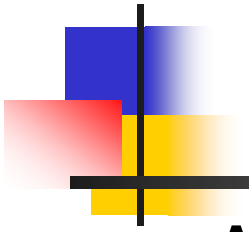
Operações com ponteiros

- Existe uma estreita relação entre ponteiros e vetores,
- Ao declararmos um vetor na forma

```
int v[5];
```

- a variável `v` é um ponteiro **constante** para o primeiro elemento do vetor, ela contém o endereço base do vetor

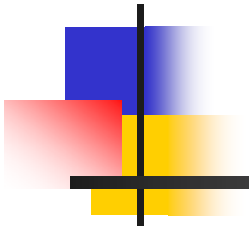




Operações com ponteiros

- Ao mesmo tempo os ponteiros são operandos válidos em expressões:
  - aritmeticas
  - de atribuição
  - de comparação

- Entretanto não todas as operações básicas são válidas com ponteiros.



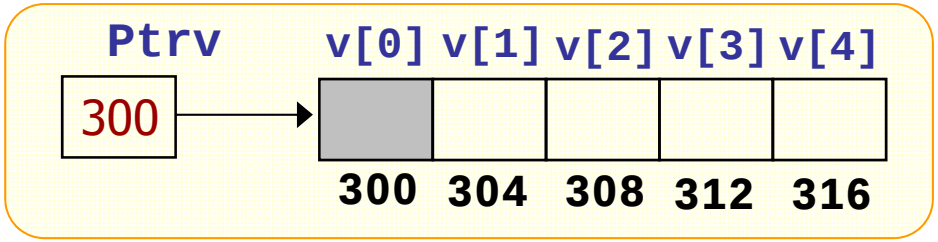
Operações aritméticas

- Um conjunto limitado de operações aritméticas pode ser realizado com ponteiros:
  - incremento (++)
  - decremento (--)
  - somar um inteiro a um ponteiro (+ ou +=)
  - substrair um inteiro a um ponteiro (- ou -=)

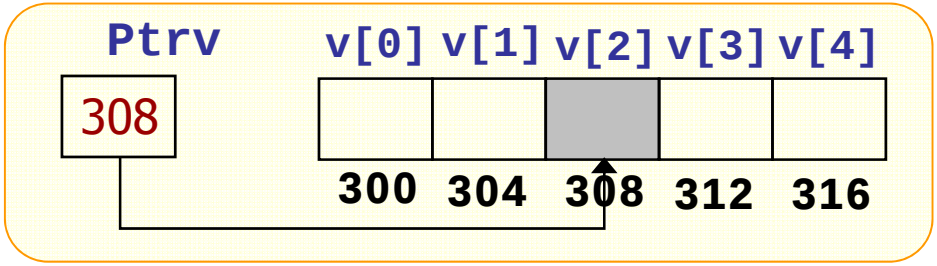
Operações aritméticas

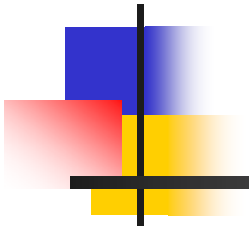
- Considere as seguintes declarações:

```
int v[5], int *Ptrv;
Ptrv = v; /* Ou Ptrv = &v[0] */
```



- Que acontece se somarmos 2 ao ponteiro `Ptrv`?





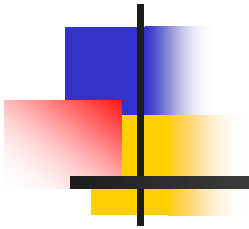
Operações aritméticas

- Quando um inteiro é adicionado a um ponteiro,
- O ponteiro não é simplesmente incrementado por tal inteiro,
- mas sim por tal inteiro vezes o tamanho do objeto ao qual o ponteiro se refere,
- o numero de bytes depende do tamanho do objeto apontado.

```c
/* Operações aritmeticas com ponteiros */
#include <stdio.h>
#include <stdlib.h>

main(){
  int v[5] = {1,2,3,4,5}, i;
  int *Ptrv;

  Ptrv = v;
  printf("%d\n", *Ptrv);
  Ptrv += 4;
  printf("%d\n", *Ptrv);
  for(i=0;i<4;i++){
    Ptrv--;
    printf("%d\n", *Ptrv);
  }
  system("PAUSE");
  return 0;
}
```

```
1
5
4
3
2
1
```

- Que imprime na tela o programa anterior?

