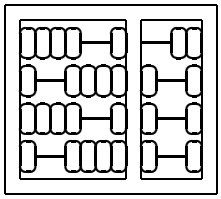

MC-102 Algoritmos e Programação de Computadores
IC-UNICAMP
Aula 27 - Alocação Dinâmica

1 Objetivos

Apresentar como funciona a alocação dinâmica, quais os comandos que são utilizados e também motivos para se utilizar alocação dinâmica.

2 Motivação

- Usar somente a memória necessária, diminuindo assim a quantidade de recursos necessária para a execução do programa.
- Não impor limites quanto aos recursos utilizados, a não ser físicos (e.g. tamanho da memória), para os programas.

3 Alocação Dinâmica de Memória

Suponhamos que você vai escrever um programa interativo e que você não conheça de antemão quantas entradas de dados serão fornecidas. Uma solução para esse tipo de problema é a de solicitar memória toda vez que precisarmos. O mecanismo para aquisição de memória é a função **malloc()**.

3.1 Malloc

A função `malloc()` (abreviatura de *memory allocation*) da biblioteca padrão aloca um bloco de bytes consecutivos na memória do computador e devolve o endereço desse bloco. Eis um exemplo que recebe um caracter e imprime o caracter seguinte:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char *ptr;
    ptr = (char *) malloc (1);
    scanf ("%c", ptr);
    printf ("%c\n", *ptr + 1); /* nao confunda com *(ptr+1) */
    free (ptr);
}
```

Eis outro exemplo, que calcula a soma de dois números:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr;
    int x;
    ptr = (int *)malloc (sizeof (int));
    scanf ("%d %d", ptr, &x);
    *ptr = *ptr + x;
    printf ("%d\n", *ptr);
    free (ptr);
}
```

A expressão `sizeof(int)` dá o número de bytes de um `int`.

A função malloc devolve um ponteiro "genérico" - ou seja, do tipo `void *` para um bloco de bytes consecutivos. Geralmente, esse ponteiro é automaticamente convertido em ponteiro-para-int uma vez que a variável `ptr` é do tipo `int *`. Mas o ideal é usar *type casts* para que o `C` não nos pregue nenhuma peça.

3.2 Conversão de Tipos

Algumas vezes precisamos forçar o compilador a retornar o valor de uma expressão com um tipo particular. Por exemplo, suponha que estamos usando uma variável `n` do tipo `float`, em nosso programa, em um dado momento queremos calcular a raiz quadrada desse valor. O **C** nos oferece uma função chamada **sqrt()**, que retorna a raiz quadrada, mas espera um argumento do tipo `double`, e dará um resultado sem sentido se uma variável de outro tipo lhe for passada.

Assim precisamos converter nossa variável `n` para o tipo `double`. Para isso, vamos usar o que se chama *type casts* (ou operador de molde) que consiste em colocar o tipo desejado entre parênteses antes do valor a ser convertido. Ex.:

```
resposta = sqrt((double)n);
```

A expressão `(double)`converte `n` para o tipo `double` antes de usá-lo.

É necessário fazer uma conversão similar com o valor retornado pelo `malloc()`, pois o compilador precisa saber duas informações sobre qualquer ponteiro: o endereço da variável apontada e o seu tipo.

A função `malloc()` retorna um ponteiro para void. Portanto devemos indicar ao compilador que o valor retornado por `malloc()` é do tipo **ponteiro para inteiro**. Ao fazer isso, o `cast` força o valor a ser desse tipo.

3.2.1 free

A função `free()` libera a porção de memória alocada por malloc. O comando `free (ptr)` avisa o sistema de que o bloco de bytes apontado por `ptr` está livre. A próxima chamada de `malloc` poderá tomar posse desses bytes.

3.3 calloc

A linguagem C também apresenta a função `calloc()` que é muito parecida com o `malloc()`.

A função `calloc()` recebe dois argumentos, o primeiro é o número de células de memória desejado e o segundo é o tamanho de cada célula em bytes. Ex.:

```
long *mem;
mem = (long *)calloc(100,sizeof(long)); /* == mem = (long *)malloc(100*sizeof(long)); */
```

Outra diferença do `calloc()` é que ele inicializa todo o conteúdo alocado com **zeros**.

As funções `malloc()`, `calloc()` e `free()` estão na biblioteca stdlib. Portanto, diga

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

no início de qualquer programa que use `malloc()` e `free()`.

3.4 A memória não é infinita

Se a memória do computador já estiver toda ocupada, `malloc()` não consegue alocar mais espaço e devolve `NULL`. Convém verificar essa possibilidade antes de prosseguir:

```
ptr = (int *)malloc (sizeof (int));
if (ptr == NULL) {
    printf ("Socorro! malloc devolveu NULL!\n");
}
```

3.5 Vetores e endereços

Os elementos de qualquer vetor têm endereços consecutivos na memória do computador. Depois da declaração:

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

A variável `v` é, essencialmente, um ponteiro para o primeiro elemento do vetor. Mais precisamente, `v` é uma espécie de "ponteiro constante": não se pode mudar o valor de `v`.

