Instituto de Ciências Matemáticas e de Computação
| Universidade de São Paulo |

SCC-814– Projetos de Algoritmos
Prof. Zhao Liang

# Revisão da Linguagem C

Prof. Lucas Antiqueira

## Noções Básicas

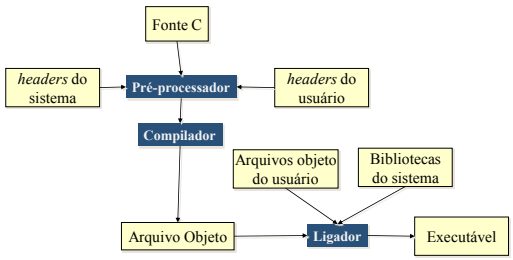


## Estrutura de um programa em C

**Programa C**

- **Diretivas ao Pré-Processador**
  - **Includes**
  - **Macros**
- **Declarações Globais**
  - **Funções**
  - **Variáveis**
- **Programa Principal**
  **int main ()**
  **{ /* begin */**
  **} /* end */**
- **Definição das Funções**

## Fluxo do Compilador C




## Começando do Zero

- Zero é ponto de início natural em C:
  - Contagens a partir de 0.
  - 0 significa falso e diferente de 0 significa verdadeiro.
  - Limite inferior de vetores é 0.
  - Sinal de fim de string é '\0'.
  - Ponteiros usam 0 para indicar um valor nulo.

## Variáveis

- Estão associadas a posições de memória que armazenam informações.
- Toda variável deve estar associada a um identificador.
- Palavras-chave de C não podem ser utilizadas como nome de variáveis: int, for, while, etc…
- C é case-sensitive:
  - contador ≠ Contador ≠ CONTADOR ≠ cOntaDor

## Variáveis

- Exemplos de nomes de variáveis:

| **Corretos** | Incorretos |
|---|---|
| Contador | 1contador |
| Teste23 | oi!gente |
| Alto_Paraiso | Alto..Paraíso |
| __sizeint | _size-int |

## Tipo de Dado

- O *tipo* de uma variável define os valores que ela pode assumir e as operações que podem ser realizadas com ela.
- Ex:
  - variáveis tipo *int* recebem apenas valores inteiros.
  - variáveis tipo *float* armazenam apenas valores reais.

## Tipos Básicos em C

- Os tipos de dados básicos, em C, são 5:
  - Caracter: **char**
    - Exemplos: 'a', '1', '+', '$', ...
  - Inteiro: **int**
    - Exemplos: -1, 1, 0, ...
  - Real: **float**
    - Exemplos: 25.9, -2.8, ...
  - Real de precisão dupla: **double**
    - Exemplos: 25.9, -2.8, ...
  - Sem valor: **void**
- Todos os outros tipos são derivados desses 5.

## Modificadores de Tipos

- Modificadores alteram algumas características dos tipos básicos para adequá-los a necessidades específicas.
- Modificadores:
  - **signed**: indica número com sinal (inteiros e caracteres).
  - **unsigned**: número apenas positivo (inteiros e caracteres).
  - **long**: aumenta abrangência (inteiros e reais).
  - **short**: reduz a abrangência (inteiros).

## typedef

- typedef, em C, permite dar novos nomes a tipos de dados existentes.
  - Composição a partir de tipos pré-existentes.
  - Não cria um novo tipo de dado!
- Forma geral:
  - `typedef tipo novo_nome;`
  - *tipo*:qualquer tipo válido em C.
  - *novo_nome*: um identificador válido em C.

## typedef

```
#include <stdio.h>
typedef float num_real;
typedef int medida;
typedef medida altura;

altura alt=20;
int x=4, i;

int main (void){
  i = alt / x;
  return(0);
}
```

## Caracteres e o tipo de dado “char”

- Em C, quaisquer variáveis de tipo inteiro podem ser usadas para representar um caracter.
  - Em geral usa-se **char** ou **int** para isso.
- Constantes como ‘a’ e ‘+’, que nós “pensamos” como caracteres, são do tipo int.