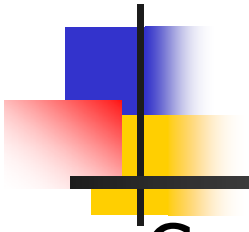


Operações aritméticas

- As operações com ponteiros podem ser utilizadas para manipular os elementos de um vetor sem usarmos a notação de subscrito.
- Dada a seguinte declaração:

```
int v[5], *Ptrv;
Ptrv = &v[0]
```

- As notações `v[3]` e `*(Ptr+3)` são equivalentes. Por que?
- Qual é a diferencia entre `*(Ptr+3)` e `(*Ptr+3)`?



Operações aritméticas

Considerando a declaração anterior a notação `*(v+3)` é inválida. Por que?

- `v` é um ponteiro constante e seu valor não pode ser modificado.
- As operações aritméticas envolvendo ponteiros não tem significado algum se não forem realizadas em um vetor.
- Não podemos assumir que duas variáveis estejam armazenadas contiguamente na memória a menos que sejam elementos de um vetor.

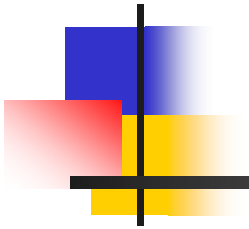
Operação de atribuição

- A atribuição de ponteiros pode acontecer se ambos forem do mesmo tipo:

```
int *Ptr1, *Ptr2;
  ...
Ptr1 = Ptr2;
```

- Caso os ponteiros não sejam do mesmo tipo devemos converter o ponteiro da direita para o tipo do ponteiro da esquerda (casting),

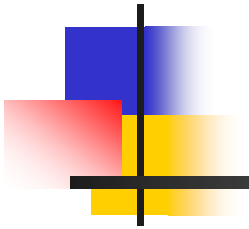
```
int *Ptr1;
float *Ptr2;
  ...
Ptr1 = (int *) Ptr2;
```



Operação de atribuição

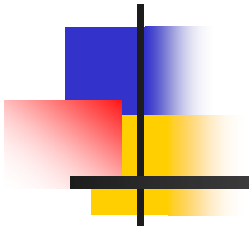
- A regra anterior não se aplica se usarmos um ponteiro void que pode representar qualquer tipo de ponteiro,
- Todos os ponteiros podem ser atribuídos a um ponteiro void,
- Um ponteiro void pode ser atribuido a um ponteiro de qualquer tipo,

```
void *Ptr1;
float *Ptr2;
  ...
Ptr1 = Ptr2;
```



Comparação de ponteiros

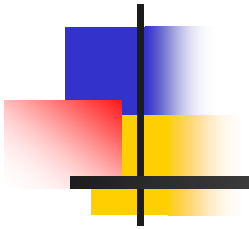
- Os ponteiros podem ser comparados usando operadores de igualdade ou relacionais,
- As comparações não tem significado se os ponteiros não apontam para membros do mesmo vetor,
- As comparações de ponteiros comparam os endereços armazenados,
- Uma comparação entre dois ponteiros do mesmo vetor mostra qual dos dois elementos tem um menor subscrito no vetor,



Comparação de ponteiros

- Um uso comum da comparação de ponteiros é para determinar se um ponteiro é NULL,
- Exemplo:

```
int *Ptr1, *Ptr2;
int v[5];
   ...
Ptr1 = v;
Ptr2 = &v[3];

if (Ptr1==NULL) ...

if (Ptr1 > Ptr2) ...
```



