Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

Alocação de Memória

Como já sabemos, em muitas situações não é possível determinar de antemão a quantidade de memória que nosso programa irá precisar e devemos recorrer a alocação dinâmica de memória. Para isso, usamos, basicamente, as funções `malloc()`, `realloc()` e `free()`. Todas essas funções estão presentes na biblioteca `stdlib.h`.

A função `malloc()`

A função `malloc()` (o nome é uma abreviatura de *memory allocation*) serve para alocar memória durante a execução do programa. Ela é responsável por solicitar ao sistema operacional um bloco de bytes consecutivos na memória RAM do computador e, em caso de sucesso, devolve o endereço desse bloco. Caso o computador não tenha memória disponível, a função retorna `NULL`. Veja como um array com n elementos inteiros pode ser alocado durante a execução de um programa (`malloc.c`):

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n;
    printf("Diginte a quantidade de numeros: ");
    scanf("%d", &n);
    int *p = malloc(n * sizeof(int));
    if (p == NULL){
        printf("Sem memória\n");
        exit(1);
    }
    for (int i = 0; i < n; i++){
        printf("Digite o valor da posicao %d: ", i);
        scanf("%d",&p[i]);
    }
    return 0;
}
```

Do ponto de vista conceitual (mas apenas desse ponto de vista) a instrução

```c
v = malloc(100 * sizeof(int));
```

tem efeito análogo ao da alocação estática

```c
int v[100];
```

A função `realloc()`

A função `realloc()` serve para alocar memória ou realocar blocos de memória previamente alocados pelas funções `malloc()` ou `realloc()`. A função recebe o endereço de um bloco previamente alocado e o número de bytes que o bloco redimensionado deve ter. A função aloca o novo bloco, copia para ele o conteúdo do bloco original, e devolve o endereço do novo bloco. Em caso de erro, a função também retorna um ponteiro nulo (`NULL`). Veja um exemplo (`realloc.c`):

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i;
    int *p = malloc(5*sizeof(int));
    if (p == NULL){
        printf("Sem memória\n");
        exit(1);
    }
    for (i = 0; i < 5; i++) {
        p[i] = i+1;
    }
    for (i = 0; i < 5; i++) {
        printf("%d\n", p[i]);
    }
    printf("\n");
    //Diminui o tamanho do array
    p = realloc(p,3*sizeof(int));
    if (p == NULL) {
        printf("Sem memória\n");
        exit(1);
    }
    for (i = 0; i < 3; i++) {
        printf("%d\n",p[i]);
    }
    printf("\n");
    //Aumenta o tamanho do array
    p = realloc(p, 10*sizeof(int));
    if (p == NULL) {
        printf("Sem memória\n");
        exit(1);
    }
    for (i = 0; i < 10; i++){
        printf("%d\n", p[i]);
    }
    return 0;
}
```

A função `free()`

Diferentemente das variáveis declaradas durante o desenvolvimento do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente por ele. É responsabilidade do programador fazer a desalocação de memória alocada dinamicamente. Dessa forma, sempre que alocamos memória de forma dinâmica, é necessário liberar essa memória quando ela não for mais necessária.

Desalocar, ou liberar, a memória previamente alocada faz com que ela se torne novamente disponível para futuras alocações. Para liberar um bloco de memória previamente alocado utilizamos a função `free()`. A instrução `free(ptr)` avisa ao sistema que o bloco de bytes apontado por `ptr` está disponível para reciclagem.

Note que nos exemplos anteriores não foi feita a desalocação de memória, modifique os códigos para que os mesmos fiquem correto. Use o `valgrind` para ter certeza seu programa não contém vazamento de memória. Para executar o valgrind, execute o seguinte comando (substituindo `<NomePrograma>` pelo nome correto):

```
valgrind --tool=memcheck --leak-check=full ./<NomePrograma>
```

Alocação de arrays multidimensionais

Existem várias soluções na linguagem C para se alocar dinamicamente um array com mais de uma dimensão. A mais tradicional é usando **ponteiro para ponteiro**. Seguindo essa abordagem, mantemos a notação de colchetes para cada dimensão.

Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array. Por exemplo, se quisermos um array com duas dimensões, precisaremos de um ponteiro com dois níveis (`**`); se quisermos três dimensões, precisaremos de um ponteiro com três níveis (`***`) e assim por diante.

O fragmento de código abaixo faz a alocação dinâmica de uma matriz com m linhas e n colunas:

```
1 int **p; //2 '*' = 2 níveis = 2 dimensões
2 p = malloc(m * sizeof(int *));
3 for (int i = 0; i < m; ++i)
4     p[i] = malloc (n * sizeof(int));
```

Alocamos no primeiro nível do ponteiro um array de ponteiros representando as linhas da matriz. Essa tarefa é realizada pela primeira chamada da função `malloc()`, a qual aloca o array usando o tamanho de um ponteiro para `int` (ou seja, `sizeof(int *)`). Em seguida, para cada posição desse array de ponteiros, alocamos um array de inteiros, o qual representa o espaço para as colunas da matriz, as quais efetivamente manterão os dados. Essa tarefa é realizada pela segunda chamada da função `malloc()`, dentro do comando for, a qual aloca o array usando o tamanho de um `int` (ou seja, `sizeof(int)`). Dessa forma, `p[i][j]` é o elemento de `p` que está na linha `i` e coluna `j`.

A Figura 1 exemplifica como funciona o processo de alocação de uma matriz usando o conceito de ponteiro para ponteiro.

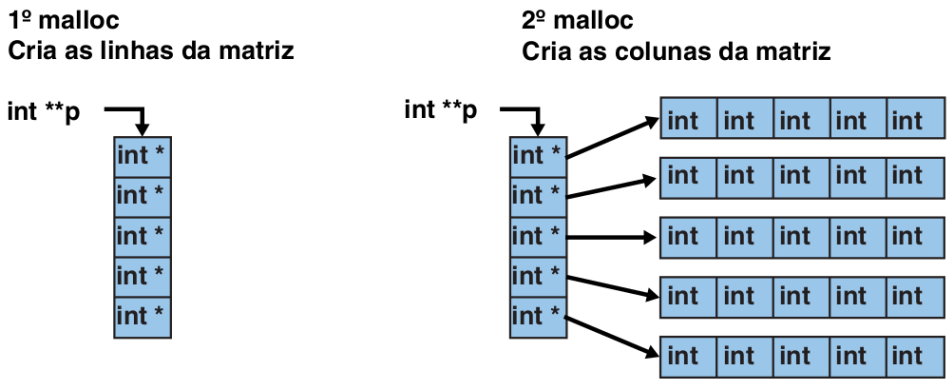

Figura 1: Representação da alocação de uma matriz bidimensional usando ponteiro para ponteiro

Para liberar a matriz, devemos liberar a memória no sentido inverso da alocação: primeiro liberamos as colunas, para depois liberar as linhas da matriz. Essa ordem deve ser respeitada porque, se liberarmos primeiro as linhas, perderemos os ponteiros para onde estão alocadas as colunas e, assim, não poderemos liberá-las. Ou seja,

```
for (int i = 0; i < m; ++i)
    free(p[i]) // Libera as colunas
free(p); // Libera as linhas
```

Exercícios

1. Analise cada trecho de código abaixo e aponte o problema de cada um.

a)

```
int *p = malloc(100*sizeof(int));
int *q = malloc(100*sizeof(int));
q = p;
```

b)

```
int *primos() {
   int v[3];
   v[0] = 1009; v[1] = 1013; v[2] = 1019;
   return v;
}
```

c)

```
char *p;
int n = 2;
if (n % 2 == 0) {
   char c = 'A';
   p = &c;
}
printf("%c\n", *p);
```

d)

```c
int *p;
int *q;
p = malloc(100*sizeof(int));
q = p;
free(q);
for(int i = 0; i < 100; i++)
    p[i] = i + 1;
```

e)

```c
int *p;
int *q;
p = malloc(100*sizeof(int));
q = p;
free(p);
free(q);
```

f)

```c
int n;
int *p;
while(1) {
    scanf("%d", &n);
    p = malloc(n*sizeof(int));
    if(n < 0)
        break;
}
```

