IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Encontro 05 – Recursividade

INSTITUTO METRÓPOLE DIGITAL

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

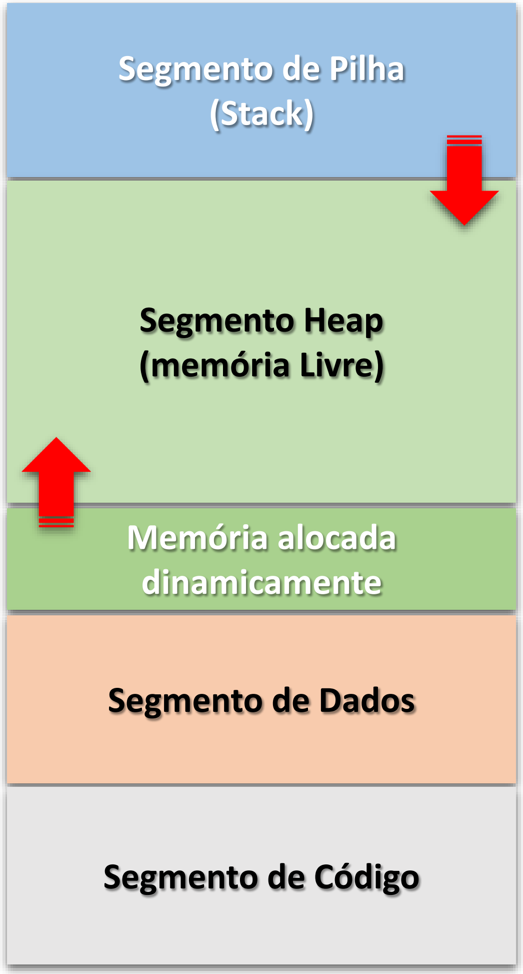
Objetivos da aula

- Aprofundar-se nos conceitos de recursividade
    - Segmentos de memória e suas utilidades
    - Entender como a recursividade funciona na memória do computador, cujas instruções são sequenciais
    - Identificar as vantagens e desvantagens do uso de recursividade
    - Conhecer técnicas para, se necessário, remover a recursividade

- Ao final da aula espera-se que o aluno seja capaz de:
    - Empregar adequadamente o uso de recursividade
    - Se necessário, remover a recursividade de seu algoritmo

Memória de trabalho do computador (RAM)

- A memória de trabalho do computador (RAM) é subdividida em vários segmentos lógicos dentro de um programa
  - **Segmento de pilha (stack)**: onde sub-rotinas e métodos alocam temporariamente suas variáveis locais
  - **Segmento heap**: onde variáveis dinâmicas são alocadas (tempo de execução)
    - – Bastante útil quando não se sabe de antemão quantas variáveis de determinado tipo serão necessárias para o programa
  - **Segmento de dados**: onde variáveis globais e estáticas são alocadas (tempo de compilação)
  - **Segmento de código**: onde instruções de máquina do programa são encontradas



Recursividade

- Conceito fundamental em Matemática e Ciência da Computação
  - Uma função recursiva é definida em termos dela mesma
  - Um programa (ou subrotina) recursivo é um programa que chama a si mesmo
- Conceito poderoso que permite definir conjuntos infinitos com comandos finitos
- Em termos gerais, a recursividade é uma estratégia que pode ser utilizada sempre que o cálculo de uma função para o valor n, pode ser descrita a partir do cálculo desta mesma função para o termo anterior (n-1)
- Exemplos
  - Sequência de Fibonnaci, Função fatorial, Árvore

Recursividade

- Recursividade é uma idéia inteligente que desempenha um papel importante na programação e na ciência da computação em geral
  - o Recursividade é o mecanismo básico para repetições nas linguagens funcionais
  - o São sinônimos: recursividade, recursão e recorrência
- De modo geral, uma definição de função recursiva é dividida em duas partes:
  - o Há um ou mais **casos base** que dizem o que fazer em situações simples, onde não é necessária nenhuma recursão
    - – Nestes casos **a resposta pode ser dada de imediato**, sem chamar recursivamente a função sendo definida
    - – **Isso garante que a recursão eventualmente possa parar (condição de parada)**
  - o Há um ou mais **casos recursivos** que são mais gerais, e definem a função em termos de uma **chamada mais simples a si mesma**

