Fundamentos de Programação

Alocação Dinâmica e ponteiros

Endereços

A memória RAM (random access memory) de qualquer computador é uma sequência de bytes. A posição (0, 1, 2, 3, etc.) que um byte ocupa na sequência é o endereço (= address) do byte. (É como o endereço de uma casa em uma longa rua que tem casas de um lado só.) Se e é o endereço de um byte então e + 1 é o endereço do byte seguinte.

Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador. Uma variável do tipo char ocupa 1 byte. Uma variável do tipo int ocupa 4 bytes e um double ocupa 8 bytes em muitos computadores. O número exato de bytes de uma variável é dado pelo operador sizeof. A expressão sizeof (char), por exemplo, vale 1 em todos os computadores e a expressão sizeof (int) vale 4 em muitos computadores.

Cada variável (em particular, cada registro e cada vetor) na memória tem um endereço

```
char c;
int i;
struct {
    int x, y;
} ponto;
int v[4];
```

c	*89421*
i	*89422*
ponto	*89426*
v[0]	*89434*
v[1]	*89438*
v[2]	*89442*

Fonte: ICMC/USP

O endereço de uma variável é dado pelo operador & . Assim, se i é uma variável então &i é o seu endereço. (Não confunda esse uso de & com o operador lógico and, representado por && em C.)

Exemplo: o segundo argumento da função de biblioteca scanf é o endereço da variável que deve receber o valor lido do teclado:

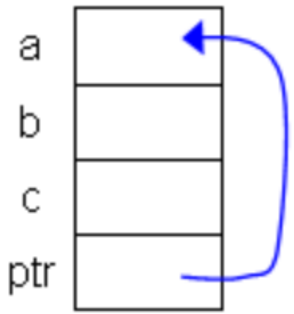
```
int i;
scanf ("%d", &i);
```

Um ponteiro (apontador = pointer) é um tipo especial de variável que armazena um endereço. Um ponteiro pode ter o valor NULL que é um endereço inválido.

A macro NULL está definida na interface stdlib.h e seu valor é 0 (zero) na maioria dos computadores.

Se um ponteiro p armazena o endereço de uma variável i, podemos dizer p aponta para i ou p é o endereço de i

```c
int a;
int *ptr;  // declara um ponteiro para um inteiro
           // um ponteiro para uma variável do tipo inteiro
a = 90;
ptr = &a;
printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
```



Acessando o Conteúdo de uma Posição de Memória através de um Ponteiro

Para acessar o conteúdo de uma posição de memória, cujo endereço está armazenado em um ponteiro, usa-se o operador de **derreferência** (*)

```c
#include <stdio.h>
void main()
{
  int x;
  int *ptr;
  x = 5;
  ptr = &x;
  printf("O valor da variável X é: %d\n", *ptr);  // derreferenciando um ponteiro
  *ptr = 10;  // usando derreferencia no "lado esquerdo" de uma atribuição
  printf("Agora, X vale: %d\n", *ptr);
}
```

Tipos de ponteiros

Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros,etc . O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro p para um inteiro, escreva

```
int *p;
```

Um ponteiro r para um ponteiro que apontará um inteiro (como no caso de uma matriz de inteiros) é declarado assim:

```
int **r;
```

Exemplo prático

```c
int *p;
int **r; // ponteiro para ponteiro para inteiro
p = &a;  // p aponta para a
r = &p;  // r aponta para p e *r aponta para a
c = **r + b;
```

Trocando valores de variáveis

```c
void troca (int *p, int *q)
{
    int temp;
    temp = *p; *p = *q; *q = temp;
}
```

Para aplicar essa função às variáveis i e j basta dizer troca (&i, &j) ou então

```
int *p, *q;
p = &i;
q = &j;
 troca (p, q);
```

Alocação Dinâmica de Memória

Durante a execução de um programa, pode-se alocar dinamicamente memória para usar como variáveis do programa. Em C, a alocação é feita com a função **`malloc`** (e, para liberar memória, usa-se a função **`free`**).

```c
int *ptr_a;

ptr_a = malloc(sizeof(int));
// cria a área necessária para 01 inteiro e
// coloca em 'ptr_a' o endereço desta área.

 if (ptr_a == NULL)
 {
   printf("Memória insuficiente!\n");
   exit(1);
 }

 printf("Endereço de ptr_a: %p\n",  ptr_a);
 *ptr_a = 90;
 printf("Conteúdo de ptr_a: %d\n", *ptr_a);  // imprime 90
 free(ptr_a);  // Libera a área alocada
```

