UNIVERSIDADE FEDERAL DO CEARÁ

UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS CRÁTEUS

CURSOS: CIÊNCIA DA COMPUTAÇÃO E SISTEMA DE INFORMAÇÃO

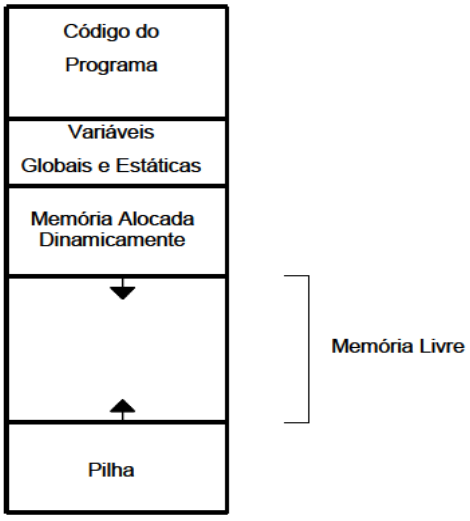
DISCIPLINA: ESTRUTURA DE DADOS

PROFS. BRUNO DE CASTRO E WELLINGTON

NOTA DE AULA SOBRE ALOCAÇÃO DINÂMICA

1. ALOCAÇÃO DINÂMICA

Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória. O sistema operacional reserva também os espaços necessários para armazenarmos as variáveis globais (e estáticas) existentes no programa.
O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à pilha de execução e, quando a função termina, é desempilhado.



Com alocação dinâmica, declaramos uma variável do tipo ponteiro que posteriormente recebe o valor do endereço de um elemento ou do primeiro elemento de uma sequência de endereços, alocado dinamicamente.

Mesmo que um espaço seja alocado dinamicamente no escopo local de uma função, podemos acessá-lo depois da função ser finalizada, pois a área de memória ocupada por ele permanece válida. Se o programa não liberar o espaço alocado dinamicamente, este será automaticamente liberado quando a execução do programa terminar.

2. FUNÇÕES DE ALOCAÇÃO DINÂMICA

As principais funções de alocação dinâmica são **malloc** e **free**, que estão na biblioteca **stdlib**. Para usar esta biblioteca, você deve incluir o correspondente arquivo-interface no seu programa por meio de

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

A função **malloc** (o nome é uma abreviatura de memory allocation) aloca um bloco de bytes consecutivos na memória do computador e retorna o endereço desse bloco. O número de bytes é especificado como parâmetro desta função. No seguinte fragmento de código, **malloc** aloca 1 byte:

```
/*
A sintaxe de malloc é: void* malloc (unsigned int num);
- onde void* significa que malloc retorna um endereço de memória genérico,
- unsigned, de forma simplificada, significa sem sinal negativo,
- int o tipo da variável ou valor a ser passado,
- e num um valor númerico
*/
char *ptr;
ptr = (char* )malloc(1);
scanf( "%c", ptr);
printf("%d",*ptr);
```

O endereço devolvido por **malloc** é armazenado em um ponteiro do tipo apropriado. No exemplo anterior, o endereço é armazenado num ponteiro-para-char. Para alocar um tipo-de-dado que ocupa mais de 1 byte, é preciso recorrer ao operador **sizeof**, que diz quantos bytes o tipo especificado tem.

É valido ressaltar que cada invocação de **malloc** aloca um bloco de bytes consecutivos maior que o solicitado: os bytes adicionais são usados para guardar informações administrativas sobre o bloco de bytes (essas informações permitem que o bloco seja corretamente desalocado, mais tarde, pela função **free**). O número de bytes adicionais pode ser grande, e não depende do número de bytes solicitado no argumento de **malloc**.

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 = malloc( sizeof (float));
if (ptr == NULL) {
  printf( "Memoria cheia! malloc devolveu NULL!\n");
  exit( EXIT_FAILURE);
}
```

Recapitulando o que foi descrito no tópico 1, as variáveis alocadas estaticamente dentro de uma função desaparecem quando a execução da função termina. Já 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**.

A função **free** libera a porção de memória alocada por **malloc**. A instrução **free(ptr)** avisa ao sistema que o bloco de bytes apontado por **ptr** está livre e pode ser utilizado por outros programas. A próxima chamada de **malloc** poderá tomar posse desses bytes.

Há três ponderações a serem feitas sobre a função **free**:

1. A função **free** não deve ser aplicada a uma parte de um bloco de bytes alocado por **malloc**; aplique **free** apenas ao bloco todo;
2. Como mencionado no parágrafo anterior, a função **free** libera um espaço de memória para ser alocado pelo mesmo programa ou por

outro programa, mas os dados guardados neste espaço de memória permanecem na memória do computador.

3. O ponteiro a qual é passado como parâmetro continua apontando para a porção de memória liberada por **free.** A estes ponteiros chamamos de *ponteiros soltos*.

Convém não deixar ponteiros "soltos" (dangling pointers) no seu programa, pois isso pode ser explorado por hackers acessar dados do seu programa. Portanto, depois de cada **free(ptr)**, atribua **NULL** a **ptr**:

```
free(ptr);
ptr = NULL;
```

Outra função importante para alocar memória dinamicamente é **calloc()**. Normalmente é utilizado para criar um vetor de tamanho dinâmico, ou seja, definido durante a execução do programa. Difere da função **malloc**, pois além de inicializar os espaços de memória ainda atribui o valor 0 (zero) para cada um deles. É útil, pois em C quando se declara um variável o espaço no mapa de memória usado por esta provavelmente contém algum valor lixo. Um exemplo de sua utilização é dado a seguir:

```
/*
sintaxe da calloc: void* calloc(n_de_elementos ,
tamanho_de_cada_elemento);
*/
 int *p;
 int n;
 ...
 p = (int*) calloc(n, sizeof(int));
```

E por fim, a linguagem C oferece ainda um mecanismo para re-alocarmos um vetor dinamicamente. Em tempo de execução, podemos verificar que a dimensão inicialmente escolhida para um vetor tornou-se insuficiente (ou excessivamente grande), necessitando um redimensionamento. A função **realloc** da biblioteca padrão nos permite re-alocar um vetor, preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação (no fragmento de código abaixo, **m** representa a nova dimensão do vetor).

