ICMC USP SÃO CARLOS
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

# SCC-601– Introdução à Ciência da Computação II

## Revisão da Linguagem C

Lucas Antiqueira

# Tópicos

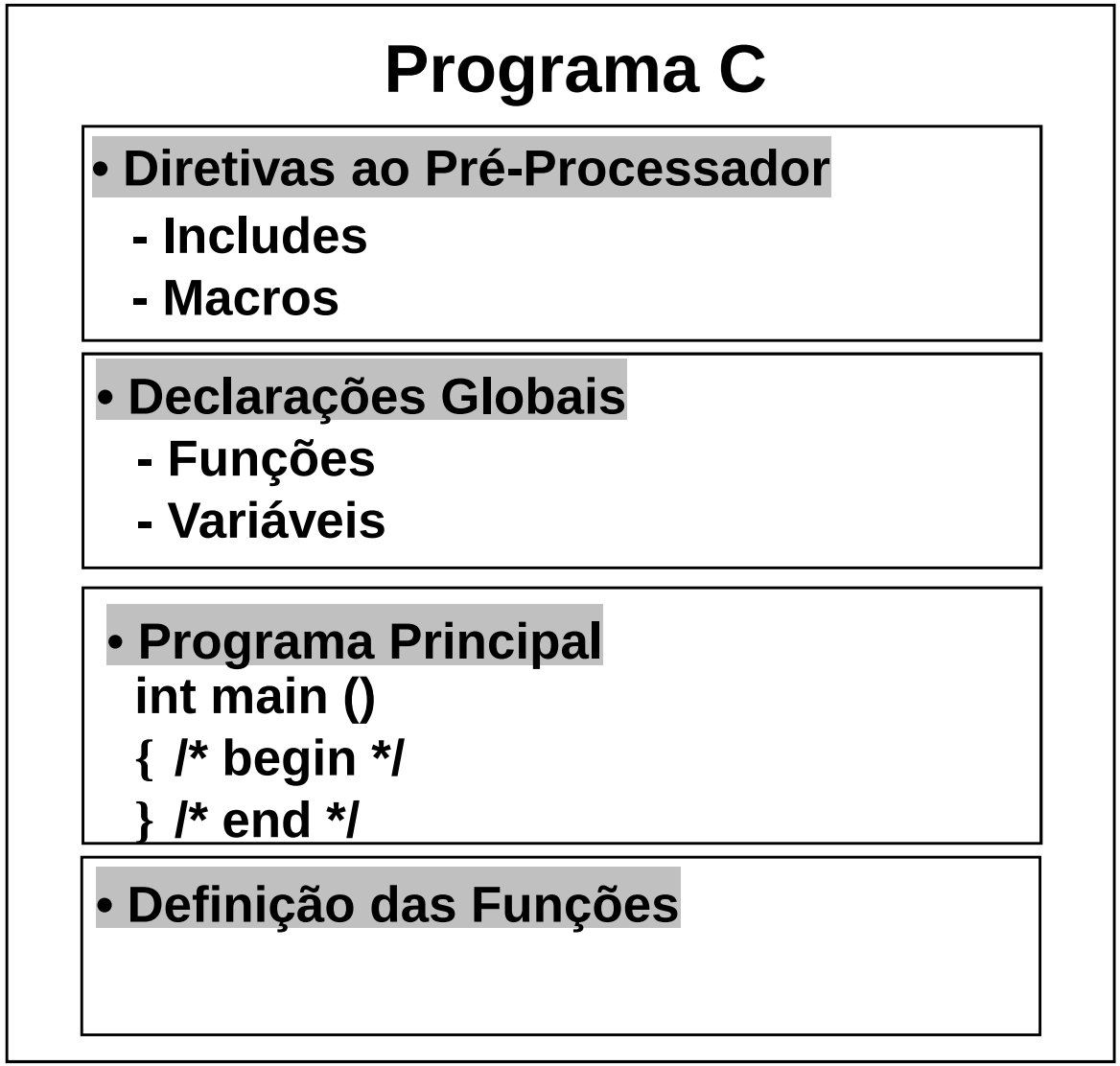
1. Constantes, Variáveis
2. Expressões, Operadores
3. printf, scanf
4. Comandos de Seleção
5. Laços de repetição
6. Arquivos
7. Ponteiros
8. Funções
9. Vetores, Strings, Matrizes
10. Estruturas
11. Alocação Dinâmica

# Tópicos

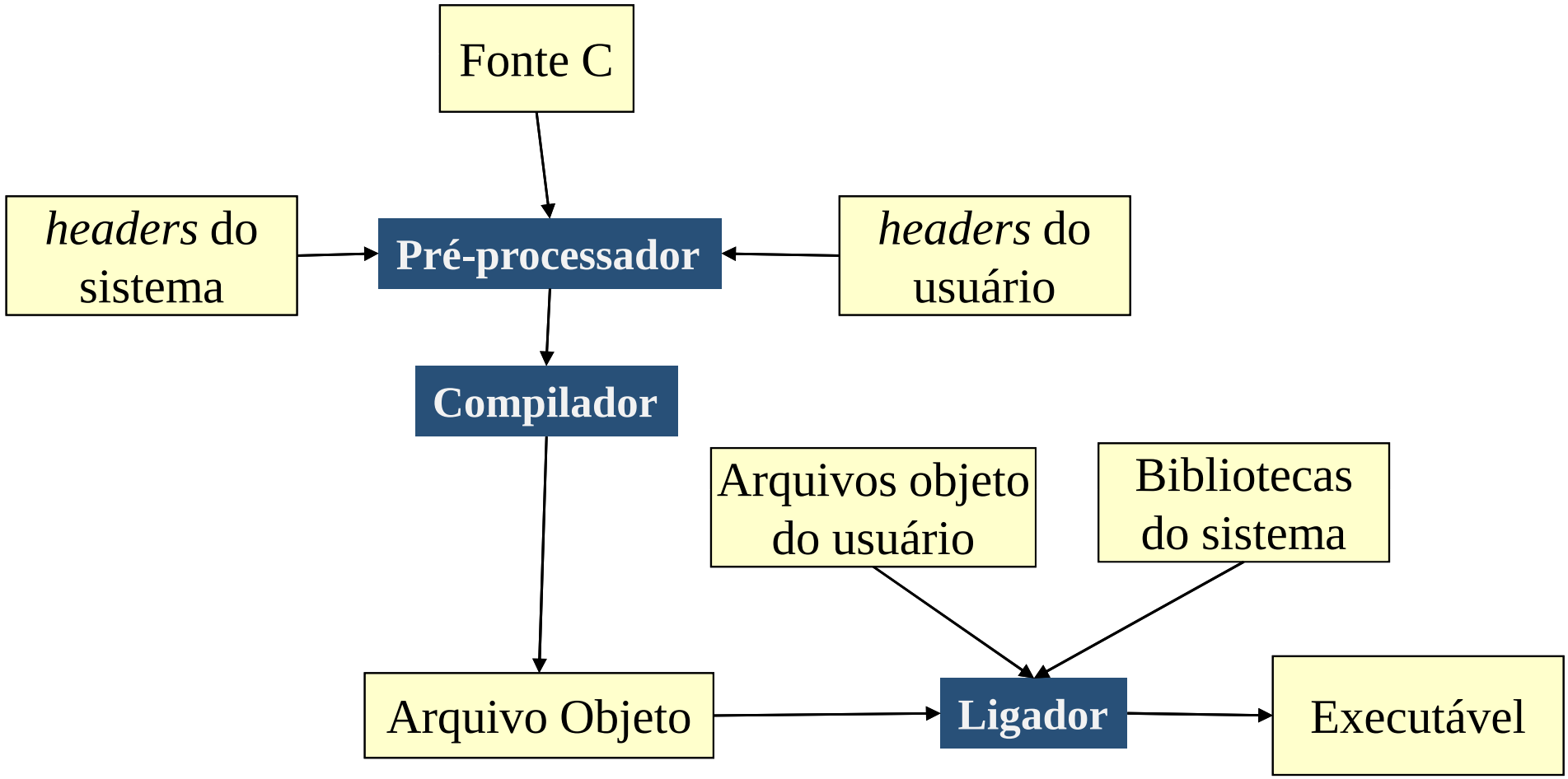
1. Constantes, Variáveis
2. Expressões, Operadores
3. printf, scanf
4. Comandos de Seleção
5. Laços de repetição
6. Arquivos
7. Ponteiros
8. Funções
9. Vetores, Strings, Matrizes
10. Estruturas
11. Alocação Dinâmica

