Recursão

Definição e exemplos

Prof. Edson Alves - UnB/FGA

2018

Sumário

Recursão

Recursão

- A definição de um objeto é dita recursiva se ela se refere ao próprio objeto a ser definido
- Uma definição recursiva tem duas partes: o caso base e as regras de construção
- No caso base, todos os elementos estruturais básicos são listados
- As regras de construção determinam como os elementos estruturais se relacionam, e envolvem o próprio objeto
- Uma função é dita recursiva se o cálculo do seu retorno envolve a chamada da própria função

Exemplo de definição recursiva: números naturais

- caso base: $0 \in N$
- regra de construção: se $n \in N$, então $(n + 1) \in N$
- definição de escopo: não há outros elementos no conjunto N (isto é, não há outras formas de se obter números naturais)

Exemplo de função recursiva: fatorial de um número natural

- O fatorial do número natural n é dado por

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1)!, & \text{se } n > 0 \end{cases}$$

- Observe que a primeira parte da definição corresponde ao caso base
- Já a segunda parte se refere à regra de construção
- A definição ade escopo fica implícita: n deve ser um número natural
- A implementação em C/C++ é quase idêntica à definição:

```
1 int factorial(int n)
2 {
3     return n == 0 ? 1 : n * factorial(n - 1);
4 }
```

Relações de pertinência

- É possível determinar se um elemento pertence ou não a um conjunto de elementos definidos recursivamente aplicando a regra de construção sucessivamente, decompondo o elemento em outros elementos do conjunto
- Por exemplo, para $n = 5$,

$$
\begin{aligned}
&5 = 4 + 1, \text{ logo se } 4 \in N, \text{ então } 5 \in N \\
&4 = 3 + 1, \text{ logo se } 3 \in N, \text{ então } 4 \in N \\
&3 = 2 + 1, \text{ logo se } 2 \in N, \text{ então } 3 \in N \\
&2 = 1 + 1, \text{ logo se } 1 \in N, \text{ então } 2 \in N \\
&1 = 1 + 0
\end{aligned}
$$

- Como $0 \in N$, então $1, 2, 3, 4, 5 \in N$

Resolução de recorrências

- Para encontrar o n-ésimo termo do conjunto, é necessário encontrar todos os termos anteriores que se fizerem presentes de acordo com as regras de construção
- Este processo impacta diretamente na complexidade computacional da rotina
- Onde for possível, a troca de uma definição recursiva por uma definição que seja independente dependa de termos anteriores leva a implementações mais eficientes
- Por exemplo, a definição recursiva

$$g(n) = \begin{cases} 1, & \text{se } n = 0 \\ 2g(n-1), & \text{se } n > 0 \end{cases}$$

equivale a definição

$$g(n) = 2^n$$

Resolução de recorrência por hipótese e demonstração

- Uma maneira de se resolver uma relação de recorrência é observar alguns termos, formular uma hipótese e demonstrar esta hipótese usando indução matemática
- Por exemplo, se $h(n) = 2h(n-1) + 1$, com $h(1) = 1$, então

$$h(2) = 3,\ h(3) = 7,\ h(4) = 15,\ \ldots$$

- Pode se observar que, para estes termos, é possível formular a hipótese de que $h(n) = 2^n - 1$
- Para provar esta hipótese, primeiramente deve-se verificar o caso base:

$$h(1) = 2^1 - 1 = 1,$$

de modo que a hipótese vale para o caso base

Resolução de recorrência por hipótese e demonstração

- Suponha que a hipótese seja verdadeira para $m \in N$. Daí, usando a definição recursiva de $h(n)$ e a hipótese, segue que

  $$h(m+1) = 2h(m)+1 = 2(2^m-1)+1 = 2^{m+1}-2+1 = 2^{m+1}-1,$$

  de modo que $h(m+1)$ é verdadeira sempre que $h(m)$ for verdadeira
- Assim, de acordo com o Princípio da Indução, a hipótese é verdadeira
- Observe que, se a hipótese fosse falsa, ou ela falharia no caso base, ou não seria possível provar a segunda parte da indução, isto é, que a veracidade para m implica a veracidade para $m+1$

Resolução de recorrência por expansão

- Outra maneira de resolver uma relação de recorrência é aplicar a recorrência em si mesmo, expandindo sua definição
- Utilizando a mesma definição de $h(n)$, tem-se que

$$\begin{aligned} h(n) &= 2h(n-1)+1 \\ &= 2(2h(n-2)+1)+1 = 4h(n-2)+3 \\ &= 4(2h(n-3)+1)+3 = 8h(n-3)+7 \\ &\dots \end{aligned}$$

