Linguagem C

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

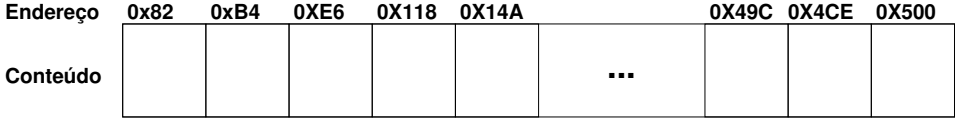
Licenciamento

Sumário

Vetores (arranjos)

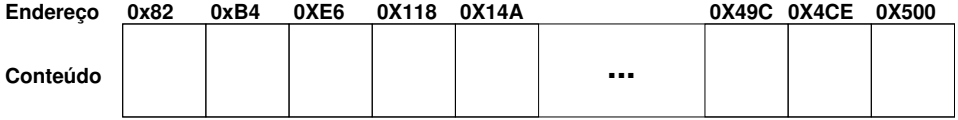
Variáveis

Qual a relação das variáveis com a memória do computador?



Variáveis

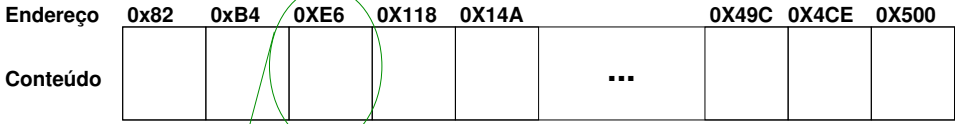
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */
```

Variáveis

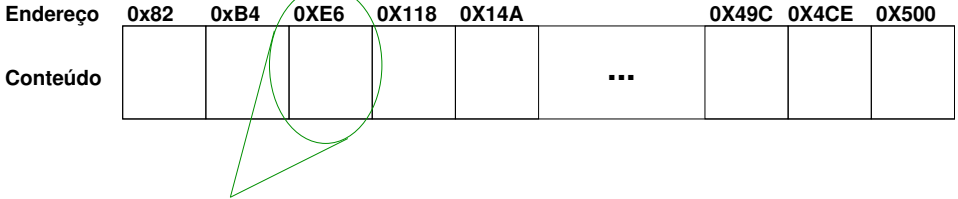
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */
```

Variáveis

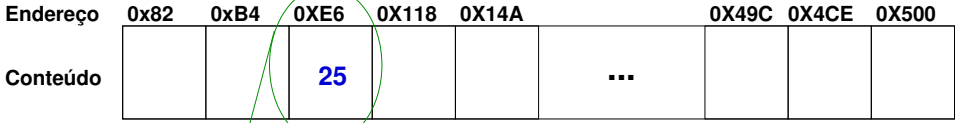
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */

dia1 = 25;         /* atribuiu um valor a variável */
```

Variáveis

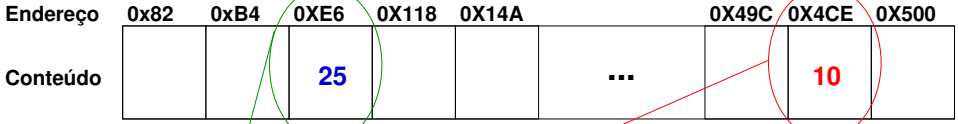
Qual a relação das variáveis com a memória do computador?




```
int dia1;          /* reserva um espaço na memória */

dia1 = 25;         /* atribuiu um valor a variável */
```

Variáveis

Qual a relação das variáveis com a memória do computador?

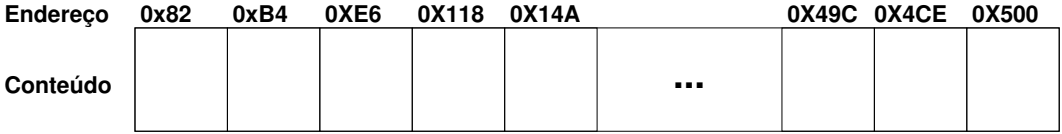



```
int dia1;          /* reserva um espaço na memória */
dia1 = 25;         /* atribuiu um valor a variável */
int dia2 = 10;
```

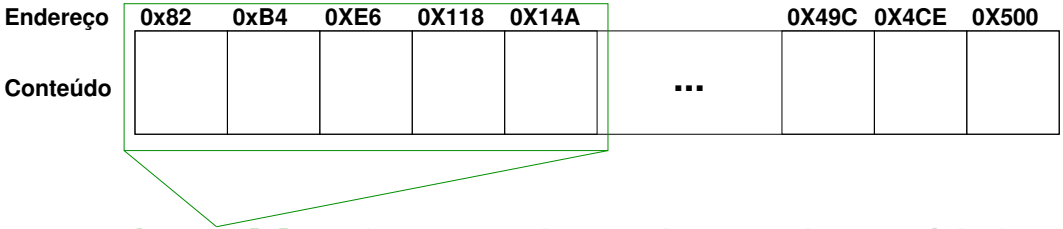
Vetores (arranjos)

- Coleção de **variáveis de um mesmo tipo** que é referenciado por um nome comum e de **tamanho fixo**
  - elementos guardados de forma contígua na memória a uma distância fixa um do outro
- Elementos de um vetor são acessados or meio de um índice, sendo 0 o índice para o primeiro elemento
  - Permite o acesso direto a qualquer posição do vetor
- Compilador não verifica se o índice informado está dentro dos limites de um vetor
  - Problemas irão aparecer somente durante a execução do programa

Vetores (arranjos)

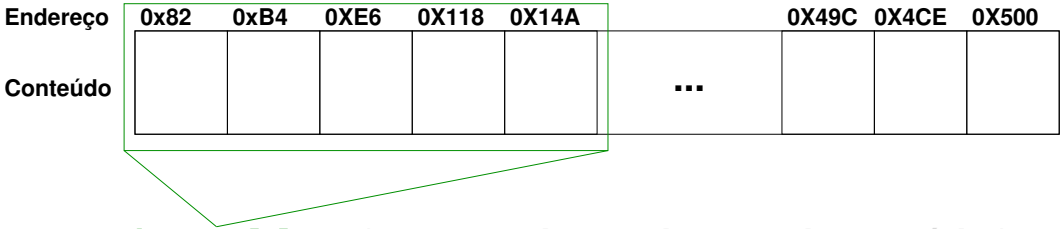


Vetores (arranjos)



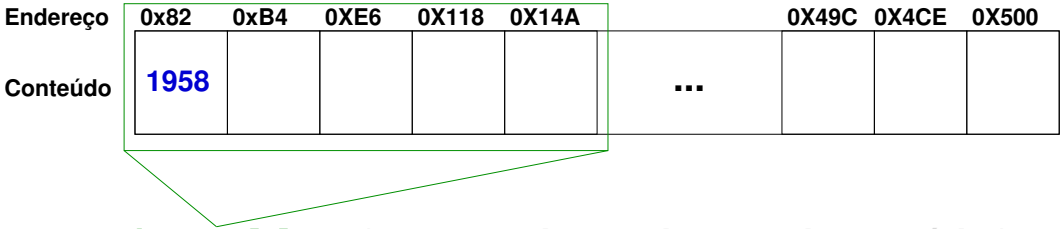