## Caracteres e o tipo de dado “char”

| ‘a’ | ‘b’ | ‘c’ | … | ‘z’ |
|---|---|---|---|---|
| 97 | 98 | 99 | | 122 |
| ‘A’ | ‘B’ | ‘C’ | … | ‘Z’ |
| 65 | 66 | 67 | | 90 |
| ‘0’ | ‘1’ | ‘2’ | … | ‘9’ |
| 48 | 49 | 50 | | 57 |

## Declaração de Variáveis

- A declaração de uma variável segue o modelo:

```
TIPO_VARIÁVEL lista_de_variaveis;
```

- Ex:

```
int x, y, z;
float f;
unsigned int u;
long double df;
```

## Escopo de Variáveis

- Escopo define onde e quando uma variável pode ser usada em um programa.
- variável declarada fora das funções (global) tem escopo em todo o programa:

```
#include <stdio.h>
int i = 0;          /* variavel global */
                    /* visivel em todo o código */
void incr_i() { i++; }
...
void main() { incr_i(); printf("%d", i); }
```

## Escopo de Variáveis

- Escopo de função: variável declarada na lista de parâmetros da função ou definida dentro da função.
- Ex:

```
...
void f (void) {
  printf ("%d %d", i, j); /* erro: i e j não definidos */
}
int main (void) {
  int i, j; /* i e j visiveis apenas dentro da função main*/
  f();
  ...
}
```

## Expressões

- Em C, expressões são compostas por:
  - Operadores: + , - , % , …
  - Constantes e variáveis.
- Toda expressão termina com ;
- Ex:

```
x;
14;
x + y;
(x + y)*z + w – v;
```

## Expressões

- Expressões retornam um valor:

```
x = 5 + 4 /* retorna 9 */
```

  - esta expressão retorna 9 como resultado da expressão e atribui 9 a x

```
((x = 5 + 4) == 9) /* retorna true */
```

  - na expressão acima, além de atribuir 9 a *x*, o valor retornado é utilizado em uma comparação

- Expressões em C seguem, geralmente, as regras da álgebra.
  - Porém, existem alguns aspectos específicos de C.

## Precedência

**Maior precedência**

( ) [ ] ++ (pós) -- (pós)
! ~ ++(pré) -- (pré) . - (unário) (cast) *(unário) &(unário) sizeof
* / %
+ -
<< >>
<<= >>=
== !=
&
^
|
&&
||
?
= += -= *= /= etc...
,

**Menor precedência**

## Conversão de Tipos

```
#include <stdio.h>
 int main (void)
 {
     int num;
     float f;
     num = 10;
     f = (float)num/7;
     printf ("%f", f);
}
```

Se não tivéssemos usado o modelador no exemplo ao lado, uma divisão inteira entre 10 e 7 seria efetuada. O resultado seria 1 e este seria depois convertido para **float,** mas continuaria a ser 1.0. Com o *cast* temos o resultado correto.

## Operadores unários

| | | |
|---|---|---|
| + | mais unário (positivo) | /* + x; */ |
| - | menos unário (negativo) | /* - x; */ |
| ! | NOT ou negação lógica | /* ! x; */ |
| & | endereço | /* &x; */ |
| * | conteúdo (ponteiros) | /* (*x); */ |
| ++ | pré ou pós incremento | /* ++x ou x++ */ |
| -- | pré ou pós decremento | /* -- x ou x -- */ |

## Operador ++

- Incremento
  - O operador ++ pode ser usado de modo pré ou pós-fixado em uma variável.
  - Exemplo:

```
int x=1, y=2;
x++; /* equivale a x = x + 1*/
++y; /* equivale a y = y + 1*/
```

  - x e y são incrementados em uma unidade.
  - Não pode ser aplicado a constantes nem a expressões.

## Operador ++

- Incremento
  - A instrução ++x executa a operação de incremento para depois usar x.
  - A instrução x++ primeiro usa o valor de x para depois incrementá-lo.

## Operador --

- Decremento
  - O operador -- decrementa seu operando de uma unidade.
  - Funciona de modo análogo ao operador ++.