Vetores de ponteiros

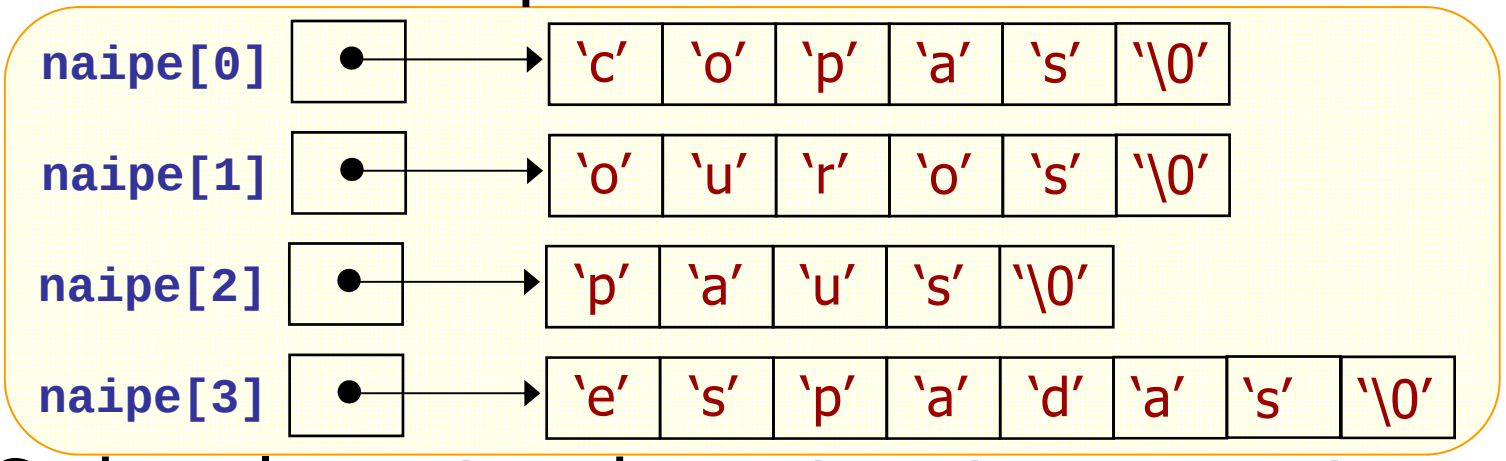
- Podemos ter vetores onde cada elemento do vetor seja um ponteiro,
- Um uso comum de tais estruturas é para termos matrizes (vetores bidimensionais) de caracteres ou um vetor de strings,
- Exemplo:

```
char * naipe[4] = {“copas”, “ouros”,
                   “paus”, “espadas”};
```

- A parte `naipe[4]` indica um vetor de quatro elementos,

Vetores de ponteiros

- A parte `char *` indica que cada elemento do vetor é um ponteiro a char,
- a lista de inicializadores coloca cada naipe no elemento correspondente ao vetor.



- Cada elemento do vetor tem um tamanho diferente o que nos brinda um uso mais eficiente da memória.