* Maior ênfase

# Estrutura de um programa em C



# 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.

# Constantes, Variáveis

# Variáveis e Constantes

- Identificadores
  - Em C, são utilizados para dar nomes a variáveis e constantes.
  - Identificadores podem ter vários caracteres.
  - Em C, apenas os 31 primeiros caracteres são considerados.
  - O primeiro caracter tem que ser uma letra ou underscore "_".
  - O restante do nome pode conter letras, dígitos e underscore.

# Constantes

- Constantes
  - São valores fixos que não podem ser modificados pelo programa.

- Exemplos:
  - Caracteres: ‘a’, ‘\n’, ‘9’
  - Valores inteiros: 123, 1, 1000, -23
  - Reais: 123.45F, 3.1415e-10F
  - Reais de precisão dupla: 123.45, -0.91254
  - Strings: “abcd”, “isto é uma string!”, “Av. São Carlos, 2350”

# 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).

# Abrangência dos Dados

- Depende da Arquitetura do Processador
- Padrão ANSI define apenas a faixa mínima de valores de cada tipo, mas não seu tamanho

| Tipo | Tamanho(bytes) | Abrangência | | |
|---|---|---|---|---|
| char | 1 | -127 | a | 127 |
| unsigned char | 1 | 0 | a | 255 |
| singned char | 1 | -127 | a | 127 |
| int | 2 | -32767 | a | 32767 |
| unsigned int | 2 | 0 | a | 65535 |
| signed int | 2 | mesmo que int | | |
| short int | 2 | mesmo que int | | |
| unsigned short int | 2 | 0 | a | 65535 |
| signed short int | 2 | mesmo que short int | | |
| long int | 4 | -2.147.483.647 | a | 2.147.483.647 |
| signed long int | 4 | o mesmo que long int | | |
| unsigned long | 4 | 0 | a | 4.294.967.295 |
| float | 4 | 6 dígitos de precisão | | $(-3,4 \cdot 10^{38}$ a $3,4 \cdot 10^{38})$ |
| double | 8 | 10 dígitos de precisão | | $(-1,7 \cdot 10^{308}$ a $1,7 \cdot 10^{308})$ |
| long double | 10 | 10 dígitos de precisão | | $(-3,4 \cdot 10^{4932}$ a $3,4 \cdot 10^{4932})$ |

- Tipos fundamentais agrupados por funcionalidade:
  - Tipos inteiros: char, signed char, unsigned char, short, int, long, unsigned short, unsigned, unsigned long.
  - Tipos de ponto flutuante: float, double, long double.

# 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”

- Cada **char** é armazenado na memória em um byte.
- $2^8 = 256$ valores possíveis.
- Código ASCII.

# 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;
```

# Atribuição e Inicialização

- Operador de atribuição: =
  - Exemplo: `int a; a = 10;`
- Variáveis podem ser incializadas na declaração:
  - `int a = 10;`
- Inicializar uma variável significar atribuir à mesma um valor inicial válido.
  - Ao se declarar a variável, a posição de memória da mesma contém um valor qualquer.

# Onde declarar variáveis?

- Em três lugares, basicamente:
  - Dentro de funções: variáveis locais.
  - Na definição de parâmetros de funções: parâmetros formais.
  - Fora de todas as funções: variáveis globais.

# 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 bloco: é visível apenas no interior do bloco

```
...
if (teste == TRUE) {
        int i;
        i = i+1;
        ...
}
else {
        i = i - 1;    /* erro: i não definida */
        ...
}
...
```

# 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 visíveis apenas dentro da função main*/
   f();
   ...
}
```

# Atribuição de Variáveis

- Forma geral: nome_da_variável = expressão;
  - Atribui o valor de *expressão* à variável à esquerda de =
  - Expressão pode ser desde uma constante até uma expressão complexa.