Recursividade

- Nenhum programa nem função pode ser exclusivamente definido por si
  - Um programa seria um **loop infinito**
  - Uma função teria definição circular
- Condição de parada
  - Permite que o procedimento pare de se executar
  - Por exemplo, **F(x) > 0** onde **x** é decrescente
- Os casos base definem as condições de parada
- **Profundidade** é o número de vezes que uma rotina recursiva chama a si própria, até obter o resultado
  - Muitas vezes a profundidade de uma recursão não é tão clara, até mesmo para definições simples

Recursividade: exemplo de uso

- Número (ou termo) de Fibonnacci (0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987..)
  - A essência do algoritmo é recursiva
  - A implementação natural (intuitiva) é, portanto, recursiva

```
#include<iostream>
using namespace std;
//função recursiva para o numero fibonacci
long int fibonacci(int n)
{
    if (n == 0) return 0;        Caso base
    if (n == 1) return 1;        Caso base
    return fibonacci(n-1) + fibonacci(n-2);        Caso recursivo
}
int main() {
    int n; cout << "Entre com o termo a ser calculado: ";
    cin >> n;
    cout << "Fib(" << n << ") : " << fibonacci(n) << endl;
    return 0;
}
```

Tipos de recursão

- **Direta**: quando numa subrotina existe uma chamada para a própria subrotina, independentemente dos valores dos parâmetros
  - **Simples**:

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

  - **Múltipla**

$$fib\ (n) = \begin{cases} n & , se\ n \in \{0,1\} \\ fib(n-1)\ + fib(n-2) & , se\ n > 1 \end{cases}$$

Tipos de recursão

- **Indireta**: Duas ou mais subrotinas são conectadas através de uma cadeia de chamadas sucessivas que acaba retornando à primeira que a desencadeou

$$par\ (n) = \begin{cases} verdadeiro & , se\ n = 0 \\ impar(n-1) & , se\ n > 0 \end{cases}$$

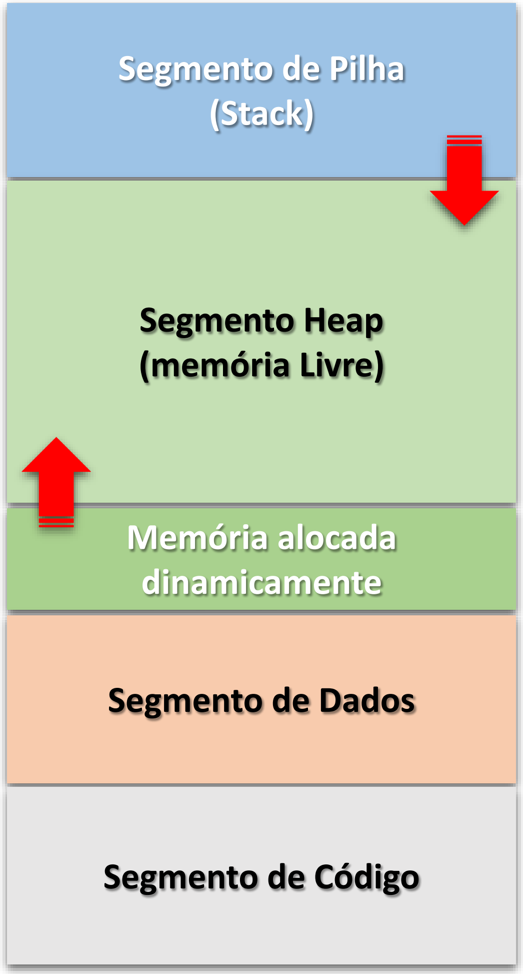
$$impar\ (n) = \begin{cases} falso & , se\ n = 0 \\ par(n-1) & , se\ n > 0 \end{cases}$$

- **Aninhada**

$$ack\ (n, m) = \begin{cases} m+1 & , se\ n = 0 \\ ack(n-1, m) & , se\ n > 0; m = 1 \\ ack\big(n-1, ack(n, m-1)\big) & , se\ n > 0, m = 1 \end{cases}$$

Como funciona internamente?