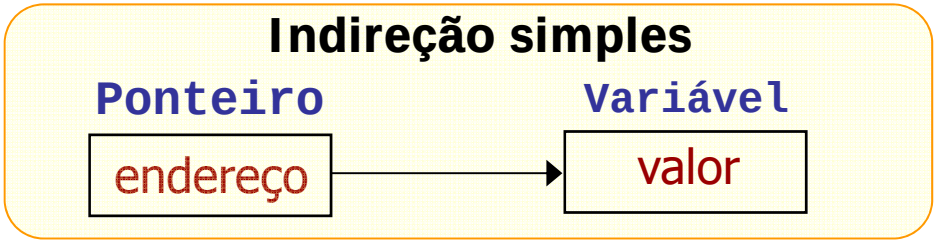
Vetores de ponteiros - Exemplo

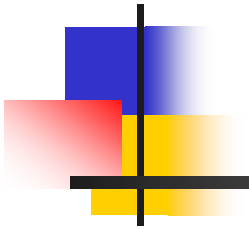
```
/* Imprimindo o vetor de ponteiros naipe */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(){
  int i;
  char * naipe[4] = {"copas", "ouros",
                     "paus", "espadas"};
  for(i=0;i<4;i++)
    puts(naipe[i]);

  system("PAUSE");
  return 0;
}
```

```c
/* Imprimindo o vetor de ponteiros naipe */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(){
  int i;
  char *naipe[4] = {"copas", "ouros",
                    "paus", "espadas"}, *Ptr;

  for(i=0;i<4;i++){
    Ptr = naipe[i];
    while(*Ptr){
      putchar(*Ptr);
      Ptr++;
    }
    printf("\n");
  }

  system("PAUSE");return 0;
}
```

Ponteiros para Ponteiros

- Podemos ter um ponteiro apontando para um ponteiro que aponta para uma variável (valor),
- Esta situação é chamada de indireção múltipla ou ponteiros para ponteiros:





Ponteiros para Ponteiros

- A indireção múltipla pode ser levada a qualquer dimensão desejada,
- Entretanto, raramente é necessário mais de três níveis,
- Um ponteiro para um ponteiro deve ser declarado da seguinte forma:

```
float **Ptr;
```

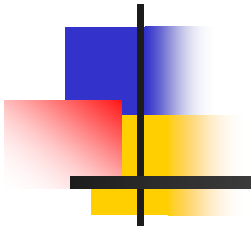
- Para acessar o valor final apontado o operador asterisco deve ser usado duas vezes.

Ponteiros para Ponteiros

- Exemplo:

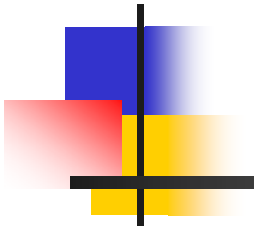
```
int main(){
   int x, *p, **q;

   x = 10;
   p = &x; // p aponta a um int
   q = &p; // q aponta a um int*

   printf("%d\n", **q);

   system("PAUSE");
   return 0;
}
```

- Ponteiros para ponteiros são utilizados em alocação dinâmica de matrizes.



Alocação dinâmica

- Até agora sempre temos utilizado estruturas de dados estáticas,
- Nas estruturas estáticas a memória é alocada em tempo de compilação,
- Exemplo: `int v[5];`
- Que fazermos quando não conhecemos o tamanho do vetor a priori?
- **Alocação dinâmica**



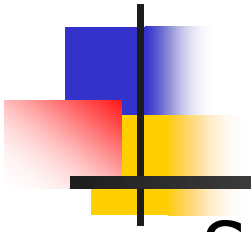
Alocação dinâmica

- O uso de alocação dinâmica permite criar, manter e destruir estruturas de dados durante a execução do programa, porem fazemos um uso mais eficiente da memória disponível,
- O limite máximo de alocação dinâmica de memória e a quantidade de memória (física ou virtual) disponível no sistema.
- Para alocação dinâmica são utilizadas as funções **`malloc`** e **`free`**, assim como o operador **`sizeof()`.**

Alocação dinâmica

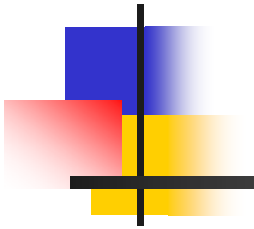
- A função **`malloc`** utiliza como argumento o numero de bytes a serem alocados e retorna um ponteiro tipo **`void`** para a memória alocada,
- Cabeçalho: **`void * malloc ( número de bytes );`**
- O ponteiro **`void`** pode ser atribuído a um ponteiro de qualquer tipo,
- A função **`malloc`** é usada normalmente com o operador **`sizeof().`**
- Syntaxe:

$$Ptr = malloc\,(sizeof(tipo\_de\_dados))$$



Alocação dinâmica

- Se não houver memória disponível `malloc` retorna um ponteiro `null`,
- Ao usar `malloc` devemos verificar se a memória foi alocada com sucesso, se o ponteiro retornado é `NULL` devemos imprimir uma mensagem de erro.
- Usamos função `free` para liberar a memória alocada.
- A função `free` retorna a memória ao sistema operacional para que possa ser realocada no futuro.

