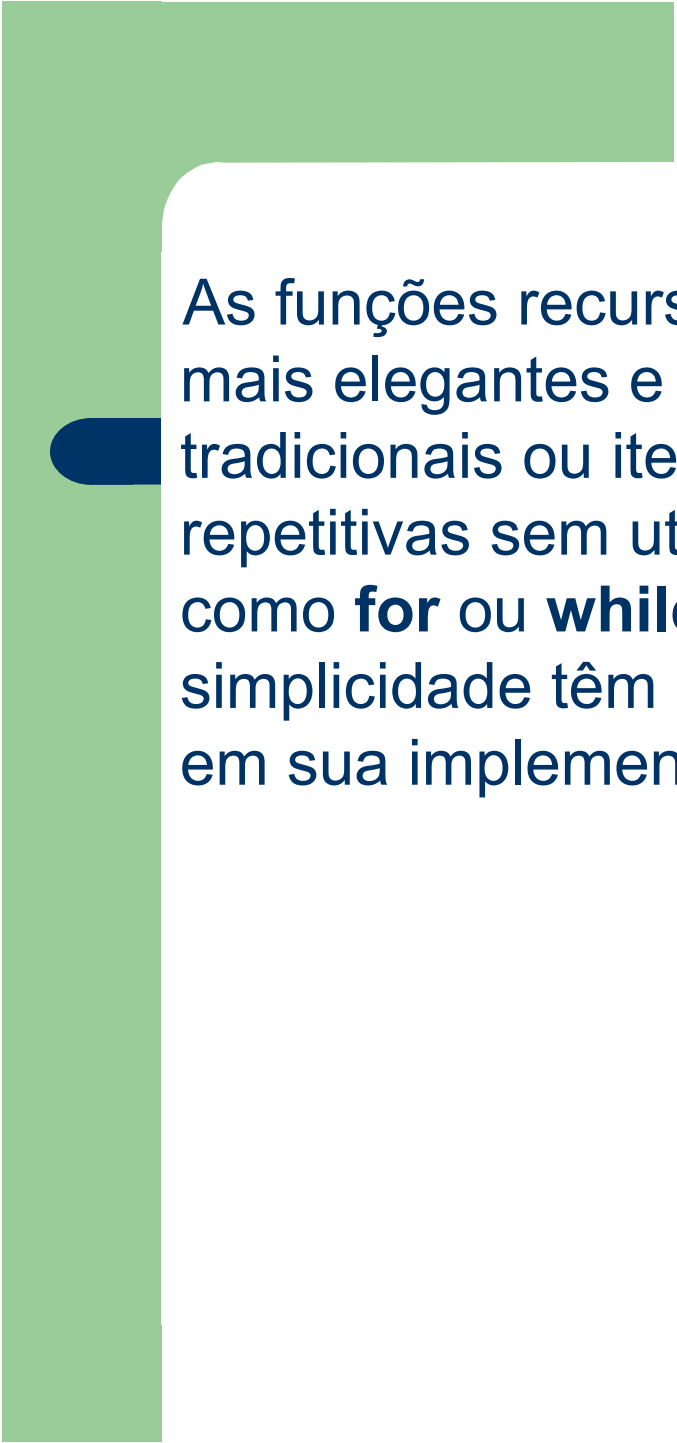


# **Recursividade na Linguagem C**

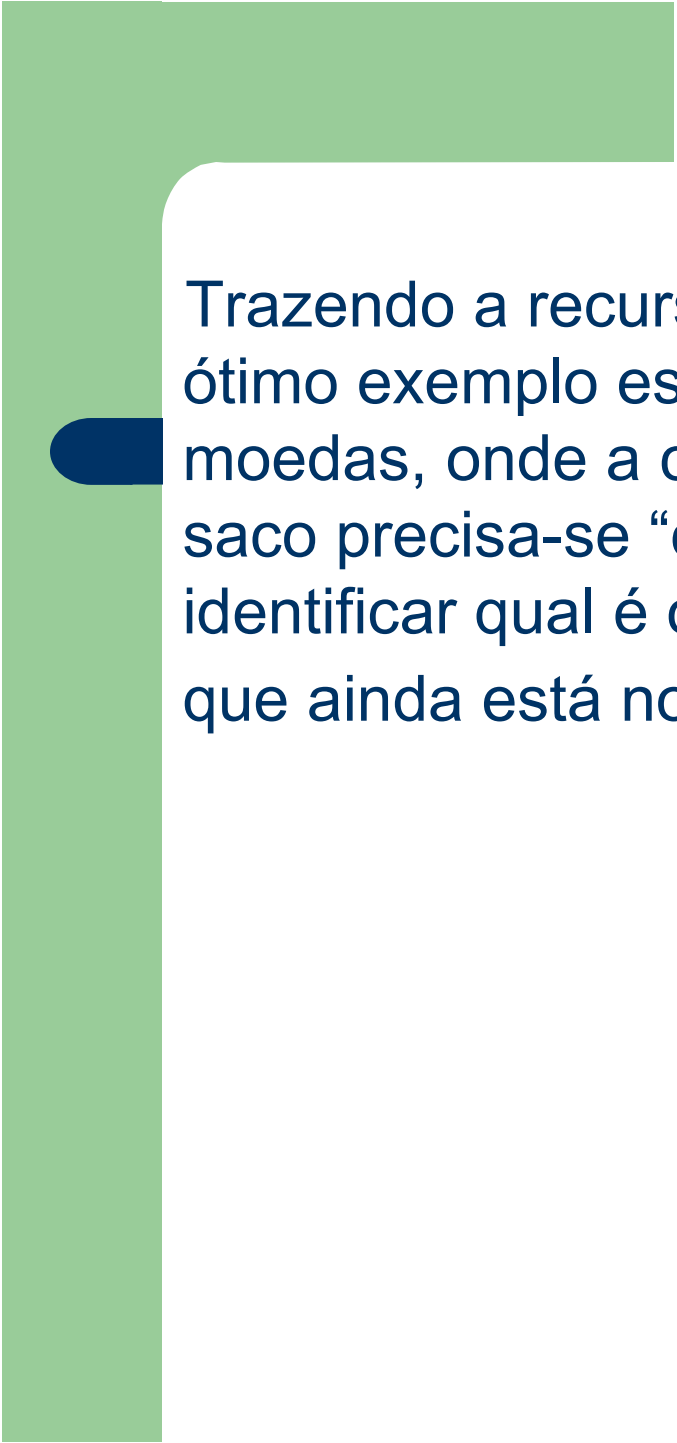


# Função Recursiva

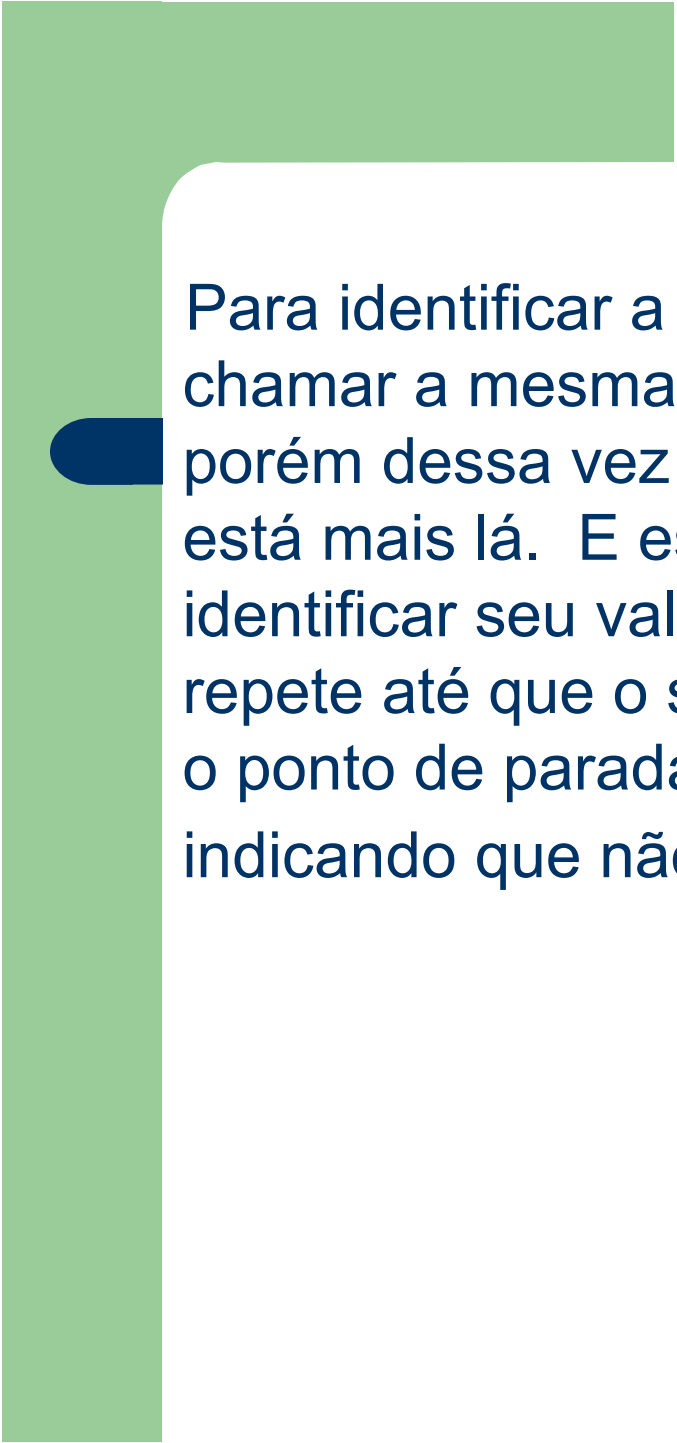
- Função recursiva é aquela que chama a si própria. Uma função poderá também ser considerada recursiva se chamar outras funções que, em algum momento, chamem a primeira função, tornando esse conjunto de funções um processo recursivo.



