



Estruturas de Dados  
Aula 2: Estruturas Estáticas

02/03/2011



## Tipos Básicos

- Quantos valores distintos podemos representar com o tipo char?

Tipo	Tamanho	Menor valor	Maior valor
char	1 byte	-128	+127
unsigned char	1 byte	0	+255
short int (short)	2 bytes	-32.768	+32.767
unsigned short int	2 bytes	0	+65.535
int (*)	4 bytes	-2.147.483.648	+2.147.483.647
long int (long)	4 bytes	-2.147.483.648	+2.147.483.647
unsigned long int	4 bytes	0	+4.294.967.295
float	4 bytes	$-10^{38}$	$+10^{38}$
double	8 bytes	$-10^{308}$	$+10^{308}$

# Operadores de Incremento e Decremento



- ++ e --
  - Incrementa ou decrementa o valor de uma variável de uma unidade
  - O incremento/decremento pode ser antes ou depois da variável ser usada
    - N++, incrementa n depois de ser usado
    - ++N, incrementa n antes de ser usado

```
n = 5;
x = n++;          /* x recebe 5; n é incrementada para 6 */
x = ++n;         /* n é incrementada para 6; x recebe 6 */
a = 3;
b = a++ * 2;     /* b termina com o valor 6 e a com o valor 4 */
```



## Operador Sizeof

- Retorna o número de bytes ocupado por um determinado tipo
  - `int a = sizeof (float);`
  - Armazena 4 na variável a pois um float ocupa 4 bytes de memória
- Também pode ser usado em uma variável, retornando o número de bytes ocupado por esta variável



## Conversão de tipo

- Ou “Type Casting”
  - Conversão é feita automaticamente pelo C na avaliação de expressões
  - Pode ser requisitado explicitamente pelo programador

```
float f;          /* valor 3 é convertido automaticamente para “float”      */
float f = 3;     /* ou seja, passa a valer 3.0F, antes de ser atribuído a f          */

int g, h;        /* 3.5 é convertido (e arredondado) para “int”                        */
g = (int) 3.5;   /* antes de ser atribuído à variável g                               */
h = (int) 3.5 % 2 /* e antes de aplicar o operador módulo “%”                          */
```



## Entrada e saída

- São feitas com uso de funções
- Função printf
  - *printf (formato, lista de constantes/variáveis/expr...);*

*%c especifica um char*

*%d especifica um int*

*%u especifica um unsigned int*

*%f especifica um double (ou float)*

*%e especifica um double (ou float) no formato científico*

*%g especifica um double (ou float) no formato mais apropriado (%f ou %e)*

*%s especifica uma cadeia de caracteres*



## Entrada e saída

```
printf ("Inteiro = %d Real = %g", 33, 5.3);
```

com saída:

Inteiro = 33 Real = 5.3

- Caracteres de escape

`\n`      *caractere de nova linha*

`\t`      *caractere de tabulação*

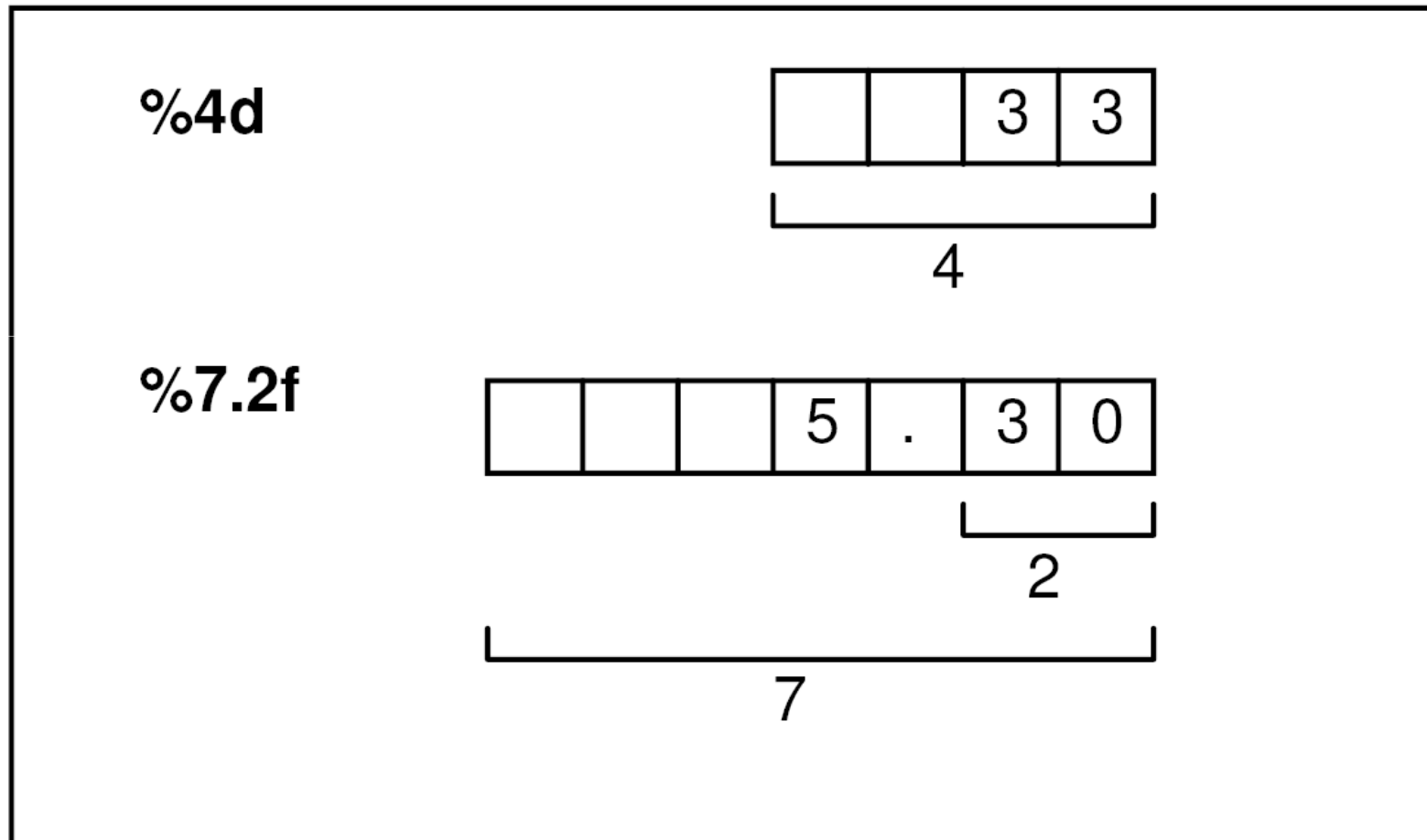
`\r`      *caractere de retrocesso*

`\"`      *caractere “*

`\\`      *caractere \*

# Entrada e saída

- Especificação do tamanho do campo







## Entrada e saída

- scanf (formato, lista de endereços das variáveis...)

```
int n;  
scanf ("%d", &n);
```

`%c`

*especifica um char*

`%d`

*especifica um int*

`%u`

*especifica um unsigned int*

`%f, %e, %g`

*especificam um float*

`%lf, %le, %lg`

*especificam um double*

`%s`

*especifica uma cadeia de caracteres*

# Funções



- Comando de definição de uma função

```
Tipo_retorno nome_funcao (lista de parametros)
{
    Corpo da função
}
```

# Definição de Funções



```
/* programa que lê um número e imprime seu fatorial (versão 2) */
```

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

```
int fat (int n);
```

```
int main (void)
```

```
{ int n, r;
```

```
printf("Digite um número nao negativo:");
```

```
scanf("%d", &n);
```

```
r = fat(n);
```

```
printf("Fatorial = %d\n", r);
```

```
return 0;
```

```
}
```

“protótipo” da função:  
deve ser incluído antes  
da função ser chamada

chamada da função

“main” retorna um inteiro:  
0 : execução OK  
≠ 0 : execução →OK

```
/* função para calcular o valor do fatorial */
```

```
int fat (int n)
```

```
{ int i;
```

```
int f = 1;
```

```
for (i = 1; i <= n; i++)
```

```
    f *= i;
```

```
return f;
```

```
}
```

