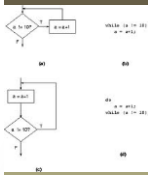
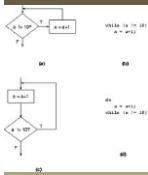
Estruturas de Dados

- Prof. Edkallenn Lima
- edkallenn@yahoo.com.br (somente para dúvidas)
- Blogs:
  - http://professored.wordpress.com (Computador de Papel – O conteúdo da forma)
  - http://professored.tumblr.com/ (Pensamentos Incompletos)
  - http://umcientistaporquinzena.tumblr.com/ (Um cientista por quinzena)
  - http://eulinoslivros.tumblr.com/ (Eu Li nos Livros)
  - http://linabiblia.tumblr.com/ (Eu Li na Bíblia)
- Redes Sociais:
  - http://www.facebook.com/edkallenn
  - http://twitter.com/edkallenn
  - https://plus.google.com/u/0/113248995006035389558/posts
  - Pinterest: https://www.pinterest.com/edkallenn/
  - Instagram: http://instagram.com/edkallenn ou @edkallenn
  - LinkedIn: br.linkedin.com/in/Edkallenn
  - Foursquare: https://pt.foursquare.com/edkallenn
- Telefones:
  - 68 98401-2103 (VIVO) e 68 3212-1211.
- Os exercícios devem ser enviados SEMPRE para o e-mail: edkevan@gmail.com ou para o e-mail: edkallenn.lima@uninorteac.edu.br



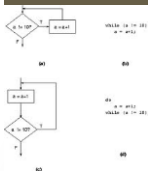
Agenda

- Funções
- Definição de funções
- Pilha de execução (simplificada)
- Ponteiros (Introdução)
- Valor x Referência
- Passando ponteiros para funções
- Variáveis Globais x Estáticas
- Recursividade (introdução)
- Pré-processador (introdução)
- Exercícios



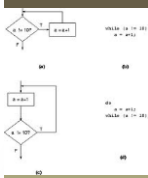
Funções

- Funções dividem as grandes tarefas de computação em tarefas menores
- A melhor maneira de escrever programas grandes é construí-lo a partir de pequenas partes (ou módulos).
- "Dividir para conquistar" ou "dividir e conquistar"
- Permitem que as pessoas trabalhem sobre o que outras já fizeram, reaproveitando código, ao invés de começar do zero (reinventar a roda).
- Modularização do código
- Função é uma unidade de código de programação autônoma desenhada para cumprir uma tarefa particular.



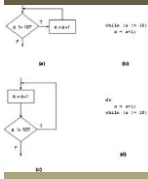
Funções

- Os módulos em C são chamados funções
- Programas em C geralmente são compostos de várias pequenas funções em vez de poucas de maior tamanho. (programa estruturado)
- Um procedimento repetido, deve, sempre ser transformado em uma função.
- Um programa em C deve ser pensado em termos de funções.
- Em C TUDO é feito usando funções.



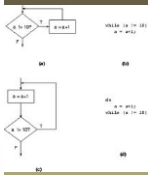
Funções

- Maiores vantagens:
  - Facilita a verificação de erros
  - Permite testar módulos individualmente
  - Desenvolvimento independente dos módulos
  - Permite reutilização de código



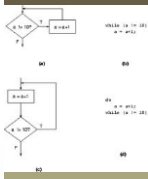
Funções

- Geralmente os programas em C são escritos combinando novas funções que o programador escreve com funções "prontas" disponíveis na biblioteca padrão do C (C standard library)
- É uma boa técnica se familiarizar com o ótimo conjunto de funções da biblioteca padrão do ANSI C
- Embora não sejam tecnicamente parte da linguagem C as funções da biblioteca padrão são sempre fornecidas em compiladores C
- As funções **`printf`**, **`scanf`**, **`pow`** e **`sqrt`** são funções da biblioteca padrão.



Funções

- As funções são chamadas, ativadas ou invocadas por uma chamada de função.
- A chamada de função menciona a função e fornece informações (como argumentos) de que a função precisa para realizar sua tarefa
- Cada argumento de uma função pode ser uma constante, uma variável ou uma expressão

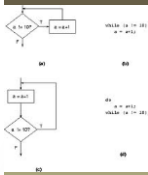


Definição de Funções

- A forma geral de uma função é:

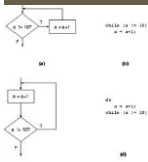
```
tipo_retornado nome_da_função ( lista de parâmetros... )
{
  corpo da função
}
```

- Onde: tipo_retornado especifica o tipo de valor que o comando return da função devolve
- Se nenhum tipo é especificado o compilador assume que seja um inteiro.
- Lista de parâmetros é uma lista de nomes de variáveis separados por vírgulas e seus tipos associados que recebem valores quando a função é chamada.
- Os argumentos da função devem ser equivalentes em número, tipo e ordem aos parâmetros na definição da função
- Argumento ≈ parâmetro



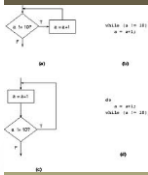
Funções

- Quando um programa encontra uma função, o controle é transferido do ponto de chamada para a referida função, as instruções da função chamada são executadas e o controle retorna a quem chamou
- Uma função pode retornar de 3 maneiras:
  - ao atingir a chave direita de final da função;
  - executando a instrução **`return;`**
  - ou retornando algum valor: **`return expressao;`**



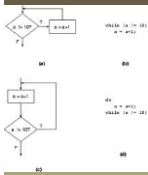
Funções - protótipos

- Um protótipo de função declara o tipo de retorno da função e declara o número, os tipos e a ordem dos parâmetros que a função espera receber
- Os protótipos permitem ao compilador verificar se as funções são chamadas corretamente
- O Compilador ignora os nomes de variáveis declarados no protótipo da função
- Protótipo é a assinatura da função



Escopo de Funções

- Em C, cada função é um bloco discreto e autônomo de código.
- Um código de uma função é privativo àquela função e não pode ser acessado por nenhum comando em outra função
- Exceto por meio de uma chamada à função
- Variáveis definidas internamente a uma função são chamadas "locais" (existe quando ocorre a entrada na função e é destruída ao sair)
- Variáveis locais não mantém seus valores entre chamadas a funções (exceção – static)
- Em C, todas as funções estão no mesmo nível hierárquico e de escopo.
- NÃO SE PODE DEFINIR UMA FUNÇÃO DENTRO DE OUTRA 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

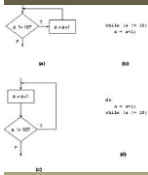
"main" retorna um inteiro:
0 : execução OK
≠ 0 : execução ¬OK

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

retorna o valor da função

Dicas de funções

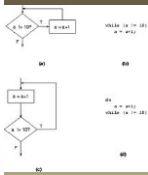
- Cada função deve se limitar a realizar uma tarefa simples e bem definida
- O nome da função deve expressar efetivamente aquela tarefa
- Isso facilita a abstração e favorece a capacidade de reutilização de código
- Usar as funções da biblioteca padrão do C torna seus programas mais portáveis
- Procure construir uma "biblioteca própria de funções"



Exemplo de função – int quadrado(int x)

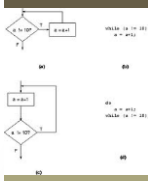
exemplo_funcao_quadrado.c

```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Funcao quadrado(x)
    Autor : Edkallenn - Data : 06/04/2012
    Observações: Eleva um numero ao quadrado
*/
int quadrado (int);  //prototipo da funcao
int cubo(int);
int main(){
    int x;
    for(x = 1;x<=10;x++)
        printf("O quadrado de %3d eh: %3d\n", x, quadrado(x));
    printf("\n");
    getch();
    return 0;
}
//definicao da funcao
int quadrado(int a)
{
    return(a * a);
}
```