```
int ano[5];     /* reservando 5 endereços de memória */
```

Vetores (arranjos)



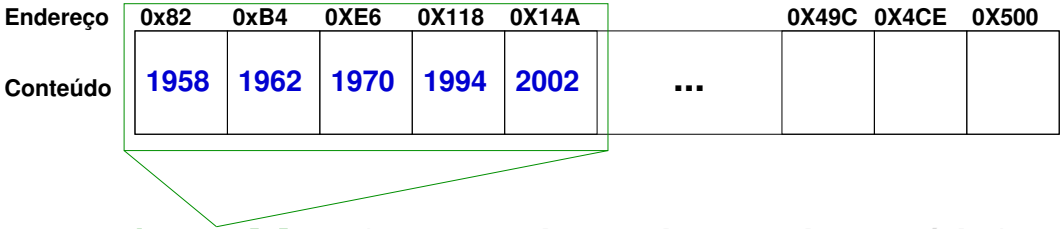

```
int ano[5];      /* reservando 5 endereços de memória */

ano[0] = 1958;
```

Vetores (arranjos)




```
int ano[5];      /* reservando 5 endereços de memória */

ano[0] = 1958;
```

Vetores (arranjos)




```
int ano[5];      /* reservando 5 endereços de memória */
```

```
ano[0] = 1958;
ano[1] = 1962;
ano[2] = 1970;
ano[3] = 1994;
ano[4] = 2002;
```

índice	valor
0	1958
1	1962
2	1970
3	1994
4	2002

Vetores (arranjos)

- Obtendo o tamanho em `bytes` ocupado por um vetor

```c
// iniciando vetor com valores: 2, 4, 8, 16 e 32
int ano[5] = {2, 4, 8, 16, 32};
int tamanho;
// ou tamanho = sizeof(int) * 5
tamanho = sizeof(ano);
```

- Em C não é feita a verificação dos limites de um vetor e cabe ao programador ficar atento a isto

```c
// posições de 0 a 4 recebem o valor 0
int ano[5] = {};

// O compilador não indicará erro
ano[6] = 2000;
```

Vetores com várias dimensões

vetores retangulares multidimensionais (matriz)

```c
// declarando vetor com duas dimensões
int matriz[3][3];

// declarando vetor com 3 dimensões
int matquad[2][3][4];

scanf("%d", &mat[1][1]);
printf("O valor armazenado em 1,1 é: %d", mat[1][1]);
```

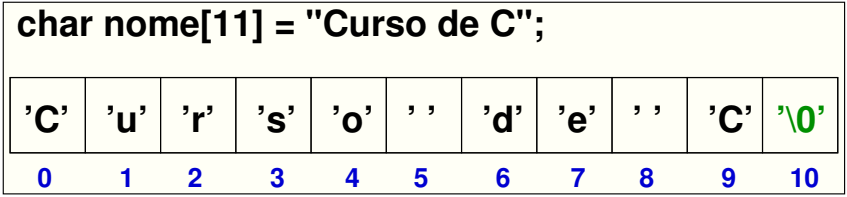
- É possível criar vetores não dimensionados com mais de uma dimensão, porém é necessário especificar todas as dimensões exceto a dimensão mais à esquerda

```c
// declarando um vetor nao dimensionado
// com base nos valores informados, terá 4 linhas e 2 colunas
int vet[][2] = {1,2,2,4,3,8,4,16};
// ou ainda int vet[][2] = {{1,2},{2,4},{3,8},{4,16}};
```

Vetores de caracteres

Vetores de caracteres

- Não existe um tipo de dados primitivo para armazenar cadeias de caracteres (*strings*)
- O caractere nulo (`\0`) é usado para indicar o fim da cadeia de caracteres
  - Não é preciso adicionar manualmente o `\0` no final de literais *strings*. O compilador já faz isso por você
- Para armazenar uma cadeia de caracteres com até 10 letras, será necessário declarar um vetor de caracteres de 11 posições



Vetores de caracteres

- Leitura com `scanf`, `fgets` ou `gets`?
  - A função `gets` foi removida do C11, pois não reserva espaço na memória, podendo gerar estouro de pilha (*buffer overflow*)

```c
char nome[20];

printf("Entre com seu nome: ");
// Outras opções: scanf("%s", nome) ou scanf("%[^\n]%*c", nome)
fgets(nome, sizeof(nome), stdin);

printf("Bom dia %s\n", nome);
```

- Se um vetor for declarado sem dimensão, o compilador criará um vetor grande o suficiente para armazenar a informação atribuída ao vetor

```c
// declarando um vetor dimensionado
char v1[12] = "Linguagem C";

// declarando um vetor nao dimensionado
char v2[] = "Facilidades da linguagem C";
```

Vetores de caracteres

- Cadeias de caracteres em C não são tipos de dados primitivos e assim **não é possível** realizar as operações que poderíamos fazer, por exemplo, com inteiros
  - Operações aritméticas: `a + b` ou `(a == b)`
  - Operações relacionais: `a > b`
  - Operações lógicas: `(a && b)`
- A biblioteca padrão **string.h** provê as funções necessárias para atender essas necessidades

Biblioteca string.h

https://cplusplus.com/reference/cstring/

Função	Descrição
`char *strcpy(destino,origem)`	Copia o conteúdo da origem para o destino
`char *strncpy(destino,origem,n)`	Copia **N** caracteres da origem para o destino
`char *strcat(destino,origem)`	Concatena o conteúdo da origem ao final do destino
`char *strncat(destino,origem,n)`	Concatena **N** caracteres da origem ao final do destino
`int strcmp(a,b)`	Retorna 0 se $a == b$, negativo se $a < b$, e positivo se $a > b$
`size_t strlen(cadeia)`	Retorna o tamanho da cadeia de caracteres
`char * strtok(cadeia, delimitador)`	Separa a cadeia de acordo com o delimitador
`char * strstr(cadeia, palavra)`	Retorna ponteiro para 1ª palavra na cadeia

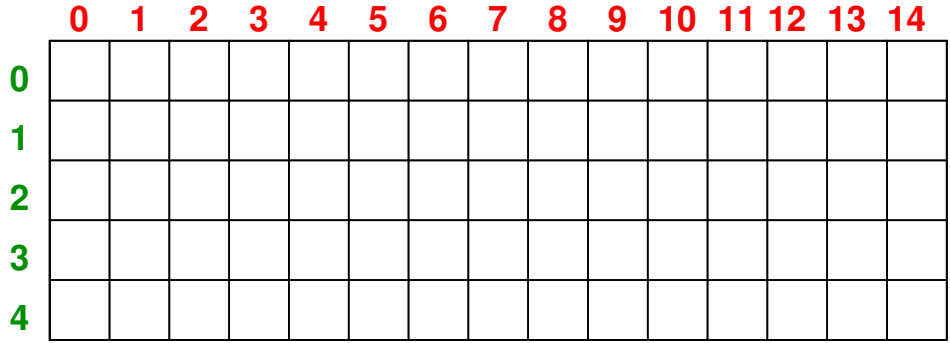
Vetores multidimensionais de caracteres

- No código abaixo, um vetor que pode armazenar 5 cadeias de caracteres, sendo que cada uma poderá ter no máximo 14 caracteres
  - Lembre-se do caractere '\0' delimita cadeia de caracteres

```
1 char alunos[5][15];
2
3 /* Fazendo a leitura do nome do 1o. aluno */
4 fgets(alunos[0], sizeof(alunos[0]), stdin);
```

Vetores multidimensionais de caracteres

