

Recursão e TAD

Prof. Denio Duarte

Recursividade

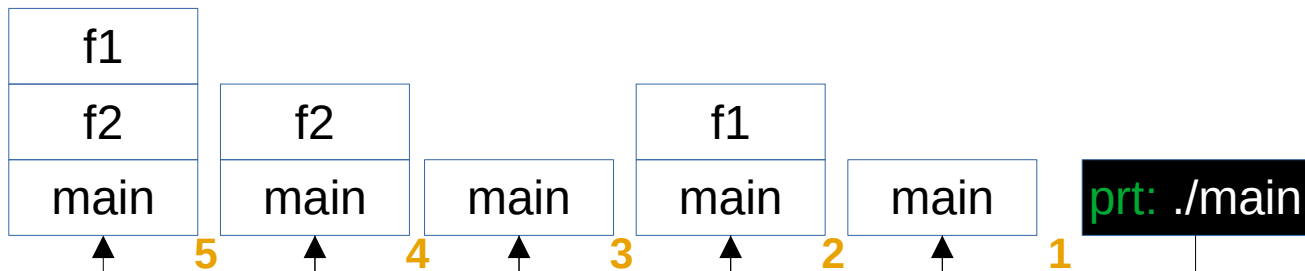
Recursividade

- É uma técnica de programação
- Baseda no conceito de uma função chamar ela mesma
- Alguns problemas são mais facilmente codificados utilizando a recursão



Recursividade

- Todo processo disparado por um programa ocupa um espaço da memória RAM
- Os processos de um programa são empilhados conforme a ordem que foram chamados (o último fica no topo da pilha)

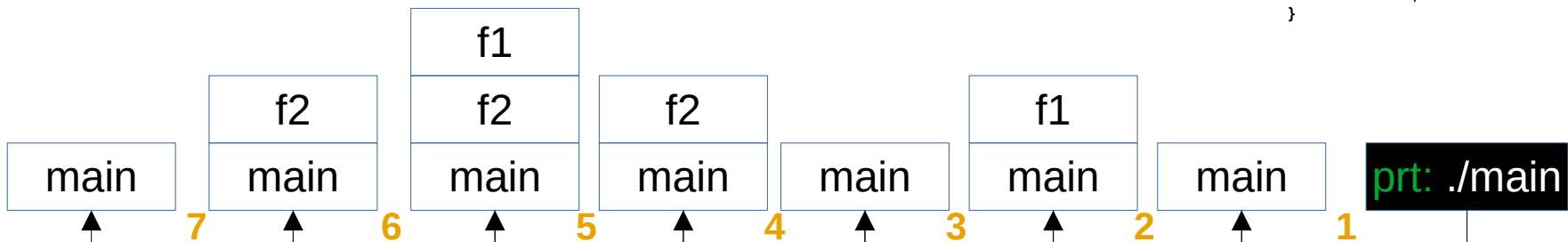


```
int f1()  
{  
    print("Um\n");  
    return 0;  
}  
int f2()  
{  
    f1();  
    return 1;  
}  
int main()  
{  
    f1();  
    f2();  
    return 0;  
}
```

Recursividade

- Todo processo disparado por um programa ocupa um espaço da memória RAM
- Os processos de um programa são empilhados conforme a ordem que foram chamados (o último fica no topo da pilha)

```
int f1()  
{  
    print("Um\n");  
    return 0;  
}  
int f2()  
{  
    f1();  
    return 1;  
}  
int main()  
{  
    f1();  
    f2();  
    return 0;  
}
```



Recursividade

- Funções iterativas
 - Funções tradicionais – que não se chamam
- Funções recursivas
 - São chamadas por elas mesmas
 - Podem causar um *looping* infinito

Recursividade

- Como usar isso para nosso benefício?
 - Quebramos o problema em partes menores, deixamos ele mais simples, e chamamos a função várias vezes até encontrar a resposta

Recursividade

- Como usar isso para nos ajudar a pensar?
 - Quebramos o problema e a função várias vezes até e



Recursividade

- Como usar isso para nosso benefício?
 - Quebramos o problema em partes menores, deixamos ele mais simples, e chamamos a função várias vezes até encontrar a forma mais simples
- Podemos decompor uma recursão por
 - **Caso base:** uma instância do problema solucionada facilmente
 - **Chamadas recursivas:** onde a função é definida em termos de si própria, realizando uma redução para seu caso básico


Recursividade

- Exemplo 1:

- Multiplicação através de adições

- $3 \times 4 = 3 + 3 + 3 + 3$, ou seja, $3+3+6$ ($3+3$) $\Rightarrow 3 + 9$ ($3+6$) $\Rightarrow 12$ ($3+9$)

- Definição formal

$m \times n$  $\begin{cases} \text{se } n==0 \text{ então } 0 \text{ (caso base)} \\ \text{caso contrário } m+(m \times (n-1)) \text{ (caso recursivo)} \end{cases}$

Recursividade

$m \times n$ $\left\{ \begin{array}{l} \text{se } n==0 \text{ então } 0 \text{ (caso base)} \\ \text{caso contrário } m+(m \times (n-1)) \text{ (caso recursivo)} \end{array} \right.$

	m		m	n						
3 x 4 =>	3	+	(3 x 3)							
	3	+	(3 x 2)							
	3	+	(3 x 1)							
	3	+	(3 x 0)							

	3	+	(9)							
3 x 4 <=	3	+	(9)							
		+	(6)							
		+	(3)							
		+	(0)							

Recursividade

$m \times n$ { se $n==0$ então 0 (caso base)
caso contrário $m+(m \times (n-1))$ (caso recursivo)

$$\begin{array}{lcl} 3 \times 4 \Rightarrow & 3 + (3 \times 3) & \\ & 3 + (3 \times 2) & \\ & 3 + (3 \times 1) & \\ & 3 + (3 \times 0) & \end{array} \quad \begin{array}{c} \uparrow \\ 3 \times 4 \leq 3 + (9) \\ 3 + (6) \\ 3 + (3) \\ 3 + (0) \end{array}$$

Diagram illustrating the recursive calculation of 3×4 . The left side shows the expansion of the recursive formula: $3 \times 4 \Rightarrow 3 + (3 \times 3)$, $3 + (3 \times 2)$, $3 + (3 \times 1)$, and $3 + (3 \times 0)$. A red arrow points down from the first line to the last, indicating the sequence of recursive calls. The right side shows the return values being calculated: $3 \times 4 \leq 3 + (9)$, $3 + (6)$, $3 + (3)$, and $3 + (0)$. Red arrows point up from the last line to the first, indicating the return values being passed back up the call stack. The final result is 12.

```
int multrec (int m, int n)
{
    if (n==0) return 0;
    return m+multrec(m,n-1);
}
```

base

recursão

Recursividade

$m \times n$ $\left\{ \begin{array}{l} \text{se } n==0 \text{ então } 0 \text{ (caso base)} \\ \text{caso contrário } m+(m \times (n-1)) \text{ (caso recursivo)} \end{array} \right.$

$$\begin{array}{rcl} 3 \times 4 = & 3 + (3 \times 3) & \\ & 3 + (3 \times 2) & \\ & 3 + (3 \times 1) & \\ & 3 + (3 \times 0) & \end{array} \quad \begin{array}{c} \uparrow \\ 3 \times 4 = \end{array} \begin{array}{rcl} 3 + (9) & \leftarrow 12 \\ 3 + (6) & \leftarrow 9 \\ 3 + (3) & \leftarrow 6 \\ 3 + (0) & \leftarrow 3 \end{array}$$

```
int multite(int m, int n)
{
    int res=0,i;
    for (i=1;i<=n;i++)
        res+=m
    return res;
}
```

```
int multrec (int m, int n)
{
    if (n==0) return 0;
    return m+multrec(m,n-1);
}
```


base
recursão

Recursividade

- Exemplo 2
 - Fatorial
 - O fatorial de um número é o resultado da multiplicação do número por seus antecessores até 1 (por definição o fatorial de 0 é 1)
 - $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
 - Definição formal

$$n! \begin{cases} \text{se } n==0 \text{ ou } n==1 \text{ então } 1 \text{ (caso base)} \\ \text{caso contrário } n \times (n-1)! \text{ (caso recursivo)} \end{cases}$$

Recursividade


$n!$ 
se $n==0$ ou $n==1$ então 1 (caso base)
caso contrário $n \times (n-1)!$ (caso recursivo)

```
int fat(int n)
{
    if (n==0 || n==1)
        return 1;
    return n * fat(n-1);
}
```

← base

← recursão

Recursividade

$n!$  se $n==0$ ou $n==1$ então 1 (caso base)
caso contrário $n \times (n-1)!$ (caso recursivo)

```
int fat_it(int n)
{
    int r=1, i;
    if (n==0 || n==1)
        return r;
    for (i=1; i<=n; i++)
        r*=i;
    return r;
}
```

```
int fat(int n)
{
    if (n==0 || n==1)
        return 1;
    return n * fat(n-1);
}
```

