Estruturas de Dados

Módulo 4 – Funções

Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados*, Editora Campus (2004)

Capítulo 4 – Funções

Tópicos

- Definição de funções
- Pilha de execução
- Ponteiros
- Variáveis globais
- Variáveis estáticas
- Recursividade
- Pré-processador e macros

Definição de Funções

- Comando para definição de função:

```
tipo_retornado  nome_da_função  ( lista de parâmetros... )
{
   corpo da função
}
```

```c
/* 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;
}

/* 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;
}
```

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

chamada da função

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

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

retorna o valor da função

Definição de Funções

- Exercício:
  – implemente uma função para calcular
    o número de arranjos de n elementos, tomados k a k,
    dado pela fórmula:

$$a = \frac{n!}{(n-k)!}$$

Pilha de Execução

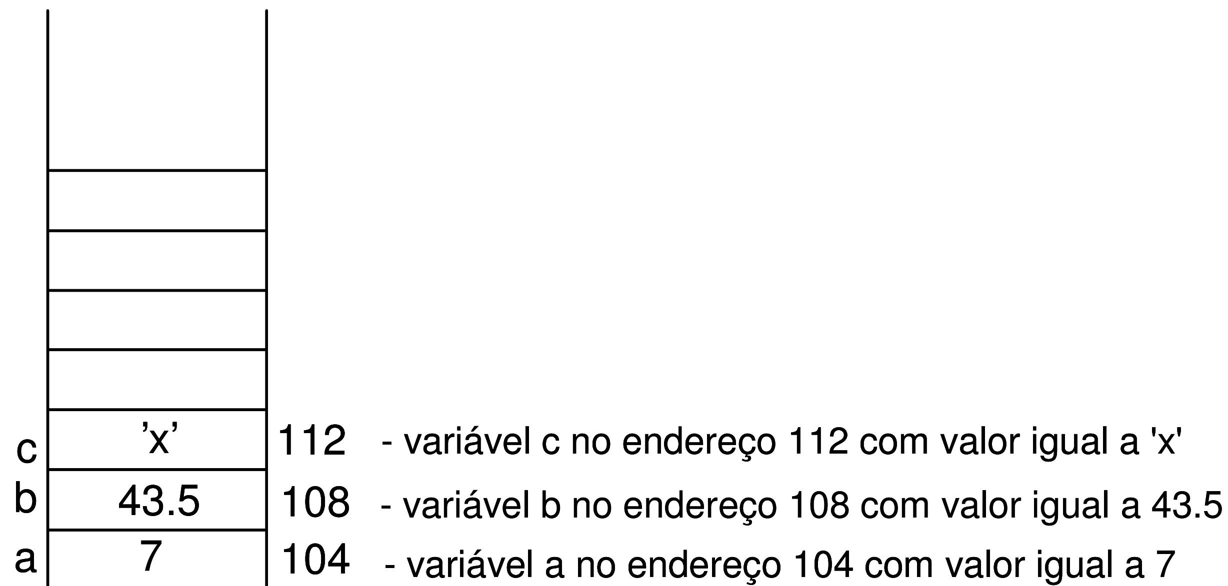
- Comunicação entre funções:
  - funções são independentes entre si
  - transferência de dados entre funções:
    - através dos parâmetros e do valor de retorno da função chamada
    - passagem de parâmetros é feita ***por valor***
  - variáveis locais a uma função:
    - definidas dentro do corpo da função (incluindo os parâmetros)
    - não existem fora da função
    - são criadas cada vez que a função é executada
    - deixam de existir quando a execução da função terminar

Pilha de Execução

- Comunicação entre funções (cont.):

  Pergunta: Como implementar a comunicação entre funções?