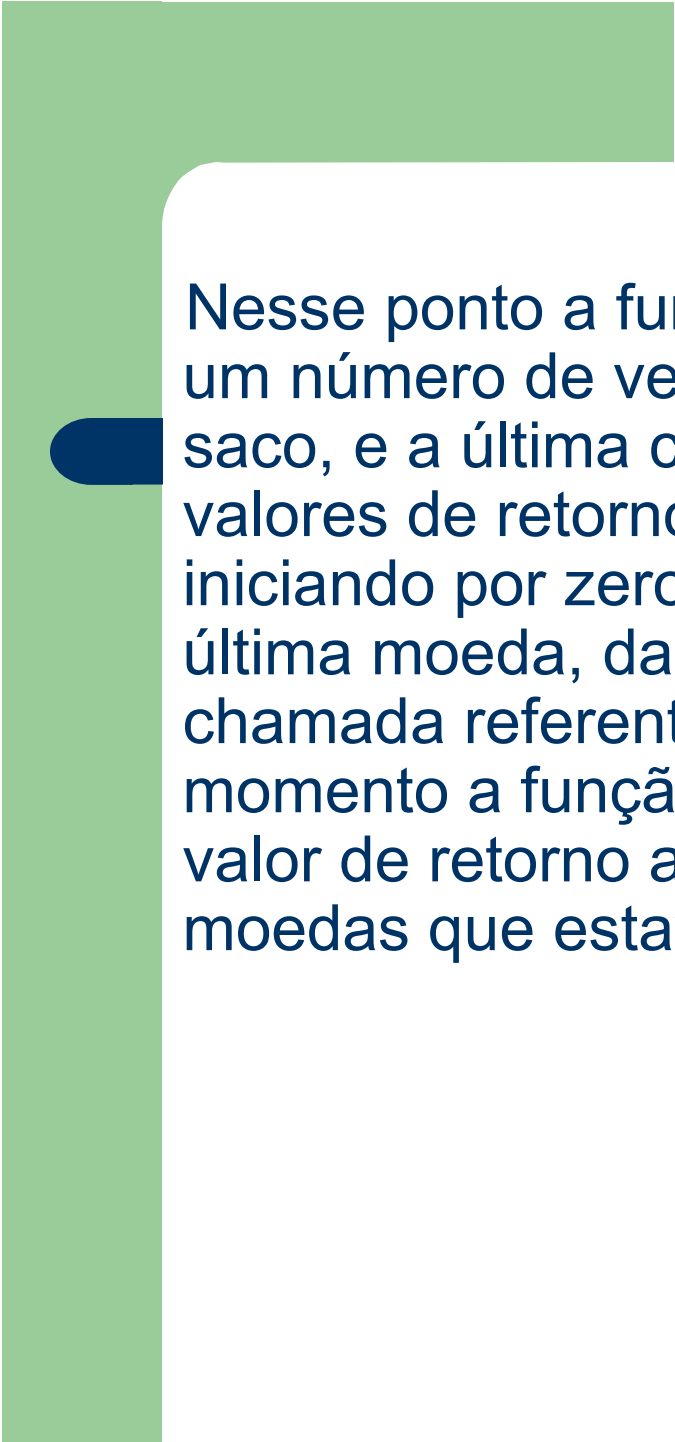
As funções recursivas são em sua maioria soluções mais elegantes e simples, se comparadas a funções tradicionais ou iterativas, já que executam tarefas repetitivas sem utilizar nenhuma estrutura de repetição, como **for** ou **while**. Porém essa elegância e simplicidade têm um preço que requer muita atenção em sua implementação.