## Operadores de Atribuição

| | | |
|---|---|---|
| = | atribui | x = y; |
| += | soma e atribui | x += y; <=> x = x + y; |
| -= | subtrai e atribui | x -= y; <=> x = x - y; |
| *= | multiplica e atribui | x *= y; <=> x = x * y; |
| /= | divide e atribui quociente | x /= y; <=> x = x / y; |
| %= | divide e atribui resto | x %= y; <=> x = x % y; |
| &= | E bit-a-bit e atribui | x &= y; <=> x = x & y; |
| \|= | OU bit-a-bit e atribui | x \|= y; <=> x = x \| y; |
| <<= | shift left e atribui | x <<= y; <=> x = x<<y; |

## Operadores Relacionais

- Aplicados a variáveis que obedeçam a uma relação de ordem, retornam 1 (true) ou 0 (false)

| Operador | Relação |
|---|---|
| > | Maior do que |
| >= | Maior ou igual a |
| < | Menor do que |
| <= | Menor ou igual a |
| == | Igual a |
| != | Diferente de |

## Operadores Lógicos

- Operam com valores lógicos e retornam um valor lógico verdadeiro (1) ou falso (0)

| Operador | Função | Exemplo |
|---|---|---|
| && | AND (E) | (c >='0' && c<='9') |
| \|\| | OR (OU) | (a=='F' \|\| b!=32) |
| ! | NOT (NÃO) | (!var) |

## O Comando printf()

- Características:
  - Definido em stdio.h
  - Permite escrever dados em vários formatos.
  - Forma geral:
    - `printf("string_de_controle", lista de variáveis);`
    - A string de controle pode conter:
      - Caracteres que serão impressos na tela.
      - Comandos de formato. Começam com o símbolo %
    - A lista de variáveis contém os nomes das variáveis cujos valores serão impressos de acordo com o formato especificado na string de controle.

## O Comando printf()

| Comando | Formato |
|---|---|
| %c | Caractere |
| %d | Inteiro |
| %e | Notação científica |
| %f | Float |
| %o | Octal |
| %s | String |
| %u | Inteiro sem sinal |
| %x | Hexadecimal |
| %% | Escreve o símbolo % |

## O Comando printf()

- Especificador de Precisão
  - Um ponto seguido de um número inteiro.
  - Limita o número de casas decimais a serem impressas.
  - Exemplo:

```
double num = 3.456789;
printf("%.2f", num);
```

Resultado: 3.45

## O Comando scanf()

- Características:
  - Definido em stdio.h
  - Permite ler dados, em vários formatos, vindos do teclado.
  - Forma geral:
    - `scanf("string_de_controle", lista de variáveis);`
    - A string de controle pode conter:
      - Especificadores de formato.
    - A lista de variáveis contém os nomes das variáveis cujos valores serão lidos do teclado, no formato especificado, e armazenados, respectivamente, nas variáveis.

## O Comando scanf()

- Exemplo:

```
int num;
scanf("%d", &num);
printf("%d", num);
```

- IMPORTANTE: scanf necessita que toda variável, exceto strings, usem o operador &.
- Outro exemplo:

```
int num;
char ch;
scanf("%d %c", &num, &ch);
printf("o número é %d \n", num);
printf("e o caracter é %c", ch);
```

## O Comando scanf()

| Comando | Formato |
|---|---|
| %c | Caracter |
| %d | Inteiro |
| %e | Número em ponto flutuante |
| %f | Número em ponto flutuante |
| %o | Octal |
| %s | String |
| %x | Hexadecimal |
| %u | Inteiro sem sinal |

## O Comando scanf()

- Lendo strings
  - O comando:

```
scanf ("%s", str);
```

Lê uma string até encontrar um espaço em branco.

  - Coloca '\0' no fim da string.

## O Comando if

- Forma geral:

**if (***expressão***)** *sentença1***;**
**else** *sentença2***;**

- *sentença1* e *sentença2* podem ser uma única sentença, um bloco de sentenças, ou nada.
- O **else** é opcional.

## O Comando if

**if (***expressão***)** *sentença1***;**
**else** *sentença2***;**

- Se *expressão* é verdadeira (!= 0), a sentença seguinte é executada. Caso contrário, a sentença do **else** é executada.
- O uso de if-else garante que apenas uma das sentenças será executada.