Alocação dinâmica de vetores

```c
int i;
int *v;
v = (int*)malloc(sizeof(int)*10);  // 'v' é um ponteiro para uma área que
                                   // tem 10 inteiros.
                                   // 'v' funciona exatamente como um vetor
v[0] = 10;
v[1] = 11;
v[2] = 12;
// continua...
v[9] = 19;

for(i = 0; i < 10; i++)
  printf("v[%d]: %d\n", i, v[i]);

printf("Endereço de 'v': %p", v);  // imprime o endereço da área alocada para 'v'
free(v);
```

Exercícios

Crie um programa que vai solicitar ao usuário que informe a quantidade de números que deseja armazenar e, em seguida, vai alocar dinamicamente um array para armazenar esses números. Após a alocação, o programa deve pedir que o usuário insira os números e, por fim, vai exibir os números na ordem inversa.

Crie um programa o qual solicita ao usuário 3 números. Em seguido imprima o endereço de memória onde cada número foi armazenado. Use ponteiros

Matrizes bidimensionais são implementadas como vetores de vetores. Uma matriz com m linhas e n colunas é um vetor de m elementos cada um dos quais é um vetor de n elementos. O seguinte fragmento de código faz a alocação dinâmica de uma tal matriz

```
int **M;
M = malloc (m * sizeof (int *));
for (int i = 0; i < m; ++i)
   M[i] = malloc (n * sizeof (int));
```

Matrizes bidimensionais são implementadas como vetores de vetores. Uma matriz com m linhas e n colunas é um vetor de m elementos cada um dos quais é um vetor de n elementos. O seguinte fragmento de código faz a alocação dinâmica de uma tal matriz

```
int **M;
M = malloc(m * sizeof(int *));


//Conversão explícita de tipo
int **M = (int**) malloc(m * sizeof(int *));
```

Declaração de M:
•Aqui, M deve ter sido previamente declarado como um ponteiro duplo para inteiro (int **).

•Esta linha só faz a alocação de memória, sem definir o tipo de M. Assim, o compilador espera que M já tenha sido declarado.

Alocação de Memória:
•malloc(m * sizeof(int *)) aloca memória suficiente para armazenar m ponteiros para inteiros. O tipo de retorno de malloc é void *, que é um ponteiro genérico.

Tipo de M:
•Para que essa linha funcione, M já deve ter sido declarado anteriormente, como o primeiro exemplo

```
int **M;
M = malloc(m * sizeof(int *));
```

Declaração de M e Alocação de Memória:

•Aqui, M é declarado como um ponteiro duplo para inteiro (int **) e a memória é alocada em uma única linha.
•(int**) malloc(m * sizeof(int *)) faz a conversão explícita do tipo de retorno de malloc de void * para int **.

Alocação de Memória e Conversão:
•malloc(m * sizeof(int *)) ainda aloca memória suficiente para armazenar m ponteiros para inteiros.
•A conversão (int**) é usada para informar ao compilador que o ponteiro genérico retornado por malloc deve ser tratado como um ponteiro duplo para inteiro.

```
//Conversão explícita de tipo
int **M = (int**) malloc(m * sizeof(int *));
```

Alocação de memória para a linhas

```c
int rows = 2; // número de linhas
int cols = 3; // número de colunas

int **matrix = (int **)malloc(rows * sizeof(int *));
if (matrix == NULL) {
    fprintf(stderr, "Erro na alocação de memória\n");
    return 1;
}
```

Alocação de Memória para Cada Linha:

```c
for (int i = 0; i < rows; i++) {
    matrix[i] = (int *)malloc(cols * sizeof(int));
    if (matrix[i] == NULL) {
        fprintf(stderr, "Erro na alocação de memória\n");
        // Liberar memória já alocada antes de sair
        for (int j = 0; j < i; j++) {
            free(matrix[j]);
        }
        free(matrix);
        return 1;
    }
}
```

Atribuição de dados permanece igual

```java
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        matrix[i][j] = i * cols + j; // Apenas um exemplo de valores
    }
}
```

Impressão de dados permanece igual

```c
printf("Matriz 2x3:\n");
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

Liberação de dados da memória

```
for (int i = 0; i < rows; i++) {
    free(matrix[i]);
}
free(matrix);
```

Faça um programa que lê um vetor de n elementos e uma matriz de m x n elementos. Em seguida o programa deve fazer a multiplicação do vetor pelas colunas da matriz. Use alocação dinâmica.

É possivel alocar uma matriz diretamenete sem alocar linha a linha?