- Múltiplas atribuições
  - C permite a atribuição de mais de uma variável em um mesmo comando:

```
x = y = z = 0;
```

- As atribuições do programa abaixo estão corretas?

```
int main (void){
  int i = 1; char c = ‘A’; float f = 5.0;
  c = i;
  i = f;
  f = c;
  f = i;
  return(0);
}
```

# Conversões de Tipos na Atribuição

- Quando uma variável de um tipo é atribuída a uma de outro tipo, o compilador automaticamente converte o tipo da variável a direita de "=" para o tipo da variável a esquerda de "=".
- Ex: (assumindo char de 1 byte)

```
int i; char c; float f;
c = i;   /* c recebe 8 bits menos significativos de i */
i = f;   /* i recebe parte inteira de f */
f = c;   /* f recebe os 8 bits de c convertidos para real */
f = i;   /* idem para inteiro i */
```

# Expressões, Operadores

# 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**

# Precedência

- Conflito de avaliação:
  - Primeiro precedência do operador, depois, regra de associatividade.
- Exemplo:
  - 1 + 2 * 3 é equivalente a 1 + (2 * 3), diferente de (1 + 2) * 3.
  - Usa-se a precedência de * sobre +.
- Exemplo:
  - 1 + 2 – 3 + 4 – 5 é equivalente a (((1+2) – 3) + 4) – 5.
  - Usa-se associatividade esquerda para a direita dos operadores + e -.
- Obs.:
  - Parênteses são úteis para alterar a precedência.

# Conversão de Tipos

- Os operandos de uma expressão precisam ser do mesmo tipo.

- Quando tipos diferentes são misturados em uma mesma expressão, é necessária uma conversão.
  - A conversão é feita automaticamente.
  - Essa conversão é chamada de promoção de tipos.

# Conversão de Tipos

- Casts
  - Também chamados de modeladores.
  - São usados para forçar uma expressão a assumir um determinado tipo.
  - Forma geral: (tipo) expressão;
  - Exemplo: (float) (3 + 4);

# 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 ++

```
int main (void){
  int x = 10;
  int y = 0;
  y = ++x;
  printf (“%d %d”, x, y);
  y = 0; x = 10;
  y = x++;
  printf (“%d %d”, x, y);
  return(0);
}
```

# 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) |

# printf, scanf

# 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.

# Comandos de Seleção

# 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 mínimo 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

- Compara o valor da expressão com constantes inteiras ou caracteres, somente.
- Padrão ANSI especifica que switch pode ter, pelo menos, 257 comandos case.
- switch só pode testar igualdade (com os case's).
- Duas constantes case no mesmo switch não podem ter valores idênticos.
- break é opcional.
- default é opcional.
- switch pode ser aninhado.

# 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(“ ”);
}
```

# Laços de repetição

# 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

- As três seções – inicialização, condição e incremento - devem ser separadas por ponto-e-vírgula (;)
- Quando condição se torna falsa, programa continua execução na sentença seguinte ao for.

# 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

- Diferente de *for* e de *while*, o comando *do-while* testa a *condição* no fim do laço.
  - O *comando* será executado, pelo menos, uma vez.
  - As chaves não são necessárias em *comandos* simples. Porém, evitam confusão com o *while*.
    - Boa prática de programação!

# O Comando do

- Exemplo:

```
scanf(“%d”, &i);
j = j - 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

# 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.

# Streams em C

- Stream (significa fluxo ou seqüência)
  - Interface (abstração) entre programador e dispositivo fornecida pelo sistema de Entrada/Saída em C.
  - Abstração = stream
  - Dispositivo = arquivo

# Streams em C

- Stream de texto
  - É uma seqüência de caracteres.
  - Caracter de nova linha & traduções.

- Stream binária
  - É um seqüência de bytes.
  - Número de bytes escritos/lidos é o mesmo encontrado no dispositivo.

# 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

# Ponteiros

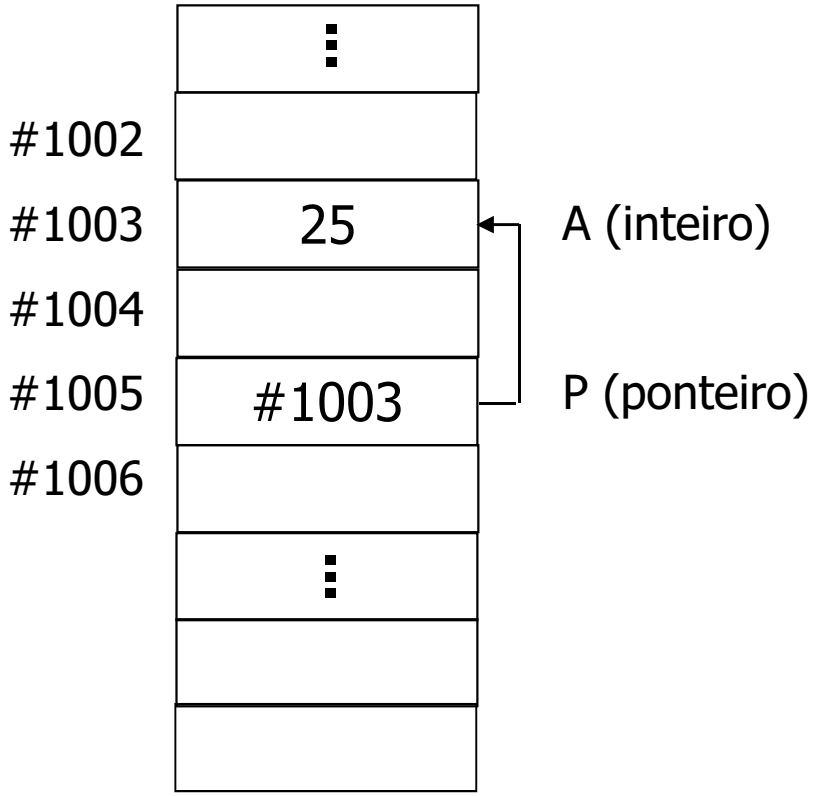
- Por quê ponteiros são importantes?
  - Ponteiros fornecem meios para funções modificarem seus argumentos.
  - São usados para alocar memória dinamicamente.
  - Pode aumentar a eficiência de certas rotinas.

- **O entendimento e uso correto de ponteiros em C é crítico!**

# 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**.

# Declaração de ponteiros em C

- Exemplo:

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

# Operadores de Ponteiros

- Os operadores de ponteiros são:
  &
  *

- Precedência:

**Maior precedência**

( ) [ ] ++ (pós) -- (pós)

! ~ ++(pré) -- (pré) . - (unário) (cast) *(unário) &(unário) sizeof

* / %

+ -

<< >>

...