Trazendo a recursividade para o nosso cotidiano um ótimo exemplo está na ação de contar um saco de moedas, onde a cada ato de retirar uma moeda do saco precisa-se “contar dinheiro” que corresponde a identificar qual é o valor da moeda e somá-la à quantia que ainda está no saco.

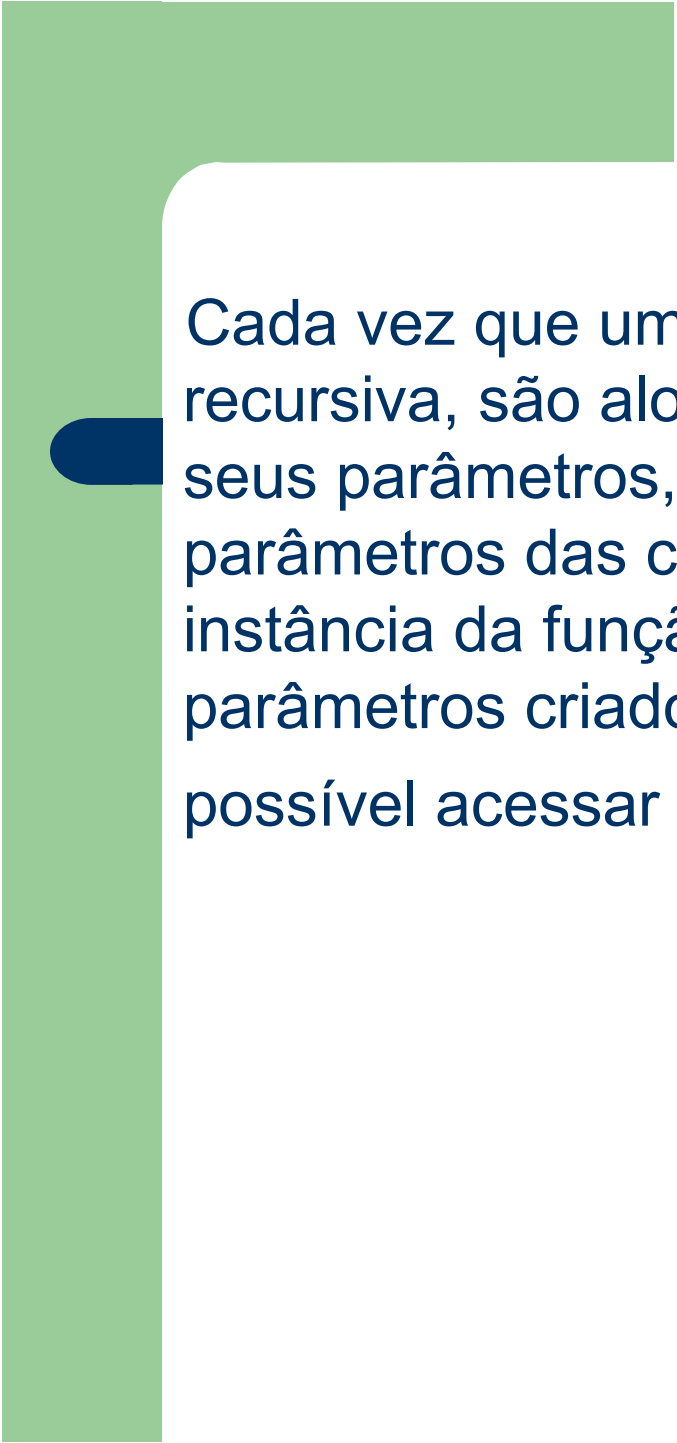


Para identificar a quantia que ainda está no saco basta chamar a mesma função “contar dinheiro” novamente, porém dessa vez já considerando que essa moeda não está mais lá. E este processo de retirar uma moeda, identificar seu valor e somar com o restante do saco se repete até que o saco esteja vazio, quando atingiremos o ponto de parada e a função retornará o valor zero, indicando que não há mais moedas no saco.