declaração da função:  
indica o tipo da saída e  
o tipo e nome das entradas

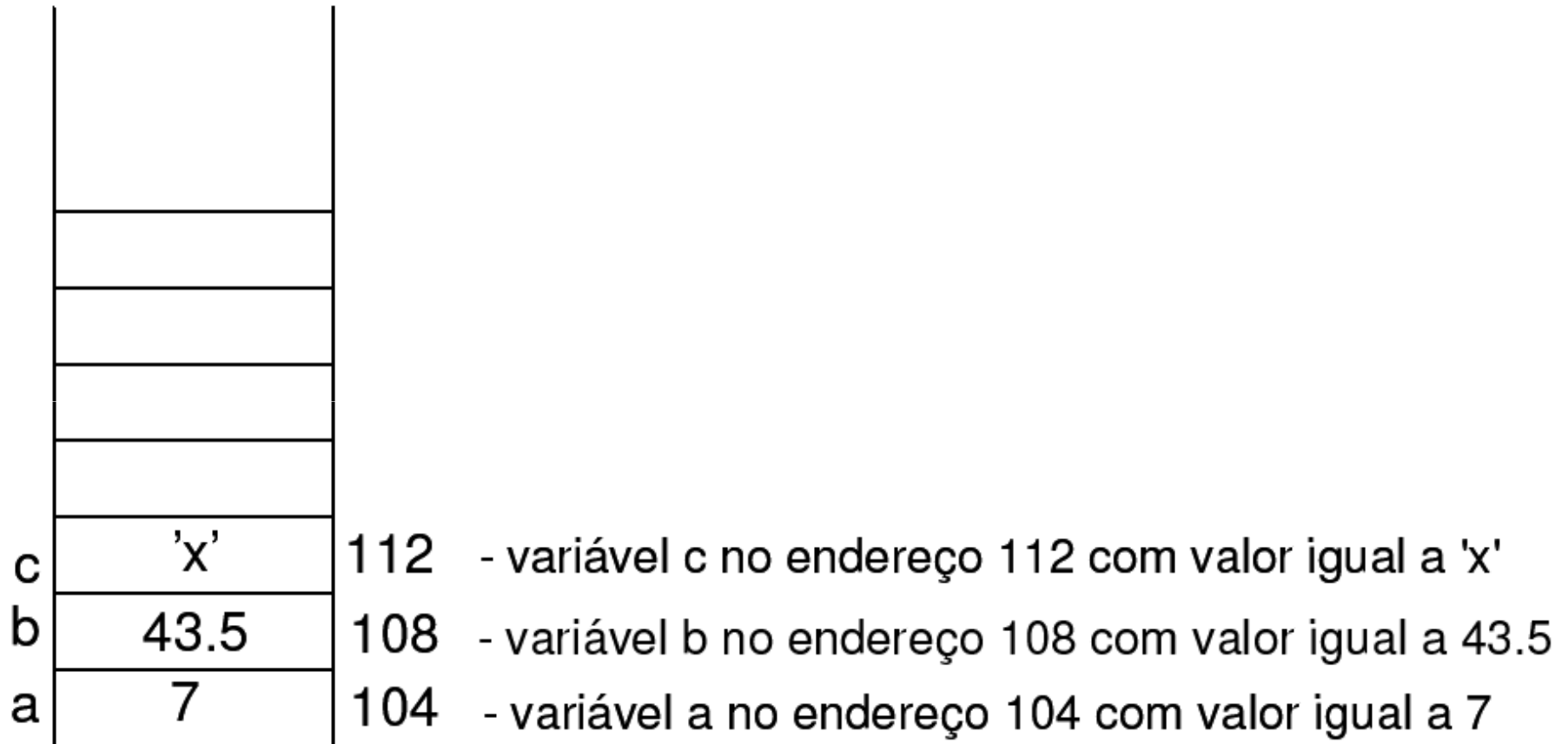
retorna o valor da função



## Pilha de Execução

- Variáveis locais têm escopo local
- Funções são independentes entre si
- Transferência de dados entre funções através de
  - Passagem de parâmetros
  - Valor de retorno
- Parâmetros em C são passados **por valor**
- **Pilha de Execução:** Coordena comunicação entre a função que chama e a função chamada
  - Permite passagem de parâmetros e valores de retorno

# Esquema representativo da memória



## Exemplo fat (5)

```
/* programa que lê um numero e imprime seu fatorial (versão 3) */
```

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

```
int fat (int n);
```

```
int main (void)
```

```
{ int n = 5;
```

```
  int r;
```

```
  r = fat ( n );
```

```
  printf("Fatorial de %d = %d \n", n, r);
```

```
  return 0;
```

```
}
```

```
int fat (int n)
```

```
{ int f = 1;
```

```
  while (n != 0) {
```

```
    f *= n;
```

```
    n--;
```

```
  }
```

```
  return f;
```

```
}
```