  Resposta: Através de uma pilha



Pilha de Execução

- Exemplo:
  - implementação da função *fat*
  - simulação da chamada *fat(5)*
    - a variável *n* possui valor 0 ao final da execução de *fat*, mas
    - o valor de *n* no programa principal ainda será 5

```c
/* 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*

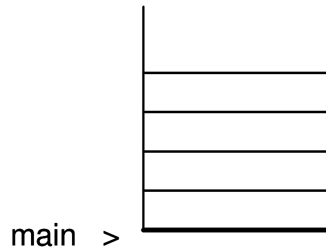
declaração das variáveis *n* e *f*, locais à função *fat*

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

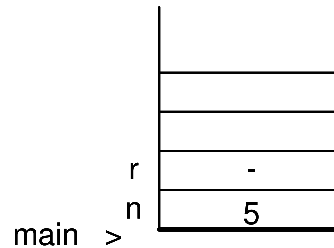
Pilha de Execução

- Exemplo (cont.): comportamento da pilha de execução

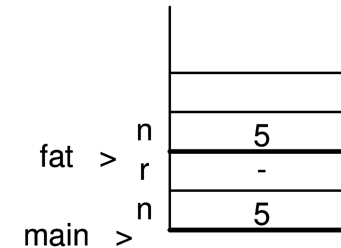
1 - Início do programa: pilha vazia



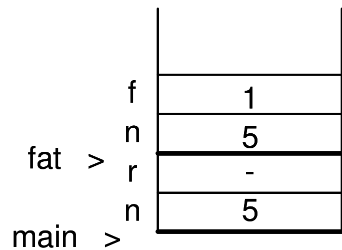
2 - Declaração das variáveis: n, r



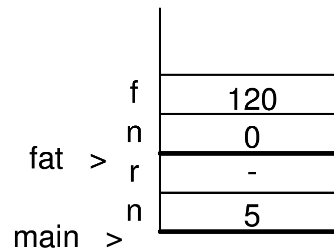
3 – Chamada da função : cópia do parâmetro



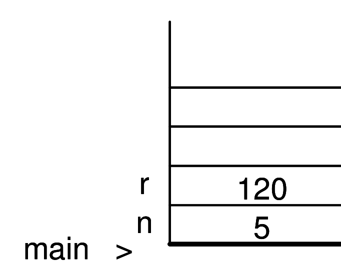
4 - Declaração da variável local: f



5 - Final do laço



6 - Retorno da função: desempilha

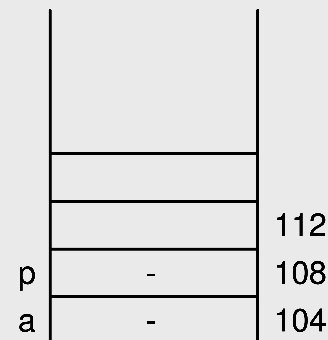


Ponteiros

- Variável do tipo ponteiro:
  - C permite o armazenamento e a manipulação de valores de endereços de memória
  - para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados

```
/*variável inteiro */
int a;

/*variável ponteiro p/ inteiro */
int *p;
```



Ponteiros

- Operador unário & (“endereço de”):
  - aplicado a variáveis, resulta no endereço da posição de memória reservada para a variável

- Operador unário * (“conteúdo de”):
  - aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro

```
/ * 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

Ponteiros

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

imprime o valor 2

Ponteiros

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

- erro na atribuição *p = 3
  - utiliza a memória apontada por p para armazenar o valor 3, sem que p tivesse sido inicializada, logo
  - armazena 3 num espaço de memória desconhecido

Ponteiros

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

- atribuição *p = 3
  - p aponta para c
  - atribuição armazena 3 no espaço de memória reservado para c

Ponteiros

- Passagem de ponteiros para funções:
  - função g chama função f
    - f não pode alterar diretamente valores de variáveis de g, porém
    - se g passar para f os valores dos endereços de memória
      onde as variáveis de g estão armazenadas,
      f pode alterar, indiretamente, os valores das variáveis de g

Ponteiros

```
/* 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;
}
```

1 -Declaração das variáveis: a, b

			112
	b	7	108
main >	a	5	104

2 - Chamada da função: passa endereços

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

3 - Declaração da variável local: temp

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

4 - temp recebe conteúdo de px

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

5 -Conteúdo de px recebe conteúdo de py

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

6 -Conteúdo de py recebe temp

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

Resumo

Operador unário & (“endereço de”)

```
p = &a;     /* p aponta para a */
```

Operador unário * (“conteúdo de”)

```
b = *p;     /* b recebe o valor armazenado na posição apontada por p */

*p = c;     /* posição apontada por p recebe o valor da variável c */
```

Variáveis Globais

- Variável global:
  - declarada fora do corpo das funções:
    - visível por todas as funções subseqüentes
  - não é armazenada na pilha de execução:
    - não deixa de existir quando a execução de uma função termina
    - existe enquanto o programa estiver sendo executado
  - utilização de variáveis globais:
    - deve ser feito com critério
    - pode-se criar um alto grau de interdependência entre as funções
    - dificulta o entendimento e o reuso do código

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

- Variável estática:
  - declarada no corpo de uma função:
    - visível apenas dentro da função em que foi declarada
  - não é armazenada na pilha de execução:
    - armazenada em uma área de memória estática
    - continua existindo antes ou depois de a função ser executada
  - utilização de variáveis estáticas:
    - quando for necessário recuperar o valor de uma variável atribuída na última vez que a função foi executada

Variáveis Estáticas

- Exemplo:
  - função para imprimir números reais:
    - imprime um número por vez, separando-os por espaços em branco e colocando, no máximo, cinco números por linha