[*] funcao_POT.c

```c
#include <stdio.h>
#include <stdlib.h>
/* Função pot(m,n) - Autor: Edkallenn - Data: 06/04/2012 */
int pot(int, int);  //protótipo da função

main(){
    int n,p, resultado;
    char c;
    do{
        printf("Digite um numero inteiro: ");
        scanf("%d",&n);
        printf("Digite a potencia para elevar %d: ", n);
        scanf("%d",&p);
        resultado = pot(n,p);
        printf("O número %d elevado a %d eh: %d", n,p,resultado);
        printf("\n\nContinua: (S/N)?");
        c = toupper(getche(c));
        printf("\n\n");
    }while(c=='S');
}
int pot (int x, int n){
    int p;
    for(p=1;n>0;--n){
        p=p*x;
    }
    return p;
}
```



Versão DevC++

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
/* Função pot(m,n) - Autor: Edkallenn - Data: 06/04/2012 */
int pot(int, int);  //protótipo da função
main(){
  int n,p, resultado;
  char c;
  do{
    printf("Digite um numero inteiro: ");
    scanf("%d",&n);
    printf("Digite a potencia para elevar %d: ", n);
    scanf("%d",&p);
    resultado = pot(n,p); //chamda da função
    printf("O número %d elevado a %d eh: %d", n,p,resultado);
    printf("\n\nContinua: (S/N)?");
    scanf("%s", &c);
    c = toupper(c);
    printf("\n\n");
  }while(c=='S');
}
int pot(int x, int n){
  int p,i;
  p=1;
  for(i=1;i<=n;i++){
    p=p*x;
  //  printf("i= %d - p = %d - n = %d\n",i,p,n);
  }
  return p;
}
```

Erros comuns

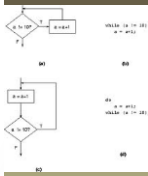
- Omitir o tipo de valor de retorno em uma definição de função causa um erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de int (ver a função quadrado)
- Esquecer de retornar um valor de uma função que deve fazer isso pode levar a erros inesperados.
- Retornar um valor de uma função cujo tipo de retorno foi declarado **`void`** causa um erro de sintaxe.
- Embora a omissão de um tipo de retorno seja assumido como **`int`**, sempre declare explicitamente o tipo de retorno. Entretanto, o tipo de retorno de **`main()`** geralmente é omitido.
- Declarar parâmetros da função do mesmo tipo como **`float x, y`** em vez de **`float x, float y`**. A declaração de parâmetros **`float x, y`** tornaria na realidade **`y`** um parâmetro do tipo **`int`** que é o default.

Erros comuns

- Definir um parâmetro de uma função novamente como variável local dentro da função é um erro de sintaxe
- Definir uma função dentro de uma função é um erro de sintaxe
- Esquecer o ponto e vírgula em um protótipo causa erro de sintaxe

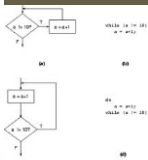
Exemplo – maiordetres(int, int, int)

```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Funcao maiordetres(x, y, z)
    Autor : Edkallenn
    Data : 06/04/2012
    Observações: Devolve o maior de 3 inteiros
*/
int maiordetres(int, int, int);  //prototipo da funcao

int main()
{
    int a,b,c;

    printf("Entre com 3 inteiros ( separados por virgulas - a, b, c): ");
    scanf("%d, %d, %d", &a, &b, &c);
    printf("O maior eh: %d", maiordetres(a,b,c) );
    getch();
    return 0;
}
//definicao da funcao
int maiordetres(int x, int y, int z)
{
    int max = x;
    if (y>max)
        max = y;

    if(z>max)
        max = z;

    return max;
}
```



Exercício

- Fazer uma função chamada cubo que retorna o cubo de um número
- Fazer uma função que retorna o valor do produto notável *quadrado da soma de dois termos* $\Rightarrow (x + y)^2 = x^2 + 2xy + y^2$ entrando com x e y e retornando o produto notável (usar a função quadrado vista anteriormente)
- Fazer o mesmo para o cubo da soma de dois termos$\Rightarrow (x + y)^3 = x^3 + 3x^2y + 3\,xy^2 + y^3$
- Usar a função cubo escrita agora há pouco e a função quadrado

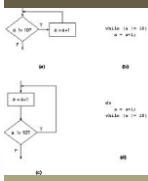


Definição de Funções (Exercício)

- Implemente uma função para calcular o número de combinações de **n** objetos diferentes, onde **r** objetos são escolhidos de cada vez. A fórmula é:

$$C_r^n = \binom{n}{r} = \frac{n!}{r! \cdot (n-r)!}$$

- Na combinação, a ordem em que os elementos são tomados não é importante
- Dica: reaproveite o código da função fatorial (!) anterior. Aplicação de reuso de código. (deve ser feito em sala de aula)

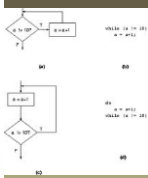


Definição de Funções (Exercício)

- Implemente uma função para calcular o número de arranjos de **n** objetos diferentes, onde **r** objetos são escolhidos de cada vez. A fórmula é:

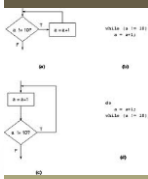
$$A_r^n = \frac{n!}{(n-r)!}$$

- Na combinação, a ordem em que os elementos são tomados é importante
- Dica: reaproveite o código da função fatorial (!) anterior. Aplicação de reuso de código. (deve ser feito em sala de aula)



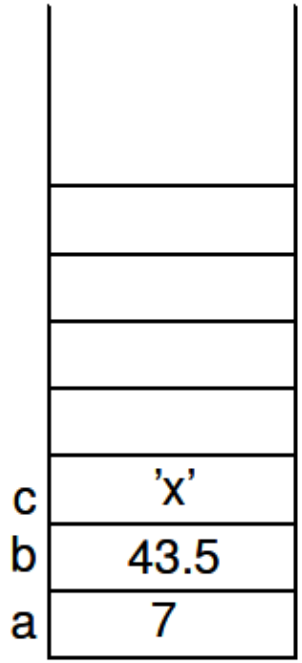
Pilha de Execução

- Comunicação entre funções
  - Funções são independentes entre si
  - A 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
    - 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

- A comunicação entre funções é realizada mediante uma PILHA (estrutura de dados que será estudada em breve)



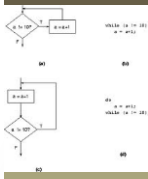
112 - variável c no endereço 112 com valor igual a 'x'

108 - variável b no endereço 108 com valor igual a 43.5

104 - variável a no endereço 104 com valor igual a 7

Pilha de Execução

- Implementação da função fat
- Simulação para fat(5)
- A variável n possui o valor 0 ao final da execução de fat, mas
- O valor de n no programa principal ainda será 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;
}
```

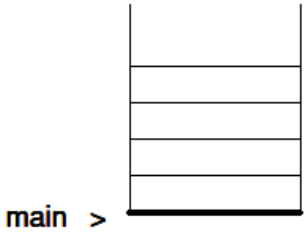
declaração das variáveis *n* e *r*, locais à função *main*

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

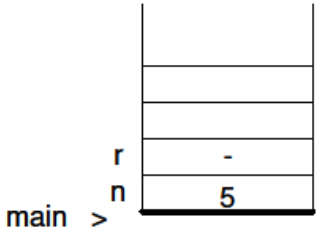
alteração no valor de *n* em *fat* não altera o valor de *n* em *main*

Pilha de Execução

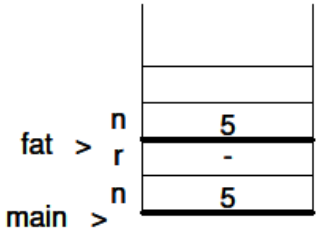
1 - Início do programa: pilha vazia



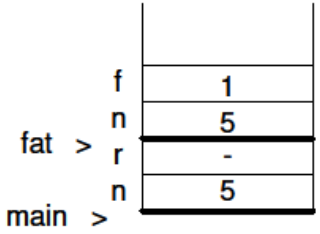
2 - Declaração das variáveis: n, r



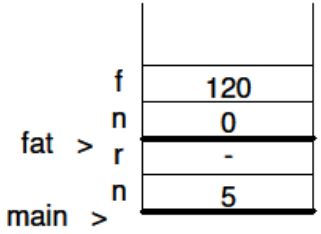
3 – Chamada da função : cópia do parâmetro