← base

← recursão

Recursividade

- Exemplos “bobinhos”

```
void impvetasc(int *m, int t)
{
    if (t < 1) return;
    impvetasc(m, t-1);
    printf("%d\n", m[t-1]);
}
:
int v[4]={1,2,3,4};
impvetasc(v, 4);
1
2
3
4
```

```
void impvetdesc(int *m, int t)
{
    if (t < 0) return;
    printf("%d\n", m[t-1]);
    impvetdesc(m, t-1);
}
:
int v[4]={1,2,3,4};
impvetdesc(v, 4);
4
3
2
1
```

Exercícios

1. Implemente uma função recursiva que, dados dois números inteiros base (b) e expoente (e), calcula o valor de b^e ($e \geq 0$).

$$b^e \begin{cases} \text{se } e==0 \text{ então } 1 & (\text{caso base 1}) \\ \text{se } e==1 \text{ então } b & (\text{caso base 2}) \\ \text{caso contrário } b \times b^{(e-1)} & (\text{caso recursivo}) \end{cases}$$

2. Implemente uma função recursiva que calcule o somatório se um vetor passado por parâmetro.

$$\sum_{i=0}^{(n-1)} v[i] \begin{cases} \text{se } n==0 \text{ então } v[0] & (\text{caso base 1}) \\ \text{caso contrário } v[n] + v[n-1] & (\text{caso recursivo}) \end{cases}$$

Exercícios

3. Implemente uma função recursiva que calcule o máximo divisor comum (mdc) entre dois números.

- Por exemplo, o mdc de 12 e 18 é 6

Definição (Algoritmo de Euclides):

$$\text{mdc}(m,n) \begin{cases} \text{se } n==0 \text{ então } m & (\text{caso base 1}) \\ \text{caso contrário } \text{mdc}(n,m\%n) & (\text{caso recursivo}) \end{cases}$$

TAD

Tipos Abstratos de Dados

Introdução

- Se considerarmos a definição de estruturas complexas da última aula (**structs**), podemos criar tipos compostos e que representam de maneira mais fidedigna elementos do mundo real
 - Lembrando que a linguagem C tem apenas o conjunto restrito de tipos: int, float, char ... e operações sobre eles: +, -, * ...
- Utilizando **structs**, podemos criar tipos mais complexos e operações que possam ser executadas sobre estes tipos
 - Podemos criar um **tipo fração** (numerador e denominador) e operações sobre o tipo criado (ou o tipo abstrato de dado criado)

```
tipofrac myfrac1, myfrac2, myfrac3;  
myfrac1=atrib_fracao(4,8);  
myfrac2=atrib_fracao(3,8);  
myfrac3=soma_frac(myfrac1,myfrac2);  
imp_frac(myfrac3); // pode imprimir 7/8
```

```
tipofrac atrib_fracao (int n, int d)  
{  
    tipofrac f;  
    f.numerador=n;  
    f.denominador=d;  
    return f;  
}
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana? (distância entre os pontos)

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
int main(){  
    Ponto p1 = {-2,3}, p2 = {-5,-9};  
    double distancia;  
  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
  
    printf("%.5lf \n", distancia);  
  
    return 0;  
}
```


Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

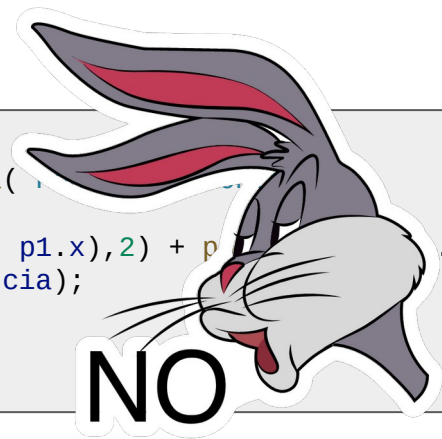
Se eu precisar utilizar essa função de novo, eu tenho que copiar colar no novo programa?