← declaração das variáveis *n* e *r*,  
locais à função *main*

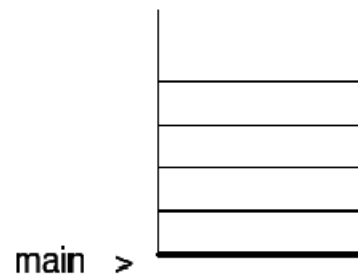
← declaração das variáveis *n* e *f*,  
locais à função *fat*

← alteração no valor de *n* em *fat*  
não altera o valor de *n* em *main*

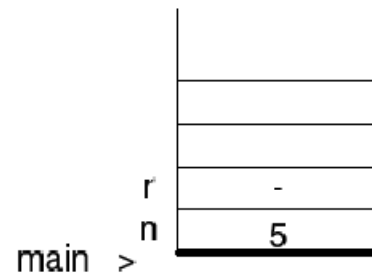
# Pilha de execução



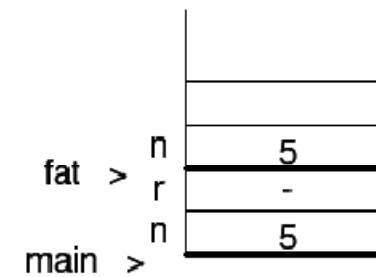
1 - Início do programa: pilha vazia



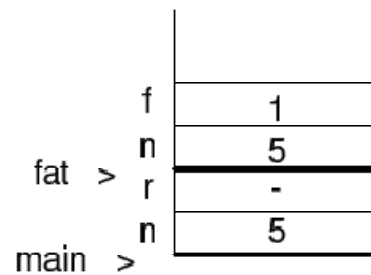
2 - Declaração das variáveis: n, r



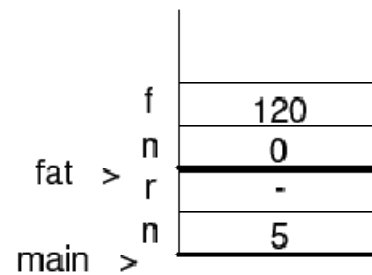
3 - Chamada da função : cópia do parâmetro



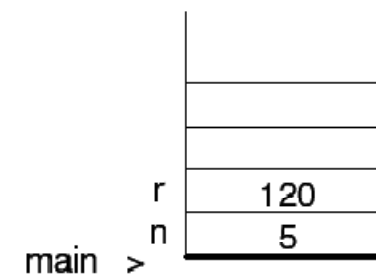
4 - Declaração da variável local: f



5 - Final do laço



6 - Retorno da função: desempilha





## Ponteiro de variáveis

- Pode ser necessário comunicar mais de um valor de retorno para função que chama
- Por exemplo, uma função que deve calcular a soma e o produto de dois números

```
#include <stdio.h>

void somaprod (int a, int b, int c, int d)
{
    c = a + b;
    d = a * b;
}

int main (void)
{
    int s, p;
    somaprod (3, 5, s, p);
    printf ("soma = %d produto =%d\n", s, p);
    return 0;
}
```





# Ponteiros

- Permitem manipulação direta de endereços de memória no C
- Variáveis do tipo ponteiro
  - Armazenam endereços de memória
  - É possível definir um ponteiro para cada tipo do C que seja capaz de armazenar endereços de memória em que existem valores do tipo correspondente
  - `int a;`
  - `int* p; // p armazena endereço de memória em que há valor inteiro`



## Operadores de ponteiros

- Operador & (“endereço de”)
  - Aplicado a variáveis, retorna o endereço da posição de memória reservada para variável
- Operador \* (“conteúdo de”)
  - Aplicado a ponteiros, acessa o conteúdo de memória do endereço armazenado pela variável ponteiro

# Exemplo

- `int a; int* p; int c;`

```
/* a recebe o valor 5 */
a = 5;
```

c	-	112
p	-	108
a	5	104

```
/* p recebe o endereço de a
ou seja, p aponta para a */
p = &a;
```

c	-	112
p	104	108
a	5	104

```
/* posição de memória apontada por p
recebe 6 */
*p = 6;
```

c	-	112
p	104	108
a	6	104

```
/* c recebe o valor armazenado
na posição de memória apontada por p */
c = *p;
```

c	6	112
p	104	108
a	6	104



## Exemplos

```
int main (void)
{
    int a;
    int *p;
    p = &a;
    *p = 2;
    printf (" %d ", a);
    return;
}
```

Imprime o valor 2



## Exemplos

```
int main (void)
{
    int a, b, *p;
    a = 2;
    *p = 3;
    b = a + (*p);
    printf (" %d ", b);
    return 0;
}
```

ERRO!



