Estrutura de Dados

Prof. Rogerio Atem de Carvalho, D. Eng.

Aula 1: Revisão de Linguagem C

C: Características Gerais

- Base para todas linguagens modernas
- Total interação com o Sistema Operacional: era dita de “médio” nível, atualmente classificada como “baixo” nível
- Código altamente otimizado e portanto rápido e compacto
- Compilada
- C++: *superset* que incorpora Orientação a Objetos

Tipos Básicos de Dados

tipo	bytes	escala
char	1	-128 a 127
int	2	-32.768 a 32.767
float	4	3.4e-38 a 3.4e+38
double	8	1.7e-308 a 1.7e+308

Tipos Básicos de Dados

```
#include <stdio.h>
main()
{
    int inteiro = 10;
    float ponto_flutuante = 5.5;
    char caracter = 'X';
    double precisao_dupla = 3.5E6;
    printf("Exemplo de Inteiro: %d \n", inteiro);
    printf("Exemplo de Ponto Flutuante: %f \n", ponto_flutuante);
    printf("Exemplo de Caracter: %c \n", caracter);
    printf("Exemplo de Precisao Dupla: %e \n", precisao_dupla);
}
```

Strings de Controle

%c -> caracter

%d -> inteiro

%e -> número ou notação científica

%f -> ponto flutuante

%o -> octal

%x -> hexadecimal

%s -> string (cadeia de caracteres)

%lf -> double

Operador de Endereço

- Variáveis são *aliases* para endereços de memória, onde se encontram os valores aos quais elas referenciam
- A variável armazena o endereço do primeiro byte ocupado pelo valor
- O operador & indica que o endereço da variável está sendo referenciado. Obviamente, este endereço varia de execução para execução do programa

Operador de Endereço

```
#include <stdio.h>

main()

{

    int numero;

    numero = 2;

    printf("Conteudo: %d, endereco: %lu \n", numero,
    &numero);

}
```

Estrutura Condicional: if-else

```
if(<expressão>)
{
    <comandos>
}
else
{
    <comandos>
}
```

Estrutura Condicional: ?

Expressão_1 ? Expressão_2 : Expressão_3

Equivale a:

if(<Expressão_1>)

    <Expressão_2>

else

    <Expressão_3>

Estrutura Condicional: case

```
switch(<expressão: int ou char>)
{
    case constante_1:
                <comandos>
                break;
    case constante_n:
                <comandos>
                break;
    default:
                <comandos>
}
```

Estrutura de Repetição: while

```
while(<condição>)
{
    <comandos>
}

do
{
} while(<condição>);
```

Estrutura de Repetição: for

```
for(<comandos de inicialização>;
<condição>;
<incremento ou decremento>)
{
    <comandos>
}
```

Funções: declaração

```
<tipo de retorno> nome_da_função(
<tipo do argumento> argumento_1,
<tipo do argumento> argumento_2, ...)
{
    <corpo da função>
    return <valor de retorno>;
}
```

- Após o *return* comandos não são executados

Funções: parâmetros

- Parâmetros ou argumentos são valores passados para uma função
- Parâmetros **Formais** são aqueles presentes na declaração da função
- Parâmetros **Reais** são os empregados em chamadas à função
- Parâmetros se comportam como variáveis locais à função*: criados na entrada (chamada) e destruídas na saída

Funções: parâmetros

- Passagem por **Valor**:
  - Os valores são copiados para as variáveis locais à função
  - Alteração nos valores dos parâmetros terão efeito apenas local, ou seja, não refletem nas variáveis passadas como parâmetro
  - Se uma variável de grande porte está sendo passada, um tempo relativamente grande pode ser empregado para copiar seu conteúdo para o parâmetro

Funções: parâmetros

- Passagem por **Referência**:
  - Os endereços das variáveis são passados para a função
  - Alteração nos valores dos parâmetros terão efeito nas variáveis passadas como tal
  - Fonte de muitos erros
  - Acesso:
    - & corresponde ao endereço da variável
    - * corresponde ao conteúdo da variável

Funções: parâmetros

- Passagem de **Vetores**:
  - Vetores possuem comportamento diferente, tanto como parâmetros, quanto como valores de retorno
  - O compilador interpreta o parâmetro como o endereço do primeiro elemento do vetor
  - Assim**, vetores são sempre passados por referência**, mesmo sem qualquer notação específica

Funções: parâmetros

- Passagem de **Vetores**:
  - Vetores declarados internamente e empregados como valor de retorno simplesmente retornarão ponteiros para áreas já desalocadas*
  - Ao passar um vetor como parâmetro não é necessário passar seu tamanho. Se o vetor for multidimensional, apenas a primeira dimensão é liberada de ter seu tamanho declarado

```
void manipula_vetores(int vetor[], int matriz[][10])
{
}
```

Funções: recursividade

- Um código recursivo é aquele que chama a si próprio
- Divide um problema em problemas menores da mesma natureza
- Um processo recursivo consiste de duas partes:
  - O caso trivial, cuja solução é conhecida e que vai gerar o **critério de parada** da recursão
  - O método geral que reduz o problema original a um ou mais problemas menores da mesma natureza