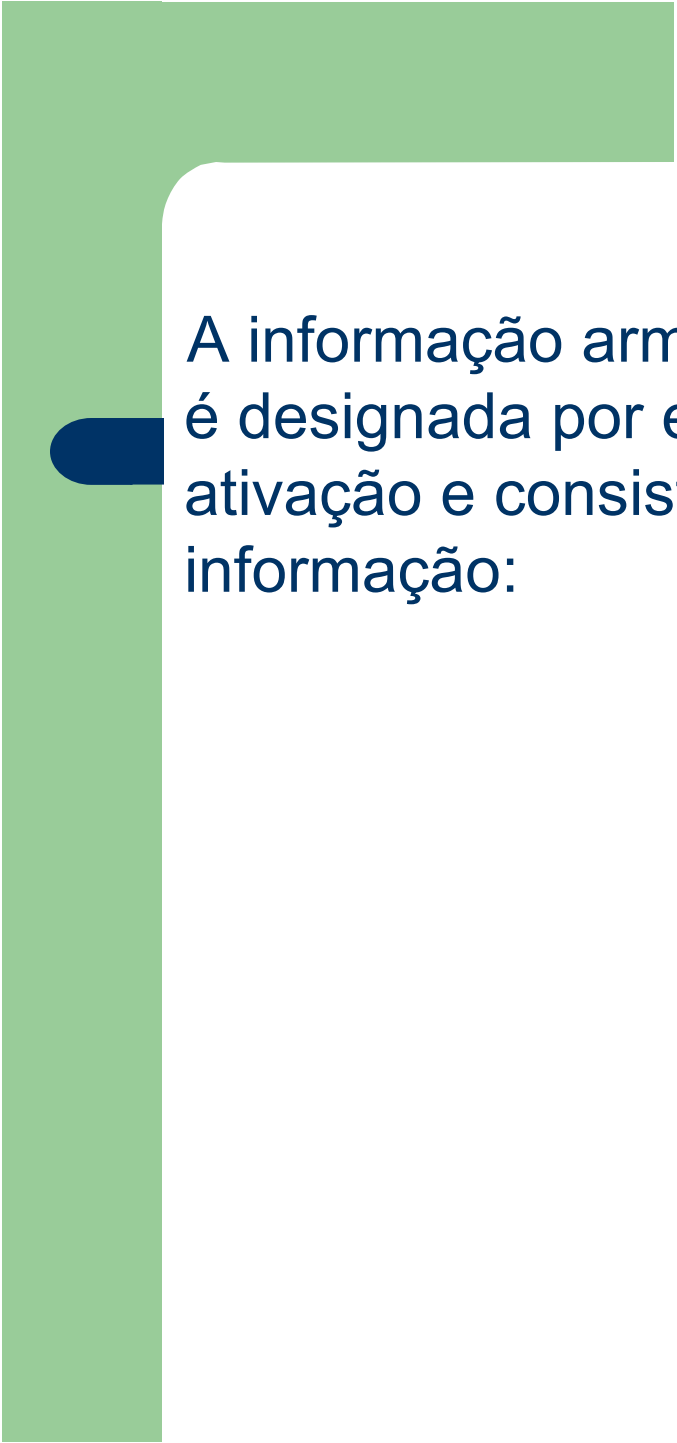
Nesse ponto a função “contar dinheiro” foi chamada um número de vezes igual a quantidade de moedas no saco, e a última chamada começa a devolver os valores de retorno de cada instância da função, iniciando por zero (saco vazio), somado ao valor da última moeda, da penúltima, etc, até retornar à primeira chamada referente a primeira moeda, e nesse momento a função inicial se encerra trazendo como valor de retorno a soma dos valores de todas as moedas que estavam no saco.

Uma função pode chamar a si própria por um número limitado de vezes. Esse limite é dado pelo tamanho da pilha (que poderá ser melhor compreendido após a apresentação do trabalho do grupo B). Se o valor correspondente ao tamanho máximo da pilha for atingido, haverá um estouro da pilha ou “Stack Overflow”. Não conseguimos resultados muito conclusivos na avaliação desse estouro de pilha nos testes realizados, porém deu pra perceber num programa para gerar uma seqüência de Fibonacci com valores grandes, que o programa estava usando acima de 90% dos recursos da CPU, o que demonstra como é pesado para o computador realizar uma tarefa recursiva.



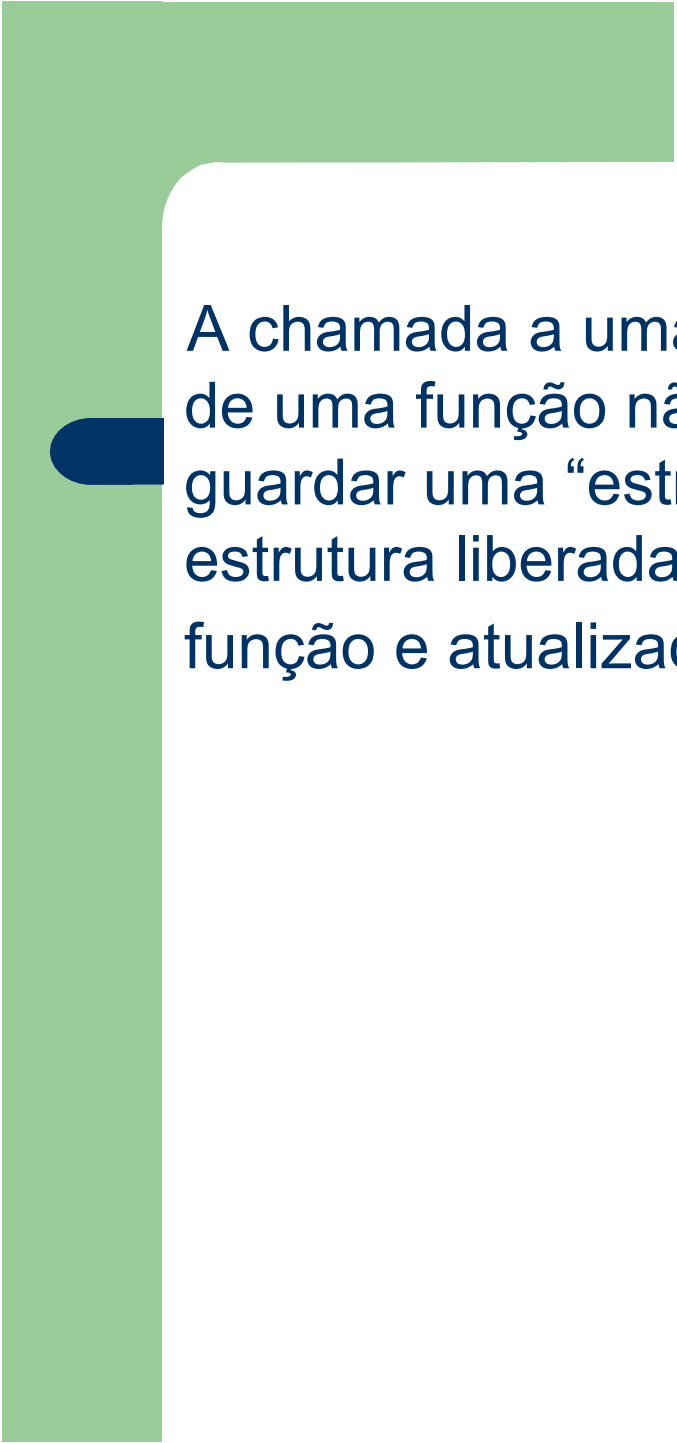
Cada vez que uma função é chamada de forma recursiva, são alojados e armazenados uma cópia dos seus parâmetros, de modo a não perder os valores dos parâmetros das chamadas anteriores. Em cada instância da função, só são diretamente acessíveis os parâmetros criados para esta instância, não sendo possível acessar os parâmetros das outras instâncias.