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um Registro de Ativação na Pilha de Execução do programa (**stack**)
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou a função
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função



Como funciona internamente?

- As chamadas são empilhadas na stack
  - Variáveis locais e seus valores são conservados
  - O estado é retornado ao voltar da chamada
- Em geral, a área alocada para a stack é menor que a heap
  - Um grande número de chamadas recursivas pode estourar a stack (**stack overflow**)
  - A maior causa de estouro de pilha é a **recursão de cauda**
- Tem-se uma **recursão de cauda** quando a última ação de uma função é a chamada recursiva
  - Na maioria das linguagens, a recursão de cauda é tratada pelo compilador, gerando uma versão iterativa correspondente

Recursão e iteração

- Algoritmos recursivos que apresentam recursão em cauda, possuem uma solução iterativa (não recursiva) sempre mais eficiente
- Recursão e iteração possuem o mesmo poder de expressividade
  - Algumas linguagens funcionais não possuem instruções para laços (while, for...)
  - Todo laço é realizado de forma recursiva
- É possível traduzir a recursão para a forma iterativa...
  - Eliminando a recursão de cauda
  - Manipulando com índices
  - Usando uma estrutura auxiliar para "simular" a stack (pilha)
- Mesmo algumas sub-rotinas que não possuem recursão de cauda podem se beneficiar da estratégia (com algumas pequenas adaptações)

Recursão e iteração

- Na recursão de cauda a chamada recursiva está no final do código, tendo como função criar um laço que será repetido até a condição de parada
- Assim, para eliminar a recursão de cauda:
    - Se uma rotina F(x) tem como última instrução uma chamada recursiva F(y), então troca-se F(y) pela atribuição $x \leftarrow y$, seguido de um desvio para o início de F
    - Utiliza-se uma repetição condicionada à expressão de teste (condição de parada) usada na versão recursiva

Recursão e iteração

- Removendo a recursão de cauda
  - o Transformando a versão recursiva em iterativa (forma geral)

```
int funcao( parametros )
{
 if( condicao )
   return caso_base( parametros );
 operacao1;
 operacao2;
 operacao3;
 ajusta( parametros );
 return funcao( parametros );
}
```

```
int funcao( parametros )
{
 int result = caso_base( parametros);
 while( !condicao ) {
   operacao1;
   operacao2;
   operacao3;
   ajusta( parametros );
   result = caso_base( parametros );
  }
 return result;
}
```

Recursão e iteração

- Transformando a versão recursiva em iterativa (forma geral)
  - o Exemplo de função de potência: base^{expoente}

```
int pot(int base, int exp)
{
  if (!exp) // caso base
    return 1;
  /* else */
  // operação + caso recursivo
  return (base*pot(base, exp-1));
}
```

```
int potIterarivo(int base, int exp)
{
  int result = 1;
  while (exp)
  {
    result *= base; // operação
    exp=exp-1; // ajusta parâmetros
  }
  return result;
}
```

Quando vale a pena usar recursividade?

- Recursividade vale a pena para Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
  - Dividir para Conquistar (Ex. Quicksort)
  - Caminhamento em Árvores (pesquisa, backtracking)
- É sempre importante avaliar as complexidades de tempo e espaço (devido a pilha de execução) do algoritmo recursivo
  - Com isso, nota-se que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos
- Dificuldade na depuração de programas, particularmente se a recursão for profunda
- Então, por que usar recursão?
  - Se bem empregada, torna o algoritmo muito elegante, isto é, claro, simples e conciso

Alguma Questão?