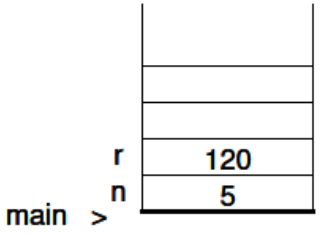
4 - Declaração da variável local: f



5 - Final do laço

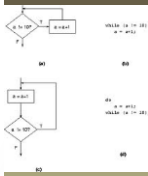


6 - Retorno da função: desempilha



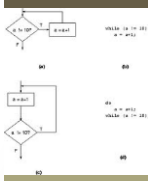
Chamada por valor x referência

- Em C, por padrão todas as chamadas de função são **por valor**
- Na chamada por valor, quando a função é solicitada é feita uma "cópia" dos valores dos argumentos e ela cria, então, variáveis temporárias para armazenar estes valores
- Uma função não pode, portanto, alterar o valor original de uma variável da função que chama (compartimentação de código)
- Isso só pode ser feito usando uma chamada por referência (passando o endereço das variáveis).
- Na chamada por referência a função recebe o endereço das variáveis e pode alterar seus valores
- Nas próximas aulas veremos como simular (com ponteiros) a chamada por referência usando operadores de endereços e de ações indiretas
- Os vetores e matrizes (arrays) são passados automaticamente através de uma simulação de chamada por referência.



Exercícios

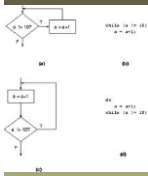
1. Fazer uma função para calcular a área de uma esfera, dado $A = 4\pi r^2$ (considere π = 3,141592)
2. Criar uma função que calcule a diferença entre duas horas distintas de um mesmo dia e retorne o intervalo de tempo entre os dois horários (ler no formato hh:mm e imprimir no formato hh:mm)
3. Escrever uma função inteira pot(m,n) que eleva m à potência n.
4. Escrever uma função para calcular a velocidade média de um percurso e apresentar no formato km/h. (entradas: deslocamento em km e tempo gasto em horas



```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Duas horas e diferença entre elas.
**   Autor : Edkallenn  -  Data : 06/04/2012
**   Observações:  */
int dif_horas(int hora1, int hora2); //Prototipo

main()
{
    int h1, min1, h2, min2, diferenca, horas, minutos;
    printf("\n\nPrograma que calcula o intervalo de tempo entre duas horas de um mesmo dia\n\n");
    printf("\nDigite a primeira hora (formato hh:mm):" );
    scanf("%d:%d", &h1, &min1);
    printf("\nDigite a segunda hora (formato hh:mm):" );
    scanf("%d:%d", &h2, &min2);

    diferenca = dif_horas(h1*60+min1, h2*60+min2);

    //escreve no formato hh:mm
    horas = (int) diferenca/60;
    minutos = (int) diferenca % 60;

    printf("\n\n%d:%d - %d:%d eh: %d:%d\n", h1,min1, h2, min2, horas, minutos  );
    printf("\nOu o intervalo de tempo entre %d:%d e %d:%d eh: %0d:%0d\n\n",
                                            h2,min2, h1, min1, horas, minutos);
    getch();
}
//a funcao dif_horas recebe os parametros em minutos
int dif_horas(int hora1, int hora2)
{
    return(hora2-hora1);
}
```

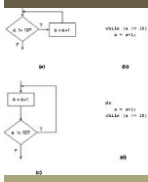
Exercícios

5. Implementar uma função que diga se um numero é ou não primo (números primos são os divisíveis somente por um e por eles mesmos). (Retornar 1 se primo e 0 se não primo).
6. Faça uma função que retorne o somatório de um número N. (somatório=soma dos números de 1 a N)
7. Faça uma função que retorne o produtório de um número N. (produtório= multiplicação dos números de 1 a N)
8. Faça uma função que imprima a tabuada de um número N informado pelo usuário (função do tipo void)

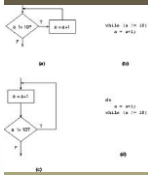


Exercícios (sala de aula) – 0.5 ponto

- Fazer uma função para calcular a área de uma esfera, dado $A = 4\pi r^2$.
  Considere $\pi = 3.14159265$.
  Protótipo:
  ```
  double areaEsfera (float raio);
  ```
- Escrever uma função para calcular a velocidade média de um percurso e apresentar no formato km/h.
  Protótipo:
  ```
  float calculaVM (float espaco, float tempo);
  ```
- Faça uma função que retorne o somatório de um número num.
  (somatório=soma de todos os números de 1 a N)
  Protótipo:
  ```
  int somatorio (int num);
  ```
- Faça uma função que leia um ângulo em graus e retorne o valor do mesmo convertido para radianos.
  A fórmula de conversão é $R = G * \pi /180$,
  sendo G o ângulo em graus e R em radianos e pi = 3.14159265
  Protótipo:
  ```
  float converteGrauRadiano (float angulo);
  ```

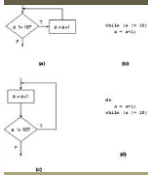


```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Funcao e_primo e uso da funcao.
**  Autor : Edkallenn  -  Data : 06/04/2012
**  Observações:
*/
int e_primo(int);    //prototipo da funcao

main(){
    int num;
    do{
        printf("\nEntre com um numero inteiro positivo: ");
        scanf("%d", &num);
    }while(num<=0);

    int primo;
    primo = e_primo(num);
    if (primo==1)
        printf("\nO numero %d eh primo\n", num);
    else
        printf("\nO numero %d nao eh primo\n", num);
    getch();
}
e_primo(int x){
    int cont_primo = 0;
    int i;
    for (i = 1;i<=x;i++){
        if ((x%i)==0)
           cont_primo+=1;
    }
    if(cont_primo==2)
        return 1;    //é primo
    else
        return 0;    //nao eh primo
}
```



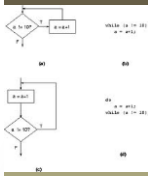
Variáveis globais

- Declarada fora do corpo das funções:
  - visível por todas as funções subsequentes
- 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



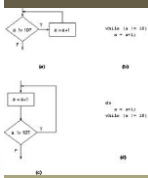
Variáveis globais

```
#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;
}
```



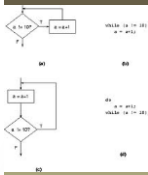
Variáveis Estáticas

- 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



Resumo - variáveis

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

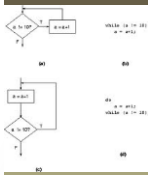


Recursividade (introdução)

- Uma função recursiva é uma função que chama a si mesma, ou diretamente ou indiretamente por meio de outra função.
- A recursão requer certo nível de abstração, mas é algo imprescindível na ciência da computação (e na matemática)
- Vamos aprender hoje exemplos simples de recursão (versões recursivas mais complexas serão vistas nas próximas aulas)

Recursividade (introdução)

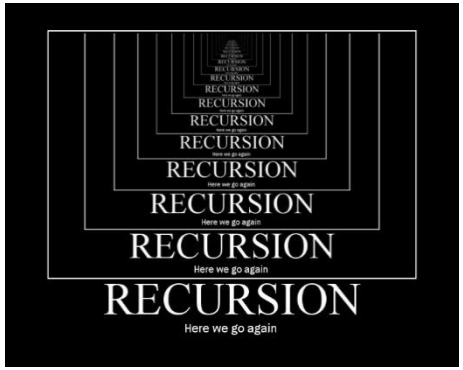
- A recursividade dá a ideia de coisas que se repetem
- Os métodos recursivos para a solução de problemas têm todos elementos em comum
- Quando uma função recursiva é chamada para resolver um problema, ela, na verdade é como se só soubesse resolver apenas os casos mais simples, ou os chamados casos básicos.(ou condições de parada)
- Em casos complexos ela divide o problema em duas partes: a parte que ela resolve (que ela sabe) e outra que ela não resolve (que ela não sabe).
- A segunda parte (para a recursão ser viável) deve ser parecida com o problema original, mas ser mais simples ou menor que ele



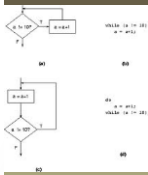
Recursividade

- Para esse novo problema (menor) a função lança (chama) uma nova cópia de si mesma para lidar com o problema menor
- Que é conhecido por chamada recursiva ou etapa de recursão
- A etapa de recursão também inclui o comando return porque seu resultado será combinado com a parte que a função sabe para resolver o problema.