Como `v` contém o endereço do primeiro elemento do vetor, a expressão `v+1` é o endereço do segundo elemento, `v+2` é o endereço do terceiro elemento, etc. Se `i` é uma variável do tipo `int` então as expressões `v + i` e `&v[i]` têm exatamente o mesmo valor. Portanto,

```
*(v+i) = 87;
```

tem o mesmo efeito que

```
v[i] = 87;
```

Para "carregar" um vetor v[10] pode-se dizer

```
for (i = 0; i < 10; i++) scanf ("%d", &v[i]);
```

ou

```
for (i = 0; i < 10; i++) scanf ("%d", v + i);
```

As duas formas têm exatamente o mesmo efeito.

Exemplo:

Eis como um vetor (*array*) com **n** elementos inteiros pode ser alocado durante a execução de um programa:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *v;
    int n, i;

    scanf ("%d", &n);
    v = (int *)malloc(n * sizeof (int));
    for (i = 0; i < n; i += 1)
        scanf ("%d", &v[i]);
    for (i = 0; i < n; i += 1)
        printf ("%d ", v[i]);
    free (v);
}
```

Do ponto de vista conceitual (mas apenas desse ponto de vista) o comando

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

tem o mesmo efeito que

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

A propósito, convém lembrar que o C padrão não permite escrever `int v[n]`, a menos que `n` seja uma constante, definida por um `#define`.

3.6 Exercícios

1. Escreva uma função que receba um caracter `c` e transforme `c` em uma cadeia de caracteres, ou seja, devolva uma cadeia de caracteres de comprimento 1 tendo `c` como único elemento.

2. Escreva um programa que leia um número inteiro positivo `n` seguido de **n** números inteiros e imprima esses **n** números em ordem invertida. Por exemplo, ao receber

   ```
   5  222 333 444 555 666
   ```

   o seu programa deve imprimir

   ```
   666 555 444 333 222
   ```

   O seu programa não deve impor limitações sobre o valor de n.

   Resposta:

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int n;
   int *vetor;

   printf("Entre com o numero de elementos: ");
   scanf ("%d", &n);

   vetor = (int *) malloc(n * sizeof (int)); /* Alocando espaco para n inteiros *

   for (i = 0; i < n; i += 1) {
      printf(" Digite elemento %d: ");
      scanf ("%d", &vetor[i]);
   }

   printf(" Imprimindo na ordem inversa: \n");
   for (i = n-1; i >=0; i += 1)
      printf ("%d \n", vetor[i]);

   free (v); /* Nao esquecer de liberar o espaco alocado */
}
```

3. Escreva um programa que leia uma matriz de um arquivo texto e mostre a sua transposta. A primeira linha contém dois números, que são o número de linhas e o número de colunas respectivamente. Após cada linha do arquivo contém as linhas da matriz. Por exemplo, o arquivo abaixo apresenta uma matriz 3 x 5:

```
3 5
10 20 30 40 50
60 70 89 90 99
80 79 69 59 49
```

Após mostre sua matriz transposta.

obs.: Faça duas implementações desse problema, uma usando matrizes e outra usando vetores para simular matrizes.

Resposta 1 - Usando somente vetor:

```c
/* Vetores Simulando Matrizes */

#include <stdio.h>
#include <stdlib.h>
int main()
{
   int x,y,lin,col;
   int *v;
   FILE *arq;
```

```
    if((arq=fopen("matriz.txt","r"))==NULL) {
      printf("Erro ao tentar abrir o arquivo!! \n");
      return;
    }

    fscanf (arq,"%d %d", &lin,&col);

    v = (int *) malloc(lin * col * sizeof (int)); /* Alocando espaco */

    for (y=0;y<lin;y++)
      for (x=0;x<col;x++)
        fscanf (arq,"%d",&v[y*col+x]);

    fclose(arq);

    for (y=0;y<lin;y++){
      for (x=0;x<col;x++)
        printf ("%2d ", v[y*col+x]);
      printf("\n");
    }

    free (v); /* Nao esquecer de liberar o espaco alocado */
}
```

Resposta 2 - Usando matriz

```
/* Matrizes */

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int x,y,lin,col;
    int **m;
    FILE *arq;

    if((arq=fopen("matriz.txt","r"))==NULL) {
      printf("Erro ao tentar abrir o arquivo!! \n");
      return;
    }

    fscanf (arq,"%d %d", &lin,&col);

    m = (int **) malloc(lin * sizeof (int)); /* Numero de linhas*/
    for(y=0;y<lin;y++)
      m[y] = (int *) malloc(col * sizeof (int)); /* Para cada linha col colunas */
```

```
    for (y=0;y<lin;y++)
      for (x=0;x<col;x++)
        fscanf (arq,"%d",&m[y][x]);

    fclose(arq);

    for (y=0;y<lin;y++){
      for (x=0;x<col;x++)
        printf ("%2d ", m[y][x]);
      printf("\n");
    }

    /* Nao esquecer de liberar o espaco alocado */
    /* Ordem inversa a alocacao */

    for(y=0;y<lin;y++)
      free(m[y]); /* Liberando todas as colunas de cada linha */
    free (m); /* Liberando o vetor que continha o inicio de cada linhas */
}
```

4 Bibliografia Utilizada

Paulo Feofiloff - Projeto de Algoritmos. disponível em: http://www.ime.usp.br/ pf/algoritmos