**Menor precedência**

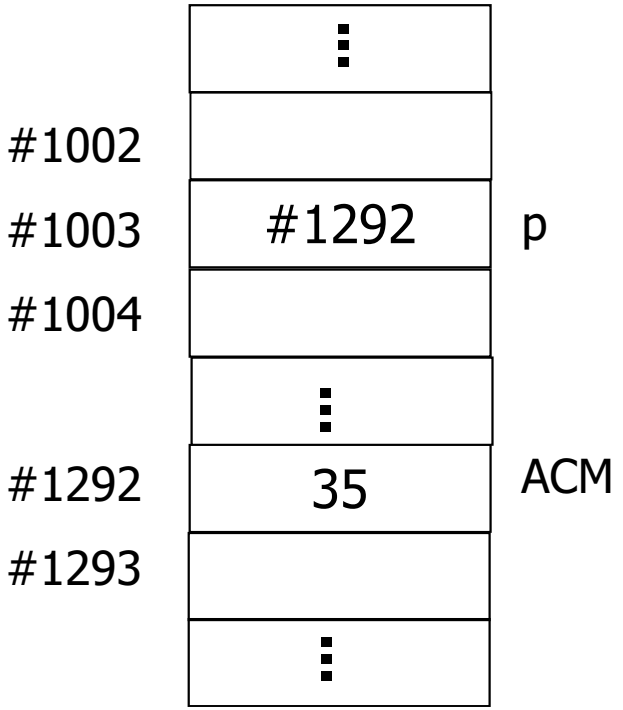
# O Operador &

- &: 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

# Atenção

- Tecnicamente, qualquer ponteiro pode apontar para qualquer posição de memória.
  - O tipo do ponteiro pode ser importante. Por exemplo, em aritmética de ponteiros.

```
float x = 3.5, y = 0.96;
int *p;
p = &x;   /* p (int *) aponta para um float!!!*/
y = *p;   /*compila, mas está errado*/
```

# Observações

- `NULL`.
  - Valor nulo de ponteiro.
- Ponteiros `NULL`.
  - Ponteiro para um tipo específico que, por ora, aponta para nada. Não tem valor.
- Ponteiros `void`.
  - Ponteiros genéricos, apontam para qualquer tipo de dado.

# Expressões com Ponteiros

- Somente 3 operações são possíveis:
  - Atribuição de ponteiros.
  - Aritmética de ponteiros.
  - Comparação de ponteiros.

# Atribuição

- A atribuição entre ponteiros é igual a qualquer variável.
  - Exemplo:

```
int x=8, *p1, *p2;
p1 = &x;
p2 = p1;
printf (“%d %d”, p1, *p1);
printf (“%d %d”, p2, *p2);
```

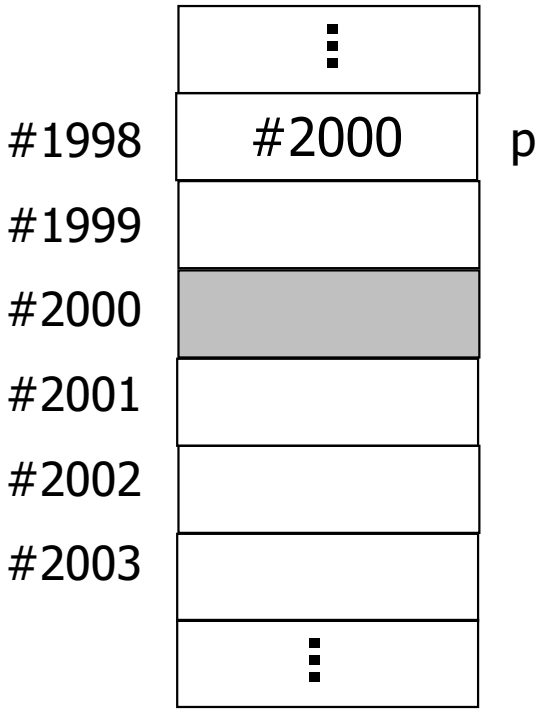
  - Cuidado com atribuições entre ponteiros de tipos diferentes!

# Aritmética de Ponteiros

- 2 operações apenas: adição e subtração.
- Cada incremento (ou decremento) aumenta (ou diminui) o valor do ponteiro em quantidade de bytes correspondente ao tipo base.

# Aritmética de Ponteiros

- Aritmética de Ponteiros
- Ex.:




```
inteiro de 2 bytes:
int *p;
...
p++;
/*p--;*/

p = p + 2; /*válido*/
```

# Comparação de Ponteiros

- Comparação entre ponteiros é possível.
  - Compara-se posições de memória.
  - Exemplo:

```
int *p1, *p2, x =10;
p1 = &x;
p2 = p1;
if (p1 == p2) ...
   ou
if (p1) ... /* p == 0: ponteiro nulo*/
```

# Indireção Múltipla

- Também conhecida como ponteiros para ponteiros.
- float **ptr;
  - ptr é um ponteiro para um ponteiro float.

```
float x = 5.7, *p, **q;
p = &x;
q = &p;                 /* q = p; é errado! */
printf (“%d”,**q);
```

- Indireção pode ser levada a qualquer dimensão
  - `int *********ptr;`

# 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

# Funções

- O coração e a alma da programação em C
- Todas as funções estão no mesmo nível.
  - Não podem ser aninhadas.
- Precisam ser declaradas antes de serem usadas.

# Funções

- A execução de um programa começa pela função *main()*, a qual pode chamar outras funções.
  - Funções definidas pelo programador.
  - Funções de biblioteca: printf(), scanf(), sqrt(), … .
- Funções trabalham com variáveis do programa. Regras de escopo determinam onde cada uma dessas variáveis está disponível.

# 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.

# Forma Geral

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

- **tipo:** especifica o tipo do valor que a função deve retornar (*return*).
  - Pode ser qualquer tipo válido.
  - Se a função não retorna valores, seu tipo é **void**.
  - Se o tipo não é especificado, tipo *default* é **int**.
  - Se necessário, o valor retornado será convertido para o tipo da função.

# Forma Geral

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

- **nome_da_função**: pode ser qualquer identificador válido em C.

# Forma Geral

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

- **lista parâmetros**: é uma lista de nomes de variáveis separadas por vírgula e seus tipos associados.
  - Cada variável **deve** ter seu tipo associado.
  - f (tipo1 var1, tipo2 var2, …, tipon varn);
- Os parâmetros recebem os valores dos **argumentos** quando a função é chamada.
- Função sem parâmetros -> lista de parâmetros vazia.
  - Obs.: Mesmo assim os parênteses são necessários.

# Forma Geral

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

- **declarações**:
  - toda variável declarada dentro do corpo de um função é dita **variável local** dessa função.
  - uma variável declarada externamente a uma função é dita **global**.