Recursividade (introdução)



- 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

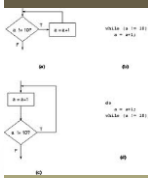


Requisito fundamental (recursão)

- A chamada recursiva DEVE estar sujeita a uma condição que em um determinado momento pare de ativar a recursão (uma condição que se torne falsa ou uma contagem que atinja um valor determinado dentro do código da função
- CONDIÇÃO DE PARADA (caso básico)
- Problema associado à recursão: recursão infinita!

PORTANTO:

- Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema.
- Dizemos que esses problemas têm *estrutura recursiva*. Para resolver um tal problema podemos aplicar o seguinte método:
    - se a instância em questão é pequena, resolva-a diretamente (use força bruta se necessário);
    - senão, *reduza-a* a uma instância menor do mesmo problema,
    - aplique o método à instância menor e
    - volte à instância original.
- A aplicação desse método produz um algoritmo *recursivo*. Para mostrar como isso funciona, examinaremos um exemplo concreto.

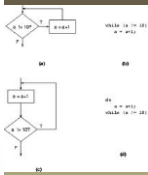


Recursividade (introdução)

- Exemplo: 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);
}
```



Recursividade

fat(5) = 5 x fat(4)

fat(4) = 4 x fat(3)

fat(3) = 3 x fat(2)

fat(2) = 2 x fat(1)

fat(1) = 1 x fat(0)

fat(0) = 1

Valor final = 120

5! = 5 * 24 = 120 (valor retornado)

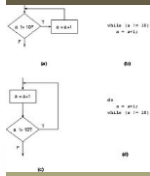
4! = 4 * 6 = 24 é o valor retornado

3! = 3 * 2 = 6 é o valor retornado

2! = 2 * 1 = 2 é o valor retornado

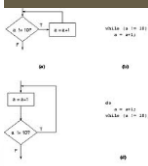
1! = 1 * 1 = 1 é o valor retornado

0! = 1 é o valor retornado (caso básico)



Roteiro

- Eis um roteiro que pode ajudar a verificar se a função está correta:
  - 1. Escreva o que a função deve fazer.
  - 2. Verifique que a função de fato faz o que deveria quando n é pequeno, ou seja, quando n=1.
  - 3. Agora imagine que n é grande, ou seja, n > 1; suponha que a função faria a coisa certa se no lugar de n tivéssemos algo menor que n; verifique que a função faz o que dela se espera.

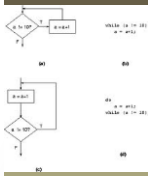


Comentários

- Para que uma função recursiva seja compreensível, é muito importante que o autor da função diga, explicitamente, *o que* a função faz.
- Algumas pessoas acreditam que funções recursivas são inerentemente ineficientes e consomem muito tempo, mas isso é lenda.
- Talvez a lenda tenha nascido com uma aplicação descuidada da recursão, como a sequencia de Fibonacci

M.D.C. (máximo divisor comum)

- O **máximo divisor comum ou MDC** (**mdc**) entre dois ou mais números inteiros é o maior número inteiro que é fator de tais números.
- Por exemplo, os divisores comuns de 12 e 18 são 1,2,3 e 6, logo mdc(12,18)=6.
- A definição abrange qualquer número de termos, por exemplo mdc (10,15,25,30)=5.
- O máximo divisor comum também pode ser representado só com parênteses.
- Com esta notação, dizemos que dois números inteiros *a* e *b* são primos entre si, se e somente se mdc(*a*, *b*)=1.
- Esta operação é tipicamente utilizada para reduzir equações a outras equivalentes:

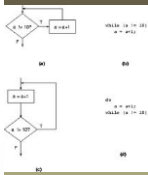


M.D.C.

- Há duas formas de determinar o máximo divisor comum de dois números:
- A primeira é fatorar os números e a partir daí, pegar os fatores comuns a todos números e deixá-los com o menor expoente que o fator analisado apresentar entre todos os números. Exemplo: Achemos o MDC de 30 e 12. Note que $30 = 2 \times 3 \times 5$ e $12 = 3 \times 2^2$
- Então (30, 12) = 2 x 3 (fatores comuns aos números e o menor expoente do fator. No caso do 2 tínhamos expoentes 1 e 2, mas pegamos o menor, daí ficou só 2 e não 2 ao quadrado).

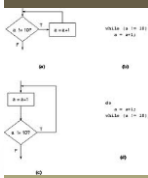
MDC

- A segunda consiste em escrever os dois números, separados por um traço vertical; em seguida, compara-se os números, e em baixo do maior deles coloca-se a diferença entre os dois.
- Agora compara-se o último número que se escreveu, com o que ficou na outra coluna, repetindo-se o processo até que se obtenha igualdade entre os números nas duas colunas, que é o resultado procurado



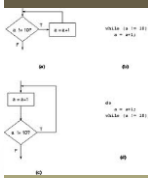
Algoritmo de Euclides (recursivo)

- O algoritmo de Euclides é um dos mais antigos e famosos que existem. Ele busca encontrar o MDC (Máximo Divisor Comum) entre dois números inteiros. (diferentes de zero).
- Fez sua primeira aparição no livro sétimo dos Elementos de Euclides por volta do ano 300 a.C.
- O processo é também conhecido como método das divisões sucessivas. É bem simples e eficiente pois dispensa fatoração.



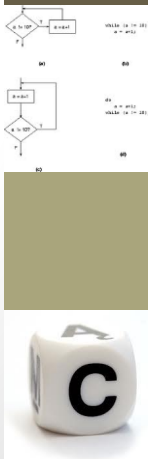
Algoritmo de Euclides (pseudocódigo)