```c
// Alocar um único bloco de memória para a matriz
int *matrix = (int *)malloc(rows * cols * sizeof(int));
if (matrix == NULL) {
    fprintf(stderr, "Erro na alocação de memória\n");
    return 1;
}
```

É possivel alocar uma matriz diretamenete sem alocar linha a linha?

```
// Inicializar a matriz
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        matrix[i * cols + j] = i * cols + j;
    }
}
```

Aritmética de Ponteiros

Quando incrementamos um ponteiro estamos fazendo ele apontar para o próximo elemento do vetor

– Ex.: se p é um ponteiro para inteiro (ou seja, declarado como int *p), e p armazena o endereço 100, então p++ faz p armazenar o endereço 104 (pois um inteiro ocupa 4 bytes)

– Por esta razão precisamos indicar o tipo na declaração do ponteiro

- A mesma regra vale para decremento
- Para incrementar o conteúdo da região apontada por p fazemos (*p)++;

Aritmética de Ponteiros

Soma/subtração de ponteiros com inteiros

- Para fazer um ponteiro p apontar para N elementos a
frente fazemos p += N; ou p = p+N;

- Para acessar o conteúdo de N elementos a frente
usamos *(p+N)

- A subtração funciona de forma similar

```c
int v[] = {10, 20, 30};
printf("%p\n", v);
printf("%p\n", v+2);
printf("%d\n", *(v+2));
```

Aritmética de Ponteiros

Podemos utilizar os operadores relacionais para testar:

== apontam para a mesma posição
!= apontam para posições diferentes
> >= < <= qual aponta para a posição mais alta

```
int v[] = {1, 2, 3, 4, 5, 6}, *p;
for (p = v; p <= &v[5]; p++)
printf("%d\n", *p);
```

Aritmética de Ponteiros

```c
int main() {
    int arr[10];     /* arr: arranjo com 10 inteiros */
    int *el;         /* el:  ponteiro para um inteiro */
    int i;

    el = &arr[0];  /* inicializa ponteiro */

    /* inicializa conteudo do arranjo via ponteiro */
    for (i=0; i<10; ++i){
        *(el + i) = 0;
    }
```

Exercício

Escreva um programa em C que faça o seguinte:
1. Solicite ao usuário a quantidade de elementos do array.
2. Alocar dinamicamente memória para o array.
3. Solicite ao usuário os valores dos elementos do array.
4. Imprima os elementos do array usando aritmética de ponteiros.
5. Calcule a soma de todos os elementos do array usando aritmética de ponteiros.
6. Libere a memória alocada.

Função Realloc

A função **realloc()** é usada para redimensionar um espaço alocado previamente com **malloc()**. Seus parâmetros são um ponteiro para o início de uma área previamente alocada, e o novo tamanho a ser redimensionado na alocação, podendo este ser maior ou menor que o tamanho inicialmente alocado.

```c
// Redimensionar o array usando realloc
int *new_array = (int *)realloc(array, new_n * sizeof(int));
if (new_array == NULL) {
    fprintf(stderr, "Erro na realocação de memória\n");
    free(array); // Liberar memória original em caso de falha
    return 1;
}
```

Função Realloc

```c
// Alocar memória para o array inicial
int *array = (int *)malloc(n * sizeof(int));
if (array == NULL) {
    fprintf(stderr, "Erro na alocação de memória\n");
    return 1;
}
```

```c
// Redimensionar o array usando realloc
int *new_array = (int *)realloc(array, new_n * sizeof(int));
if (new_array == NULL) {
    fprintf(stderr, "Erro na realocação de memória\n");
    free(array); // Liberar memória original em caso de falha
    return 1;
}
array = new_array;
```

Função Realloc

Mas ao realocar perdermos os dados anteriores???

NÃO!

Strings

Um string nada mais é que uma seqüência especial de caracteres. Em C, eles são colocadas entre ". Assim, "oba", "teste\n" (que usamos muito no printf) são exemplos de string.

Agora não confunda: 'x' é o CARACTER x, enquanto que "x" é o STRING x. E qual a diferença? Já vamos ver.

O C, infelizmente, não tem um tipo string pré-definido. Ou seja, ele tem que representar string de outra forma. E como ele representa? Como um vetor de caracteres terminado pelo caracter '\0'. Sendo assim, a única diferença entre um vetor de caracteres e uma string é a obrigatoriedade do '\0' no final da string.

Agora sim, podemos ver a diferença entre 'x' e "x". Quando escrevemos "x" o compilador, na verdade, cria um vetor com 2 caracteres: 'x' e '\0'. Por isso você não deve fazer confusão.

Lendo Strings