- Esta expansão leva a nova hipótese de que

$$h(n) = 2^k h(n-k) + (2^k - 1),$$

  para k natural
- Esta hipótese pode ser provada por indução
- Fazendo $k = (n-1)$ e lembrando que $h(1) = 1$, segue que $h(n) = 2^n - 1$

Chamadas Recursivas

Chamadas de funções

- No momento em que uma função é chamada, algumas tarefas são realizadas em plano de fundo para que o programa se comporte como esperado
- Se a função tem parâmetros, eles são inicializados com os valores passados na chamada
- O sistema operacional tem que saber de onde continuar após o encerramento da função chamada, de modo que o ponto de retorno é armazenado
- Quem chamou a função tem que ter acesso ao retorno da função chamada, logo uma área de memória é reservada para armazenar este valor

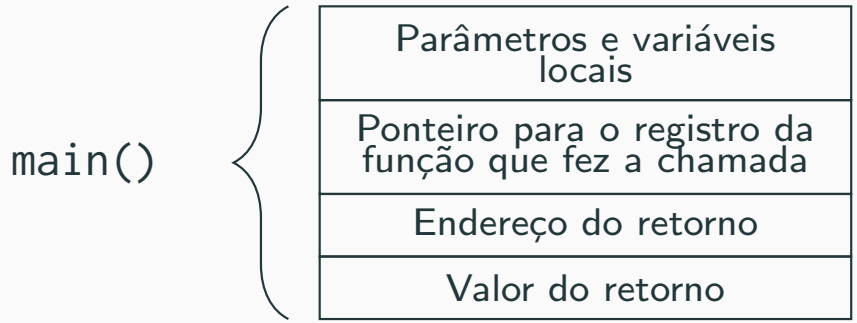
Chamadas de funções

- Algumas informações devem ser preservadas ao se chamar uma função, para se manter o estado correto do programa
- Em relação a função que realizou a chamada de um outra função, devem ser preservadas as variáveis locais e os seus parâmetros
- Também deve ser armazenado o endereço de memória que aponta para o lugar onde o programa deve prosseguir após a execução da função chamada

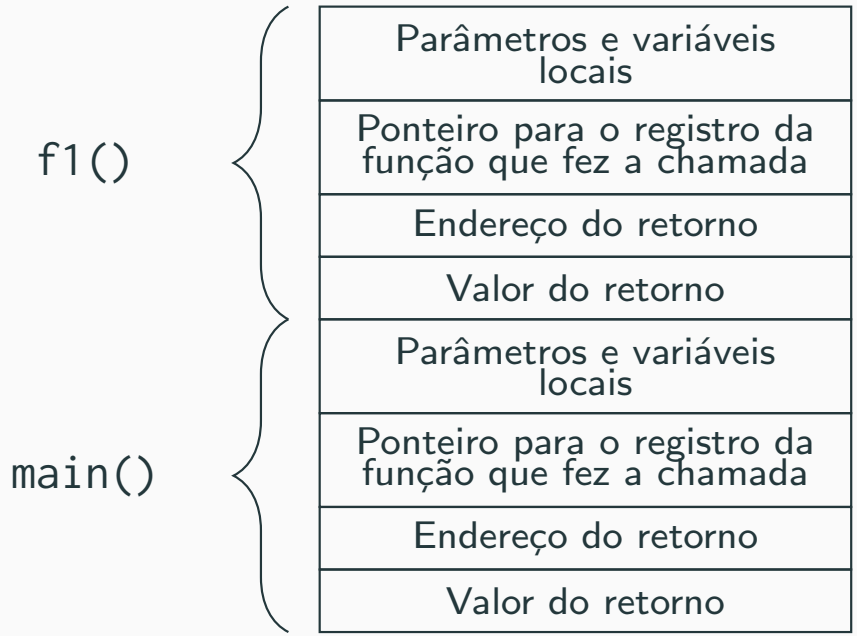
Registro de ativação

- A área de dados que armazena as informações listadas anteriormente é denominada registro de ativação ou *stack frame*
- O registro de ativação é alocado dinamicamente na pilha de execução
- Este registro existe enquanto a função que ele se refere está sendo executada
- O registro de ativação guarda todas as informações necessárias para a correta execução e retorno de uma função
- Um registro de ativção é criado quando a função é chamada e é destruído no retorno da função
- Apenas o registro de ativação da função `main()` fica ativo durante toda a execução do programa