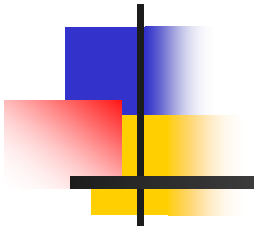


Alocação dinâmica

- Cabeçalho: `void free(void *p);`
- Syntaxe:

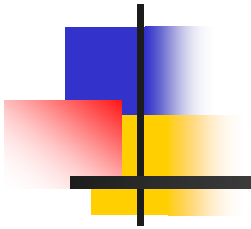
  free(Ptr)

- A função `free()` tem como parâmetro um ponteiro void,
- Indica que você deve passar o endereço de memória da área a liberar, e o tipo do ponteiro não é relevante,



Alocação dinâmica

- Toda memória alocada dinamicamente deve ser liberada usando `free` antes de finalizar a execução do programa,
- Não liberar memória alocada pode fazer com que o sistema esgote prematuramente sua memória,
- Note que não precisamos indicar ao comando `free` quantos bytes ele tem que liberar, (diferente de `malloc`),



Alocação dinâmica

- Ao executarmos nosso programa é gerada uma tabela interna, com os endereços de áreas alocadas dinamicamente, e a quantidade de bytes nessas áreas,
- Desta forma, basta que indiquemos qual endereço queremos liberar, o programa automaticamente consulta essa tabela e libera a área correta, isto é, a quantidade de bytes correta,
- É muito importante **NUNCA** chamar a função `free` com um parâmetro inválido, pois isto pode destruir a tabela de áreas alocadas dinamicamente!!!

Alocação dinâmica - Exemplos

- Alocando um float

```
float *Ptrf;
 ...
Ptrf = malloc (sizeof(float));
if (Ptrf==NULL) printf("Erro!!!");
 ...
free(Ptrf);
```

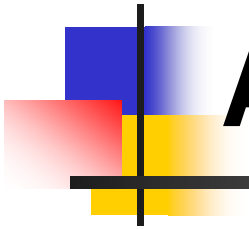
- Alocando um vetor de n inteiros

```
int *Ptrv, n;
 ... /* Leitura de n */
Ptrv = malloc (n * sizeof(int));
if (Ptrv==NULL) printf("Erro!!!");
 ...
free(Ptrv);
```

Alocação dinâmica - Exemplos

- Alocando uma matriz de m x n inteiros

```
int **Ptrm, n, m, i;
 ... /* Leitura de m e n */
Ptrm = malloc (m * sizeof(int *));
if (Ptrm==NULL) printf("Erro!!!");
for(i=0;i<m;i++){
  Ptrm[i] = malloc (n * sizeof(int));
  if (Ptrm[i]==NULL) printf("Erro!!!");
}
 ...
for(i=0;i<m;i++)
  free(Ptrm[i]);
free(Ptrm);
```



Alocação dinâmica - Exemplos

Escreva um programa que multiplique uma matriz de lm linhas e cm colunas por um vetor de lv elementos. As dimensões da matriz (lm, cm) e do vetor (lv) serão informadas pelo usuário.