Funções: recursividade

- Vantagens:
  - Redução do tamanho do código fonte
  - Algoritmos mais concisos
- Desvantagens
  - Aumento do tempo de execução devido ao empilhamento sucessivo de chamadas*
  - Depuração mais complexa, especialmente quando ocorrem recursões profundas

Funções: recursividade

- Funcionamento:
  - Cada vez que a função é chamada, como de praxe, é criado um *__Registro de Ativação__* que contém, dentre outros elementos, (i) um ponteiro para a chamada anterior, (ii) variáveis locais à chamada e (iii) parâmetros da chamada
  - Ao atingir o critério de parada, a última chamada retorna a penúltima e assim sucessivamente, desempilhando as chamadas recursivas até atingir a primeira delas, que retornará o resultado final

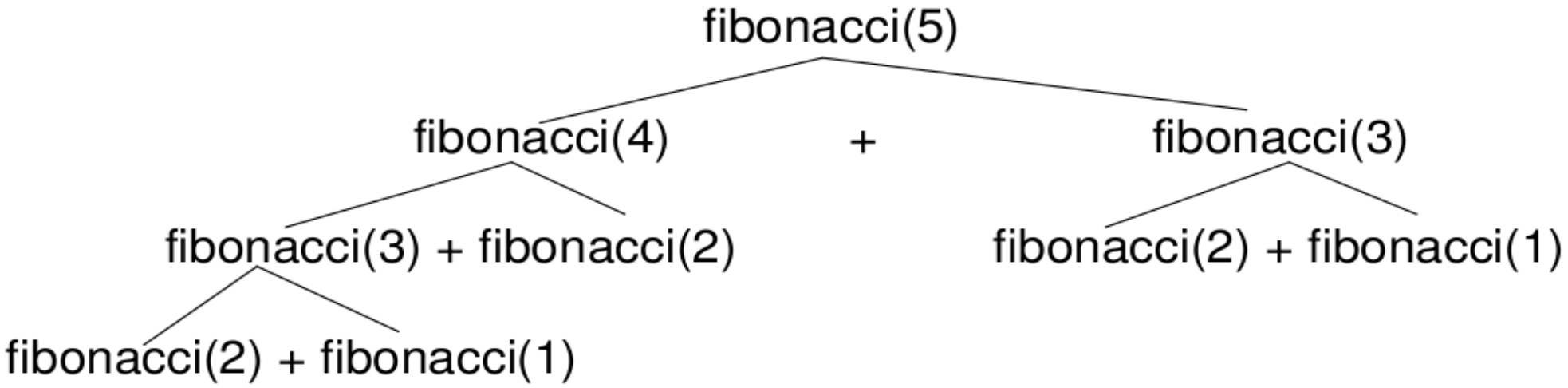
Funções: recursividade

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

```
int fatorial(int n)
{
    if( n != 1 ) return n*fatorial(n-1);
    else return 1;
}
```

Funções: recursividade

```
int fibonacci(int n)
{
    if ( n == 1 ) return 0;
    if ( n == 2 ) return 1;
    return (fibonacci(n-1) + fibonacci(n-2));
}
```



Funções: Registros de Ativação

- RAs: também conhecidos como *stack frames*
- Armazenam na pilha* as informações de uma chamada a uma função (recursiva ou não)
- A pilha é implementada pelo compilador (ou interpretador) e instanciada em tempo de execução como um vetor dinâmico* e com comportamento LIFO* (*Last In, First Out*)
- A subrotina (função) em execução estará no topo da pilha, ao término de sua execução, o marcador de topo se move para a subrotina que a chamou