```
char alunos[ 5 ][ 15];
```



Vetores multidimensionais de caracteres

```
char alunos[ 5 ][ 15];
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'									
1	'J'	'u'	'c'	'a'	'\0'										
2	'M'	'a'	'r'	'i'	'a'	'\0'									
3	'A'	'l'	'i'	'c'	'e'	'\0'									
4															

Vetores multidimensionais de caracteres

char alunos[5][15]; L = LIXO

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'	L	L	L	L	L	L	L	L	L
1	'J'	'u'	'c'	'a'	'\0'	L	L	L	L	L	L	L	L	L	L
2	'M'	'a'	'r'	'i'	'a'	'\0'	L	L	L	L	L	L	L	L	L
3	'A'	'l'	'i'	'c'	'e'	'\0'	L	L	L	L	L	L	L	L	L
4	L	L	L	L	L	L	L	L	L	L	L	L	L	L	L

Caracteres largos (*wide char*)

- Tipo primitivo `char` ocupa `1 byte` e permite representar caracteres ASCII[1]
- UTF-8[2] é de codificação binária de tamanho variável (1 a 4 bytes) que permite representar qualquer caractere universal
  - `1 byte` – para representar os 128 caracteres ASCII
  - `2 bytes` – para representar caracteres latinos com diacríticos (i.e. acentos)
  - `3 bytes` – para alfabetos Grego, Cirílico, Armênio, Hebraico, Sírio e Thaana
  - `4 bytes` – para representar outros caracteres

UTF-8 é o formato de codificação padrão da maioria dos sistemas operacionais e também de todos os protocolos da Internet padronizados pela IETF

[1] https://www.asciitable.com
[2] https://pt.wikipedia.org/wiki/UTF-8

Caracteres largos (*wide char*)

Padrão C90

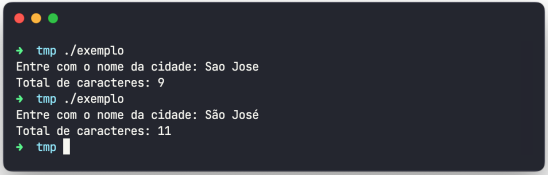
Introduziu a biblioteca `wchar.h`, juntamente com o tipo `wchar_t` e algumas funções para lidar com caracteres UTF-8

- Funções da biblioteca `string.h` que dependem de caracteres terminadores em ASCII (\n, \0, \t, ...) funcionarão com UTF-8
  - `strcpy`, `strcmp`, `strstr` e `fgets`
- `strlen` e `strncpy` ficam em função do número de *bytes* da *string* e não em função do número de caracteres
- É possível guardar uma cadeia de caracteres em UTF-8 em uma variável do tipo `char`, porém garanta que o tamanho do vetor seja suficiente
  - São necessários 2 bytes para caracteres acentuados em português

Caracteres largos (*wide char*)

Caracteres latinos com diacríticos

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

int main(){
    char nome[80];

    printf("Entre com o nome da cidade: ");
    fgets(nome, sizeof(nome), stdin);
    printf("Total de caracteres: %ld\n", strlen(nome));
    return 0;
}
```



- Em ASCII conta 8 caracteres
- Em UTF-8 conta 10 bytes

Biblioteca wchar.h

https://cplusplus.com/reference/cwchar

```
1  // Equivalente à função fgets
2  wchar_t* fgetws (wchar_t* ws, int num, FILE* stream);
3
4  // Equivalente à função scanf
5  int wscanf (const wchar_t* format, ...);
6
7  // Equivalente à função printf
8  int wprintf (const wchar_t* format, ...);
9
10 // Equivalente à função strlen
11 size_t wcslen (const wchar_t* wcs);
12
13 // Equivalente à função strcpy
14 wchar_t* wcscpy (wchar_t* destination, const wchar_t* source);
15
16 // Equivalente à função strcmp
17 int wcscmp (const wchar_t* wcs1, const wchar_t* wcs2);
18
19 // Equivalente à função strcat
20 wchar_t* wcscat (wchar_t* destination, const wchar_t* source);
```

Caracteres largos

Bibliotecas `wchar.h` e `locale.h`

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h> // necessário para usar setlocale

int main(){
    // Digite locale no terminal para ver os parâmetros de localidade em uso
    // Digite locale -a no terminal para ver as localidades disponíveis
    // LC_CTYPE é a categoria de localidade para caracteres usada pelas funções printf e
     scanf

    // LC_ALL é uma macro que define todas as categorias de localidade
    // Para usar o padrão do sistema operacional, use "" como segundo parâmetro
    // definindo a localidade para pt_BR.UTF-8
    setlocale(LC_ALL,"pt_BR.UTF-8");

    wchar_t nome[80];
    wprintf(L"Entre com o nome da cidade: ");
    fgetws(nome, sizeof(nome), stdin );
    wprintf(L"Total de caracteres: %u\n", wcslen(nome));
    return 0;
}
```

Caracteres largos

Bibliotecas `wchar.h` e `locale.h`

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main(){
    setlocale(LC_ALL,"pt_BR.UTF-8");

    wchar_t caractere;
    wchar_t *frase;

    caractere = L'á';
    frase = L"áéíóú"; // é terminada pelo caractere largo L'\0' e não apenas '\0'

    wprintf(L"Caractere: %lc\n", caractere);
    wprintf(L"Frase: %ls\n", frase);
    wprintf(L"Variável caractere tem %lu bytes\n", sizeof(caractere));
    wprintf(L"Frase tem %u caracteres\n", wcslen(frase));

    return 0;
}
```

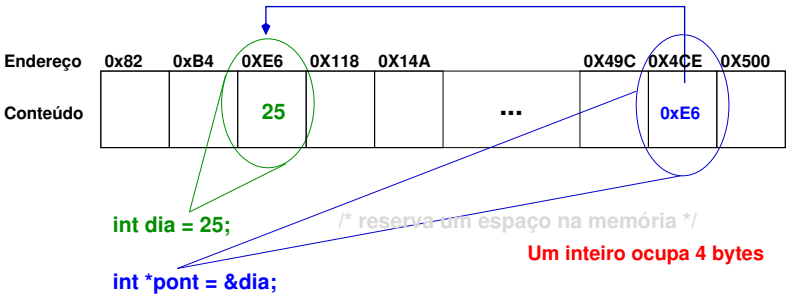
Ponteiros

Ponteiros

- Variável que armazena o endereço de memória de outra variável ou funções
- Permite acesso de baixo nível a memória
- Usados para fazer alocação dinâmica de recursos
- Como parâmetros de funções (passagem por referência)
- Pode tornar a escrita de códigos mais concisa

```
1 // tipo *ponteiro; // operador * para referenciar
2 int *ponteiro;
3 int dia = 25;
4
5 // operador & para obter o endereço de memória da variável
6 ponteiro = &dia;
```

Ponteiros



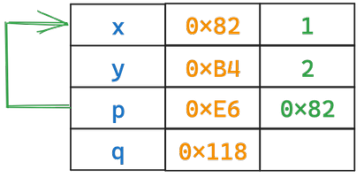
Um ponteiro para `long` ocupa 4 bytes em sistemas de 32-bits e 8 bytes em sistemas de 64-bits

Ponteiros

x	0×82	1
y	0×B4	2
p	0×E6	
q	0×118	