## O Comando if

- O comando *if* pode ser aninhado.
  - Um comando *if* aninhado é um *if* que é sentença de outro comando *if* ou *else*.
  - ANSI C especifica máximo de 15 níveis.
- Cuidado: um *else* se refere, sempre, ao *if* mais próximo, que está dentro do mesmo bloco do *else* e não está associado a outro *if*.

## O Comando if

```
if (cond1)
  if (cond2)
      comando1;
else
  comando2;
```

## O Comando if

```
if (cond1){
  if (cond2)
      comando1;
}
else
  comando2;
```

## O Operador ?

- É um operador ternário.

```
expressão1 ? sentença1 : sentença2
```

- Pode ser usado para substituir o *if*.

```
if (expressão1) sentença1;
else sentença2;
```

As sentenças devem ser expressões simples. Nunca um outro comando em C.

## Exemplo

```
x = 10;
y = x>9 ? 100 : 200;
```

y é igual a 100.

```
x = 10;
if (x > 9) y = 100;
else y = 200;
```

## O Comando switch

```
switch (expressão){
     case constante1: sequência1; break;
     case constante2: seqüência2; break;
     ...
     default: seqüência_n;
}
```

## O Comando switch

```
char ch;
ch  = getchar();
switch (ch){
  case '1': printf("1"); break;
  case '2': printf("2"); break;
  case 'a': printf("a"); break;
  case '8': printf("8"); break;
  case '3': {printf("3");
             printf("\n três");
             break;
            }

  default: printf(" ");
}
```

## O Comando for

- Encontrado, de um modo ou de outro, em praticamente todas as linguagens estruturadas.
- Em C, fornece maiores flexibilidade e capacidade.
- Forma geral:

```
for (inicialização ; condição ; incremento) comando;
```

## O Comando for

- Exemplo

```
#include <stdio.h>
int main (void){
  int i;
  for (i=0; i<10;i++)
       printf("%d \n", i);

  return(0);
}
```

## O Comando while

- Forma geral

```
while(condição)
     comando;
```

- *condição*: é qualquer expressão. Determina o fim do laço quando a condição é falsa. Execução continua na sentença seguinte ao while.
- *comando*:pode ser vazio, simples ou um bloco.

## O Comando while

Assim como o for, o while testa uma *condição* antes de entrar no laço.

```
char ch ='\0'; /*caracter nulo*/

while (ch != 'Z')
  ch = getchar();
printf ("Z: fim do laço while");
```

## O Comando do

- Forma geral

```
do{
   comando ;
}while(condição);
```

- *comando*: pode ser vazio, simples ou um bloco.
- *condição*: pode ser qualquer expressão. Se falsa, o comando é terminado e a execução continua na sentença seguinte ao do-while.

## O Comando do

- Exemplo:

```
scanf("%d", &j);
j--;
i = 1;
do{
  i = i * j;
  j--;
}while(j > 0);
```

## Exercício 1

- Implemente um programa em C que
  1. Leia um número positivo do usuário (escolha o que fazer se o número lido for negativo).
  2. Calcule e imprima a seqüência de Fibonacci até o primeiro número superior ao número lido do usuário

Exemplo:
Se o usuário informou o número 30, a seqüência a ser impressa é
0 1 1 2 3 5 8 13 21 34

## Arquivos

52

## Arquivos

- O que é um arquivo?
  - Modelo para representar informações.
- Para que serve?
  - Armazenar informações de modo permanente em meio externo.
- Quando é necessário?
  - Armazenar informações de modo permanente.
  - Volume de dados muito grande.

## Arquivos em C

- Um arquivo em C pode ser qualquer coisa, desde um arquivo em disco (um .doc, por exemplo) até uma impressora.
- Associação stream-arquivo é feita via operação de abertura.
- Após aberto, informações pode ser trocadas entre o programa e o arquivo.