A informação armazenada na chamada de uma função é designada por estrutura de invocação ou registro de ativação e consiste basicamente na seguinte informação:

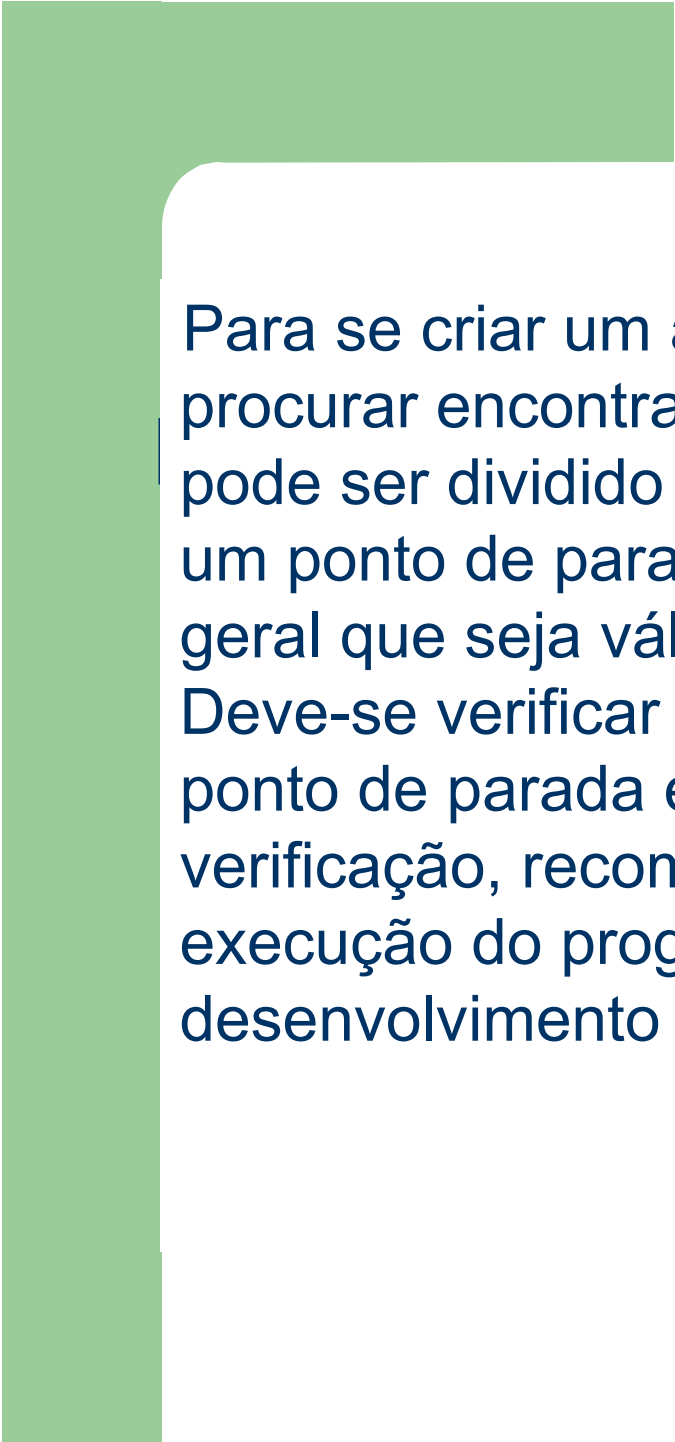
- Endereço de retorno (quando a função terminar o programa deve continuar a sua execução na linha seguinte à invocação da função)
- Estado dos registros e flags da CPU
- Variáveis passadas como argumentos para a função (por valor, referência, etc.)
- Variável de retorno (por valor, referência, etc.)