Funções: Registros de Ativação

argumentos de entrada

locais de sub-blocos

registradores salvos

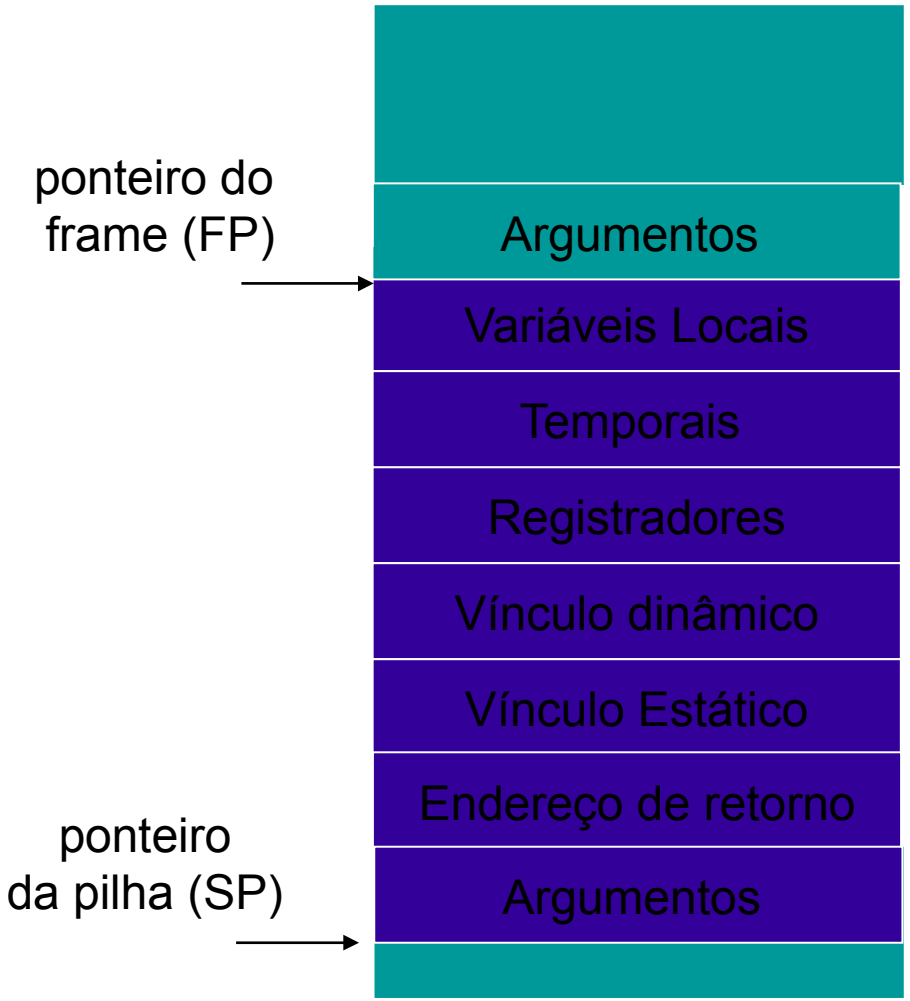
ponteiro para o topo do registro de ativação do chamador

ponteiro para a base do registro de ativação do pai estático

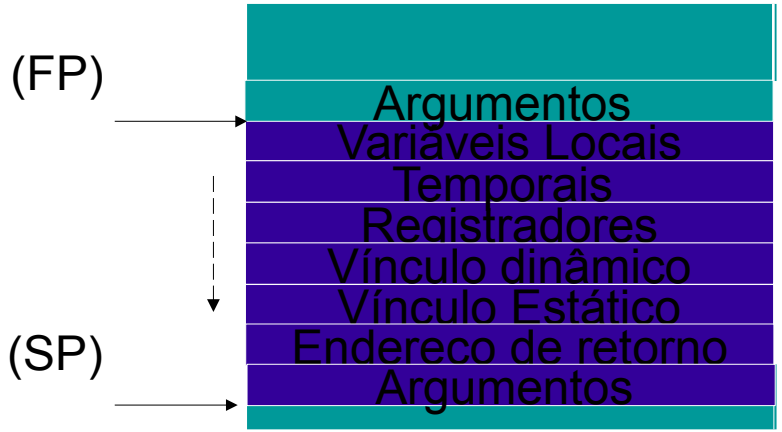
ponteiro para o código

argumentos de saída

Funções: Registros de Ativação

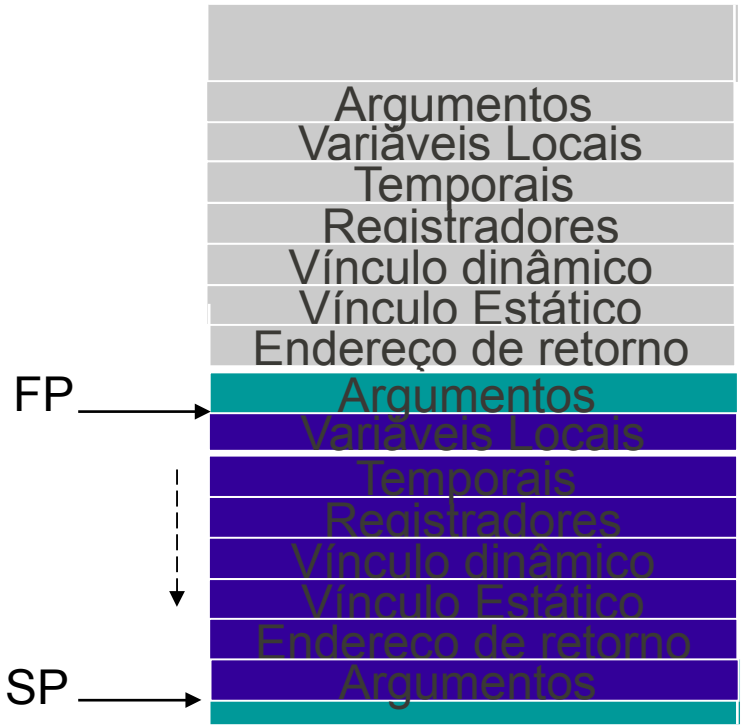


Funções: Registros de Ativação



1. O subprograma atual faz chamada a um outro subprograma

Funções: Registros de Ativação



2. Um novo frame é alocado. Os ponteiros FP e SP devem ser atualizados

O novo FP = FrameSize - SP.

O antigo FP é guardado em memória

SP sempre aponta para o início da pilha.

Funções: Registros de Ativação

3. Ao desalocar o frame.

FP e SP são restaurados

Próximo Tópico

Variáveis Compostas