```
algoritmo "algoritmo de Euclides"
// Função : Algoritmo de Euclides
// Autor : Ed
// Data : 12/05/2009
// Seção de Declarações
var
    a,b,c, dividendo, divisor:inteiro
inicio
//entrada de dados
escreval("Algoritmo de Euclides para encontrar o MDC entre 2 números")
escreva("Digite o primeiro numero:")
leia(a)
escreva("Digite o segundo numero:")
leia(b)
//algotimo propriamente dito
dividendo <- a
divisor <- b
enquanto ((dividendo%divisor) <> 0)  faca
     c <- (dividendo%divisor)
     dividendo <- divisor
     divisor <- c
fimenquanto
escreva(" O MDC entre", a, " e ", b, " é ", divisor)    //apresentacao na tela
fimalgoritmo
```

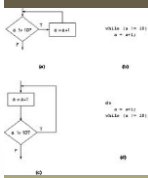


Algoritmo de Euclides – Pseudocódigo recursivo

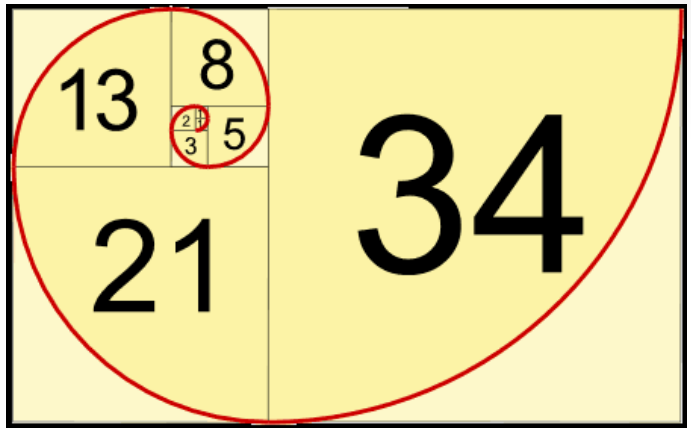
```
1  algoritmo "Algoritmo Euclides recursivo"
2  // Função : Calcula o MDC usando o método de Euclides
3  // Autor : Edkallenn
4  // Data : 25/09/2012
5  // Seção de Declarações
6
7  var
8  dividendo, divisor, resto, resultado: inteiro
9
10 funcao mdc_r(x: inteiro; y: inteiro): inteiro
11 //Entradas: x,y     --> valores para o cálculo do MDC
12 //Sáida - retorno  --> máximo divisor comum entre x e y
13 var
14 inicio
15 se y=0 entao
16    retorne x
17 senao
18      se x<y entao
19         retorne mdc_r(y,x)
20      senao
21           retorne mdc_r(y, x % y)
22      fimse
23 fimse
24 fimfuncao
25
26 inicio
27 // Seção de Comandos
28 escreval("MDC usando o algoritmo de Euclides - recursivo")
29 escreva("Digite o primeiro numero: ")
30 leia(dividendo)
31 escreva("Digite o segundo numero: ")
32 leia(divisor)
33 resultado <- mdc_r(dividendo, divisor)
34 escreval ("O mdc de ", dividendo, " e ", divisor, " é ", resultado)
35 fimalgoritmo
36
```



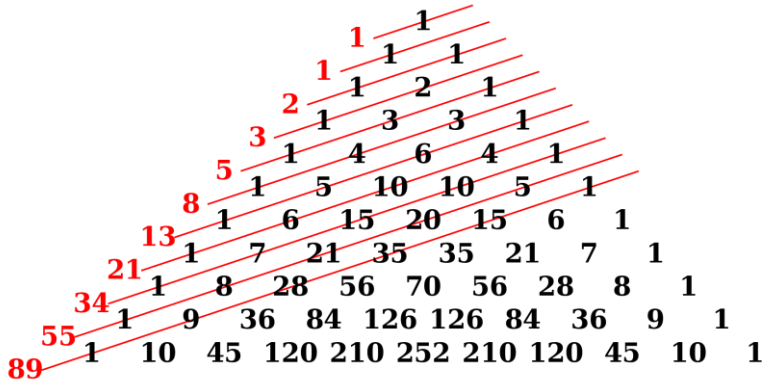
```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : MDC entre dois números recursivo (Euclides)
    Autor : Edkallenn  -  Data : 06/04/2012
    Observações: arquivo-> mdc_recursivo.cpp em /basico
*/
int mdc(int, int);

main()
{
    int i, num1, num2;
    do{
        printf("\nEntre com dois numeros inteiro positivos: ");
        scanf("%d, %d", &num1, &num2);
    }while((num1<=0) || (num2<=0));

    printf("O MDC entre %d e %d eh %d\n\n",num1, num2, mdc(num1, num2));
    system("pause");
}

int mdc(int x, int y)
{
    if(y==0)
        return x;
    else
         if (x<y)
            return mdc(y,x);
        else
            return mdc(y, x % y);
}
```



Recursividade

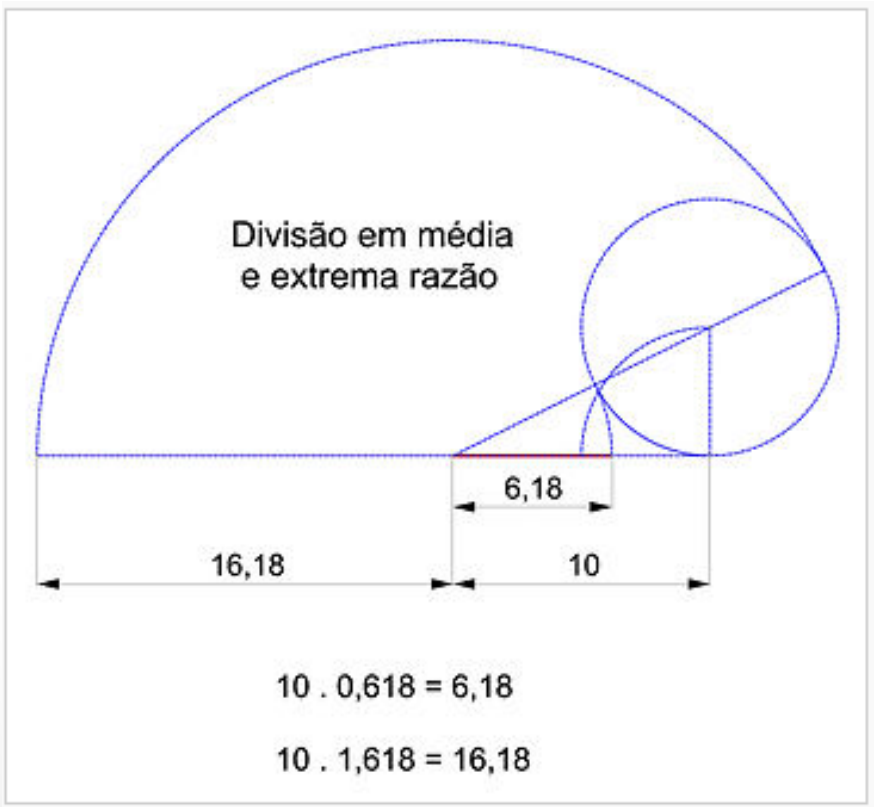


- A sequência de Fibonacci:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- Começa com 0 e 1 e tem a propriedade de que cada número subsequente de Fibonacci é a soma dos dois anteriores;
- A sequência ocorre na natureza e descreve uma forma de espiral.
- A taxa dos números sucessivos converge sob um valor constante de 1,618...
- Razão áurea ou taxa áurea. Número Fi. Número de ouro.

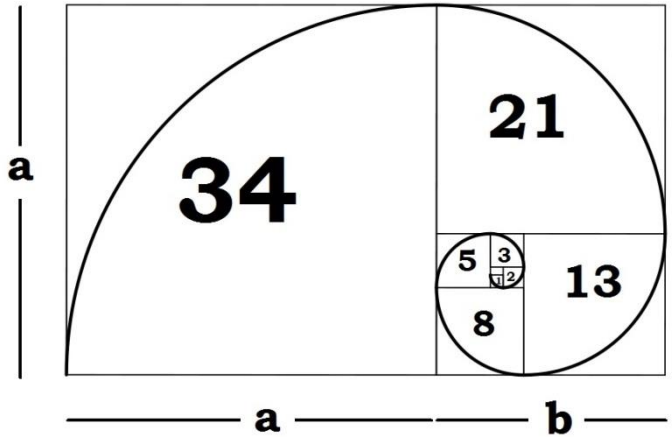


Recursividade

- Proporção áurea:



Divisão em média e extrema razão. A partir de um segmento de 10 unidades, determina-se a sua seção áurea multiplicando-o por 0,618 (média). Para encontrar-se um segmento maior, em extrema razão, deve-se multiplicar as dez unidades iniciais por 1,618.



Recursividade

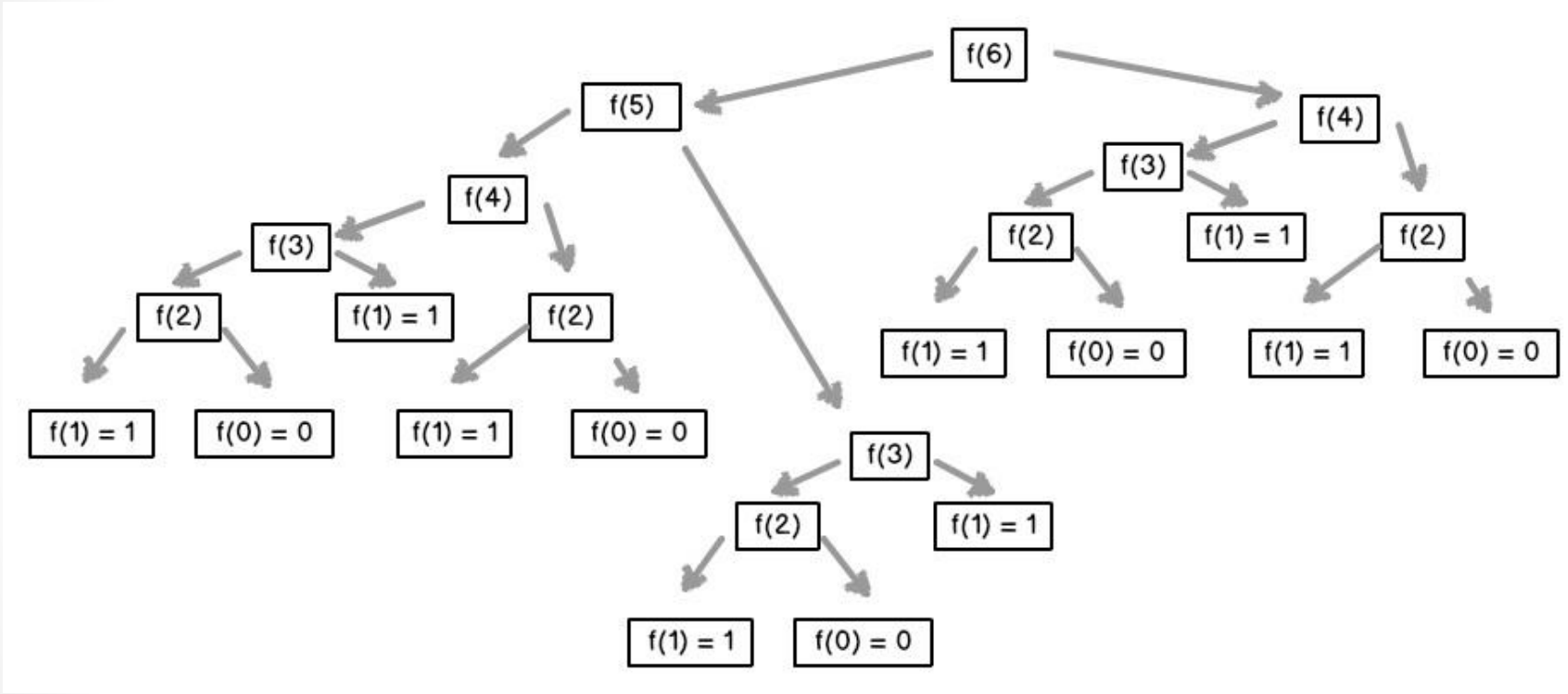
- A sequencia de Fibonacci pode ser definida recursivamente como se segue:

- `fibonacci(0) = 0`
- `fibonacci(1) = 1`
- `fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)`

Fibonacci recursivo

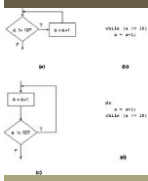
```c
#include <stdio.h>
#include <stdlib.h>