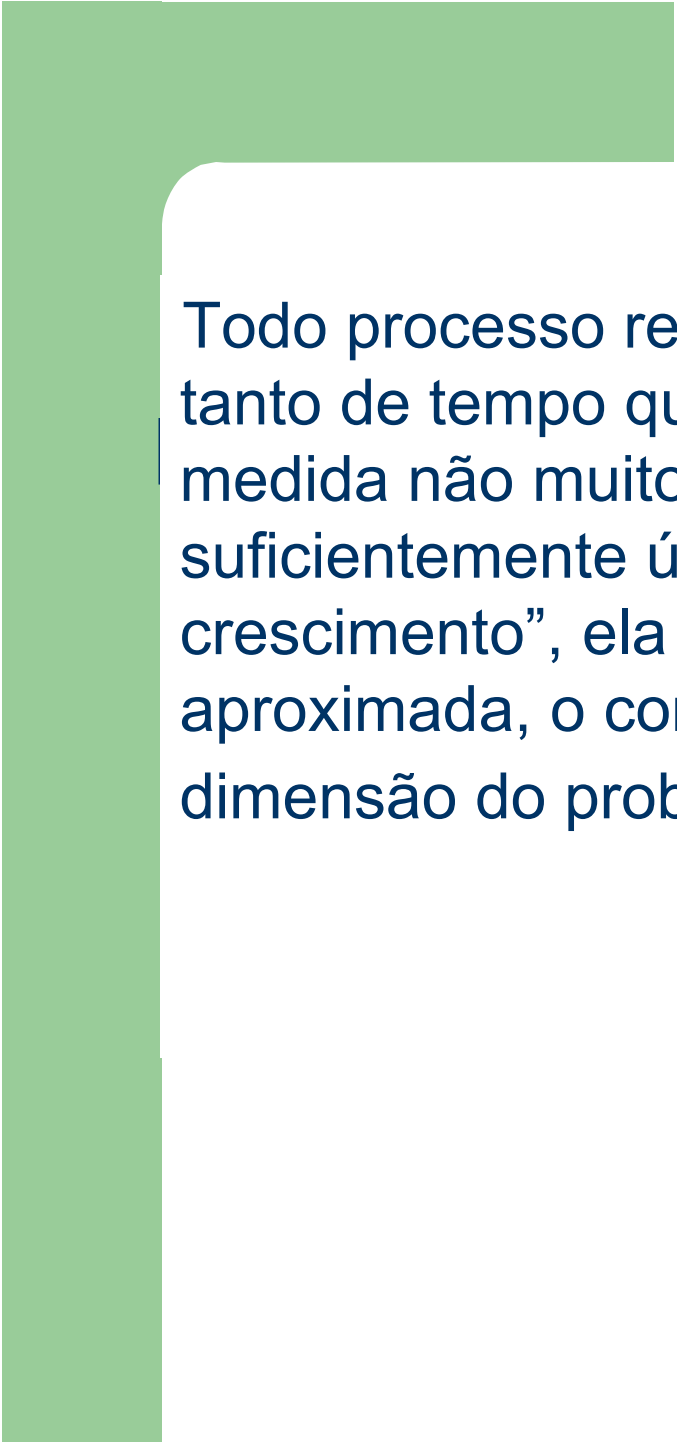
A chamada a uma função recursiva é igual à chamada de uma função não recursiva, na qual é necessário guardar uma “estrutura de invocação”, sendo esta estrutura liberada depois do fim da execução da função e atualização do valor de retorno.

## Funções recursivas contem duas partes fundamentais:

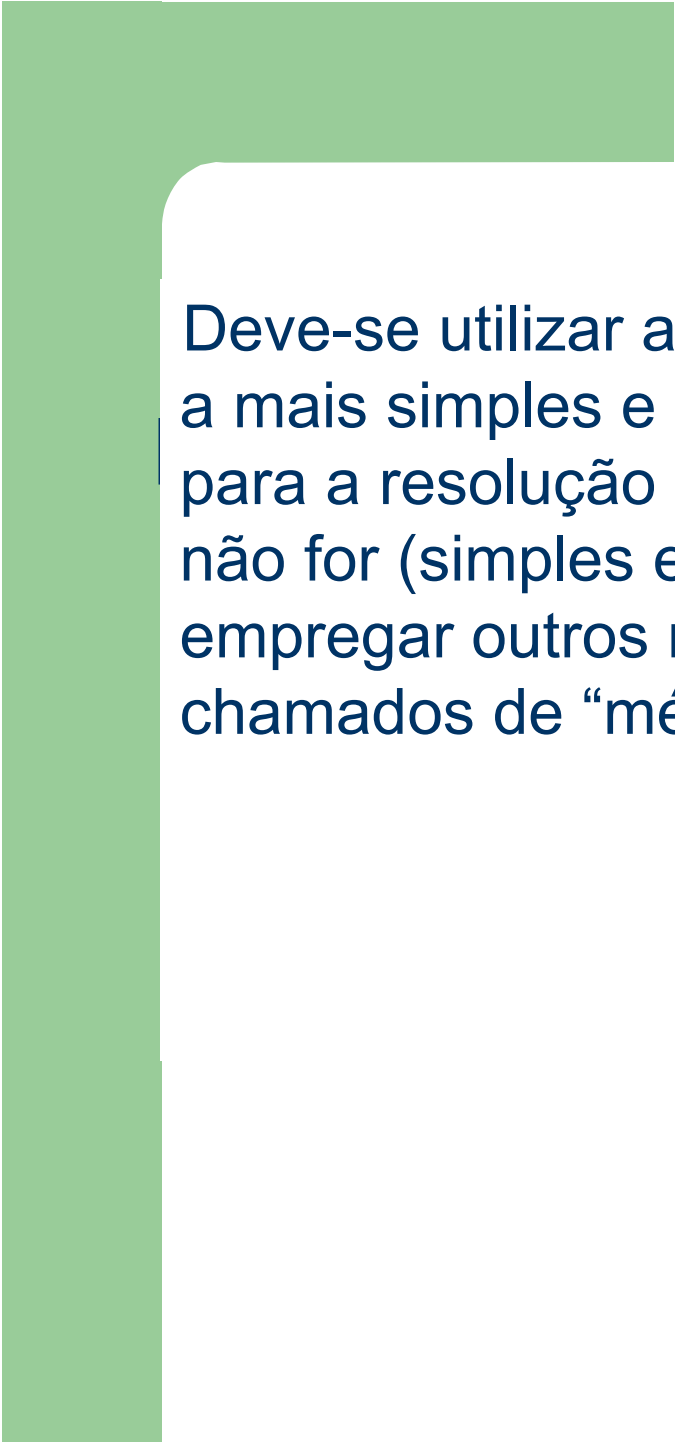
- **Ponto de Parada ou Condição de Parada:** que é o ponto onde a função será encerrada, e é geralmente um limite superior ou inferior da regra geral.
- **Regra Geral:** é o método que reduz a resolução do problema através da invocação recursiva de casos menores, que por sua vez são resolvidos pela resolução de casos ainda menores pela própria função, assim sucessivamente até atingir o “ponto de parada” que finaliza o método.



Para se criar um algoritmo recursivo, deve-se primeiro procurar encontrar uma solução de como o problema pode ser dividido em passos menores. Depois definir um ponto de parada. Em seguida, definir uma regra geral que seja válida para todos os demais casos. Deve-se verificar se o algoritmo termina, ou seja, se o ponto de parada é atingido. Para auxiliar nessa verificação, recomenda-se criar uma árvore de execução do programa, como um chinês, mostrando o desenvolvimento do processo.