## Arquivos em C

- Estrutura FILE
  - Estrutura de controle para streams.
  - Cabeçalho stdio.h.
  - Uso de FILE é feito via um ponteiro:

```
FILE *fp;
```

## Arquivos em C

- Protótipos em stdio.h:
  - `fopen()`
  - `fclose()`
  - `putc()`
  - `fputc()`
  - `getc()`
  - `fgetc()`
  - `fseek()`
  - `fprintf()`
  - `fscanf()`
  - `feof()`
  - `ferror()`
  - `rewind()`
  - `remove()`
  - `fflush()`

## Abertura de arquivo

```
FILE *fopen(const char *nomearq, const char *modo);
```

| Modo | Significado |
|---|---|
| r | Abre p/ leitura (texto) |
| w | Cria p/ escrita (texto) |
| a | Anexa ao fim (texto) |
| rb | Abre p/ leitura (binário) |
| wb | Cria p/ escrita (binário) |
| ab | Adiciona ao fim (binário) |
| r+ | Abre p/ leitura/escrita (texto) |
| r+b | Abre p/ leitura/escrita (binário) |

## Abertura de arquivo

- Exemplo:

```
FILE *fp;
if ((fp = fopen("teste.dat", "w")) == NULL){
  printf("Erro na abertura do arquivo!");
  exit(1);
}
```

## Fechamento de arquivo

- `int fclose(FILE *fp);`
- `fclose()` fecha um arquivo que foi aberto via `fopen()`.
- Uma chamada à `fclose()`:
  - Grava os dados (buffer).
  - Fecha o arquivo.

## Fechamento de arquivo

- Exemplo

```
FILE *fp;
if ((fp = fopen("teste.dat", "w")) == NULL){
  printf("Erro na abertura do arquivo!");
  exit(1);
}
...
fclose(fp);
```

## Fim de arquivo

- `int feof(FILE *fp);`
  - Devolve verdadeiro se o fim de arquivo for encontrado.

```
while (!feof(fp))
  ch = getc(fp);
```

## Outros comandos

- fflush()
  - int fflush(FILE *fp);
  - Esvazia um stream. Se for chamada com valor nulo, descarrega todos os streams abertos.

## Outros comandos

- `fprintf()` e `fscanf()`
  - Equivalentes a `printf()` e `scanf()`.
  - `int fprintf(FILE *fp, const char *control_str, ...);`
  - `int fscanf(FILE *fp, const char *control_str, ...);`

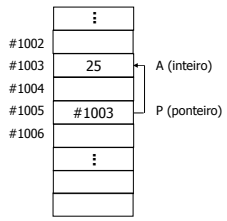
## Ponteiros



## Definição

- **É uma variável que contém um endereço de memória.**
  - Normalmente, esse endereço é a posição de outra variável na memória.
  - Dizemos que um ponteiro "aponta" para uma variável.

## Exemplo



## Declaração de ponteiros em C

- Forma geral: **`tipo *identificador;`**
  - **tipo**: qualquer tipo válido em C.
  - identificador: qualquer identificador válido em C.
  - *****: símbolo para declaração de ponteiro. Indica que o **identificador** aponta para uma variável do tipo **tipo**.
- Exemplos:

```
int *p;
char *p1;
float *pf1, *pf2;
```

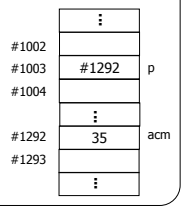
## O Operador &

- &: operador unário. Devolve o endereço de memória de seu operando.
  - Seu uso mais comum é durante inicializações de ponteiros.
  - Ex.:

```
int *p, acm = 35;
p = &acm;     /*p recebe "o endereço de"
                acm*/
```

  - O valor de p é 1292
  - Outro modo de inicializar:

```
p = 0;
p = NULL; /*equivale a p = 0*/
```




## O Operador *

- *: operador unário. Devolve o valor da variável apontada.
  - Ex.:

```
int *p, q, acm = 35;
p = &acm;
q = *p;     /*   q recebe o conteúdo da variável
                 "no endereço" p   */
```

  - O valor de q é 35

## Erros Comuns

- Ponteiro não inicializado.

```
int x, *p;
x = 10;
*p = 25; /*antes, deveria haver: p = &x; */
```

  - p não aponta para um endereço válido.
  - **Boa prática: inicializar todo ponteiro!**

## Erros Comuns

- Ponteiro inicializado de modo errado.

```
int x, *p;
x = 10;
p = x;  /*errado*/
printf("%d",*p);
```

  - Não irá imprimir o valor de x.
  - p = x; está errado. p deve conter um endereço e não um valor.
  - O correto é p = &x;

## Erros Comuns

- Comparação entre ponteiros:

```
char s, y;
char *p1, *p2;
s='s'; y=' y';
p1 = &s; p2 = &y;
if (p1 < p2)  ...
```

  - Comparações de ponteiros comparam posições de memória.
  - Contudo, não sabemos onde p1 e p2 estão alocados.