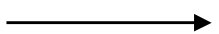
```
#include <stdio.h>

int func (int x);
int w;

int main (void){
  int i, j, w;                  /* i é local p/ main*/
  ...
  w = (i * 3.5)/ i*i;
  j = func(w);                  /* w é local*/
}

int func (int x) {
  char c; int y;                /*x, c e y são locais para func*/

  c = scanf(“%c”, &c);
  y = c + x + w;                /*w é global*/
  return (y);
}
```

# Forma Geral

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

- **sentenças**: além de declarações, o corpo de uma função pode conter qualquer sentença válida em C.
  - Sentenças podem ser expressões, comandos de controle de fluxo, atribuições, chamadas a funções, etc.

```
int func (int x, char y, float z) {
    char c;
    x = y + z;
    if (x > 10)
            func2();
    else{
            scanf("%c", &c);
            x = func3(c);
    }
    return (x);
}
```



sentenças

# Regras de Escopo

- Regra geral: identificadores de variáveis são acessíveis somente dentro do bloco no qual eles foram declarados.

- Função é um caso de bloco lógico.
- Em C, o código de uma função é privativo àquela função. Não pode ser acessado por nenhum comando em outra função, exceto por uma chamada à função.

# Regras de Escopo

- E quando programadores usam o mesmo identificador em diferentes declarações?
- A qual variável o identificador se refere?
  1. à variável declarada no mesmo bloco lógico, se existir.
  2. à variável global.

# Regras de Escopo

```
{
  int a = 2;
  printf(“%d”, a);          /*imprime 2*/
  {
       int a = 5;
       printf(“%d”, a);     /*imprime 5*/
  }
  printf(“%d”, ++a);        /*imprime 3*/
}
```

# Regras de Escopo

```
#include <stdio.h>

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

int w = 0;  /*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*/
   j = f2(y);
   return (j);
}
```

```
int f2(int y){

        int x;

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

# Argumentos e Parâmetros

- Valores passados à função na sua chamada.
- Uma função pode ou não ter argumentos.
- Se uma função usa argumentos, ela deve declarar variáveis que recebam os valores dos argumentos: os parâmetros.

# 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.

# Argumentos e Parâmetros

- Os parâmetros são variáveis locais.
- Como qualquer variável local, são criadas na entrada da função e são destruídas na saída da mesma.
- Cabe ao programador assegurar a compatibilidade entre argumentos e parâmetros.
  - Nem sempre o compilador gera mensagem de erro!

# 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);
   i = func(2, 'C', 2.35);
}
```

# 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*/
```

# Retorno de Valores

- Funções devolvem valores, mas não é necessário usar esse valor.
  - printf () devolve o número de caracteres escritos.

```
x = printf("caracteres escritos: ");
printf("%d", x);
```

# 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;`

# O Comando return

- Se o comando return contém uma expressão, o valor da expressão é passado ao ponto em que a função foi chamada.

- Se necessário, o valor é convertido para o tipo de retorno especificado na definição da função.

# O Comando return

- Pode haver zero ou mais comandos *return* em uma função.
- Somente um é executado.
- Se não houver um comando return, o controle volta ao ponto de chamada da função quando o fecha-chaves "}" da função é executado.

# O Comando return

- Todas as funções, exceto as do tipo void, retornam valores.
  - Esse valor deve estar especificado, explicitamente, no comando return.
  - Se não houver um comando return, o valor de retorno da função é indefinido.

# O Comando return

```
int func (int a, int b){
     if (a > b)
       return(a);
     else
       return(b);
}
int func (int a, int b){
     return(a+b);
     return(b*a);
}
int func (int a, int b){
     b = b * a;
}
```

# Passando Ponteiros como Argumentos

```
#include <stdio.h>
void f (int *p);

int main(void) {
   int x=10, *p;
   p = &x;
   f(p);
   printf("x = %d\n",x);
   char c = getch();
   return(0);
}

void f (int *p) {
   *p += 2;
}
```

**equivalente**

```
int x =10;
f(&x);
```

# Retornando Ponteiros

- Para retornar um ponteiro, a função deve ser declarada como tendo tipo de retorno ponteiro:

```
int *mod(int a, int b) {
  int *p, m;
  p = &m;
  m = a % b;
  return(p);
}
```

# Retornando Ponteiros

```
#include <stdio.h>
char *match(char c, char *s);

int main (void){
   char s[80], *p, ch;
   gets(s);
   ch = getch();
   p = match(ch, s);
   if (p)
         printf (“%s ”, p);
   else
         printf(“não encontrei”);
}

char *match(char c, char *s){
   while (c != *s && *s) s++;
   return(s);
}
```

# Argumentos para main()

- `main()` recebe argumentos via linha de comando.
- Exemplo:

```
> gcc nome_prog.c -o nome_prog
```

- Argumentos de `main()`:
  - **`argc`**: inteiro que contém o número de argumentos da linha de comando.
  - **`argv`**: vetor de ponteiros para caracteres. Cada posição do vetor (string) é um argumento.
  - O primeiro argumento é o nome do programa.

# Argumentos para main()

```
include <stdio.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
        printf(“Uso: nome <seu nome>”);
        exit(1);
  }
  printf(“Olá %s!”, argv[1]);
  char c = getchar();
  return(0);
}
```

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

int main (int argc, char *argv[]) {
  float op1, op3;
  op1 = atof(argv[1]);
  op3 = atof(argv[3]);

  switch(argv[2][0]) {
     case '+': printf("%f\n",op1+op3); break;
     case '-': printf("%f\n",op1-op3); break;
     case '*': printf("%f\n",op1*op3); break;
     case '/': printf("%f\n",op1/op3); break;
  }

  getchar();
  return(0);
}
```

# O que main() devolve?

- Devolve um valor inteiro para o processo que chamou seu programa.
- Geralmente o chamador é o sistema operacional.
- Se não está explícito, o valor devolvido é tecnicamente indefinido.
- Pode ser declarada como void.
  - Alguns compiladores geram uma advertência.

# Vetores

# Vetores

- Um vetor é uma coleção de **variáveis do mesmo tipo** referenciadas por um nome comum.
- Uma variável do vetor é chamada de **elemento** do vetor.
- Os elementos de um vetor podem ser acessados, individualmente, por meio de índices.

# Vetores

- Em C, os elementos de um vetor ocupam posições contíguas na memória.
- Em C, um vetor é considerado uma matriz - ou array - unidimensional.
- Em C, arrays e ponteiros são assuntos intimamente relacionados.