Introdução

- Imagine a representação de ponto em um plano de duas dimensões
 - Teremos duas coordenadas (x,y)
 - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana(Ponto p1, Ponto p2)  
{  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```



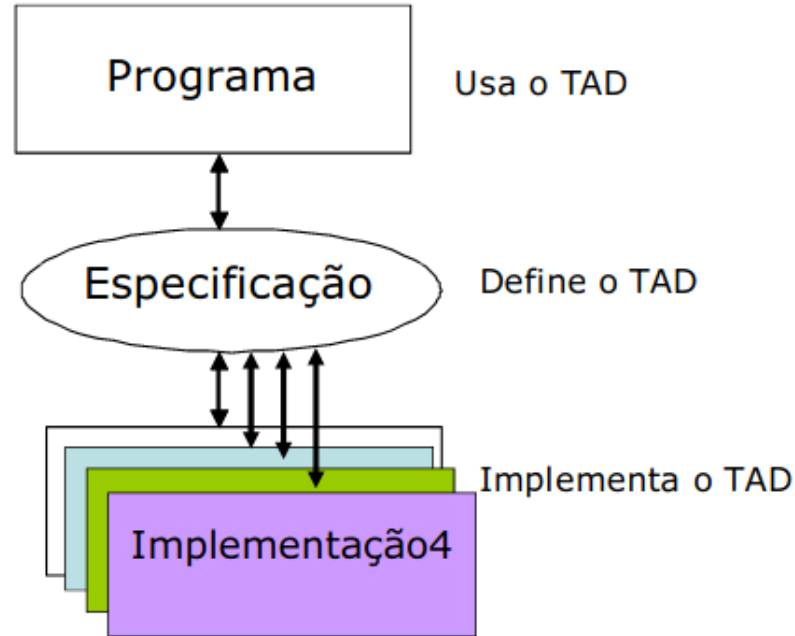
Se eu precisar utilizar essa função de novo, eu tenho que copiar colar no novo programa?

TAD

- Utilizamos nesses casos os Tipos Abstratos de Dados (TAD)
 - TAD especifica o tipo de dado (domínio de operações) sem referência a detalhes da implementação
 - Permite maior flexibilidade no desenvolvimento, principalmente na manutenção do código
 - O programador não sabe como o TAD foi implementado, as implementações ficam “escondidas”
- O TAD especifica tudo o que precisa saber para usar um determinado tipo
- O TAD divide o sistema em:
 - Programas de Usuário
 - Implementação

TAD

- Utilizamos nesse contexto (TAD)
 - TAD específico para cada implementação
 - Permite maior reutilização de código
 - O programador não precisa lidar com os detalhes “escondidos”
- O TAD especifica a interface a ser utilizada
- O TAD divide o trabalho em partes menores
 - Programas de interface
 - Implementações



TAD)

ênncia a detalhes da

a manutenção do código
implementações ficam

n determinado tipo

TAD

- O TAD é representado por dois documentos
 - Especificação
 - Chamado de arquivo de cabeçalho em C ([header](#))
 - É nomeado como [nome_arquivo.h](#)
 - Implementação
 - Efetivamente implementa as funções declaradas no cabeçalho
 - Geralmente tem o mesmo nome do .h mas .c ([nome_arquivo.c](#))

TAD

- Cabeçalho

planoCartesiano.h

```
typedef struct {  
    int x;  
    int y;  
} Ponto;  
  
double distanciaEuclidiana( Ponto p1, Ponto p2);  
void setPonto( Ponto *p1, int x, int y);
```

TAD

- implementação

planoCartesiano.c

```
#include <stdlib.h>
#include <math.h>
#include "planoCartesiano.h"

double distanciaEuclidiana( Ponto p1, Ponto p2){
    double distancia;
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);
    distancia = sqrt(distancia);
    return distancia;
}

void setPonto( Ponto *p1, int x, int y){
    p1->x=x;
    p1->y=y;
    return;
}
```


TAD

- Uso por outros programas

```
#include <stdlib.h>
#include <math.h>
#include "planoCartesiano.h"
int main()
{
    Ponto pt1, pt2;
    setPonto(&pt1, 10, 15);
    setPonto(&pt2, 60, 35);
    printf("%.5lf", distanciaEuclidiana(pt1, pt2));
    return 0;
}
```

- Compilação `gcc -Wall principal.c planoCartesiano.c -o principal`

TAD

- Exercício
 - Implemente uma TAD que represente frações e as operações sobre as mesmas
 - Atribuição
 - Multiplicação
 - Divisão
 - Opcional:
 - Adição e subtração (tem que calcular o MMC)

```
int mmc(int a,int b)
{
    int div;
    if(b == 0) return a;
    else
        div = (a*b) / (mdc(a,b));
    return (div);
}
```