```
int x = 1, y = 2;
int *p, *q;
```

Ponteiros



x	0×82	1
y	0×B4	2
p	0×E6	0×82
q	0×118	

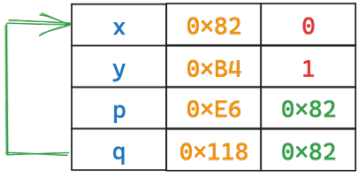
```
int x = 1, y = 2;
int *p, *q;

p = &x;
```

Ponteiros

x	0×82	0
y	0×B4	1
p	0×E6	0×82
q	0×118	

```
int x = 1, y = 2;
int *p, *q;

p = &x;

y = *p;
*p = 0;
```

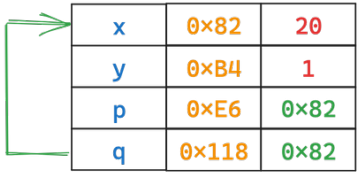
Ponteiros



x	0×82	0
y	0×B4	1
p	0×E6	0×82
q	0×118	0×82

```
int x = 1, y = 2;
int *p, *q;

p = &x;

y = *p;
*p = 0;

q = p;
```

Ponteiros

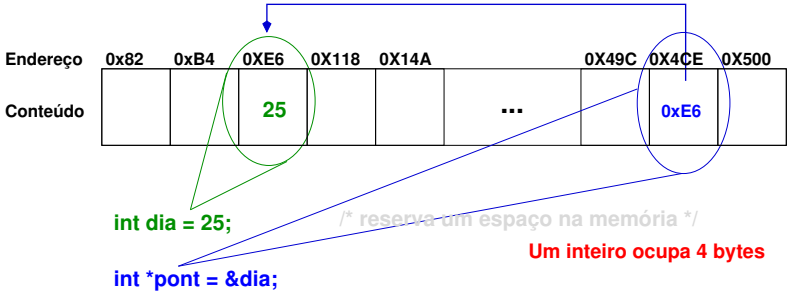


x	0×82	20
y	0×B4	1
p	0×E6	0×82
q	0×118	0×82

```
int x = 1, y = 2;
int *p, *q;

p = &x;

y = *p;
*p = 0;

q = p;
*q = 20;
```

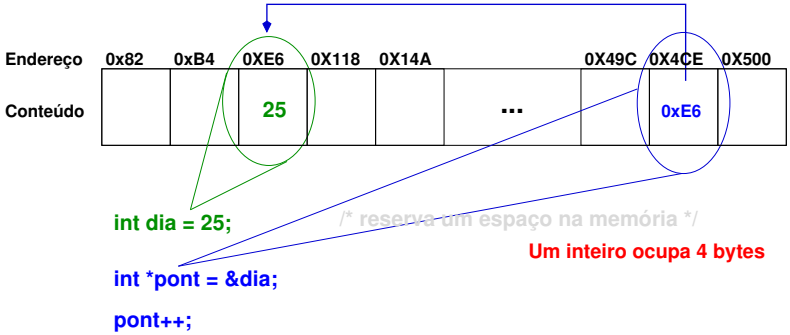
Aritmética de ponteiros

- Operações aritméticas (soma, subtração, comparação) com ponteiros
- Tem diversas utilidades, sendo a navegação em arranjos uma delas



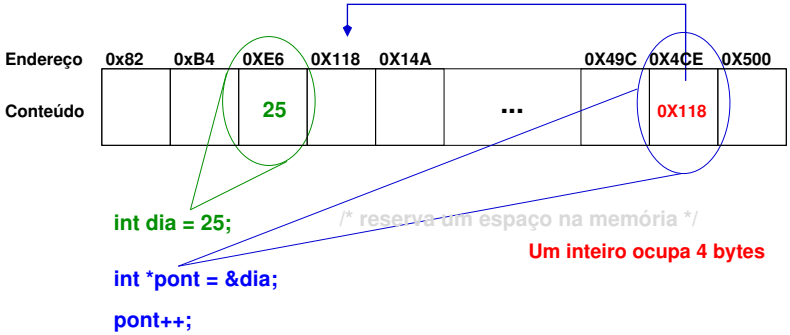
Aritmética de ponteiros

- Operações aritméticas (soma, subtração, comparação) com ponteiros
- Tem diversas utilidades, sendo a navegação em arranjos uma delas



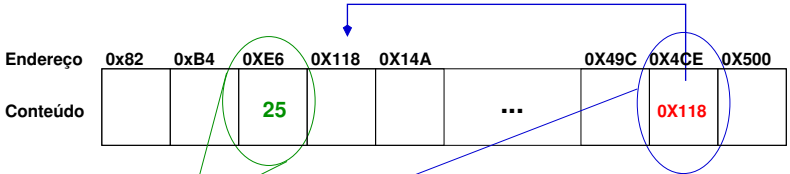
Aritmética de ponteiros

- Operações aritméticas (soma, subtração, comparação) com ponteiros
- Tem diversas utilidades, sendo a navegação em arranjos uma delas



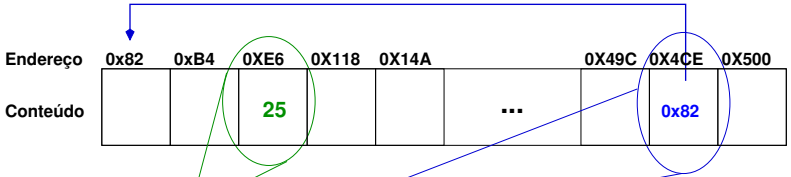
Aritmética de ponteiros

- Operações aritméticas (soma, subtração, comparação) com ponteiros
- Tem diversas utilidades, sendo a navegação em arranjos uma delas



Aritmética de ponteiros

- Operações aritméticas (soma, subtração, comparação) com ponteiros
- Tem diversas utilidades, sendo a navegação em arranjos uma delas



Aritmética de ponteiros

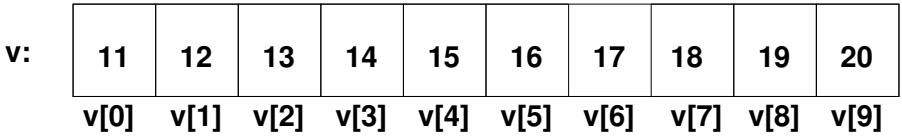
- **Exercício:** após a execução de cada linha, imprimia o valor de i, de j, o endereço para onde **p** aponta e o valor contido nesse endereço. Exemplo na última linha do código.