# Declaração

- 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];
```

- Considere `int A[10], c=8;`

1. A[1.5] – não é permitido.
2. A[‘f’] – é permitido.
3. A[(1 >= 10)] – é permitido.
4. A[c] – é permitido.
5. A[(c+2)*3] – é permitido.
6. A[-1] – não é permitido.

# 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};`

# Indexação

- Em C, todo vetor começa pelo índice 0.
  - `char p[10];`
  - é um vetor de caracteres que possui 10 elementos, de p[0] a p[9].

- Portanto, o *i*-ésimo elemento de um vetor pode ser acessado usando-se i-1 como índice.

```
int A[10];
```

Para acessar o 1º elemento: `A[0]`

```
A[0] = 387;
```

Para acessar o 8º elemento: `A[7]`

```
A[7] = 126;
```

# Indexação

- O quê acontecerá com o trecho de código abaixo?

```
int c[10], i;
for (i = 0; i < 100; i++)
     c[i] = i;
```

# Indexação

- O código anterior compila sem erros, mas está **incorreto**.
- C não possui verificação de limites de arrays. É tarefa do programador fazer a verificação dos limites onde for necessário!
- No exemplo anterior ocorrerá invasão de memória – escrevendo nos dados de outra variável ou mesmo na área de código do programa.

# 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] = {0, 1, 2, 3, 4};
int *p, i;
p = A;
```

`A[i]` é equivalente a `*(A + i)`

`p[i]` é equivalente a `*(p + i)`

# 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

- Uso mais comum de arrays unidimensionais: strings de caracteres.

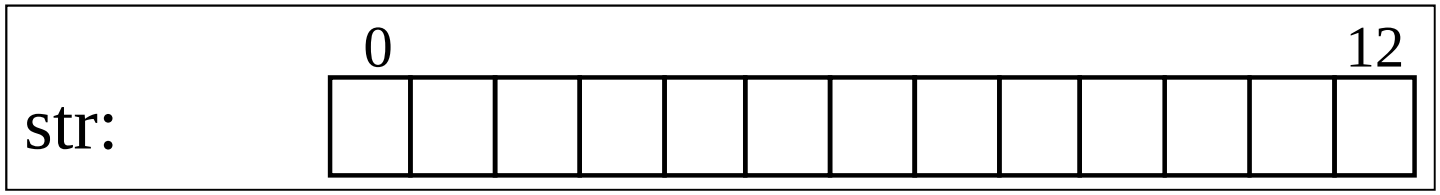
- C não possui o **tipo de dado** string. Permite apenas constantes string:
  - “isto é uma constante string”
  - Em constantes string não precisa adicionar o caracter nulo ‘\0’. O compilador faz isso automaticamente.

# 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'.

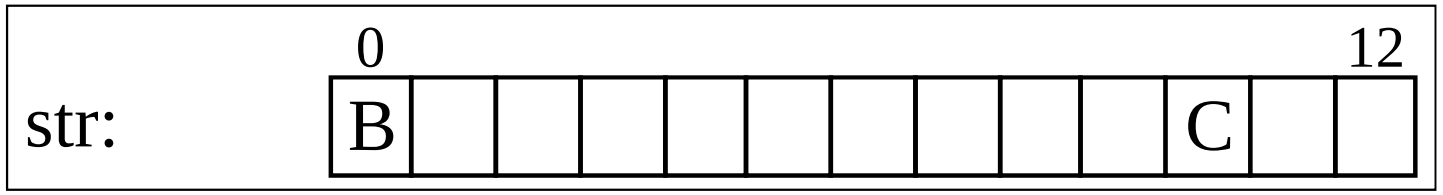


# Strings em C

- Os caracteres podem ser individualmente acessados por indexação:
- Ex: char str[13];

  str[0] = 'B';

  str[10] = 'C';



# Inicialização e Atribuições

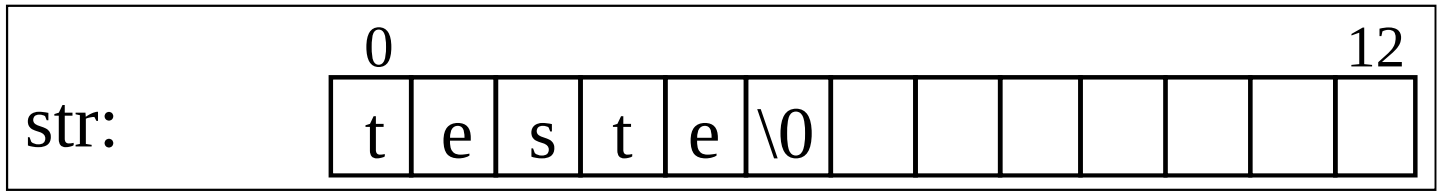
- `char str[13];`

- Atribuição errada:

  `str = “teste”; /*Erro!!!*/`

- Correta:

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



# Inicialização e Atribuições

```
char str[13], nome[13];
str[0] = 'a'; str[1] = '\0';
```

- atribuição: não se pode atribuir uma string a outra string:

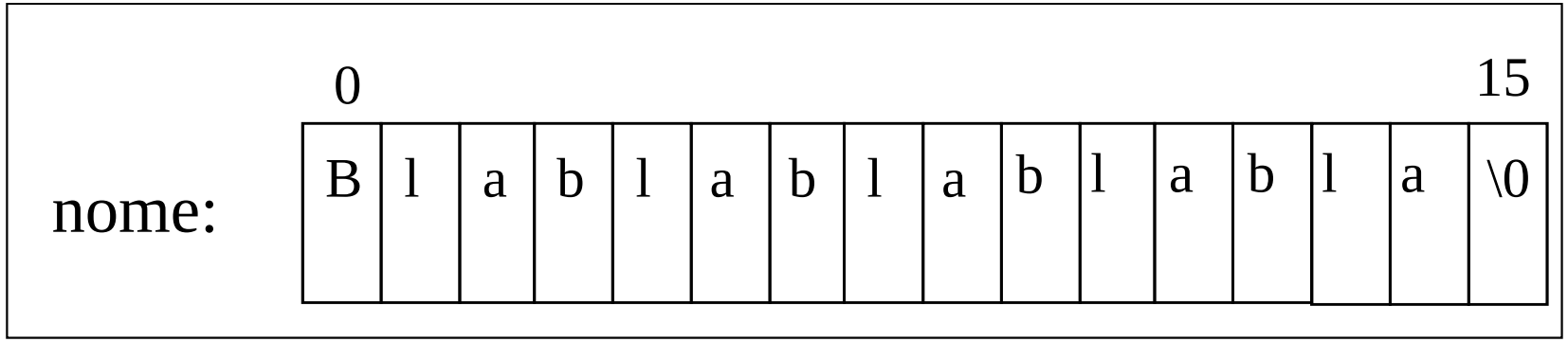
```
nome = str; /* erro */
```

- Para copiar uma string para outra:
  - Faz-se elemento a elemento ou
  - Usa-se funções da biblioteca string.h

# Inicialização na Declaração

```
char nome[16] = “Blablablablabla”;
```

- Define uma string “nome”, reserva para ela um espaço de memória de 16 (15 + ‘\0’) bytes e inicia-a com o texto indicado.