2. (ex2.c) Teste o código abaixo, fornecendo o valor 5 como entrada. Para cada posição do array alocado, o programa imprime seu endereço e valor armazenado.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *a;
    int n, i;
    scanf("%d", &n);
    a = malloc(n * sizeof(int));
    for (i = 0; i < n; i++) {
        printf("%p: %d\n", &(a[i]), a[i]);
    }
    return 0;
}
```

a) Note que os endereços variam de 4 em 4 (valores em hexadecimal). O que isso indica?

b) Execute novamente o programa algumas vezes, e note que os endereços mudam. O que isso indica?

c) São alocadas n posições. Modifique o comando `for` para imprimir $2 * n$ posições e teste o programa. Qual o "perigo" de se fazer isso?

d) Os valores impressos são um lixo qualquer, valores que já estavam no local antes da alocação dinâmica. Coincidentemente, alguns valores (até todos) podem ser zeros. Muitas vezes é interessante já iniciar o array preenchido com zeros (0). Para isso, podemos usar a função `calloc`, dessa forma, garantimos que em todas as posições teremos o valor 0. Substitua a função `malloc` por `calloc` e teste o programa.

3. (`ex3.c`) Analise o código abaixo:

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *x = NULL;
    int i, v, n = 0;
    printf("Escreva varios numeros inteiros (0 encerra)\n");
    scanf("%d", &v);
    while (v) {
        n++;
        x = realloc(x, n * sizeof(int));
        x[n - 1] = v;
        scanf("%d", &v);
    }
    for (i = 0; i < n; i++) {
        printf("x[%d] = %d\n", i, x[i]);
    }
    return 0;
}
```

Note que a variável `x` foi inicializada com `NULL`, portanto ela não aponta para nenhuma lugar quando inicia-se a execução do programa, ela está "zerada". Assim, a primeira execução do `realloc` funciona como um `malloc`.

a) O que faz o programa?

b) Por que o valor de `v` é guardado na posição `n-1` e não na posição `n` do array `x`?

c) Qual a vantagem desse programa em relação a um programa que usa array de tamanho estático (declarando o array como `int x[500]` por exemplo)? E qual a desvantagem?

4. (`ex4.c`) As funções `malloc`, `calloc` e `realloc` retornam o endereço do local alocado, ou `NULL` caso não se consigam alocar uma área do tamanho requisitado. Em programas que utilizam constantemente alocação e realocação de grande quantidade de memória é sempre bom verificar o valor retornado, para evitar erros de execução ao acessar posições "inexistentes", ou seja, memória não efetivamente alocada para o programa.

a) Analise o código abaixo. Você o executaria no seu computador? Caso afirmativo, execute e confira o resultado. Caso contrário, por que você não o executaria?

