# Alocação Dinâmica

Profa Dra. Eloize Seno

# Introdução

- As variáveis de um programa podem ser alocadas estaticamente ou dinamicamente.
- **Alocação estática**: a memória é alocada para uma variável em tempo de compilação e permanece alocada durante toda a execução do programa.
- **Alocação dinâmica**: permite alocar memória para variáveis durante a execução de um programa.
  - Uso mais eficiente da memória: variáveis não mais utilizadas podem ser liberadas da memória.
  - Não há desperdiço

# Alocação Estática

- As declarações abaixo alocam memória para diversas variáveis. A alocação é ***estática***, pois acontece antes que o programa começa a ser executado:

```
char c;
int i, v[10];
```

**Obs:** Às vezes, a quantidade de memória a ser alocada só se torna conhecida durante a execução do programa. Para lidar com essa situação é preciso recorrer à alocação ***dinâmica*** de memória.

# Alocação Dinâmica

- O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**:
  - **malloc, calloc, realloc, free**.

**Obs**: Existem diversas outras funções que são amplamente utilizadas, mas dependentes do ambiente e compilador. Neste curso serão abordadas somente estas funções padronizadas.

# Função malloc()

- A função **malloc** (abreviatura de *memory allocation*) aloca um bloco de bytes consecutivos na memória do computador e devolve o endereço desse bloco.

- O tamanho do bloco de bytes deve ser especificado no argumento da função.

# Função malloc (cont.)

- A função malloc() tem o seguinte protótipo:
  **void *malloc (unsigned int num);**
- A função toma o número de bytes que se deseja alocar (num), aloca a memória e retorna um ponteiro void * para o primeiro byte alocado ou **NULL**, se não houver memória suficiente.
- O ponteiro void * pode ser atribuído a qualquer tipo de ponteiro.

# Função malloc() (cont.)

- Exemplo: No seguinte fragmento de código, **malloc** aloca 1 byte:

```
char *ptr;
ptr = (char *) malloc (1);
scanf ("%c", ptr);
```

O ponteiro void* que malloc() retorna precisa ser convertido para char* antes de ser atribuído a ptr.

- O endereço devolvido por malloc é do tipo "genérico" **void ***. O programador armazena esse endereço num ponteiro de tipo apropriado. No exemplo acima, o endereço é armazenado num ponteiro para **char**.

# Função malloc() (cont.)

- Para alocar um tipo de dado que ocupa vários bytes, é preciso recorrer ao operador **sizeof**, que diz quantos bytes o tipo especificado tem.

- Exemplo:

```
typedef struct {
   int dia, mes, ano;
} data;
data *d;
d = (data *) malloc (sizeof (data));
d->dia = 31;
d->mes = 12;
d->ano = 2008;
```

# Função free()

- As variáveis alocadas **estaticamente** dentro de uma função desaparecem quando a execução da função termina.
- As variáveis alocadas **dinamicamente** continuam a existir mesmo depois que a execução da função termina.
- Se for necessário liberar a memória ocupada por essas variáveis, é preciso recorrer à função **free ()**.

# Função free() (cont.)

- A função **free()** libera a porção de memória alocada por **malloc()**.
  - A próxima chamada de malloc poderá tomar posse desses bytes que foram liberados por essa função!

- Protótipo da função:

  **void free (void *p);**

  - onde: p é o ponteiro que aponta para o início da memória alocada.

# Função free() (cont.)

- Exemplo:

```
char *ptr;
ptr = (char *) malloc(1);
printf(“Digite um caractere:”)
scanf ("%c", ptr);
printf("%c", *ptr);
free(ptr);
```

# Vetores Dinâmicos

- Vetores e matrizes também podem ser alocados dinamicamente.
- Exemplo:

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

# Exemplo de vetor alocado dinamicamente com malloc()

```
#include <stdio.h>
#include <stdlib.h>/* Para usar malloc() */
main ()
{ int *p, n, i;
  printf(“Quantos números deseja armazenar no vetor?\n”);
  scanf(“%d”, &n);
  p=(int *) malloc(n * sizeof(int)); /* Aloca espaço para armazenar n
                                        números inteiros.*/
  if (!p)
  {   printf ("Erro: Memoria Insuficiente !");
      exit(0);   }
  for (i=0; i<n ; i++)
  {  p[i] = i*i; /* p pode ser tratado como um vetor de n elementos */
     printf(“Elemento da posicao %d = %d\n”, i, p[i]);  }
   free(p); } // fecha a funcao main()
```