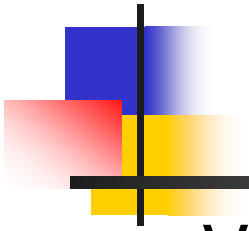
Alocação dinâmica - Exemplo

- Multiplicação de matriz por vetor:

$$A\vec{b} = \vec{c}$$

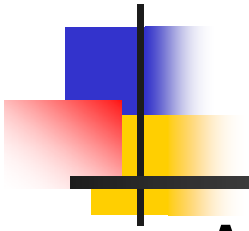
$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-10} & a_{m-10} & \cdots & a_{m-1n-1} \end{bmatrix}_{(m-1)\times(n-1)} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}_{(n-1)} = \begin{bmatrix} a_{00}b_0 + a_{01}b_1 + \cdots + a_{0n-1}b_{n-1} \\ a_{10}b_0 + a_{11}b_1 + \cdots + a_{1n-1}b_{n-1} \\ \vdots \\ a_{m-10}b_0 + a_{m-11}b_1 + \cdots + a_{m-1n-1}b_{n-1} \end{bmatrix}_{(m-1)}$$

- Numero de colunas da matriz = Numero de linhas do vetor
- Mostrar $m=3$, $n=2$



Alocação dinâmica - Exemplo

- Variáveis necessárias:
  - Numero de linhas da matriz – lm
  - Numero de colunas da matriz – cm
  - Elementos do matriz – A
  - Numero de elementos do vetor – lv
  - Elementos do vetor – b
  - Vetor resultante – c
  - Controladores de laços



Alocação dinâmica - Exemplo

- Algoritmo:
  - Inicio
  - Leitura de lm, cm e lv
  - Verificar se é possível fazer a multiplicação
  - Alocação dinâmica de memória
  - Multiplicar matriz por vetor
  - Mostrar Resultados
  - Liberar memória
  - Fim

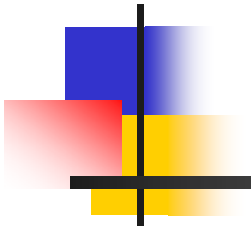
```c
/* Multiplica matriz por vetor (Ab = c)
   utilizando alocação dinâmica de memória */
#include <stdio.h>
#include <stdlib.h>

main(){
  int lm, cm, lv, //dimensoes da matriz e o vetor
      i, j, k;    //controladores de laços

  float **A, //Elementos da matriz
        *b, //Elementos do vetor de entrada
        *c; //Elementos do vetor resposta

  //Leitura das dimensoes da matriz e o vetor
  printf("Digite o numero de linhas da matriz: ");
  scanf("%d", &lm);
  printf("Digite o numero de colunas da matriz: ");
  scanf("%d", &cm);
  printf("Digite o numero de elementos do vetor: ");
  scanf("%d", &lv);
```

```
//Verificacao - multiplicacao definida
if(cm!=lv){
  printf("Multiplicacao indefinida!!!\n");
  system("PAUSE");
  return -1;
}
//Alocacao de memoria
b = malloc (lv * sizeof(float));
c = malloc (lv * sizeof(float));
A = malloc (lm * sizeof(float *));
//Verificando alocacao bem sucedida
if ((A==NULL)||(b==NULL)||(c==NULL)){
  printf("Erro: Memoria insuficiente!!!\n");
  system("PAUSE");
  return -1;
}
for(i=0;i<lm;i++){
  A[i] = malloc (lv * sizeof(float));
  if (A[i]==NULL){
    printf("Erro: Memoria insuficiente!!!\n");
    system("PAUSE");
    return -1;
  }
}
```

```
//Entrada de dados
for(i=0;i<lm;i++)
  for(j=0;j<cm;j++){
    printf("Digite o elemento [%d][%] de A: ", i, j);
    scanf("%f", &A[i][j]);
  }
for(i=0;i<lv;i++){
   printf("Digite o elemento [%d] de b: ", i);
   scanf("%f", &b[i]);
  }

//Multiplicacao de matriz por vetor
for(i=0;i<lm;i++){
  c[i] = A[i][0] * b[0];
  for(j=1;j<cm;j++)
    c[i] += A[i][j] * b[j];
}

//Impressao dos resultados
printf("\nResultados:\n");
for(i=0; i<lm; i++)
  printf("c[%d] = %8.3f\n", i, c[i]);
```