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

int main() {
    int *x;
    int i = 0;
    while(1) {
        printf("%d ", i++);
        x = malloc(1000000*sizeof(int));
        if (x == NULL){
            break;
        }
    }
    return 0;
}
```

b) Acrescente o comando `free(x)` antes da chave que fecha o bloco `while`. Agora você executaria o código no seu computador? Por quê?

5. (`ex5.c`) Em uma matriz simétrica $n \times n$, não há necessidade, no caso de $i \neq j$, de armazenar ambos os elementos `mat[i][j]` e `mat[j][i]`, já que os dois têm o mesmo valor. Portanto, basta guardar os valores dos elementos da diagonal e da metade dos elementos restantes (por exemplo, os elementos abaixo da diagonal, para os quais $i > j$). Dessa forma, podemos fazer uma economia de espaço usado para alocar a matriz. Complete as funções do arquivo `ex5.c` para que o mesmo funcione como solicitado. Obrigatoriamente, você deve fazer a alocação da matriz como descrito no exercício.

6. (`ex6.c`) Na matemática um número não tem limites de dígitos, mas isso nem sempre é verdade quando estamos trabalhando com algumas linguagens de programação. Esse é o caso de C, onde, por exemplo, uma variável do tipo `unsigned long long` pode armazenar um valor entre 0 e 18.446.744.073.709.551.615 (ou seja, um número com menos de 20 dígitos). Quando precisamos armazenar valores com mais dígitos, devemos recorrer a outras linguagens (como Python) ou armazenar cada dígito do número em um array (normalmente um array de `char`). Usando essa estratégia, um número pode ter uma quantidade bem grande de dígitos, mas qualquer operação matemática deve ser implementada pelo programador. Nesse exercício você deve ler um número inteiro (que será armazenado em um array de `char`), converter cada dígito para o seu correspondente inteiro (por exemplo, o caractere '1' deve ser convertido para o inteiro 1) e armazená-lo em um array de `int`. Não se esqueça de liberar qualquer espaço alocado dinamicamente. Use o Valgrind para ter certeza que não há vazamento de memória.

   Dica: Você pode usar a função `atoi` para converter um dígito em número ou:

```c
char um = '1';
char dois = '2';
char nove = '9';
printf("%d\n", um - '0');
printf("%d\n", dois - '0');
```

```
6 printf("%d\n", nove - '0');
7 printf("%d\n", (dois - '0')*(nove - '0'));
```

7. Utilizando alocação dinâmica, implemente um programa para resolver o problema abaixo. Lembre-se de, ao final do programa, desalocar toda a área alocada.

   Em uma empresa, os funcionários trabalham de forma escalonada e recebem de acordo com a quantidade de horas trabalhadas em cada dia do mês. Seu objetivo é fazer um programa que leia a quantidade de horas trabalhadas por dia de cada funcionário, calcular e imprimir algumas informações. O programa deve:

   - Ler a quantidade de funcionários da empresa;
   - Alocar dinamicamente a memória para armazenar os dados desses funcionários;
   - Para cada funcionário, ler o seu código, quantos dias trabalhou e as horas trabalhadas de cada dia;
   - Sabendo que a empresa paga R$25,00/hora, imprimir o valor que cada funcionário deve receber;
   - Imprimir o valor total gasto com funcionários pela empresa;
   - Desalocar toda a memória alocada.

   Não se sabe de antemão o número de funcionários da empresa, nem o número de dias que cada funcionário trabalhou no mês, por isso o programa deverá usar alocação dinâmica. Utilize a seguinte definição de tipo em seu programa.

```
1 typedef struct {
2     int codigo;                //codigo do funcionario
3     int qtd_dias_trabalhados; //qtd de dias que o funcionario trabalhou
4     int *horas_trabalhadas;   //horas trabalhadas em cada dia pelo func.
5 }Funcionario;
```

   Exemplo (os valores destacados foram fornecidos pelo usuário):

```
Quantidade de funcionarios: 3

--- Funcionario 1 ---
Codigo: 1111
Quantidade de dias trabalhados: 5
Horas trabalhadas em cada dia: 8 4 10 12

--- Funcionario 2 ---
Codigo: 2222
Quantidade de dias trabalhados: 2
Horas trabalhadas em cada dia: 6 10
```

```
--- Funcionario 3 ---
Codigo: 3333
Quantidade de dias trabalhados: 10
Horas trabalhadas em cada dia: 8 8 8 8 8 8 8 8 8 8

--- Relatorio Final ---
Funcionario - Horas Trabalhadas - Salário
    1111    -         34        - R$850,00
    2222    -         16        - R$400,00
    3333    -         80        - R$2000,00

Total pago aos funcionarios: R$3250,00
```