# Função calloc()

- A função **calloc()** é similar a **malloc()**, mas possui um protótipo um pouco diferente:

  **void *calloc (unsigned int num, unsigned int size);**

- A função aloca uma quantidade de memória igual a num * size, isto é, aloca memória suficiente para um vetor de num objetos de tamanho size e retorna um ponteiro void * para o primeiro byte alocado, ou **NULL** se não houver memória suficiente.

- A principal diferença de **calloc()** é que ela inicializa o espaço alocado com 0.

# Exemplo de vetor alocado dinamicamente com calloc()

```
#include <stdio.h>
#include <stdlib.h>/* Para usar malloc() */
main ()
{ int *p, n, i;
  printf(“Quantos números deseja armazenar no vetor?\n”);
  scanf(“%d”, &n);
  p=(int *) calloc (n, sizeof(int)); /* Aloca espaço para armazenar n
                                        números inteiros.*/
  if (!p)
  {   printf ("Erro: Memoria Insuficiente !");
      exit(0);   }
  for (i=0; i<n ; i++)
  {  p[i] = i*i; /* p pode ser tratado como um vetor de n elementos */
     printf(“Elemento da posicao %d = %d\n”, i, p[i]);  }
   free(p); } // fecha a funcao main()
```

# Função realloc()

- A função realloc() serve para realocar memória e tem o seguinte protótipo:

  **void *realloc (void *ptr, unsigned int num);**

- A função modifica o tamanho da memória previamente alocada, apontada por *ptr, para o valor especificado por num.
  - O valor de num pode ser maior ou menor que o original.
  - Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco (antigo) para aumentar seu tamanho.
  - Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida.
  - Se ptr for NULL, aloca size bytes e devolve um ponteiro; senão, a memória apontada por ptr é liberada. Se não houver memória suficiente, um ponteiro NULL é devolvido e o bloco original fica inalterado.

# Exemplo de uso da Função realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{  int *p, n, i;
    printf("Quantos números deseja
    armazenar no vetor?\n");
    scanf("%d", &n);
    p=(int *) malloc(n * sizeof(int)); /*
    Aloca espaço para armazenar n
    números inteiros.*/
    if (!p)
    {  printf ("Erro: Memoria
                Insuficiente !");
     exit(0);   }
```

continua->