float fibonacci(int);

main(){
        int numero, i;

        printf("Entre com um inteiro: ");
        scanf("%d", &numero);
        for(i = 1;i<=numero;i++)
                printf("Fibonacci(%3d) - %3f\n", i, fibonacci(i));

        system("pause");
}

float fibonacci(int n)      //fibonacci recursiva
{
     if(n==0 || n==1)
        return n;
     else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Fibonacci recursivo



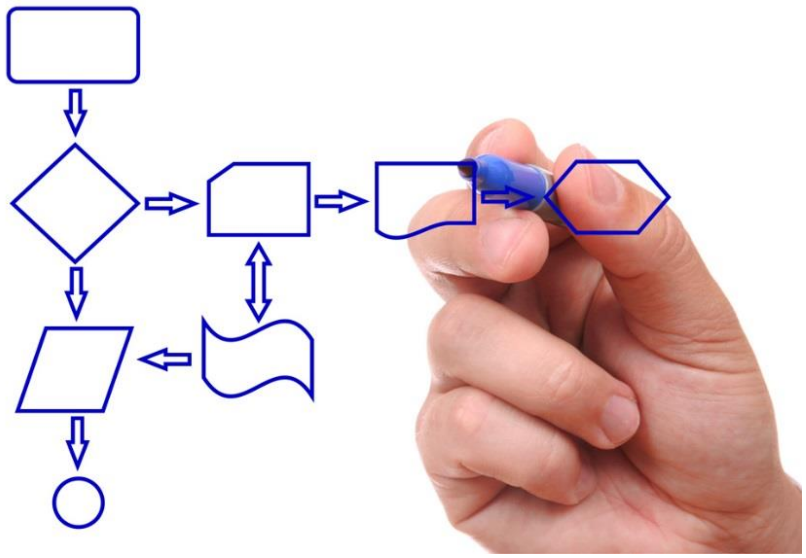
Recursividade x iteratividade (pontos a considerar)

- Evite usar recursividade em em que o desempenho é fundamental
- As chamadas recursivas são um pouco mais demoradas e consomem memória adicional
- Um método recursivo é escolhido quando reflete melhor o problema e resulta em um programa mais fácil de entender e depurar
- Outro motivo para se escolher recursividade é que o método iterativo pode não ser aparente
- Tanto a iteração quanto a recursão se baseiam em uma estrutura de controle: a iteração usa uma estrutura de repetição, a recursão uma estrutura de seleção
- Tanto a iteração quanto a recursão envolvem repetições (uma direta e a outra com funções)
- Ambas envolvem testes de encerramento (condição de parada)
- A iteração e a recursão podem ocorrer indefinidamente. Cuidado! A iteração infinita ocorre quando a condição do loop nunca se torna falso. A recursão infinita quando não há convergência para o caso básico.

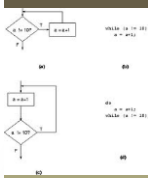


Funções que retornam void

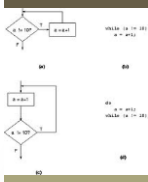
- Funções que retornam void são consideradas procedimentos
- Diferença: função x procedimento (lembrar)
- As funções que retornam `void` executam algo (realizam uma tarefa e não retornam valores)
- Não é necessário o `return`
- Exemplos.



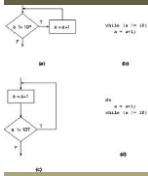
```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Pula quantidade x de linhas
    Autor : Edkallenn  -  Data : 03/10/2013
    Observações:
*/

void pula();

int main(){

    printf("Teste da funcao");
    pula(); pula();
    printf("Deve pular uma linha");
    getchar();
}
void pula(){
    printf("\n");
}
```



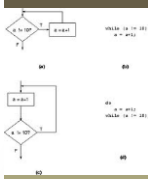
```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Pula quantidade x de linhas
    Autor : Edkallenn  -  Data : 03/10/2013
    Observações:
*/

void pula(int); //pula n linhas

int main(){

    printf("Teste da funcao");
    pula(10);
    printf("Deve pular uma linha");
    pula(15);
    getchar();
}
void pula(int x){
    int i;
    for(i=1;i<=x;i++)
        printf("\n");
}
```