## Passando ponteiros para função

- Ponteiros permitem modificar o valor das variáveis indiretamente
- Possível solução para passagem por ref!

```
void somaprod (int a, int b, int *p, int *q)
{ *p = a + b;
  *q = a * b;
}
```

```
int main (void)
{
    int s, p;
    somaprod (3, 5, &s, &p);
    printf ("soma = %d produto =%d\n", s, p);
    return 0;
}
```

# Exemplo

```

/* função troca */
#include <stdio.h>
void troca (int *px, int *py )
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int main ( void )
{
    int a = 5, b = 7;
    troca(&a, &b); /* passamos os endereços das variáveis */
    printf("%d %d \n", a, b);
    return 0;
}

```

# Exemplo



1 - Declaração das variáveis: a, b    2 - Chamada da função: passa endereços

		112
b	7	108
a	5	104
main	>	

			120
py	108		116
px	104		112
>b	7		108
a	5		104
troca			
main	>		

3 - Declaração da variável local: temp    4 - temp recebe conteúdo de px

			120
temp	-		116
py	108		112
px	104		108
>b	7		104
a	5		
troca			
main	>		

			120
temp	5		116
py	108		112
px	104		108
>b	7		104
a	5		
troca			
main	>		

5 - Conteúdo de px recebe conteúdo de py    6 - Conteúdo de py recebe temp

			120
temp	5		116
py	108		112
px	104		108
>b	7		104
a	7		
troca			
main	>		

			120
temp	5		116
py	108		112
px	104		108
>b	5		104
a	7		
troca			
main	>		





## Variáveis Globais

- Declaradas fora do escopo das funções
- São visíveis a todas as funções
- Existem enquanto o programa existir (não estão na pilha de execução)
- Utilização:
  - Devem ser usadas com critério
  - Pode criar muita dependência entre as funções
  - Dificulta o entendimento e o reuso de código

# Exemplo de Variáveis Globais



```
#include <stdio.h>

int s, p; /* variáveis globais */

void somaprod (int a, int b)
{
    s = a + b;
    p = a * b;
}

int main (void)
{
    int x, y;
    scanf("%d %d", &x, &y);
    somaprod(x,y);
    printf("Soma = %d produto = %d\n", s, p);
    return 0;
}
```



# Variáveis Estáticas

- Declaradas no escopo de funções
- Existem enquanto o programa existir (não estão na pilha de execução)
- Somente são visíveis dentro das funções nas quais são declaradas
- Utilização
  - Quando for necessário recuperar o valor de uma variável na execução passada da função



## Exemplo de variável estática

- Função que imprime números reais
  - Imprime um número por vez (máximo de 5 números por linha)

```
void imprime (float a)
{
    static int n=1;
    printf (" %f ", a);
    if ((n%5) == 0) printf ("\n");
    n++;
}
```

# Sobre variáveis estáticas e globais...



- Variáveis estáticas e globais são inicializadas com zero, quando não forem explicitamente inicializadas
- Variáveis globais estáticas
  - São visíveis para todas funções subsequentes
  - Não podem ser acessadas por funções de outros arquivos
- Funções estáticas
  - Não podem ser acessadas por funções de outros arquivos



# Pré-processador e Macros

- Código C antes de ser compilado é passado pelo pré-processador
- O Pré-processador
  - Reconhece diretivas
  - Altera o código e envia para o compilador
- Diretiva `#include`
  - O pré-processador substitui pelo corpo do arquivo especificado



## Pré-processor e Macros

- # include "nome\_do\_arquivo"
  - Procura o arquivo do diretório local
  - Caso não encontre, procura nos diretórios de include especificados para compilação
- # include <nome\_do\_arquivo>
  - Não procura no diretório local



# Pré-processador e Macros

- Constantes
  - #define PI 3.1415
- Macros
  - Definição com parâmetros
  - #define MAX (a,b) ((a)>(b)?(a):(b))
  - O pré-processador substituirá o trecho de código:

```
v = 4.5;  
c = MAX (v, 3.0);
```

- Por:

```
v = 4.5;  
c = ((v) > (3.0) ? (v): (3.0));
```