# Inicialização na Declaração

```
char nome[] = “Testando...”;
```

- Define uma string e reserva espaço suficiente para os caracteres presentes, inclusive para ‘\0’.

# Strings

Funções para manipulação de strings:

```
char str[80], str2[80];
```

- gets(str) – lê uma string do teclado e armazena em str.
- puts(str) – imprime o conteúdo de str no monitor.

# 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

- Uma matriz é uma variável composta homogênea multidimensional.
- Fornece meios para se armazenar dados em modo tabular.
- C permite que arrays sejam declarados e indexados de modo multimensional.
- Aplicações: tabelas, matrizes matemáticas, composição de variáveis.
  - Usos em problemas matemáticos, de engenharia e de computação em geral.

# Declaração

- 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];
```

# Matrizes Bidimensionais

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |

```
int mat[3][4];
```

- Conceito de linha-coluna.

```
/* 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);
...
```

# Matrizes Multidimensionais

- C permite matrizes com mais de duas dimensões.
  `tipo nome [tam 1] [tam 2] [tam 3] ... [tam n];`
- O limite, se existente, é determinado pelo compilador.

- Matrizes de 3 ou mais dimensões são pouco usadas.
  - Quantidade de memória.
  - Ex.: float mat[10][3][5][10];
  - tamanho = 10 * 3 * 5 * 10 * sizeof(float) = 12000 bytes (assumindo float de 8 bytes).

- Problema:
  - Espaço não utilizado nas matrizes (matrizes esparsas).
  - Para esse tipo de matriz usa-se alocação dinâmica.

# Inicialização de Matrizes

- C permite inicializar matrizes no momento da declaração:

```
int a[5] = {1, 2, 3, 4, 5};
```

```
char str[4] = “USP”;
```

equivale a

```
char str[4] = {‘U’,’S’,’P’,’\0’};
```

# Inicialização de Matrizes

- Inicialização de matrizes multidimensionais

```
int mat[5][2] = {1, 3, 5, 8, 3, 1, 2, 2, 7, 4};
```

ou

```
int mat[5][2] = {
                   1, 3,
                   5, 8,
                   3, 1,
                   2, 2,
                   7, 4
                 };
```

# Inicialização de Matrizes

- Inicialização de matrizes não dimensionadas: somente a dimensão mais à esquerda não precisa ser especificada.

```
int mat[][2] = {
                    1, 3,
                    5, 8,
                    3, 1,
                    2, 2,
                    7, 4
                };
```

# Arrays como Parâmetros

- Arrays são exceções à passagem default em C.
- Arrays como argumentos sempre passam seu endereço (o do primeiro elemento) ao parâmetro.

- Modos de passar um array como parâmetro:

```
int t[10];
display(t);

void display (int num[10]);
void display (int num[]);
void display (int *num);
```

# Arrays como Parâmetros

- Modos de passar um array como parâmetro:

```
int t[10][20];
display(t);

void display (int num[10][20]);
void display (int num[][20]);
void display (int (*num)[20]);
```

# 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;
```

- Definição x Declaração
- O nome da estrutura ou a lista de variáveis podem ser omitidos. Não ambos!

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

# 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;
}
```

# Atribuição

- Padrão ANSI estabelece que uma estrutura poder ser atribuída à outra estrutura de mesmo tipo.

- Exemplo:

```
struct {
  int x;
  char y;
} a, b;

int main (void) {
  a.x = 10;
  b = a;
  printf("%d", b.x);
}
```

# Inicialização

```
struct S {
  int x;
  char y;
};

int main (void) {
  struct S  b, a = {10,'a'};
  b = a;
  printf("%d %c", b.x, b.y);
}
```

# Operações

Operações com campos de estruturas devem ser feitas membro a membro:

```
struct S {
  int x;
  char y;
};

int main (void){
  struct S  b, a = {10,'a'};
  b.x = a.x * 2;
}
```

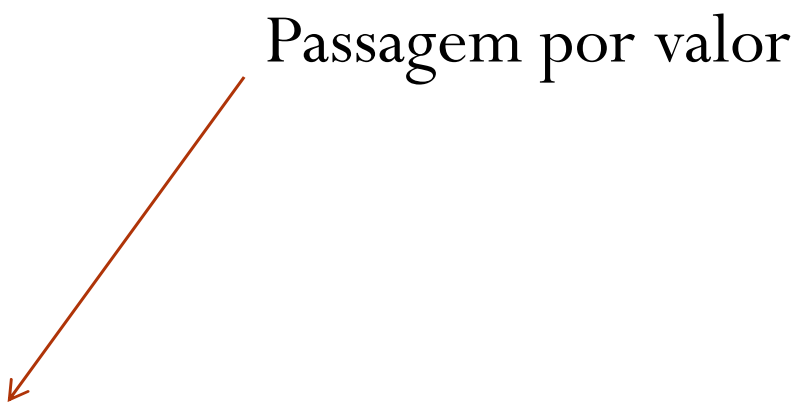
# Encadeamento

- Estruturas podem ser autoreferenciadas:

```
struct ponto{
  int x;
  int y;
  struct ponto *p;
};
```

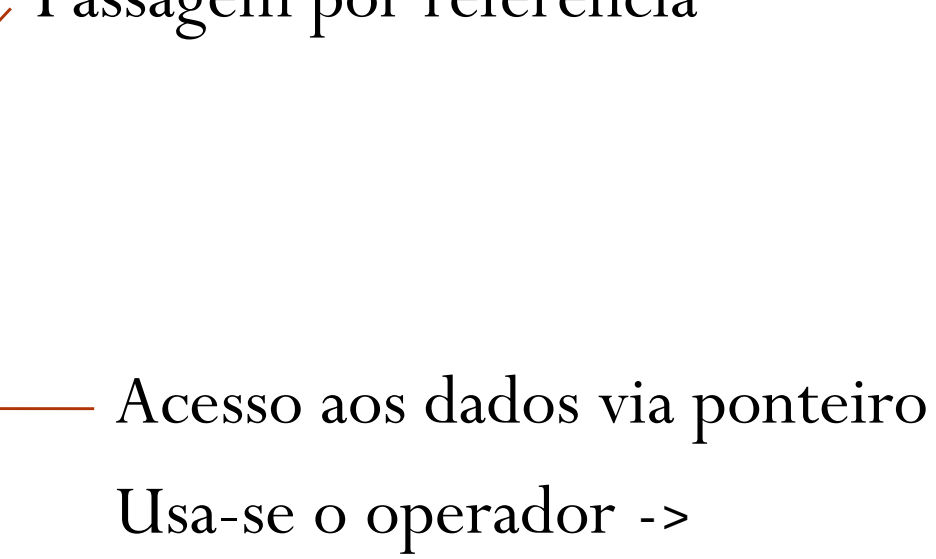
# Estruturas como Parâmetros

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

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

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



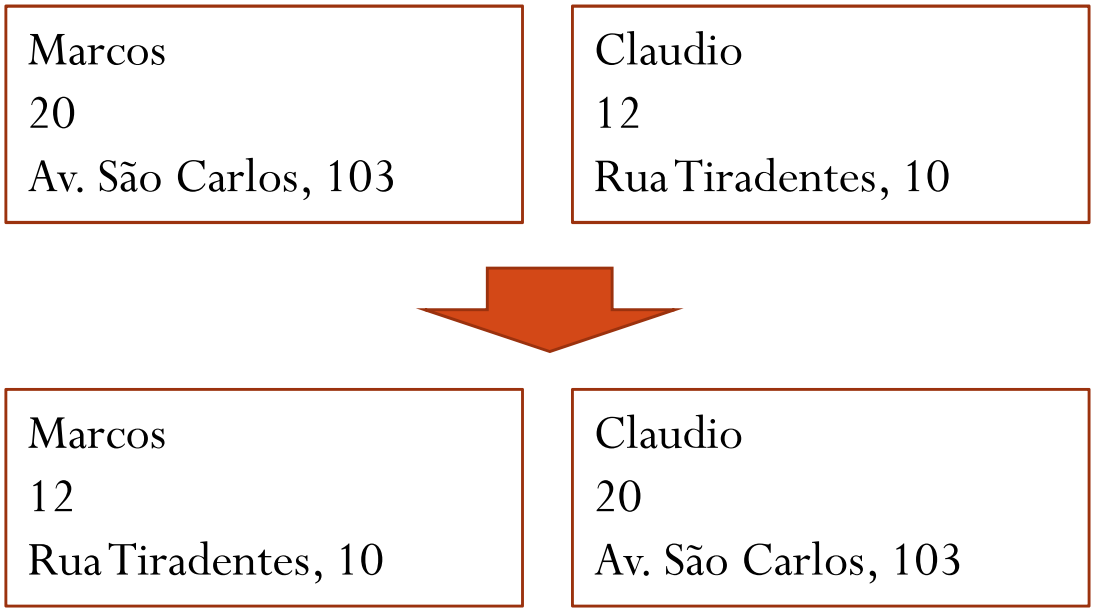
# Estruturas como Parâmetros

```
...
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);
}
...
```



# 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

# 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, 4); /*supondo inteiro de 4 bytes*/
```

ou

```
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) ) {
  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.

# Exercício 5

- Um conjunto é uma coleção de membros (ou elementos); cada membro ou é um conjunto ou um elemento primitivo chamado de átomo. No caso deste exercício, considera-se conjuntos formados por números inteiros.

- Todos os membros são diferentes: nenhum conjunto contém 2 cópias do mesmo elemento

- Ex:

  {1,4} ok

  {1,4,1} não ok

# Exercício 5 – cont.

- **Operações básicas**
  - Se A e B são conjuntos, então $A \cup B$ é o conjunto de elementos que são membros de A ou de B ou de ambos
  - Se A e B são conjuntos, então $A \cap B$ é o conjunto de elementos que estão em A e em B
  - Se A e B são conjuntos, então $A - B$ é o conjunto de elementos em A que não estão em B
  - Exemplo: $A = \{a,b,c\}$ $B = \{b,d\}$
    $A \cup B = \{a,b,c,d\}$
    $A \cap B = \{b\}$
    $A - B = \{a,c\}$

# Exercício 5 – cont.

- Decida como representar/implementar em C um conjunto

# Exercício 5 – cont.

- Decida como representar/implementar em C um conjunto

```
#define N 100
//conjunto que tem números de 1 a 100

int conjunto[N];
//conjunto[i-1]=1 se i está no conjunto; 0, caso contrário
```

# Exercício 5 – cont.

- Implemente em C uma função para cada uma das operações básicas vistas
  - União(A,B,C): toma os argumentos A e B que são conjuntos e atribui $A \cup B$ à variável C
  - Intersecção(A,B,C): toma os argumentos A e B que são conjuntos e atribui $A \cap B$ à variável C
  - Diferença(A,B,C): toma os argumentos A e B que são conjuntos e atribui A - B à variável C

# Exercício 5 – cont.

- Implemente as demais funções abaixo
  - `membro(x,A)`: toma o conjunto A e o objeto x cujo tipo é o tipo do elemento de A e retorna um valor booleano – true se $x \in A$ e false caso contrário
  - `cria_conj_vazio(A)`: faz o conjunto vazio ser o valor do conjunto A

# Exercício 5 – cont.

- Implemente as demais funções abaixo
  - Inserir(x,A): toma o conjunto A e o objeto x cujo tipo é o tipo do elemento de A e faz x um membro de A. O novo valor de A é $A \cup \{x\}$. Se x já é um membro de A, então a operação não modifica A
  - Remover(x,A): remove o objeto x, cujo tipo é o tipo do elemento de A, de A. O novo valor de $A = A - \{x\}$. Se x não pertence a A então a operação não altera A

# Exercício 5 – cont.

- Implemente as demais funções abaixo
  - Atribui(A,B): seta o valor da variável conjunto A igual ao valor da variável conjunto B
  - Minimo(A): retorna o valor mínimo do conjunto A.
  - Maximo(A): retorna o valor máximo do conjunto A.
  - Igual(A,B): retorna true se e somente se os conjuntos A e B contém os mesmos elementos

# Créditos

Aula baseada no material do Prof. Rudinei Goularte

Exercícios desenvolvidos pelo Prof. Thiago Pardo