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

int leia_int(void);
void par(int);

void main(void){
    int num=0;
    while (num!=-1){
        num = leia_int();
        if (num==-1) break;
        par(num);
    }
    getch();
}
int leia_int(){
    int numero;
    printf("Insira um numero inteiro: ");
    scanf("%d", &numero);
    return (int) numero;
}
void par(int epar){
    if ((epar%2) == 0)
        printf("\n\n O numero %d e par!\n\n", epar);
    else
        printf("\n\n O numero %d e impar!\n\n", epar);
}
```



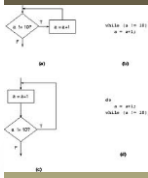
```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : exibe menu
    Autor : Edkallenn  -  Data : 03/10/2013
    Observações:
*/

void exibe_menu(); //menu
int leia_int();
void opcoes(int n);
void pula(int x);

main(){
    int num;
    do{
        exibe_menu();
        num = leia_int();
        opcoes(num);
        getch();
    }while(num!=0);
    getch();
}
void pula(int x){
    int i;
    for(i=1;i<=x;i++)
        printf("\n");
}
```

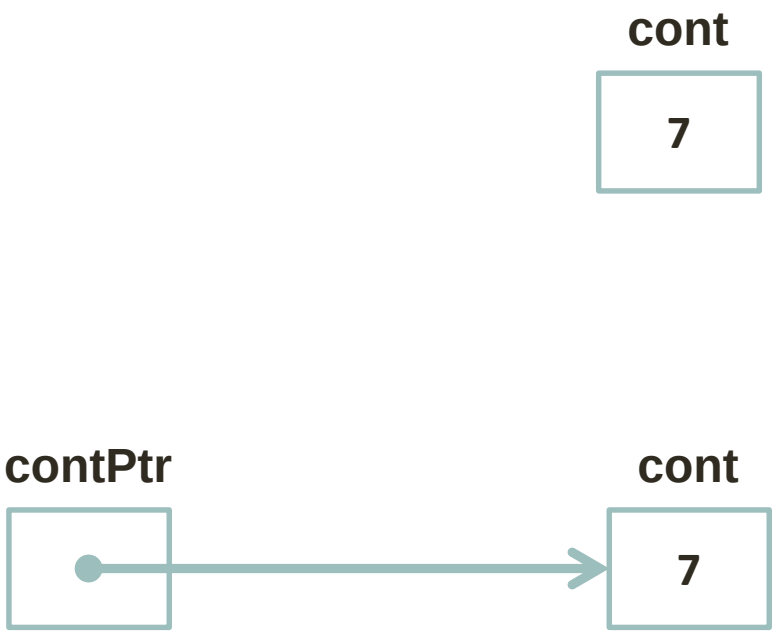


```
28  void exibe_menu(){
29      system("cls");
30      pula(2);
31      printf(" ====== MENU PRINCIPAL ======"); pula(2);
32      printf(" 1  -  I n s e r i r       \n");
33      printf(" 2  -  C o n s u l t a r   \n");
34      printf(" 3  -  C a l c u l a r     \n");
35      printf(" 0  -  S A I R             \n"); pula(2);
36  }
37  int leia_int(){
38      int numero;
39      printf("Insira um numero inteiro: ");
40      scanf("%d", &numero);
41      return (int) numero;
42  }
43  void opcoes(int n){
44      switch(n){
45      case 1: printf("\nOpcao 1\n"); break;
46      case 2: printf("\nOpcao 2\n"); break;
47      case 3: printf("\nOpcao 3\n"); break;
48      case 0: return; break;
49      default: printf("\nValor digitado eh invalido\n"); break;
50      };
51  }
52
```

Ponteiros (introdução)

- Um dos recursos mais poderosos da linguagem de programação C são *os ponteiros*
- Ponteiros são variáveis que contém endereços de memória como valores
- Normalmente um a variável faz referência direta a um valor específico
- Um ponteiro contém um endereço de uma variável que contém um valor
- Um nome de variável é uma referência direta a um valor
- Um ponteiro é uma referência indireta





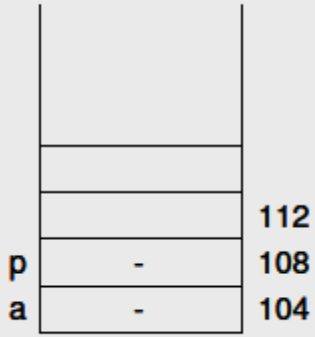
cont faz uma referência direta a uma variável cujo valor é 7

contPtr faz uma referência indireta a uma variável cujo valor é 7

Ponteiros (introdução)

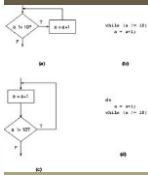
- C permite o armazenamento e a manipulação de valores de endereços de memória
- Para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados
- Os ponteiros devem ser declarados como qualquer outra variável

```
/*variável inteiro */
 int a;

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



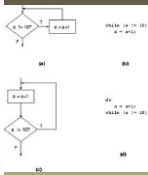
Inicialização

- Um ponteiro pode ser inicializado com 0, NULL ou um endereço.
- Um ponteiro com o valor NULL não aponta para lugar algum
- Inicializar um ponteiro com 0 é equivalente a inicializar um ponteiro com NULL, mas NULL é mais recomendado
- O valor 0 é o único valor inteiro que pode ser atribuído diretamente a uma variável do tipo ponteiro



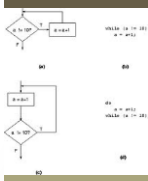
Prática recomendável

- Inicialize os ponteiros para não ter resultados inesperados
- Inclua as letras ptr (ou p, ou pt) em nomes de variáveis de ponteiros para tornar claro que essas variáveis são ponteiros e precisam ser manipuladas apropriadamente.



Ponteiros (introdução)

- **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

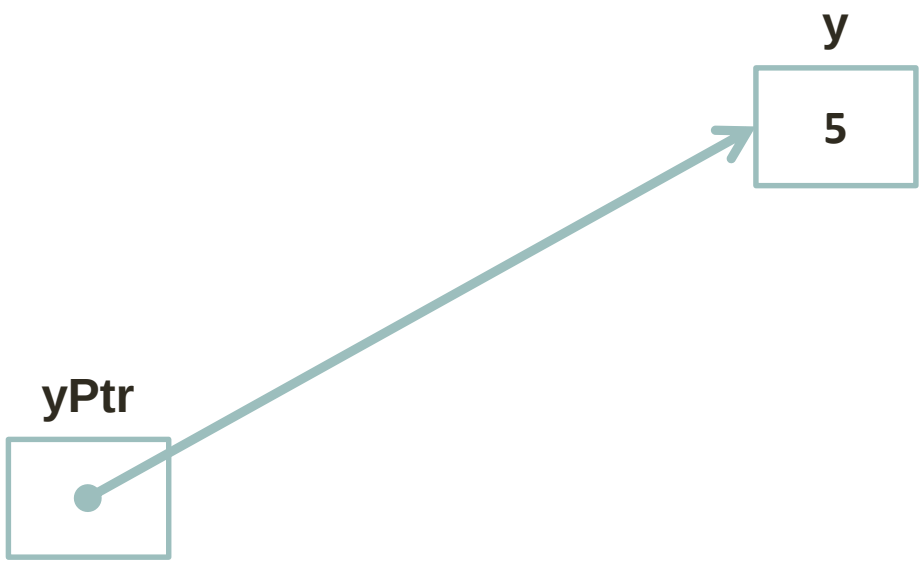
```
/* c recebe o valor armazenado
   na posição de memória apontada por p */
c = *p;
```

c	6	112
p	104	108
a	6	104

Exemplo

```
int y = 5;
int *yPtr;
yPtr = &y;
```

Diz-se que a variável **yPtr** "aponta para" **y**



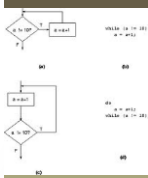
Representação esquemática de y e yPtr na memória

```c
#include <stdio.h>
#include <stdlib.h>
/*  Função : Usando os operadores & e *
    Autor : Edkallenn  -  Data : 06/04/2012
    Observações: Exemplos de utilização dos operadores de ponteiro
*/
main(){
    int a;        //a é um inteiro
    int *aPtr;    //aPtr é um ponteiro para um inteiro

    a = 7;
    aPtr=&a;

    printf("\nO endereco de a eh: %p\n", &a);
    printf("O valor de aPtr eh: %p\n\n", aPtr );

    printf("O valor de a eh: %d\n", a);
    printf("O valor de *aPtr eh: %d\n\n", *aPtr);

    printf("Sabendo que * e & complementam-se mutuamente");
    printf(".\n&*aPtr= %p\n*&aPtr= %p\n", &*aPtr, *&aPtr);

    return 0;
}
```

Ponteiros

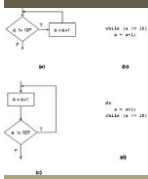
```
int main ( void )
{
   int a;
   int *p;
   p = &a;
   *p = 2;
   printf(" %d ", a);
   return;
}
```

imprime o valor 2



```
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

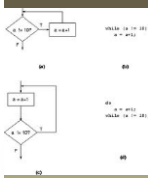