Todo processo recursivo requer recursos da máquina, tanto de tempo quando de espaço de memória. Uma medida não muito precisa desses recursos, mas suficientemente útil, baseia-se na “ordem de crescimento”, ela nos permite caracterizar, de forma aproximada, o consumo de recursos em função da dimensão do problema.



Deve-se utilizar a recursividade quando esta forma for a mais simples e intuitiva de implementar uma solução para a resolução de um determinado problema. Se não for (simples e intuitiva), será então melhor empregar outros métodos não recursivos, também chamados de “métodos iterativos”.

# Aplicações práticas de funções recursivas na linguagem C

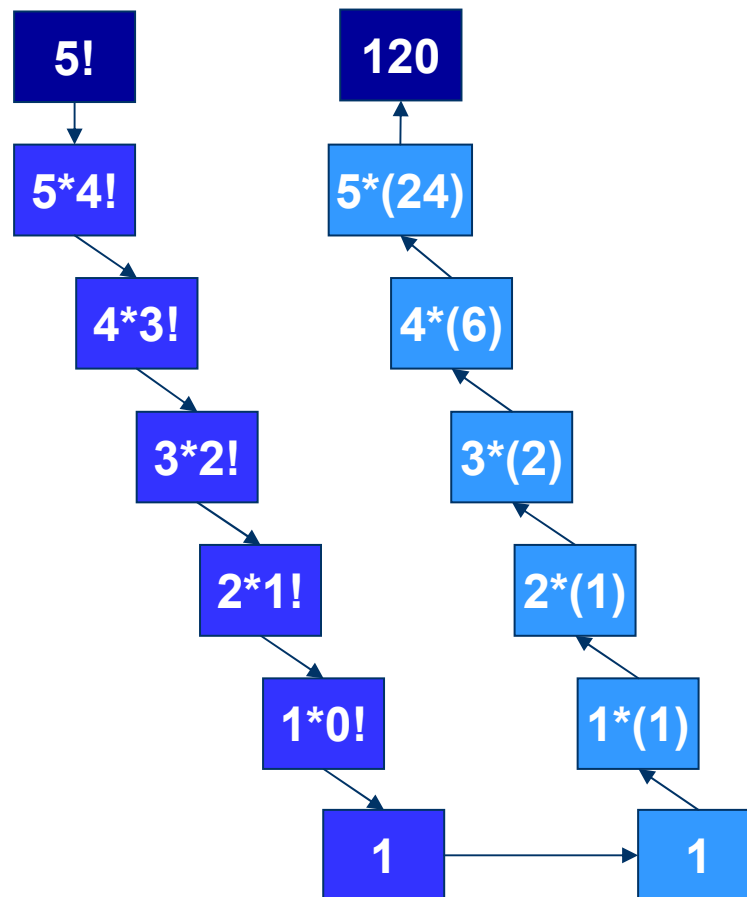
- Como primeiro exemplo de função recursiva, vamos ver o cálculo de fatorial.



# Fatorial

```
1 - //Recursividade na Linguagem C
2 - //PRC - Prof. João
3 - //FATORIAL
4 - #include "stdio.h"
5 - int fatorial(int x){
6 -     if(( x == 0 ) || ( x ==1))
7 -         return 1;
8 -     else
9 -         return(x * fatorial(x-1));
10 - }
11 - main(){
12 -     int num;
13 -     printf("Entre com um número: ");
14 -     scanf("%d", &num);
15 -     printf("O fatorial de %d é %d.", num, fatorial(num));
16 - }
```