```
1  int i = 10, j = 0, *p;
2
3  p = &i;
4  *p = *p + 1;
5  j = *p + 2;
6  *p += 1;
7  (*p)++;
8  *p++;
9  p++;
10
11 printf("i: %d\tj: %d\tp: %p\t*p: %d\t*p\n", i, j, p, *p);
```

Ponteiros e vetores

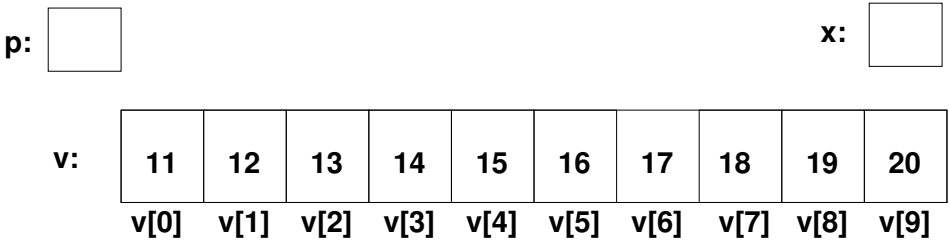
- Ponteiros e vetores possuem uma estreita relação
- Qualquer operação que possa ser obtida através da **indexação de vetores** também poderá ser feita com **ponteiros**
- A versão usando ponteiro geralmente é mais rápida que a versão usando indexação de vetores multidimensionais

Ponteiros e vetores



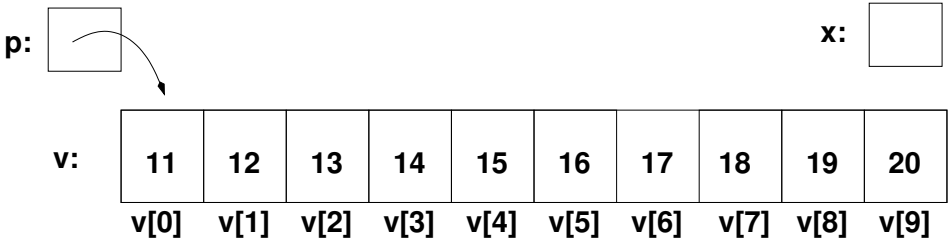

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
```

Ponteiros e vetores



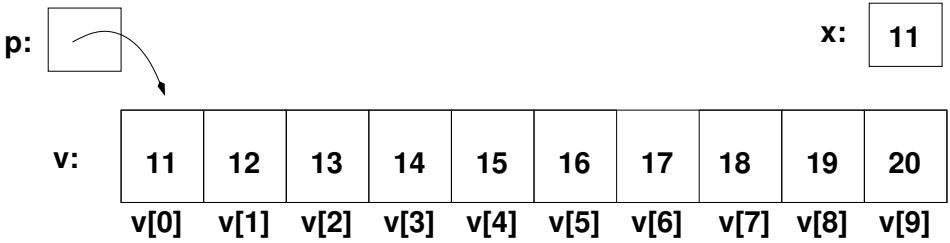

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
```

Ponteiros e vetores



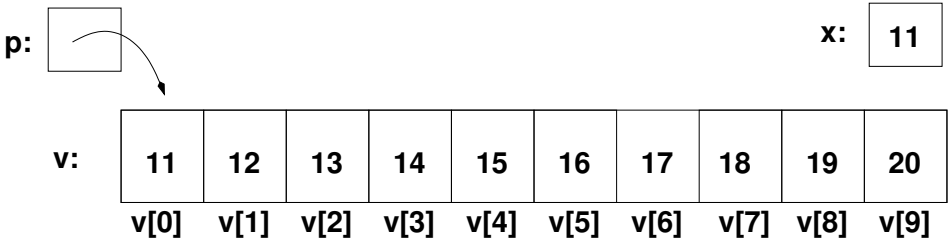

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
```

Ponteiros e vetores



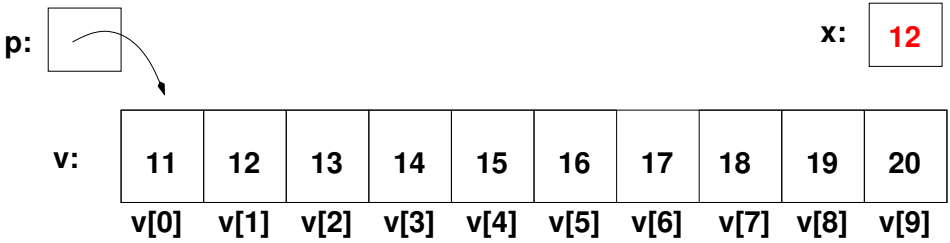

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
```

Ponteiros e vetores



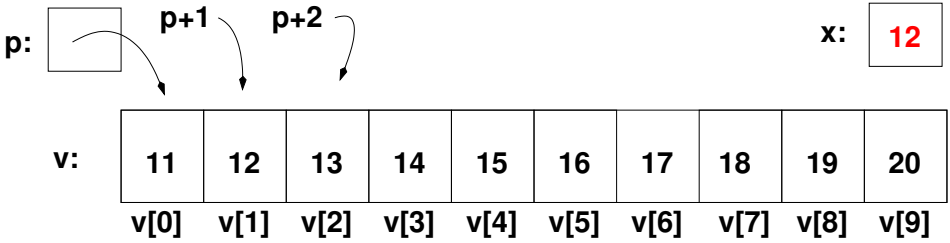

```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores




```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores




```
int v[10] = {11,12,13,14,15,16,17,18,19,20};
int *p, x;
p = &a[0];      // p = a também é permitido
x = *p;
x = *(p+1);
```

Ponteiros e vetores

```c
#include <stdio.h>

void imprime(int *v, size_t v_size) {
  for (int i = 0; i < v_size; ++i) {
    // deslocando usando ponteiros
    printf("%d: %d\n", i, *(v + i));
  }
}

int main() {
  int ano[5] = {1958, 1962, 1970, 1994, 2002};
  int meses[3] = {1, 3, 5};

  // usando indices de vetores
  for (int i = 0; i < sizeof(ano) / sizeof(ano[0]); i++) {
    printf("%d: %d\n", i, ano[i]);
  }

  imprime(meses, sizeof(meses) / sizeof(meses[0]));
  return 0;
}
```

Ponteiros e vetores

- 9 deslocamentos (for aninhado) *vs* 9 incrementos

```
1  int i, j, vet[3][3], vet2[3][3], *p;
2
3  // usando indices de vetores
4  for(i = 0; i < 3; i++){
5    for(j = 0; j < 3; j++){
6      vet[i][j] = 0;
7    }
8  }
9
10 // usando ponteiros
11 p = &vet2[0][0];
12
13 for(i = 0; i < 9; i++){
14   *(p+i) = 0;
15 }
```

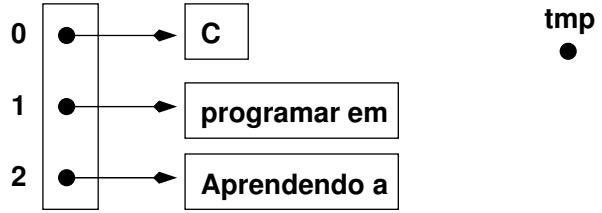
Vetores de ponteiros

- Ponteiros também são variáveis, logo é possível definir vetores de ponteiros

```c
char *frases[3];

frases[0] = "Aprendendo a";
frases[1] = "programar em";
frases[2] = "C";

printf("%s %s %s\n", frases[0], frases[1], frases[2]);
```

Vetores de ponteiros

Exemplo: Ordenando um vetor

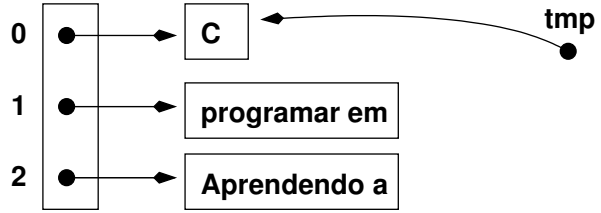



```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
```

Vetores de ponteiros

Exemplo: Ordenando um vetor

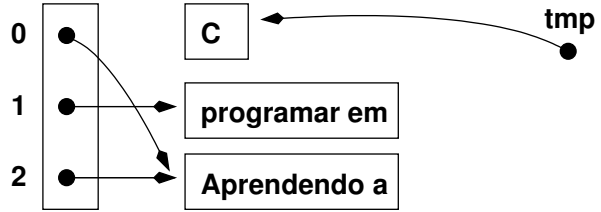