```
int main ( void )
{
   int a, b, c, *p;
   a = 2;
   p = &c;
   *p = 3;
   b = a + (*p);
   printf(" %d ", b);
   return 0;
}
```

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

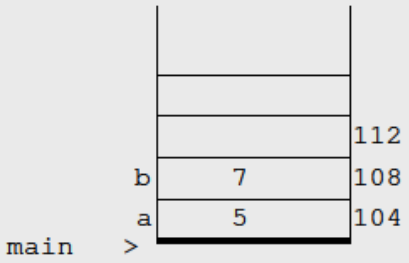
Ponteiros

- 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**

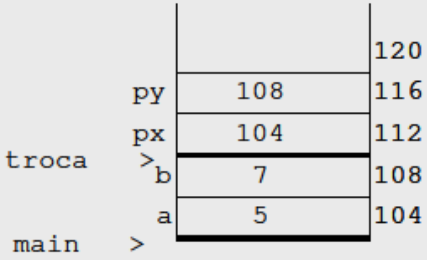


```c
/*  Função : |Trocando variáveis com ponteiros
    Autor : Edkallenn  -  Data : 06/04/2012
    Observações: Exemplos de utilização de ponteiros
*/
#include <stdio.h>
#include <stdlib.h>

void troca_var(int *, int *);

main(){
    int a,b;
    a = 5; b = 7;
    troca_var(&a, &b); //passamos o endereço das variaveis
    printf("%d %d \n", a, b);
    system("pause");
}
void troca_var(int *px, int *py){
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

```

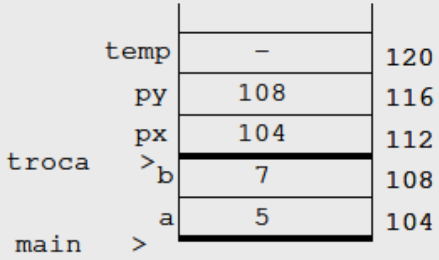
1 -Declaração das variáveis: a, b



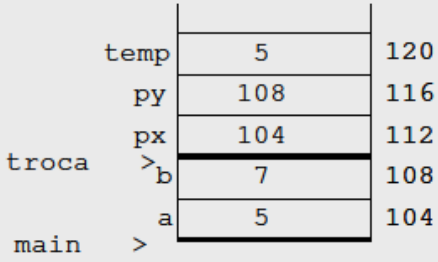
2 - Chamada da função: passa endereços



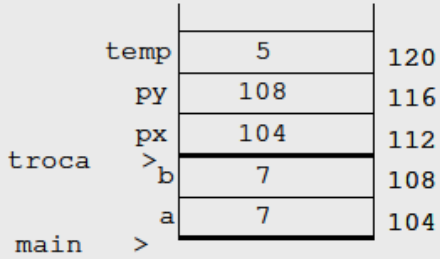
3 - Declaração da variável local: temp



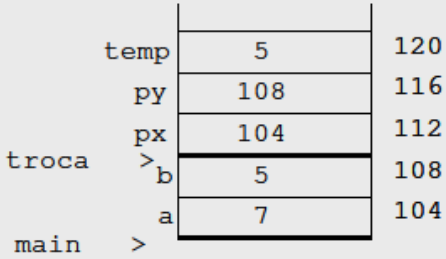
4 - temp recebe conteúdo de px



5 -Conteúdo de px recebe conteúdo de py

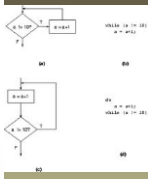


6 -Conteúdo de py recebe temp



Exercício

- Faça **duas funções** chamadas **void quad_por_valor(int)** e **void quad_por_referencia(int *)** para elevar um número ao quadrado, usando passagem por valor e por referência respectivamente
- Faça **duas funções** chamadas **void cubo_por_valor(int)** e **void cubo_por_referencia(int *)** para elevar um número ao cubo, usando passagem por valor e por referência respectivamente



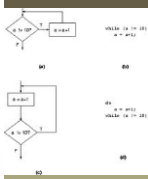
Resumo - Ponteiros

Operador unário & (“endereço de”)

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

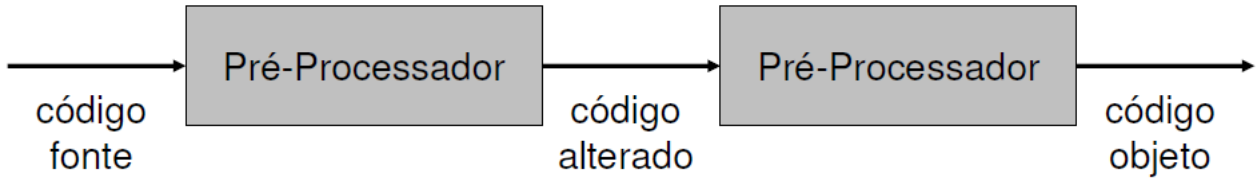
Operador unário * (“conteúdo de”)

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

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



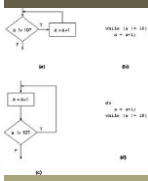
Pré-processador e macros

- O pré-processador C é um programa que examina o programa fonte em C e executa certas modificações nele, baseado em instruções chamadas diretivas
- Pré-processador:
  - Reconhece determinadas diretivas
  - Altera o código antes de enviá-lo ao compilador



Diretiva #include

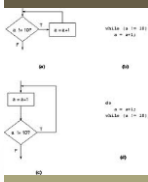
- `#include` *`nome-do-arquivo`*
- pré-processador substitui o “include” pelo corpo do arquivo especificado:
  - o texto do arquivo passa a fazer parte do código fonte
  - nome do arquivo entre aspas (*“nome-do-arquivo"):*
    - pré-processador procura o arquivo primeiro no diretório local (em geral 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 <>)



Diretiva #define

- Pode ser usada para representar constantes simbólicas
- Fortemente recomendada 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;
}
```

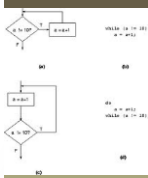


Importante

- Todas as diretivas do pré-processador começam com #
- Não há ponto e vírgula após as diretivas do pré-processador
- Use nomes significativos para as constantes simbólicas para tornar os programas mais auto-explicativos.
- Só podemos escrever um comando por linha
- #define economiza tempo e é mais rápido que a atribuição normal.

Macros

- Uma macro é uma operação definida em uma diretiva #define do pré-processador
- Como as constantes simbólicas, o identificador da macro é substituído no programa por um texto de substituição antes de o programa ser compilado.
- As macros podem ser definidas com ou sem argumentos
- Uma macro sem argumentos é processada como uma constante simbólica
- Em uma macro com argumentos, esses são substituídos no texto de substituição, então a macro é expandida – isto é, o texto de substituição substitui o identificador e alista de argumentos no programa.



Exemplos de macros

- Considere a seguinte definição de macro:

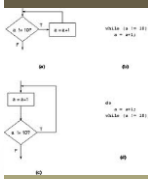
```
#define AREA_CIRCULO(x) ( PI * (x) * (x) )
```

- Sempre que AREA_CIRCULO(x) aparecer no arquivo, o valor de x substitui x no texto de substituição, a constante simbólica PI é substituída por ser valor (já definido)
- Por exemplo, a instrução:

```
area = AREA_CIRCULO(4);
```

- É expandida em:

```
area = ( 3.14159 * (4) * (4) );
```



Exemplos de macros

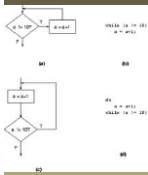
- Um exemplo é a macro já utilizada abaixo:

```
#define QL printf("\n")
```

- Sempre que o compilador encontrar:

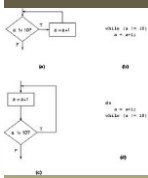
```
QL;
```

- Ele expandirá para `printf("\n");`



Cuidados na definição de macros

- Erros são difíceis de serem detectados
- 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)
- Como precaução envolva cada parâmetro, além da macro como um todo, entre parênteses



VER A LISTA DE EXERCÍCIOS QUE ESTARÁ DISPONÍVEL NO BLOG E NO DROPBOX.