```
for (i=0; i<n ; i++)
{  p[i] = i*i;
   printf("Elemento da    posicao
     %d = %d\n", i, p[i]);  }
/* O tamanho de p deve ser
modificado, por algum motivo */
 printf("Quantos números deseja
     armazenar agora no vetor?\n");
 scanf("%d", &n);
 p = (int *)realloc (p, n*sizeof(int));
 for (i=0; i<n ; i++)
 {  p[i] = n*i*(i-6);
    printf("Elemento da posicao
     %d = %d\n", i, p[i]);  }
 free(p); } fecha funcao main()
```

# Exercício de Fixação

- Modifique o programa anterior mostrando que no caso do realloc, os dados previamente alocados permanecem inalterados.

# Matrizes Dinâmicas

- C só permite alocação dinâmica de **conjuntos unidimensionais.**

- É necessário, portanto, criar **abstrações conceituais** com vetores para representar matrizes alocadas dinamicamente.

# Matrizes Dinâmicas (cont.)

- A matriz bidimensional pode ser representada por um vetor simples (unidimensional )

- Onde:

  – primeiras posições do vetor armazenam elementos da primeira linha

  – seguidos dos elementos da segunda linha, e assim por diante.

- Exige disciplina para acessar os elementos da matriz

# Matrizes Dinâmicas (cont.)

- Exemplo:

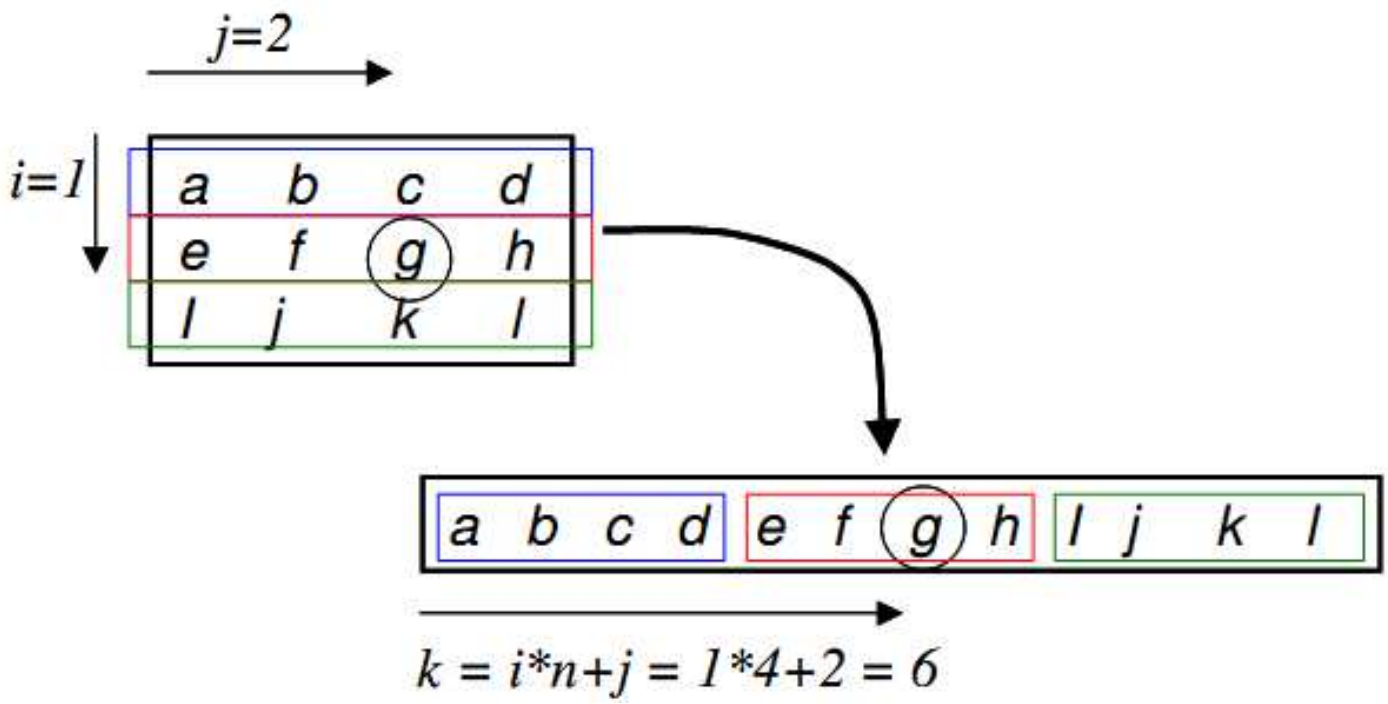
```
float *mat; /* matriz m x n representada por um
    vetor */
//...
mat = (float *) malloc(m * n * sizeof(float));
```

- Disciplina de acesso:

  – **mat[ i ][ j ]** é mapeado em **v[k]** para **k= i * n + j**, onde n é o número de colunas de mat.

# Matrizes Dinâmicas (cont.)

- **mat[ i ][ j ]** mapeado em **v[ i * n + j ]:**



# Exercícios

1- Faça uma função que receba um valor n e crie dinamicamente um vetor de n elementos e retorne um ponteiro. Crie uma função que receba um ponteiro para um vetor e um valor n e imprima os n elementos desse vetor. Construa também uma função que receba um ponteiro para um vetor e libere esta área de memória. Por fim, crie uma função principal que leia um valor n e chame a função de alocação de memória. Depois, a função principal deve ler os n elementos desse vetor. Então, a função principal deve chamar a função de impressão dos n elementos do vetor criado e, finalmente, liberar a memória alocada através da função criada para liberação.

# Exercícios (cont.)

2- Construa uma função que receba dois parâmetros m e n, aloque uma matriz de ordem m x n e retorne um ponteiro para a matriz alocada em tempo de execução. Crie ainda outra função que receba por parâmetro um ponteiro para matriz e libere a área de memória alocada. Finalmente, crie um programa (main) que teste/use as duas funções criadas.