```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
```

Vetores de ponteiros

Exemplo: Ordenando um vetor

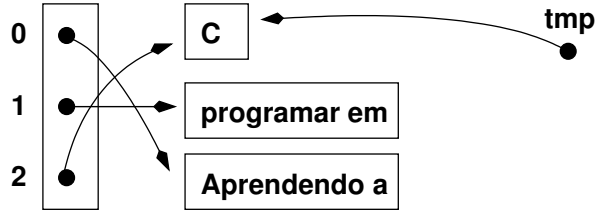



```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
```

Vetores de ponteiros

Exemplo: Ordenando um vetor




```
char *frases[3];
frases[0] = "C";
frases[1] = "programar em";
frases[2] = "Aprendendo a";
```

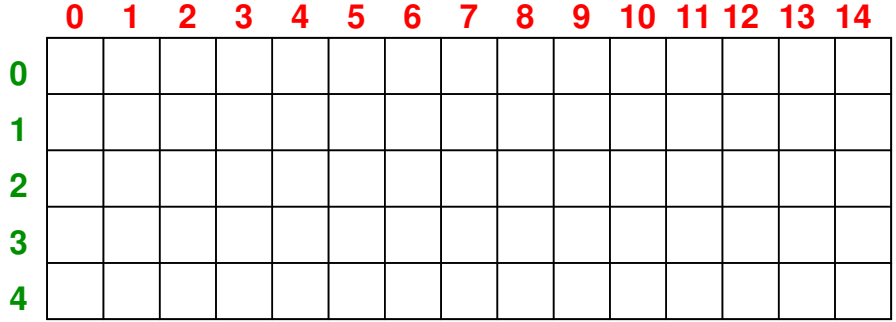
```
char *tmp;
tmp = frases[0];
frases[0] = frases[2];
frases[2] = tmp;
```

Ponteiros *vs* vetores multidimensionais

- A definição **char a[5][15]** reserva na memória 75 posições para caracteres
  - É possível encontrar o elemento `a[lin][col]` fazendo $15 \times lin + col$
- A definição **char *b[5]** reserva na memória 5 posições para ponteiros
- Um vetor de ponteiros permite que cada elemento de **b** aponte para um vetor de caracteres de diferente tamanho

Ponteiros *vs* vetores multidimensionais

```
char alunos[  5 ][ 15] = {  "Pedro", "Juca", "Maria", "Alice"};
```



Ponteiros *vs* vetores multidimensionais

```
char alunos[  5 ][ 15] = {  "Pedro", "Juca", "Maria", "Alice"};
```

L = LIXO

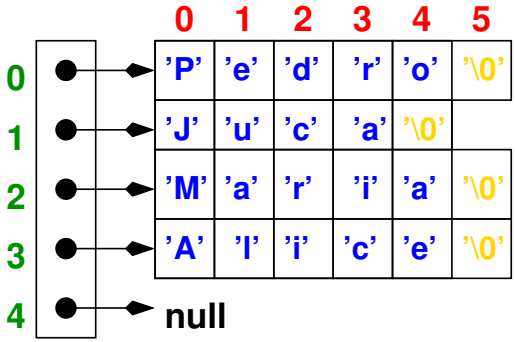
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	'P'	'e'	'd'	'r'	'o'	'\0'	L	L	L	L	L	L	L	L	L
1	'J'	'u'	'c'	'a'	'\0'	L	L	L	L	L	L	L	L	L	L
2	'M'	'a'	'r'	'i'	'a'	'\0'	L	L	L	L	L	L	L	L	L
3	'A'	'l'	'i'	'c'	'e'	'\0'	L	L	L	L	L	L	L	L	L
4	'\0'	L	L	L	L	L	L	L	L	L	L	L	L	L	L

Ponteiros *vs* vetores multidimensionais

```
char *nomes[ 5 ] = { "Pedro", "Juca", "Maria", "Alice"};
```

Ponteiros *vs* vetores multidimensionais

```
char *nomes[ 5 ] = { "Pedro", "Juca", "Maria", "Alice"};
```



Diferenças entre `char s[]` e `char *p`

- `char s[] = "prog2"` cria um vetor de caracteres de tamanho 6
  - Pode realizar qualquer operação que seja permitida com vetores
  - `s[0] = 'P'` é permitido
- `char *p = "prog2"` cria uma cadeia de caracteres literal que é armazenada na memória de somente leitura
  - Qualquer tentativa de atualização irá resultar em comportamento inesperado
  - `p[0] = 'P'` resultará em *segmentation fault* no gcc

Struct, Union e Enum

Struct

Tipo de dados definido pelo usuário que permite agrupar variáveis de diferentes tipos

- Definição tipos de dados compostos personalizados
- Permite uma função retornar múltiplos valores
- Para criação de estruturas de dados como listas, árvores

Struct

Criar uma estrutura para representar um ponto no plano cartesiano

```c
struct ponto{
    int x;
    int y;
};

int main(){
    struct ponto inicial;
    //struct ponto final = {10, 5}; modo antes do padrão C99
    struct ponto final = {.x = 10, .y = 5}; // estilo padrão C99
    struct ponto *qualquer;

    // operador . para acessar membro
    inicial.x = 0;
    inicial.y = 4;

    // operador -> para acessar membro quando a referência é um ponteiro
    qualquer->x = 20;
    qualquer->y = 40;

    return 0;
}
```

Struct

Criar uma estrutura para representar um retângulo

```c
struct retangulo{
    struct ponto p1;
    struct ponto p2;
};

int main(){
    struct retangulo r;
    struct retangulo ret2 = {{2,1}, {3, 4}};
    struct retangulo ret = {.p1.x = 5, .p1.y = 3, .p2.x = 5, .p2.y = 3}; // estilo C99
    struct retangulo *rp;

    r.p1.x = 0; r.p1.y = 4;

    r.p2.x = 10; r.p2.y = 5;

    // operador -> para acessar membro
    rp->p1.x = 10; rp->p1.y = 20;

    return 0;
}
```

typedef – atribuição de nome para tipo de dados

```c
typedef struct pessoa{
    char nome[20];
    char cpf[11];
    double salario;
    struct pessoa* dependente;
} pessoa_t; // _t é uma convenção de escrita de código em C

int main(){
    pessoa_t pai = {"Joao", "123", 1000, NULL};
    pessoa_t filha = {"Ana", "456", 0, NULL};

    pai.dependente = &filha;

    printf("%s é pai de %s\n", pai.nome, pai.dependente->nome);

    return 0;
}
```

União

- Estrutura que permite **agrupar variáveis de diferentes tipos**, porém diferentemene das `structs`, **somente um membro por vez pode armazenar dados**
- O tamanho ocupado por uma união na memória é igual ao tamanho do maior de seus membros
  - Útil para estruturas de dados que queiram usar a mesma memória reservada para ser usada por um de seus membros
  - O tamanho de uma `struct` é igual ou maior ao somatório do tamanho de todos os seus membros