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

Comentários

- variáveis estáticas e variáveis globais:
  - são inicializadas com zero,
    se não forem explicitamente inicializadas
- variáveis globais estáticas:
  - são visíveis para todas as funções subseqüentes
  - não podem ser acessadas por funções definidas
    em outros arquivos
- funções estáticas:
  - não podem ser chamadas por funções definidas
    em outros arquivos

Funções Recursivas

- Tipos de recursão:
  - direta:
    - uma função A chama a ela própria
  - indireta:
    - uma função A chama uma função B que, por sua vez, chama A
- Comportamento:
  - quando uma função é chamada recursivamente,
    cria-se um ambiente local para cada chamada
  - as variáveis locais de chamadas recursivas são independentes
    entre si, como se estivéssemos chamando funções diferentes

Funções Recursivas

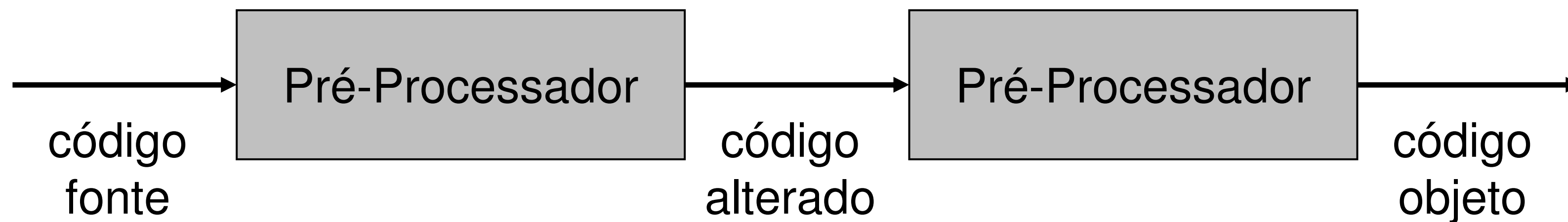
- Exemplo: definição recursiva de fatorial

$$n! = \begin{cases} 1, \; se \; n = 0 \\ n \times (n-1)!, \; se \; n > 0 \end{cases}$$

```
/* Função recursiva para cálculo do fatorial */
int fat (int n)
{
   if (n==0)
      return 1;
   else
      return n*fat(n-1);
}
```

Pré-processador e macros

- Pré-processador:
  - reconhece determinadas diretivas
  - altera o código antes de enviá-lo ao compilador



Pré-processador e macros

- #include *nome-do-arquivo*
  - pré-processador substitui o “include”
    pelo corpo do arquivo especificado:
    - o texto do arquivo passa a fazer parte do código fonte
    - nome do arquivo entre aspas (“*nome-do-arquivo*"):
      - pré-processador procura o arquivo primeiro no diretório local
        (em geral denominado diretório de trabalho)
      - caso não o encontre, o procura nos diretórios de *include* especificados para compilação
    - nome do arquivo entre os sinais de menor e maior (<*arquivo*>):
      - pré-processador não procura o arquivo no diretório local
        (os arquivos da biblioteca padrão de C devem ser incluídos com <>)

Pré-processador e macros

- Diretiva de definição para representar constantes:
  - fortemente recomendável para uniformidade e clareza do código
  - exemplo:
    - função para calcular a área de um círculo
      - antes da compilação, toda ocorrência de PI
        (desde que não envolvida por aspas) será trocada pelo número 3.14159F

```
#define PI  3.14159F
float area (float r)
{
   float a = PI * r * r;
   return a;
}
```

Pré-processador e macros

- Macro:
  – diretiva de 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));
```

Pré-processador e macros

- Cuidados na definição de macros:
  - erro de sintaxe pode ser difícil de ser detectado
  - compilador indicará erro na linha em que se utiliza a macro e não na linha de definição da macro
    (onde efetivamente encontra-se o erro)
  - **regra básica para a definição de macros:**
    - **envolva cada parâmetro, além da macro como um todo, entre parênteses**

Pré-processador e macros

```
#include <stdio.h>
#define DIF(a,b)   a - b
int main (void)
{
   printf(" %d ", 4 * DIF(5,3));
   return 0;
}
```

– problema:
  - o resultado impresso é 17 e não 8 pois o compilador processa
    *printf(" %d ", 4 * 5 - 3)*
    e a multiplicação tem precedência sobre a subtração

– solução:
  - parênteses envolvendo a macro:
    #define DIF(a,b) ((a) – (b))

Pré-processador e macros

```
#include <stdio.h>
#define PROD(a,b)   (a * b)
int main (void)
{
   printf(" %d ", PROD(3+4, 2));
   return 0;
}
```

- problema:
  - o resultado é 11 e não 14
- solução:
  - parênteses envolvendo a macro:
    #define PROD(a,b)   ((a) * (b))

Resumo

Operador unário & (“endereço de”)

Operador unário * (“conteúdo de”)

#include *nome-do-arquivo*

#define *código*