# Funções



## Forma Geral

```
tipo nome_da_função (lista de parâmetros) {
  declarações
  sentenças
}
```

- Tudo antes do "abre-chaves" compreende o **cabeçalho** da **definição** da função.
- Tudo entre as chaves compreende o **corpo** da **definição** da função.

## Regras de Escopo

```
#include <stdio.h>

int f1(int x);
int f2(int y);

int w = 0;   /*var. global: evitar*/

int main (void){
   int A=2, B;

   B = f1(w);
}

int f1 (int x) {
   char c; int y, j;
   y = c + x + A;        /*erro: A não está acessível*/
   j = f2(y);
   return (j);
}
```

```
int f2(int y){

        int x;
        x = y + w;
        return(x);
}
```

## Argumentos e Parâmetros

```
int func (int x, char y, float z) {
  char c;
  x = y + z;
  return (x);
}

int main (void){
  int i, a=2;
  char b='C';
  float c=2.35;
  ...
  i = func(a, b, c);
}
```

Os **argumentos** e os **parâmetros** devem ser em igual número. Além disso, os tipos devem estar na mesma ordem.

## Passagem por Valor

- Em C, a passagem de parâmetros padrão (*default*) é por valor.
- Esse método copia o valor do argumento no parâmetro da função.
- Alterações no parâmetro, feitas dentro da função, não alteram o argumento.

## Passagem por Valor

```
int main (void)
{
   int n = 3, sum;
   printf("%d \n",n); /*imprime 3*/

   sum = soma(n);

   printf("%d \n", n); /* imprime 3*/
   printf("%d", sum); /*imprime 6*/
   return 0;
}
```

```
int soma (int n)
{
        int sum;
        for ( ; n > 0; --n);
                sum += n;
        return(sum);
}
```

## Passagem por Referência

- O endereço do argumento é copiado no parâmetro.
- Dentro da função, o endereço é usado para acessar o argumento real utilizado na chamada.
- As alterações feitas no parâmetro, dentro da função, afetam o argumento.

## Passagem por Referência

```
#include <stdio.h>
void swap (int *x, int *y);
void main (void)
{
  int i, j;
  i = 10;
  j = 20;
  swap (&i, &j);
  printf("%d %d", i, j);
}
```

```
void swap (int *x, int*y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

## Retorno de Valores

- Como as funções retornam valores, elas podem ser usadas como operandos em expressões.

```
x = max(x,y) + 100;
```

- Funções não podem receber uma atribuição:

```
max(x,y) = 100; /*errado*/
```

## O Comando return

- Comando *return*
  - Provoca a saída imediata de uma função.
    - Controle retorna ao código chamador.
  - Pode ser usado para retornar valores.
    - `return;`
    - `return ++a;`
    - `return (a * b);`
    - `return a * b;`

## Vetores



## Vetores

- Forma geral da declaração:
  - **`tipo A[`***`expressão`***`]`**
    - **tipo** é um tipo válido em C.
    - A é um identificador.
    - *expressão* é qualquer expressão válida em C que retorne um **valor inteiro positivo**.
  - Exemplos

```
float salario[100];
int numeros[15];
double distancia[43];
```

## Inicialização

- `float F[5] = {0.0, 1.0, 2.0, 3.0, 4.0};`
- `int A[100] = {0};`
  - Todos os elementos de são inicializados com 0.
- `int A[100] = {1, 2};`
  - A[0] recebe 1, A[1] recebe 2 e o restante recebe 0.
- `int A[] = {2, 3, 4};`
  - Equivale a: `int A[3] = {2, 3, 4};`

## Vetores e Ponteiros

- O nome (identificador) de um array é um ponteiro.
  - Aponta para o primeiro elemento do array.
- Exemplo:

```
int A[3] = {5, 10, 15};
printf("%d", *A);
*A = 2;
printf("%d, %d, %d", A[0], A[1], A[2]);
```

## Vetores e Ponteiros

- `int A[5];`
- Diferenças entre ponteiros e vetores:
  - Um ponteiro pode receber diferentes endereços.
  - A é um endereço, ou ponteiro, que é fixo.
  - Isso implica que não se pode mudar o valor de A.
    - `A = p; ++A; A += 2;`
      - São operações ilegais.
    - `&A;`
      - Errado, A já é um endereço.

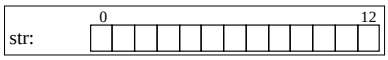
## Exercício 2

- Implemente em C um programa que leia e armazene em um vetor as notas de uma prova de toda uma turma de alunos e, ao final, calcule e imprima a média geral
  - Implemente uma função para ler as notas e outra para calcular a média geral

## Strings em C

- Strings são seqüências de caracteres adjacentes na memória. O caractere '\0' (= valor inteiro 0) indica o fim da seqüência
- Declaração:

  Ex: `char str[13];`
  - define uma string de nome **str** e reserva para ela um espaço de 13 bytes na memória.
  - Deve-se lembrar do '\0'.

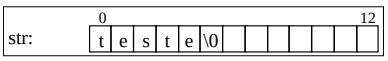



## Inicialização e Atribuições

- `char str[13];`
- Atribuição errada:

  `str = "teste"; /*Erro!!!*/`
- Correta:

```
str[0] = 't'; str[1] = 'e'; str[2] = 's';
str[3] = 't'; str[4] ='e'; str[5] ='\0';
```



## Strings

- Funções para manipulação de strings – definidas em <string.h>:
  - `strcpy(s1, s2)` – copia s2 em s1.
  - `strcat(s1, s2)` – concatena s2 ao final de s1.
  - `strlen(s1)` – retorna o tamanho de s1.
  - `strcmp(s1, s2)` – retorna:
    - 0 se s1 e s2 são iguais;
    - Valor menor que 0, se s1 < s2 (lexicograficamente);
    - Valor maior que 0, se s1 > s2 (lexicograficamente).

## Matrizes

- Forma geral da declaração - matrizes bidimensionais:
  - **`tipo M[`***`expressão`***`][`***`expressão`***`];`**
    - **tipo** é um tipo válido em C.
    - M é um identificador.
    - *expressão* é qualquer expressão válida em C que retorne um **valor inteiro**.
  - Exemplos:

```
float A[100][100];
int B[15][3];
double distancias[43][128];
```

```
/* Soma os elementos de uma matriz */
int i, j, soma, mat[3][4];
...
for(i=0; i<3; i++)
  for (j=0; j<4; j++)
       soma += mat[i][j];
printf("Soma: %d", soma);
...
```

## Arrays como Parâmetros

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

void print_upper (char *string);

int main (void){
  char s[80];
  gets(s);
  print_upper(s);
}

void print_upper (char *string){
   int t;
   for (t=0; string[t]; ++t){
      string[t] = toupper(string[t]);
      putchar(string[t]);
   }
}
```

## Estruturas



## Estruturas

- Uma estrutura é uma coleção de variáveis, possivelmente de diferentes tipos, organizadas em um único conjunto.
- As variáveis que compreendem uma estrutura são comumente chamadas de **elementos** ou **campos**.

- Definição x Declaração

```
struct pessoa {
  char nome[30];
  int idade;
};
```

- Permite declarar variáveis cujo tipo seja **pessoa**.