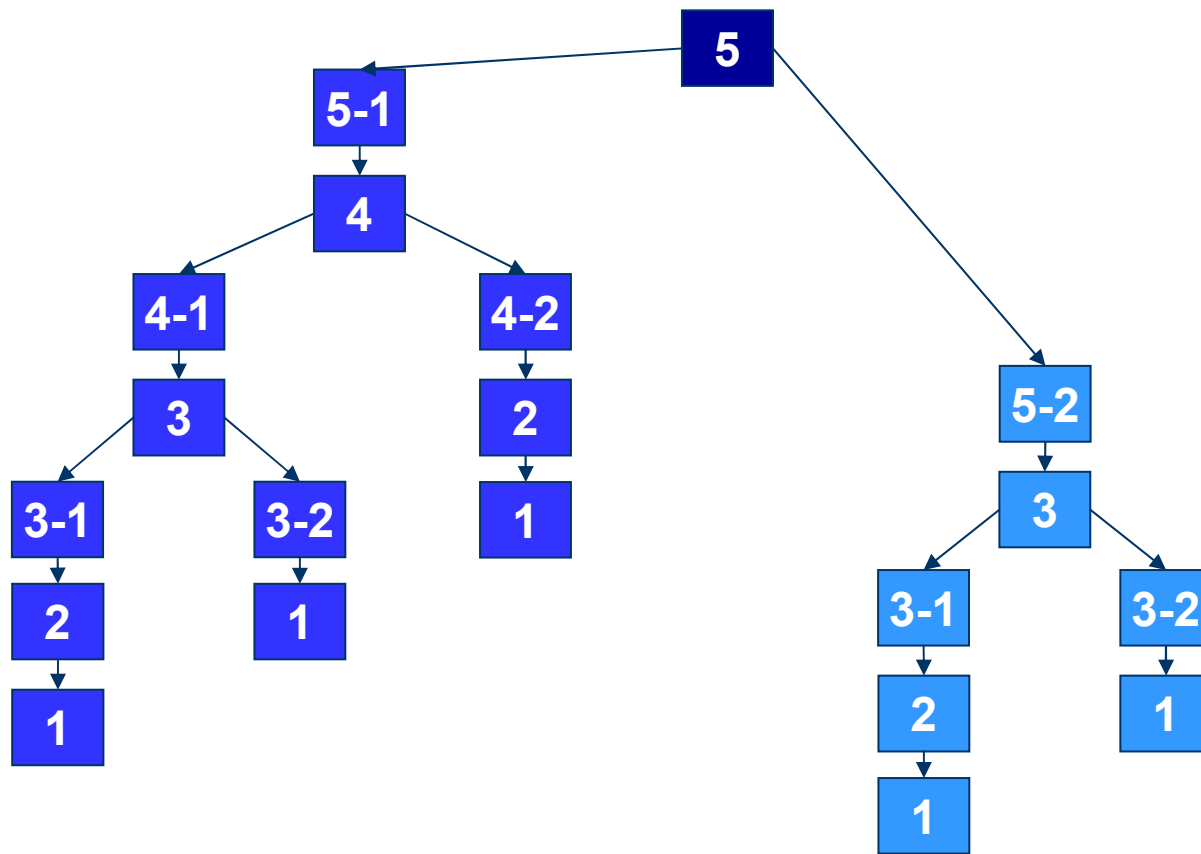
# Chinês Fatorial



# Fibonacci

```
1 - //Recursividade na Linguagem C
2 - //PRC - Prof. João
3 - //FIBONACCI
4 - #include "stdio.h"
5 - int fibonacci(int N) {
6 -     if( N == 1 )
7 -         return 1;
8 -     else
9 -         if( N == 2)
10 -             return 1;
11 -         else
12 -             return(fibonacci(N-1) + fibonacci(N-2));
13 - }
14 - main() {
15 -     int num, F;
16 -     printf("Entre com um número:   ");
17 -     scanf("%d", &num);
18 -     printf("A série de Fibonacci para %d elementos é:\n", num);
19 -     for(F=1;F <= num;F++) {
20 -         printf("%d, ", fibonacci(F));
21 -         printf("\nOk");
22 - }
```

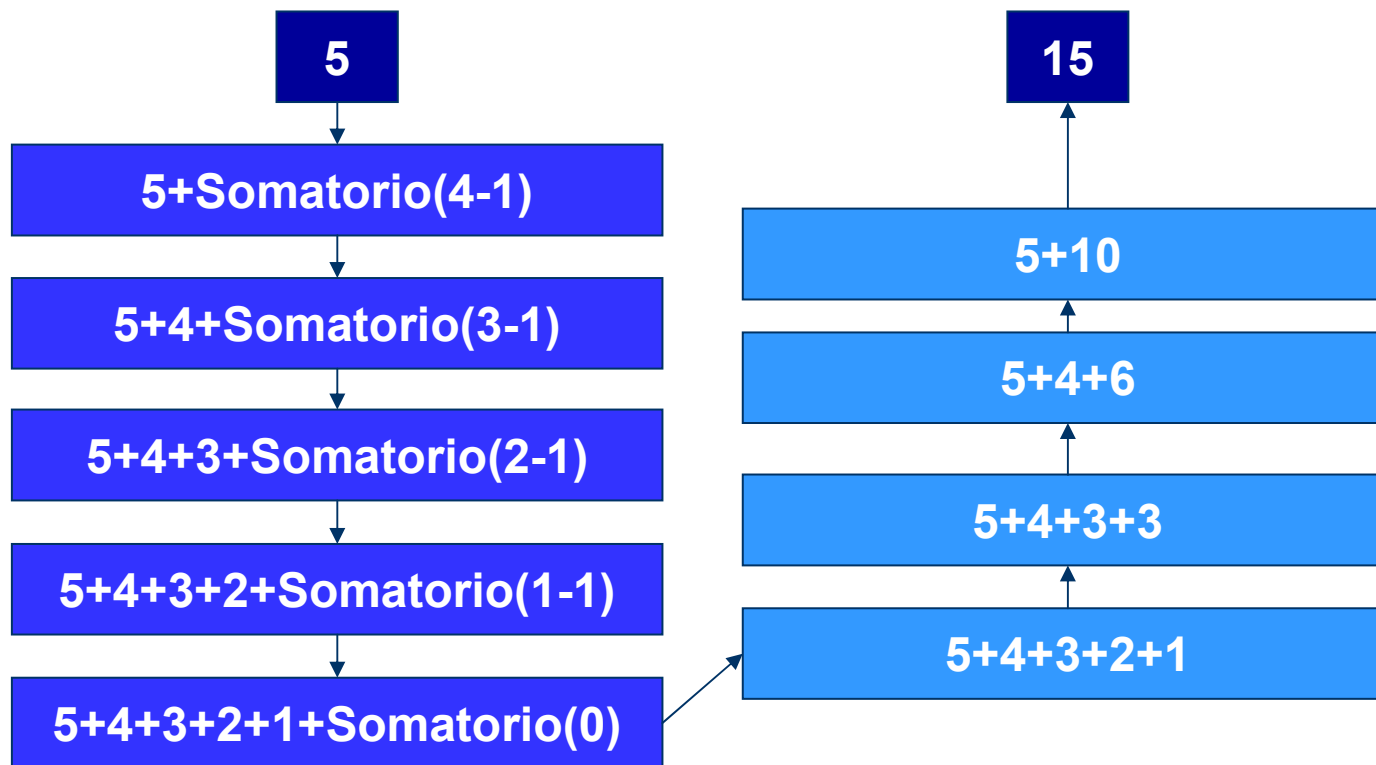
# Chinês Fibonacci



# Somatório

```
1 - //Recursividade na Linguagem C
2 - //PRC - Prof. João
3 - //SOMATORIO
4 - #include "stdio.h"
5 - int somatorio(int x) {
6 -     if( x == 1 )
7 -         return 1;
8 -     else
9 -         return(x + somatorio(x -1));
10 - }
11 - main() {
12 -     int num;
13 -     printf("Entre com um número: ");
14 -     scanf("%d", &num);
15 -     printf("O somatório de 0 até %d é %d.", num, somatorio(num));
16 - }
```

# Chinês Somatório



# Conta Dígitos

```
1 - //Recursividade na Linguagem C
2 - //PRC - Prof. João
3 - //CONTA DIGITOS
4 - #include "stdio.h"
5 - int digitos(int x) {
6 -     if( abs(x) < 10 )
7 -         return 1;
8 -     else
9 -         return(1 + digitos(x/10));
10 - }
11 - main() {
12 -     int num;
13 -     printf("Entre com um número: ");
14 -     scanf("%d", &num);
15 -     printf("O número de dígitos de %d é %d.", num, digitos(num));
16 - }
```

# Chinês Conta Dígitos

