Curso: Ciência da Computação

Disciplina: Estruturas de Dados

Profª Luciana Ap. Oliveira Betetto

Aula 05 - Recursividade

1. Introdução

Em Ciência da computação, a ***recursividade*** é a definição de uma ***subrotina*** (função ou método) ***que pode chamar a si mesma***. Um exemplo de aplicação da recursividade pode ser encontrado nos analisadores sintáticos recursivos (nos compiladores) para linguagens de programação.

2. Algoritmos recursivos

Dividir para conquistar é a idéia básica da técnica de programação, a qual consiste em pegar um determinado problema e dividi-lo em problemas menores e ao se resolver cada um dos problemas menores, chega-se a solução de todo o problema. Cada um dos problemas ***"menores"*** são resolvidos por uma função (subrotina). E quando numa linguagem de programação uma ***função chama ela mesma***, ocorre a ***recursão***.

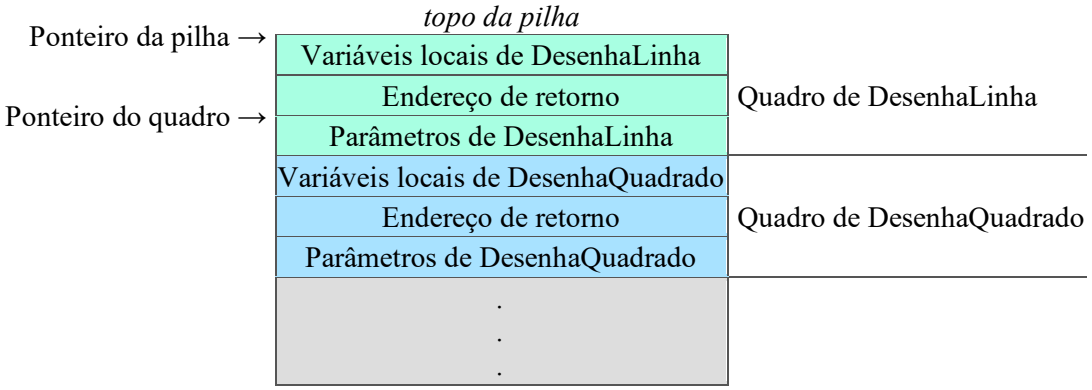
Quando uma função é chamada, o computador registra as várias instâncias de uma função em uma ***pilha de chamada****, (ou ***pilha de execução***). Embora outros métodos possam ser usados, toda função recursiva pode ser transformada em uma função iterativa usando uma pilha.

*Uma **pilha de chamada** (ou **pilha de execução**) é uma pilha que armazena informações sobre as sub-rotinas (funções) conforme elas vão sendo chamadas pelo programa. Nessa pilha são registrados o ponto em que cada sub-rotina ativa (isto é, que foi chamada) deve retornar o controle de execução quando termina de executar.

Por exemplo, se uma sub-rotina `DesenhaQuadrado` chama (invoca) a sub-rotina `DesenhaLinha` em quatro pontos diferentes, o código de `DesenhaLinha` deve saber como retornar a `DesenhaQuadrado`. Isso geralmente é feito adicionando o endereço de retorno na pilha de chamada após a chamada da sub-rotina.

Sendo organizada como uma pilha, quem invoca a sub-rotina **empilha** é o endereço de retorno. Quando termina sua execução, a sub-rotina invocada **desempilha** o endereço de retorno, desviando a execução para aquele endereço. Se durante sua execução a sub-rotina invocada também invocar outra sub-rotina, o endereço de retorno também será empilhado, e assim por diante. Quando esse processo de empilhamento consome todo o espaço alocado para a pilha de chamada, ocorre um erro chamado estouro de pilha.

Uma pilha de chamada é composta por quadros (muitas vezes chamados de registros de ativação) que dependem da implementação (Estrutura de Dados) e que contêm informações sobre o estado da sub-rotina. Cada quadro da pilha corresponde a uma chamada de sub-rotina que ainda não retornou. Por exemplo, se uma sub-rotina `DesenhaLinha` está em execução, e foi chamada por `DesenhaQuadrado`, o topo da pilha de chamada pode ser estruturada da seguinte forma:



O quadro de pilha no topo da pilha se refere a sub-rotina em execução. Geralmente o quadro inclui espaço para as variáveis locais, o endereço de retorno para a sub-rotina que invocou a outra (quadro de pilha seguinte) e os parâmetros passados para a sub-rotina. A pilha é geralmente acessada através do **registrador** "**ponteiro da pilha**", que também serve para indicar o **topo da pilha**. Alternativamente, a memória do quadro pode ser acessada por outro registrador, o "ponteiro do quadro", que geralmente aponta para um ponto específico do quadro, como o endereço de retorno.

Cada quadro possui um tamanho específico, determinado pelo número e tamanho de parâmetros e variáveis locais, ainda que o tamanho seja fixo para diferentes chamadas de uma mesma sub-rotina. Entretanto, algumas linguagens suportam alocação dinâmica de memória para variáveis locais na pilha de chamada, de forma que o tamanho do quadro de uma sub-rotina não seja fixo, variando de acordo com a chamada.

Existe uma pilha de chamada para cada ***thread**** sendo executada, ainda que mais pilhas possam ser criadas para o tratamento de ***sinais**** ou para ***multitarefa cooperativa****.

Em linguagens de alto nível, detalhes da pilha de chamada são geralmente escondidos do programador. Eles têm acesso somente a lista de sub-rotinas empilhadas, e não à memória da pilha de chamada em si. Por outro lado, a maioria das linguagens de montagem requerem que programador manipule a pilha de chamada.