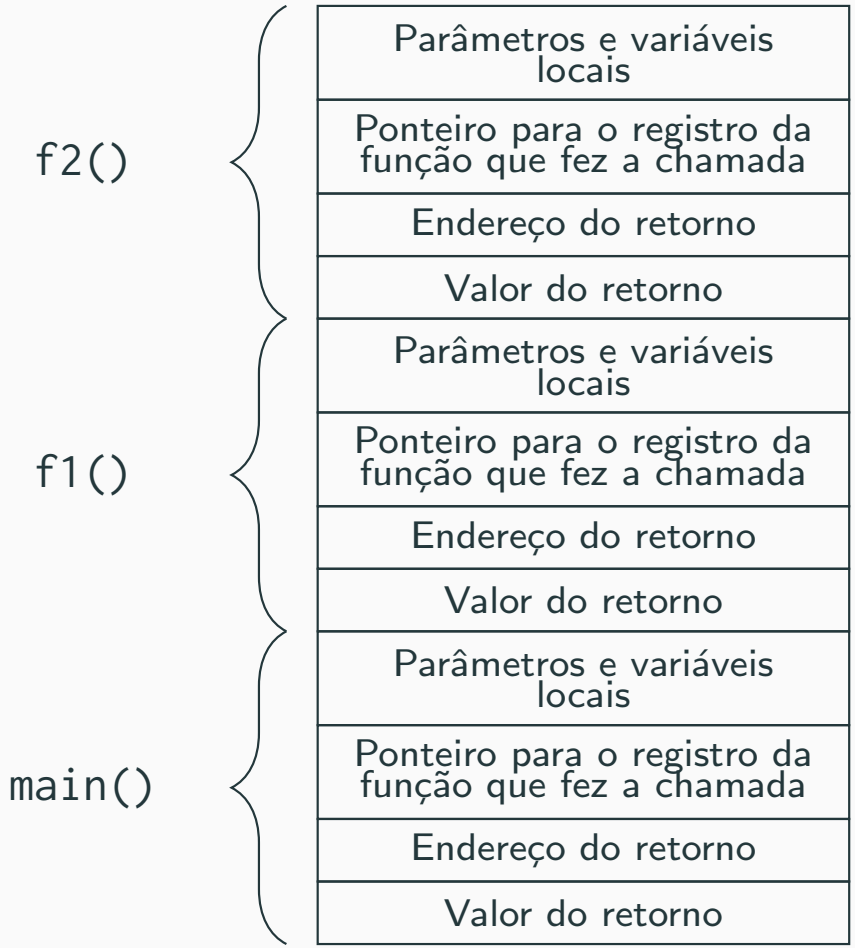
Visualização da pilha de execução no início do programa



Visualização da pilha após `main()` chamar `f1()`



Visualização da pilha após `f1()` chamar `f2()`



Notas sobre os registros de ativação

- O endereço de retorno aponta para o endereço de memória que contém a instrução imediamente após a chamada da função
- O ponteiro do registro da função que fez a chamada aponta para o elemento antecessor da pilha de execução
- Como o tamanho dos registros podem variar, o valor de retorno fica imediamente acima do registro da função que fez a chamada
- A criação de registros de ativação a cada chamada de função permitem a implementação da recursão
- De fato, a recursão consiste em chamar uma função que tem o mesmo nome da função que fez a chamada
- A função não chama a si mesma, mas a uma nova instância que tem a mesma estrutura da função original

Tail Call Optimization

- *Tail Call Optimization* (TCO) é uma técnica de otimização que permite reduzir o espaço utilizado pela pilha de execução
- Se o retorno de uma função `f()` é a chamada de uma função `g()`, torna-se desnecessário manter o registro de ativação de `f()`
- Se a função é recursiva e `f = g`, então é possível implementar `f()` usando espaço em memória constante ($O(1)$)
- Se o retorno envolver alguma outra operação que não a chamada de `g()`, não será possível usar a TCO

Implementação do fatorial que evita a TCO

```
1 int factorial(int n)
2 {
3     if (n == 0)
4         return 1;
5
6     return n * factorial(n);
7 }
```

Implementação do fatorial que permite a TCO

```
1 int factorial(int n, int ans = 1)
2 {
3     if (n == 0)
4         return ans;
5
6     return factorial(n - 1, n * ans);
7 }
```

Referências

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **ERICKSON**, Jeff. *Solving Recurrences*, acesso em 27/02/2019.[1]
3. Stack Overflow. *What is Tail Call Optimization?*, acesso em 27/02/2019.[2]

[1] http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf
[2] https://stackoverflow.com/questions/310974/what-is-tail-call-optimization