```c
char str[100];

// Solicita que o usuário insira uma string
printf("Digite uma string (máx 99 caracteres): ");
fgets(str, sizeof(str), stdin);
```

Acessando caractere por caractere

```c
char str[] = "meu string";
char letra;

printf("%c\n",str[2]); /*imprime o 3o caracter*/

letra = str[8];

printf("%c\n",letra); /*imprime o 9o caracter*/
```

strlen - Calcula o comprimento de uma string

A função strlen retorna o número de caracteres em uma string, excluindo o caractere nulo \0.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, world!";
    int length = strlen(str);
    printf("Comprimento da string: %d\n", length);
    return 0;
}
```

strcpy - Copia uma string para outra

A função strcpy copia uma string para outra.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char source[] = "Hello, world!";
    char destination[50];
    strcpy(destination, source);
    printf("Destino: %s\n", destination);
    return 0;
}
```

strncpy - Copia os primeiros n caracteres de uma string para outra

A função strncpy copia os primeiros n caracteres de uma string para outra. É mais segura que strcpy porque permite especificar o tamanho do destino.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char source[] = "Hello, world!";
    char destination[50];
    strncpy(destination, source, 5); // Copia apenas os primeiros 5 caracteres
    destination[5] = '\0'; // Adiciona o caractere nulo manualmente
    printf("Destino: %s\n", destination);
    return 0;
}
```

strcat - Concatena duas strings

A função strcat adiciona a string source ao final da string destination.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char destination[50] = "Hello, ";
    char source[] = "world!";
    strcat(destination, source);
    printf("Resultado: %s\n", destination);
    return 0;
}
```

strncat - Concatena os primeiros n caracteres de uma string a outra

A função strncat adiciona os primeiros n caracteres da string source ao final da string destination

```c
#include <stdio.h>
#include <string.h>

int main() {
    char destination[50] = "Hello, ";
    char source[] = "world!";
    strncat(destination, source, 3); // Adiciona apenas os primeiros 3 caracteres
    printf("Resultado: %s\n", destination);
    return 0;
}
```

strcmp - Compara duas strings

A função strcmp compara duas strings lexicograficamente.
Retorna:

•0 se as strings são iguais.

•Um valor negativo se a primeira string for menor que a segunda.

•Um valor positivo se a primeira string for maior que a segunda.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    int result = strcmp(str1, str2);
    if (result == 0) {
        printf("As strings são iguais\n");
    } else if (result < 0) {
        printf("str1 é menor que str2\n");
    } else {
        printf("str1 é maior que str2\n");
    }
    return 0;
}
```

strchr - Localiza a primeira ocorrência de um caractere em uma string

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, world!";
    char ch = 'o';
    char *ptr = strchr(str, ch);
    if (ptr != NULL) {
        printf("Primeira ocorrência de '%c' encontrada em: %s\n", ch, ptr);
    } else {
        printf("Caractere '%c' não encontrado\n", ch);
    }
    return 0;
}
```

strstr - Localiza a primeira ocorrência de uma substring em uma string

```c
#include <stdio.h>
#include <string.h>

int main() {
    char haystack[] = "Hello, world!";
    char needle[] = "world";
    char *ptr = strstr(haystack, needle);
    if (ptr != NULL) {
        printf("Substring '%s' encontrada em: %s\n", needle, ptr);
    } else {
        printf("Substring '%s' não encontrada\n", needle);
    }
    return 0;
}
```

Exercício

Escreva um programa em C que faça o seguinte:
1. Leia duas strings do usuário.
2. Compare as duas strings e informe se são iguais ou diferentes.
3. Concatene as duas strings e exiba o resultado.
4. Peça ao usuário um caractere específico e conte quantas vezes esse caractere aparece na string concatenada.
5. Exiba o comprimento da string concatenada.

Escreva um programa em C que faça o seguinte:

1. Leia uma string do usuário.
2. Verifique se a string é um palíndromo.
3. Informe ao usuário se a string é ou não um palíndromo.

Um palíndromo é uma palavra que escrita de trás para frente continua igual. Exemplo: ana, radar, rotor