- Definição x Declaração

```
struct pessoa pai, mae, tio, irmao;
```

ou

```
struct pessoa {
  char nome[30];
  int idade;
} pai, mae, tio, irmao;
```

## Usando typedef

```
struct a {
   int x;
   char y;
};
typedef struct a MyStruct;
...
MyStruct b; /*declaração da var b, cujo tipo é MyStruct*/
```

ou

```
typedef struct a {
   int x;
   char y;
} MyStruct;
...
MyStruct b; /*declaração da var b, cujo tipo é MyStruct*/
```

## Acesso aos Dados

- Acesso feito via o operador ponto (.).

```
struct a {
  int x;
  char y;
} MyStruct;

int main (void) {
  int num;
  MyStruct.x = 10;
  MyStruct.y = 'a';
  num = MyStruct.x;
}
```

## Estruturas como Parâmetros

```
...
struct A {
   int a;
   char b;
};

void f1 (struct A param) {
   ...
}

int main (void) {
   struct A my_s;
   ...
   f1(my_s);
}
...
```

Passagem por valor

## Estruturas como Parâmetros

```
...
struct A {
   int a;
   char b;
};

void f1 (struct A *param) {
   param->a = 10;
   param->b = 'x';
}

int main (void) {
   struct A my_s;
   ...
   f1(&my_s);
}
...
```

Passagem por referência

Acesso aos dados via ponteiro

Usa-se o operador ->

## Exercício 3

- Implemente um programa em C que leia o nome, a idade e o endereço de 2 pessoas e armazene os dados em estruturas.
- Implemente uma função que troque os dados anteriores de duas pessoas
  - Exemplo:


## Alocação Dinâmica

104

## Alocação Dinâmica

- É um meio pelo qual um programa pode obter memória enquanto está em execução.
- C fornece funções para realizar alocação dinâmica de memória.
- A memória alocada dinamicamente é obtida do *heap*.

## Alocação Dinâmica

- Funções `calloc()`, `malloc()`, `realloc()` e `free()`.
  - Funções ANSI, disponíveis no header stdlib.h.
  - *Contiguous allocation (calloc)* e *memory allocation (malloc)*.
  - `calloc` e `malloc` criam espaço na memória.
  - `free` libera espaço alocado com `calloc` ou `malloc`.
  - `realloc` realoca espaço previamente alocado.

## Alocação Dinâmica

```
void *calloc(int n, size_t tam);
```

- Aloca espaço contíguo na memória para **n** elementos, com cada elemento tendo **tam** bytes.
- O espaço é inicializado com todos os bits iguais a zero.
- Sucesso retorna um ponteiro `void` para o endereço base do espaço alocado.
- Falha retorna `NULL`.

## Alocação Dinâmica

- Exemplo:

```
int n = 5, *p;

p = (int *) calloc(n, sizeof(int)); /*versão portável*/
```

## Alocação Dinâmica

```
void *malloc(size_t tam);
```

- Aloca **tam** bytes de espaço na memória.
- Diferente de `calloc`, `malloc` não inicializa o espaço de memória alocado.
- Sucesso retorna um ponteiro `void` para o endereço base do espaço de memória alocado.
- Falha retorna `NULL`.

## Alocação Dinâmica

- Exemplo:

```
int *p;
p = (int *) malloc(2*sizeof(int));
p[0] = 1;
p[1] = 2;
```

## Alocação Dinâmica

- A memória disponível para ser alocada (*heap*) não é infinita!
- Modo correto de usar `calloc` ou `malloc`:

```
if( (p = malloc(10) == NULL ) {
  printf("Sem memória");
  exit(1); /*ou outro método de tratar erro*/
}
```

## Alocação Dinâmica

- `void *free(void *ptr);`
  - Libera espaço de memória alocado por `malloc` ou por `calloc`.
  - Exemplo:

```
int *p;
p = (int *) malloc(sizeof(int));
*p =10;
printf("%d", *p);
free(p);
```

## Alocação Dinâmica

- Memória alocada deve ser explicitamente devolvida usando `free`.
- Caso contrário, só será devolvida quando o programa (ou função) terminar.
- Boa prática de programação: usar `free` para todo ponteiro quando o mesmo não for mais necessário.

## Exercício 4

- Faça um programa em C que crie um vetor de tamanho definido pelo usuário, e leia do teclado seus valores inteiros.
  - Ao final, imprima o vetor lido.

## Créditos

Aula baseada no material do Prof. Rudinei Goularte

Exercícios desenvolvidos pelo Prof. Thiago Pardo