**** Esses termos serão vistos na disciplina de SO (Sistema Operacional).***

Toda função que puder ser produzida por um computador que pode ser escrita de forma recursiva pode também ser feita de forma iterativa, e vice-versa, isto é, e toda função que pode ser feita de forma iterativa pode também ser feita de forma recursiva.

Algumas linguagens desenvolvidas para ***programação lógica*** e ***programação funcional*** permitem recursões como única estrutura de repetição, ou seja, não podem usar laços tais como os produzidos por comandos como ***for***, ***while*** ou ***repeat***. Tais linguagens geralmente fazem uma ***recursão em cauda*** tão eficiente quanto a iteração.

A recursão está profundamente enraizada na ***Teoria da computação***, uma vez que a equivalência teórica entre as funções recursivas e as ***Máquinas de Turing*** está na base das idéias sobre a universalidade do computador moderno.

3. Programação recursiva

Em geral, uma definição recursiva é definida por casos: um número limitado de ***casos base*** e um ***caso recursivo.*** Os casos base são geralmente situações triviais e não envolvem recursão.

Um exemplo comum usando recursão é a função para calcular o **fatorial** de um natural N. Nesse caso, no **caso base** o valor de **0! é 1**. No **caso recursivo**, dado um **N > 0**, o valor de N! é calculado multiplicando por N o valor de (N-1)!, e assim por diante, de tal forma que N! tem como valor N * (N-1) * (N-2) * ... * (N-N)!, onde (N-N)! representa obviamente o caso base. Definição matemática:

$$fatorial(n) = \begin{cases} 1 & se\ n = 0 \\ n * fatorial(n-1) & se\ n > 0 \end{cases}$$

Linguagem Python:

```
input = int(input("Digite um valor: "))

def fat(n): # recebe um número
      if (n == 0):
            return 1

      else:
            return n * fat(n-1)

print(fat(input))
```

Por exemplo: **3! = 6** (isto é, fatorial de 3 = 6) ,

4! = 24 (fatorial de 4 é 24) → 4! = 4x3x2x1 = 24

Veja um exemplo de como seria montada a pilha de chamada dessa função:
Ocorre da seguinte forma:

```
fat(3) = 3 * fat(3 – 1)
       = 3 * fat(2)
       = 3 * 2 * fat(2 – 1)
       = 3 * 2 * fat(1)
       = 3 * 2 * 1 * fat(1 – 1)
       = 3 * 2 * 1 * fat(0)
       = 3 * 2 * 1 * 1
       = 6
```

Linguagem C: ***função recursiva*** para calcular um n° fatorial qualquer.

```
1   #include <stdio.h>
2   #include <locale.h>
3   int fat(int n)
4   {
5       if (n==0)
6               return 1;
7       else
8               return (n*fat(n-1));
9   }
10  int main()
11  {
12      setlocale(LC_ALL,"Portuguese");
13      int num;
14      printf("\nDigite um número inteiro para cálculo do fatorial: ");
15      scanf ("%d", &num);
16      printf ("Fatorial de %d =  %d", num, fat(num));
17      return (0);
18   }
```

Na linha 8 (ainda dentro da função), a função faz uma chamada a ela mesma. Só ocorre a parada quando n=1

n=1 é o **caso base**

chamar a si mesma é o **caso recursivo**

Veja a mesma ***função de forma iterativa***:

É importante entender que a solução iterativa requer duas variáveis temporárias. Em geral, formulações recursivas de algoritmos são frequentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas.

Linguagem Python:

```
def main():
    # leia o valor de n
    n = int(input("Digite um número inteiro não-negativo: "))

    # inicializações
    i = 1  # contador
    fat = 1

    # calcule n!
    while i <= n:
        fat = fat * i
        i = i + 1

    print("O valor de {n}! = " %(n, fat))

main()
```

Os comandos ***for*** e ***while*** são soluções **iterativas**, isto é, **repetitivas.**

Linguagem C:

```
#include <stdio.h>
#include <locale.h>
int fat( int n )
{
   int i, aux;
   aux=1;
   for (i = 1; i<= n; i++)
      aux = aux * i;
      return ( aux );
}

int main()
{
   setlocale(LC_ALL,"Portuguese");
   int  num;
   printf(" digite um n° inteiro para calcular o fatorial");
   scanf("%d", &num);
   printf(" Fatorial de %d = %d ", num, fat(num) );
}
```

Outro exemplo de recursividade está no cálculo de Fibonacci. Esta é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores. A sequência recebeu o nome do matemático italiano Leonardo de Pisa, mais conhecido por Fibonacci, que descreveu, no ano de 1202, o crescimento de uma população de coelhos, a partir desta. Esta sequência já era, no entanto, conhecida na antiguidade.
Os números de Fibonacci são, portanto, os números que compõem a seguinte sequência:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 ...

Linguagem Python:

```
def fib(n):
       if(n == 1 or n == 2):
              return 1
       return fib(n-1) + fib(n - 2)

print(fib(7)) # exibe o sétimo termo de Fibonacci = 13
```

4. Recursão *versus* Iteração

No exemplo do fatorial, a implementação iterativa tende a ser ligeiramente mais rápida na prática do que a implementação recursiva, uma vez que uma implementação recursiva precisa registrar o estado atual do processamento de maneira que ela possa continuar de onde parou após a conclusão de cada nova execução subordinada do procedimento recursivo. Esta ação consome tempo e memória.