```
1 union numeros{
2     int inteiro;
3     double real;
4 };
```

União

```c
#include <stdio.h>
#include <stdint.h>
struct s_numero{
    int16_t i;
    double r;
};
union u_numero{
    int16_t i;
    double r;
};

int main(){
    union u_numero ex_u;
    struct s_numero ex_s;

    ex_s.i = 10; ex_u.i = 10;
    printf("Union -> tamanho: %lu\td: %.2lf\ti: %d\n", sizeof (ex_u), ex_u.r, ex_u.i);
    printf("Struct-> tamanho: %lu\td: %.2lf\ti: %d\n", sizeof (ex_s), ex_s.r, ex_s.i);
    ex_u.r = 5.2; ex_s.r = 5.2;
    printf("Union -> tamanho: %lu\td: %.2lf\ti: %d\n", sizeof (ex_u), ex_u.r, ex_u.i);
    printf("Struct-> tamanho: %lu\td: %.2lf\ti: %d\n", sizeof (ex_s), ex_s.r, ex_s.i);
    return 0;
}
```

Enumerações (`Enum`)

- Tipo de dados definido pelo usuário que define a lista de valores permitidos para serem armazenados
  - Exemplo: estações do ano, dias da semana, meses do ano, planetas do sistema solar
- O uso de `enum` torna o código mais claro e fácil de manter
  - Seria equivalente as macros (`#define`), porém o código fica mais claro e seguro

Enumerações (Enum)

```c
#include <stdio.h>

typedef enum dias_da_semana {SEG, TER, QUA, QUI, SEX, SAB, DOM} dias_da_semana_t;

int main(){

    dias_da_semana_t aula = SEG;

    switch (aula) {
        // outra forma-> case SAB ... DOM:
        case SAB:
        case DOM:
            printf("Final de semana");
            break;
        default:
            printf("Dia de semana");
            break;
    }
}
```

Enumerações (Enum)

```c
#include <stdio.h>

typedef enum dias_da_semana {SEG, TER, QUA, QUI, SEX, SAB, DOM} dias_da_semana_t;

typedef struct {
    char nome[20];
    char cpf[11];
} pessoa_t;
typedef struct {
    char sigla[5];
    pessoa_t *professor;
    dias_da_semana_t dias_com_aula[2];
} disciplina_t;

int main(){
    pessoa_t prof = {"Emerson", "123"}; disciplina prg2 = {"PRG2", &prof, {SEG, TER}};
    printf("Disciplina:\t%s\nProfessor:\t%s\n", prog2.sigla, prog2.professor->nome);
    printf("Dias com aula:\n");
    for (int i = 0; i < 2; ++i) {
        printf("%d\n", prog2.dias_com_aula[i]);
    }
    return 0;
}
```

Alocação dinâmica de memória

Alocação de memória

Linguagem C trabalha com alocação estática, automática e dinâmica de memória

- **Alocação estática**
  - Variáveis são alocadas na memória principal e persistem até o término do programa
  - O tamanho alocado na memória é constante em **tempo de compilação**
- **Alocação automática**
  - São alocadas no empilhamento das chamadas de função (chamada e retorno)
  - O tamanho alocado na memória é constante em **tempo de compilação**
- **Alocação dinâmica**
  - Necessário para cenários onde o tamanho da memória a ser alocado só é conhecido durante o **tempo de execução** ou onde seria enviável reservar estaticamente um grande tamanho em memória

Alocação dinâmica de memória

Biblioteca `stdlib.h`

- `malloc` (*memory allocation*), aloca um bloco único de memória com um tamanho específico
- `calloc` (*contiguos allocation*), semelhante ao `malloc`, porém inicializa as posições com 0
- `realloc` (*re-allocation*), para alterar o tamanho da memória que foi alocada anteriormente
- `free`, para liberar uma memória que foi alocada anteriormente

```
1 // int tem tamanho de 4 bytes, assim serão alocados 400 bytes na memória
2 int *p = malloc(100 * sizeof (int));
3 int *q = (int*) calloc(100, sizeof (int)); // typecasting em C é opcional
4 // 100 era pouco, aumentando para 200
5 p = realloc(p, 200 * sizeof (int));
6 free(p);
```

Alocação dinâmica de memória

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int *vetor, tamanho = 2;
    // aloca memória para um vetor de inteiros
    vetor = (int *) calloc(tamanho, sizeof (int));

    if (vetor == NULL){
        printf("Não foi possível alocar memória\n");
        exit(EXIT_FAILURE);
    }
    vetor[0] = 10; vetor[1] = 20;
    tamanho = 4;

    vetor = realloc(vetor, tamanho * sizeof (int));
    vetor[2] = 30; vetor[3] = 40;

    for (int i = 0; i < tamanho; ++i) {
        printf("%d\n", *(vetor + i)); // ou vetor[i]
    }
    free(vetor);
    return 0;
}
```

Matriz com alocação dinâmica de memória

```c
int ler_numero(char *mensagem){
    char *ptr, cadeia[100];
    printf("%s", mensagem);
    fgets(cadeia, 100, stdin);
    long num = strtol(cadeia, &ptr, 10);
    if ((ptr == cadeia) || (*ptr != '\n' && *ptr != '\0')) {
        printf("A string contém caracteres inválidos.\n");
        exit(1);
    }
    return ((int) num);
}

int main(int argc, char *argv[]){
    char msg[20];

    int linha =  ler_numero("Entre com o número de linhas : ");
    int coluna = ler_numero("Entre com o número de colunas: ");

    /* aloca memória para uma matriz de inteiros */
    int **mat = (int**) calloc(linha, sizeof (int*));
    for (int i = 0; i < linha; ++i) {
        mat[i] = (int*) calloc(coluna, sizeof (int*));
    }

    /* inicializa a matriz */
    for (int i = 0; i < linha; ++i) {
        for (int j = 0; j < coluna; ++j) {
            sprintf(msg, "[%d][%d]: ", i, j);
            mat[i][j] = ler_numero(msg);
        }
    }
    return 0;
}
```

Manipulação de arquivos

Arquivos e fluxos

- C provê um conjunto padrão de interfaces de entrada e saída independente do dispositivo real que é acessado
  - terminais, arquivos em disco, fita
- Abstração entre o programador e o dispositivo utilizado
  - Essa abstração é chamada de **fluxos** (*streams*) e o dispositivo real é chamado de **arquivo**
- Tipos de fluxos

  Texto. Sequência de caracteres

  Binário. Sequência de *bytes*

Etapas para manipular arquivos regulares

1. Abrir o arquivo
2. Ler ou escrever no arquivo
3. Fechar o arquivo

```c
FILE *arquivo;

// abre arquivo texto somente para leitura
arquivo = fopen("agenda.txt", "r");

if (arquivo == NULL){
    printf("Não foi possível abrir o arquivo\n");
    exit(EXIT_FAILURE);
}

// código para ler linhas do arquivo
// ...
// ...

// fecha arquivo
fclose(arquivo);
```

Abrindo um arquivo

```
FILE *fopen(char *nome_arquivo, char *modo_de_acesso)
```

Fluxo		Descrição	Se o arquivo	
Texto	**Binário**		**Existir**	**Não existir**
`r`	`rb`	Abre para leitura	Abre	Erro
`w`	`wb`	Abre para escrita no início do arquivo	Sobreescreve	Cria
`a`	`ab`	Abre para escrita no final do arquivo	Abre	Cria
`r+`	`rb+`	Abre para leitura e escrita	Abre	Erro
`w+`	`wb+`	Cria um arquivo para leitura e escrita	Sobreescreve	Cria
`a+`	`ab+`	Abre para escrita no final do arquivo e leitura	Abre	Cria

