Ponteiros na Linguagem C

Mário Leite

...

Algumas linguagens de programação possuem características e recursos próprios que as diferenciam das demais; é o caso da **C**. Algumas de suas características podem ser elencadas: o famoso **;** (ponto e vírgula) no final de uma linha de código para indicar o fim da instrução, sua sensibilidade à maiúsculas e minúsculas, passagem de parâmetros por referência, etc. Entre seus recursos está a utilização de **ponteiros** que funcionam, na prática, como "setas" que apontam para o local onde um valor específico está armazenado na memória do computador; assim em vez de trabalhar diretamente com o valor, o *ponteiro* armazena o endereço onde esse valor está localizado. Isto permite, entre outras coisas, simular a passagem de parâmetros "por referência". De qualquer forma, em termos de codificação, o local mais acessado pelos programadores é, sem dúvida alguma, a memória RAM do computador; é lá onde todos os processos são efetuados e onde são armazenados os dados e extraídas as informações.

Pelo esquema da **figura 1** pode ser notado que as variáveis *locais* ocupam a região alta da memória denominada ***pilha***; as variáveis *globais* e o programa ocupam regiões fixas e de tamanhos fixos durante o processamento, sendo que o "*programa"* ocupa a chamada região baixa da memória. Deste modo, para qualquer solicitação de alocação de memória será retirada da área de memória livre e se estendendo na direção da ***pilha***. Mas, se isto for muito intenso poderá ocorrer uma "colisão" da pilha com a área de memória livre; e nesses casos poderá haver um "esgotamento" do *heap* e o pedido de alocação de memória pode falhar. Por isto, o programador sempre deve verificar se realmente existe espaço suficiente ao solicitar alocações de memória. Observe os dois exemplos simples de códigos para mostrar o uso de ponteiros nos programas: **"AlocaRAM_1"** e **"AlocaMAM_2"**; o primeiro cria um vetor de elementos inteiros digitados pelo usuário em que, embora o vetor tenha sido previamente dimensionando para 10 elementos nada impedirá que o usuário entre com uma quantidade de elementos superior a **10**; isto seria um problema, pois os elementos excedentes poderiam ser alocados "sabe lá Deus onde"! A outra situação também seria estranha: se o usuário entrar com um valor menor que **10** ocorre desperdício de memória. Na segunda versão do código do programa (**"AlocaMAM_2"**) note o que acontece quando a memória armazena os elementos dinamicamente: o vetor é declarado como um **ponteiro** para inteiros (observe os ateísticos *****); além disto, logo após o usuário entrar com a quantidade de elementos para o vetor há uma verificação para saber se existe memória suficiente para alocar seus elementos e, caso não haja memória suficiente para as alocações o programa é abortado com a função **exit**(), que pode ser usada em qualquer ponto do programa, retornando um valor inteiro para o sistema operacional, com **0** para um encerramento normal ou um ***valor inteiro não nulo*** quando ocorre um encerramento inesperado do programa.

Assim, embora ela não seja necessariamente, a "melhor linguagem" (isto vai depender de muitos fatores) podemos concluir que a linguagem **C** é uma das mais importantes; seu poder de processamento, sua simplicidade e sua eficiência permanecem relevantes até hoje. E mesmo em um ambiente de linguagens de alto nível e interfaces gráficas, com Orientação a Objetos e a Eventos, esta linguagem continua sendo obrigatória para quem deseja entender a essência da codificação de programas, especialmente em contextos de sistemas operacionais, embarcados, compiladores e desempenho extremo.

Nota: Postagem baseada no livro do autor: "Curso Básico de C: Teoria e Prática"
https://www.amazon.com.br/Curso-Basico-Pratico-F%C3%A1cil-Leite/dp/8539903377

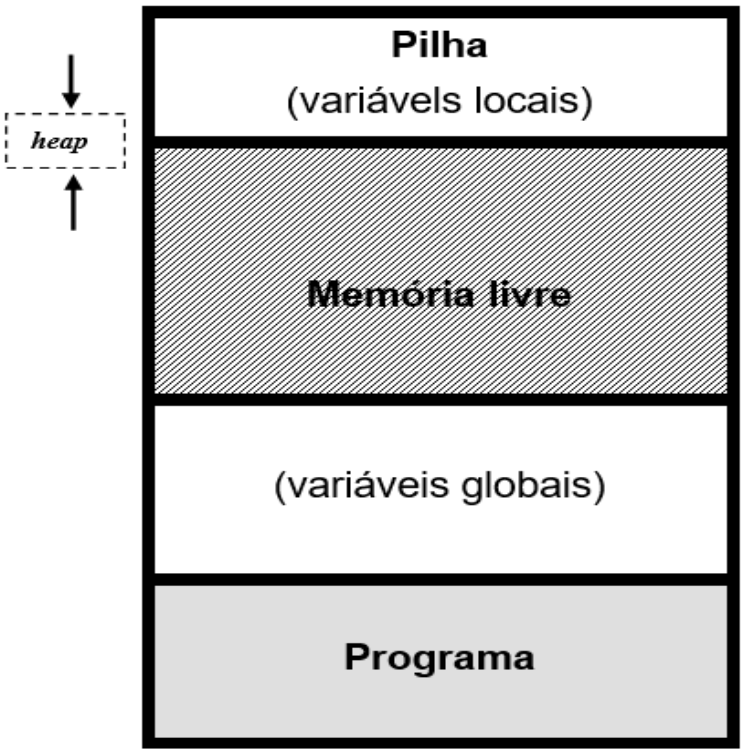

Figura 1 - Regiões da memória

```c
// AlocaRAM_1.C
//Abordagem estática
#include <stdio.h>

int main() {
    int qteElem, j;
    int Vet[10];  // vetor com tamanho fixo

    printf("Digite a quantidade de elementos para o vetor (máx 10): ");
    scanf("%d", &qteElem);

    if (qteElem > 10 || qteElem <= 0) {
        printf("Quantidade inválida. Máximo permitido é 10.\n");
        return 1;
    }

    // Leitura dos elementos
    for (j = 0; j < qteElem; j++) {
        printf("Entre com o elemento %d: ", j + 1);
        scanf("%d", &Vet[j]);
    }

    // Exibe os elementos armazenados
    printf("\nElementos armazenados:\n");
    for (j = 0; j < qteElem; j++) {
        printf("%d", Vet[j]);
    }

    printf("\n");
    return 0;
}
```

```c
 // AlocaRAM_2.C
//Abordagem dinâmica
#include <stdio.h>
#include <stdlib.h>  // para malloc(), free(), exit()

int main() {
    int qteElem, j;
    int *Vet;  // ponteiro para o vetor dinâmico

    printf("Digite a quantidade de elementos para o vetor: ");
    scanf("%d", &qteElem);

    if (qteElem <= 0) {
        printf("Quantidade inválida.\n");
        return 1;
    }

    // Aloca memória dinamicamente
    Vet = (int *)malloc(qteElem * sizeof(int));
    if (Vet == NULL) {
        printf("Memória insuficiente para alocação.\n");
        exit(1);  // encerra o programa com erro
    }

    // Leitura dos elementos
    for (j = 0; j < qteElem; j++) {
        printf("Entre com o elemento %d: ", j + 1);
        scanf("%d", &Vet[j]);
    }

    // Exibe os elementos armazenados
    printf("\nElementos armazenados:\n");
    for (j = 0; j < qteElem; j++) {
        printf("%d ", Vet[j]);
    }

    printf("\n");

    // Libera a memória alocada
    free(Vet);

    return 0;
}
```