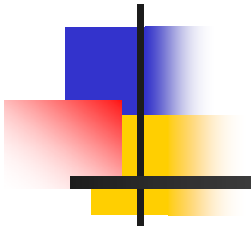
```
  //Liberando a memoria alocada
  free(c);
  free(b);
  for(i=0;i<lm;i++)
    free(A[i]);
  free(A);

  system("PAUSE");
  return 0;
}
```

```
Digite o numero de linhas da matriz: 2
Digite o numero de colunas da matriz: 3
Digite o numero de elementos do vetor: 3
Digite o elemento [0][] de A: 1
Digite o elemento [0][] de A: -1
Digite o elemento [0][] de A: 1
Digite o elemento [1][] de A: 2
Digite o elemento [1][] de A: 2
Digite o elemento [1][] de A: 3
Digite o elemento [0] de b: 1
Digite o elemento [1] de b: 0
Digite o elemento [2] de b: 1

Resultados:
c[0] =   2.000
c[1] =   5.000
```



Problemas com ponteiros

- Ponteiros são um dos aspectos mais fortes e mais perigosos do C,
- Ponteiros selvagens (não-inicializados) podem provocar a quebra do sistema,
- É muito fácil usar ponteiros incorretamente,
- Erros que envolvem ponteiros são muito difíceis de encontrar e corrigir.



Problemas com ponteiros

- Problemas comuns:
  - Erros de tipo,
  - Ponteiros não-inicializados,
  - Erros na manipulação de ponteiros,

Erros de tipo

- As variáveis ponteiros devem apontar para o tipo de dados correto,
- A declaração de um ponteiro indica ao compilador a quantidade de bytes que serão manipulados.

```
main(){
  float x=1.5, y;
  int *p;

  p = &x;
  y = *p;

  return 0;
}
```

- Valores de x e y?
- O programa compila?
- O programa executa?
- Resultados?

Usar ponteiros não-inicializados

```
main(){
   int x, *p;

   x = 10;
   *p = x;

   return 0;
}
```

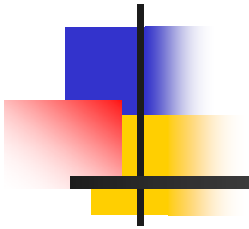
- O programa esta errado?

- Coloca o valor 10 numa posição de memória desconhecida.
- Pode passar desapercebido em programas pequenos,
- Pode gerar erros críticos em programas grandes,
- Tenha certeza que um ponteiro aponta para um endereço "válido" antes de usá-lo.

Erros de manipulação

```
main(){
   int x, *p;

   x = 10;
   p = x;
   printf("%d", *p);

   return 0;
}
```

- O programa esta errado?
- O ponteiro p assume o endereço 10 e imprime o valor armazenado nesse endereço.
- A instrução correta: `p = &x;`



Erros de manipulação

- Um erro comum é supor que duas variáveis diferentes serão colocadas em posições de memórias adjacentes.

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;

if (p1<p2) ...
```

- A posição de duas variáveis na memória é imprevisível.

- Fazer comparação entre ponteiros que não apontam para um objeto comum produz resultados inesperados.

Erros de manipulação

- Que faz o seguinte programa:

```
int main(){
  char *p1;
  char s[80];

  p1 = s;
  do{
    printf("Digite uma string: ");
    gets(s);

    while(*p1) printf("%c-", p1++);
    printf("\n");
  }while(strcmp(s, "fim"));

  system("PAUSE");
  return 0;
}
```

O programa esta correto?

Erros de manipulação

- Como corrigi-lo?

```
int main(){
  char *p1;
  char s[80];

  do{
    printf("Digite uma string: ");
    gets(s);
    p1 = s;
    while(*p1) printf("%c-", p1++);
    printf("\n");
  }while(strcmp(s, "fim"));

  system("PAUSE");
  return 0;
}
```