Funções para leitura e escrita

Funções para escrita

Função	Descrição
`int fputc(int, FILE *)`	Escreve um caractere no arquivo
`int fputs(char *, FILE *)`	Escreve uma cadeia de caractere no arquivo
`int fprintf(FILE *, char *, ...)`	Semelhante ao printf
`int fwrite(void *, int, int, FILE *)`	Escreve bytes em arquivo binário

Funções para leitura

Função	Descrição
`int fgetc(FILE *)`	Lê um caractere do arquivo
`int fgets(char *, int, FILE *)`	Lê uma cadeia de caracteres de tamanho fixo
`int fscanf(FILE *, char *, ...)`	Semelhante ao scanf
`int fread(void *, size_t, size_t, FILE *)`	Lê uma quantidade específica de bytes de arquivo binário

Outras funções para trabalhar com arquivos

Função	Descrição
`FILE *fopen(char *, char *)`	Abre um arquivo
`int fclose(FILE *)`	Fecha um arquivo
`int fseek(FILE *, long, int)`	Posiciona o arquivo em um byte específico
`long ftell(FILE *)`	Retorna a posição atual do cursor no arquivo
`int feof(FILE *)`	Retorna verdadeiro se o fim do arquivo for atingido
`int ferror(FILE *)`	Retorna verdadeiro se ocorreu algum erro
`int remove(char *)`	Exclui um arquivo
`int fflush(FILE *)`	Descarrega dados do *buffer* no arquivo

Arquivo texto

Escrita

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

int main(){
    char frase[80];
    FILE *arq;

    if ((arq = fopen("teste.txt", "w")) != NULL){
        do{
            printf("Entre com a frase: ");
            fgets(frase, sizeof(frase), stdin);
            fputs(frase, arq);
        }while(*frase != '\n');
        fclose(arq);
    }else {
        fprintf(stderr, "Erro: arquivo nao pode ser aberto\n");
        exit(EXIT_FAILURE);
    }
    fclose(arq);
}
```

Arquivo texto

Leitura

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

int main(void){
   char frase[80];
   FILE *arq;

   if ((arq = fopen("teste.txt", "r")) != NULL){
      while(!feof(arq)){
         fgets(frase, sizeof(frase), arq);
         printf("%s",frase);
      }
   }else {
      fprintf(stderr, "Erro: arquivo nao pode ser aberto\n");
      exit(EXIT_FAILURE);
   }
   fclose(arq);
   return 0;
}
```

Arquivo binário

Escrita

```c
typedef struct pessoa{
    char nome[80];
    int idade;
} pessoa;

int main(){
    FILE *arq;

    pessoa p = {"Juca", 20};

    if ((arq = fopen("contatos.dat", "wb+")) != NULL){
        fwrite(&p, sizeof(p), 1, arq);
    }else {
        fprintf(stderr,"Erro: arquivo nao pode ser aberto\n");
        exit(EXIT_FAILURE);
    }
    fclose(arq);
    return 0;
}
```

Arquivo binário

Leitura

```
1  pessoa p;
2  FILE *arq;
3
4  if ((arq = fopen("contatos.dat", "rb")) != NULL) {
5      while (!feof(arq)) {
6          fread(&p, sizeof(p), 1, arq);
7          if (!feof(arq)) {
8              printf("nome: %s\t idade: %d\n", p.nome, p.idade);
9          }
10     }
11 } else {
12     fprintf(stderr, "Erro: arquivo nao pode ser aberto\n");
13     exit(EXIT_FAILURE);
14 }
15 fclose(arq);
16 return 0;
```

Leitura com `fseek`

```
int fseek(FILE *arquivo, long int offset, int posição)
```

- **offset** – deslocamento, em bytes ou caracteres, da posição do ponteiro
- **posição** – define a posição de onde o cursor irá partir

Constantes para posição	Descrição
`SEEK_SET`	Início do arquivo
`SEEK_CUR`	Posição atual do cursor no arquivo
`SEEK_END`	Fim do arquivo

Leitura com `fseek`

`int fseek(FILE *arquivo, long int offset, int posição)`

```
1  pessoa p;
2  FILE *arq;
3  if ((arq = fopen("contatos.dat", "rb")) != NULL) {
4      printf("Posição do ponteiro do arquivo: %ld\n", ftell(arq));
5      // movendo para o final do arquivo
6      fseek(arq, 0, SEEK_END);
7      // movendo para o início do arquivo
8      fseek(arq, 0, SEEK_SET);
9      // Movendo para o 4o. registro de pessoa no arquivo
10     fseek(arq, 3*sizeof (p), SEEK_SET);
11     fread(&p, sizeof(p), 1, arq);
12     printf("nome: %s\t idade: %d\n", p.nome, p.idade);
13 }else {
14     fprintf(stderr, "Erro: arquivo nao pode ser aberto\n");
15     exit(EXIT_FAILURE);
16 }
17 fclose(arq);
```

Ferramentas para ajudar no desenvolvimento em C

Valgrind

- Ferramenta[3] que pode ajudá-los na escrita de algoritmos mais eficientes e corretos
- Pode ser usada para encontrar **vazamento de memória** e acesso a **posições inválidas de memória**
  - Casos típicos quando se está trabalhando com alocação dinâmica de memória e vetores
  - Muito útil para achar a causa de um *Segmentation fault*
- Pode ser usado de forma integrada com o CLion[4], mas é necessário que tenha o valgrind instalado em sua máquina

```
# gerando binário com informações de depuração. Parâmentro -g
gcc -g main.c -o saida

valgrind --leak-check=yes ./saida
```

[3] https://valgrind.org/docs/manual/quick-start.html
[4] https://www.jetbrains.com/help/clion/memory-profiling-with-valgrind.html

GCC

Parâmetros adicionais para encontrar vazamento de memória

```
# -Wall para apresentar todos avisos (warning) ao compilar
# -g inclui informações para depuração (-g0 a -g3)
# -fsanitize=address para detectar diferentes erros referentes a acesso a memória

gcc -Wall -g -fsanitize=address main.c -o saida
```

```
// Adaptado de https://valgrind.org/docs/manual/quick-start.html
#include <stdlib.h>

void f(void){
    int* x = malloc(100 * sizeof(int));
    x[100] = 0; // problema 1: acesso a posição inválida
}               // problema 2: vazamento de memória - x não é liberado

int main(void){
    for(int i=0; i < 1000; i++){
        f();
    }
    return 0;
}
```

Curiosidade

- UTF-8 foi criado por Ken Thompson, juntamente com Rob Pike, em um jogo americano (porta prato) durante o jantar em um restaurante em setembro de 1992 em New Jersey[5]
  - Atualmente ambos trabalham na empresa Google
- Ken Thompson e Dennis Ritchie foram responsáveis por criar a linguagem de programação B, predecessora da linguagem C
  - Ambos também atuaram no desenvolvimento do sistema operacional Unix
- Dennis Ritchie é um dos autores da linguagem C
- Ken Thompson também atuou no desenvolvimento da linguagem Go na Google

[5] https://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt

Referências

- *Introduction to Extended Characters*
  - https://www.gnu.org/software/libc/manual/html_node/Extended-Char-Intro.html
- Especificação de tamanho de argumento
  - https://learn.microsoft.com/pt-br/cpp/c-runtime-library/format-specification-syntax-printf-and-wprintf-functions?view=msvc-170#size