```
/*
sintaxe da realloc: void* realloc(void* ptr,  unsigned int novo_tamanho);
*/
int *v, n, m;
...
v = (int*) realloc(v, m*sizeof(int));
```

Vale salientar que, sempre que possível, optamos por trabalhar com vetores criados estaticamente. Eles tendem a ser mais eficientes, já que os vetores alocados dinamicamente têm uma indireção a mais (primeiro acessa-se o valor do endereço armazenado na variável ponteiro para então acessar o elemento do vetor).

3. ALOCAÇÃO DINÂMICA EM VETORES

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

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

Do ponto de vista conceitual (e apenas desse ponto de vista) a instrução:

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

tem efeito análogo ao da alocação estática:

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

Convém lembrar que a norma ANSI não permite declarar:

int v[n];

A menos que **n** seja uma constante, definida por um **#define**. Portanto, utilizar o comando **int v[n],** com **n** sendo uma variável do tipo **int**, pode não funcionar em todos os compiladores.

A seguir é exemplificado como criar um vetor com a função **calloc**:

```
int *p;
int n;
int i;

...   /* Determina o valor de n em algum lugar */
n = 3;
p = calloc(n, sizeof(int));

/* Aloca n números inteiros p pode agora ser tratado como um vetor
com n posicoes */

//p = malloc(n*sizeof(int));  /* Maneira equivalente usando malloc. */

if (!p){
  printf ("** Erro: Memoria Insuficiente **");
}

for (i=0; i<n; i++)
   p[i] = i*i;

/* p pode ser tratado como um vetor com n posicoes */
...
```

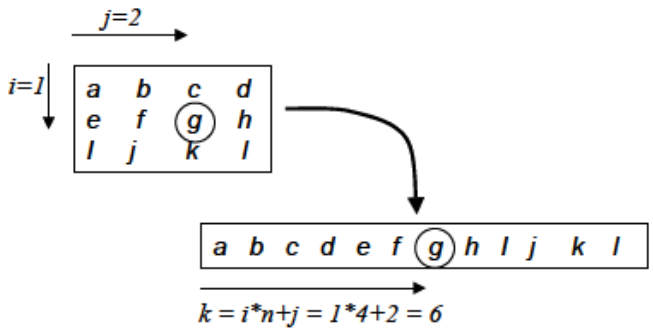
Segue o exemplo de como utilizar a função **realloc**:

```
...
 char *str;

/* Memoria inicialmente alocada */
str = (char *) malloc(11);
 strcpy(str, "www.google");
 printf("String = %s,  Endereco = %p\n", str, str);

 /* Realocação de memória */
 str = (char *) realloc(str, 15);
 strcat(str, ".com");
 printf("String = %s,  Endereco = %p\n", str, str);
 free(str);
```

4. ALOCAÇÃO DINÂMICA EM MATRIZES

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. O problema que encontramos é que a linguagem C só permite alocarmos dinamicamente conjuntos unidimensionais. Para trabalharmos com matrizes alocadas dinamicamente, temos que criar abstrações conceituais com vetores para representar conjuntos bidimensionais.

Uma forma de alocar uma matriz dinamicamente é representado-a em um vetor unidimensional. A estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento **mat[i][j]** de uma matriz, devemos acessar o elemento **v[i*n+j]**, onde **n** representa o número de colunas da matriz.



Esta conta de endereçamento é intuitiva: se quisermos acessar elementos da terceira **(i=2)** linha da matriz, temos que pular duas linhas de elementos **(i*n)** e depois indexar o elemento da linha com **j**. Com esta estratégia, a alocação da "matriz" recai numa alocação de vetor que tem **m*n** elementos, onde **m** e **n** representam as dimensões da matriz.

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

No entanto, somos obrigados a usar uma notação desconfortável, **v[i*n+j]**, para acessar os elementos, o que pode deixar o código pouco legível.

A segunda estratégia para representar matrizes por meio de alocação dinâmica é por meio do conceito de ponteiros para ponteiros. De acordo com esta estratégia, cada linha da matriz alocada dinamicamente é representada por um de ponteiro que aponta o primeiro endereço de uma sequência de endereços alocados dinamicamente. A seguir é descrita está técnica por meio de código:

```
int i;
float **mat;
/* matriz representada por um de ponteiro para ponteiros */
...
mat = (float**) malloc(m*sizeof(float*));
for (i=0; i<m; i++)
        mat[i] = (float*) malloc(n*sizeof(float));
```

A grande vantagem desta estratégia é que o acesso aos elementos é feito da mesma forma que quando temos uma matriz criada estaticamente, pois, se **mat** representa uma matriz alocada segundo esta estratégia, **mat[i]** representa o ponteiro para o primeiro elemento da linha **i**, e, conseqüentemente, **mat[i][j]** acessa o elemento da coluna **j** da linha **i**.