Alocação dinâmica de memória

Jander Moreira

1 Primeiras palavras

Na solução de problemas por meio algoritmos ou programas, é comum surgir a necessidade de manter todo o conjunto de dados a ser processado na memória. Esse é o caso, por exemplo, de se ter um conjunto de temperaturas, calcular sua média e contar quantos valores de temperatura estão abaixo ou acima dessa média calculada. Não é possível obter essas informações sem que todas as temperaturas individuais sejam mantidas em memória, pois cada valor é usado para calcular a média e, depois, deve ser consultado novamente para a contagem solicitada.

Uma decorrência desse tipo de solução é a necessidade de se manter simultaneamente todos os dados e, para isso, é preciso memória. O recurso padrão para guardar coleções de dados na memória é o emprego de arranjos (usualmente vetores).

Surge, então, o grande dilema para quem escreve a solução: qual o tamanho do vetor que deve ser definido? Em geral, existe uma estimativa da quantidade máxima de dados, o que permite estimar com certa precisão um tamanho de vetor no qual caibam os valores em questão. Ainda outras questões surgem com a definição de um tamanho fixo para o vetor de dados. A primeira é que, se houver mais dados que o tamanho definido para o vetor, não haverá espaço e a execução se torna inviável. O segundo ponto é a preocupação inversa: se há poucos dados, a quantidade de memória reservada é maior, o que se caracteriza como desperdício.

Na situação ideal, sabendo-se quantos itens individuais devem ser mantidos, deveria ser reservada memória para manter apenas essa quantidade desejada. Se há 15 temperaturas, então o vetor deveria ter 15 posições; havendo 300, ter um vetor com esse número de posições. Essa alternativa de se reservar (ou **alocar)** memória durante a execução do programa, e não durante a escrita do código, leva ao que chama **alocação dinâmica de memória**.

Neste texto, a seção 2 discorre sobre como o programa dispõe de memória para o armazenamento de dados durante sua execução, enquanto a seção 3 apresenta os conceitos da alocação dinâmica de memória, usando a linguagem C como base. A seção 4 apresenta, finalmente as considerações finais sobre os assuntos tratados.

2 Reservando espaço para dados

O uso da memória para guardar dados já é conhecimento comum. Quando se declara uma variável, uma quantidade de bytes suficiente para guardar o dado é reservada na memória. Há variáveis que ocupam mais e outras que ocupam menos bytes. Em C, por exemplo, variáveis do tipo **int** ocupam, tradicionalmente, 4 bytes, enquanto variáveis do tipo **char** ocupam apenas 1 byte. Registos, por sua vez, ocupam espaço correspondente à soma dos tamanhos de cada campo especificado.

Nos programas em C[a], **variáveis globais** são variáveis de **alocação estática**. Esse tipo de variável existe durante toda a execução e, para cada uma delas, é reservada uma área de memória fixa logo ao se iniciar a execução do programa.

Dentro das funções podem existir tanto variáveis locais quanto ser usados parâmetros. Nesse caso, somente quando a função é executada é que o espaço para estes itens se torna necessário. Não é necessário, por exemplo, alocar o espaço para uma variável local se a função nunca for executada. Essa alocação de espaço destinado a **variáveis locais e parâmetros** é feita de forma **dinâmica**, ou seja, conforme seja necessária. A função **main** também obedece a essas regras. A área da memória na qual é feita a alocação de espaço para quaisquer variáveis locais e para parâmetros é chamada de **pilha**, ou **stack**.

Uma parte do espaço da memória reservado para a pilha é usado para os dados locais logo ao se iniciar a execução de uma função. Quando a execução da função é terminada, o espaço na pilha é liberado para ser usado por outra função. É interessante observar que, quando há chamada de uma função dentro de outra, o consumo da pilha é crescente. Funções recursivas, por exemplo, ocupam uma parte da pilha a cada chamada.

Existe, porém, a situação em que o programador não quer que essa alocação, estática ou dinâmica, seja feita de forma automática. Ele pode gerenciar a reserva de espaço para dados usando comandos explícitos, indicando quando cada porção de memória que ele precisa deve ser alocada e quando ela pode ser liberada. Esse tipo de **alocação solicitada explicitamente** pelo programador é também uma **alocação dinâmica**.

Para não conflitar com a pilha, as alocações solicitadas explicitamente são feitas em outra área da memória, denominada normalmente de **monte** (ou **heap**). Todas essas alocações e desalocações são de única e exclusiva responsabilidade do programador e cabe a ele gerenciar quando devem ser feitas as alocações e também quando devem ser liberadas.

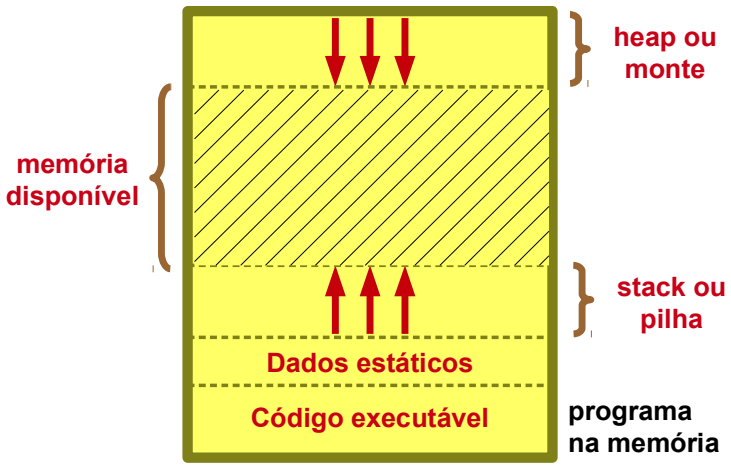

Figura 1. Ilustração do uso do espaço de memória destinado para a execução de um programa.

Na Figura 1 pode ser observada uma representação de como a memória dedicada pelo sistema operacional para a execução de um programa é usada. Uma parte da

(a) Embora a discussão aqui seja focada na linguagem C, a maioria das linguagens procedurais compiladas seguem os mesmos princípios de uso de memória.

memória é usada pelo código do programa. Existe uma área de tamanho fixo destinada para todas as variáveis globais declaradas. A pilha é uma área dinâmica, que aumenta à medida que funções chamam funções, criando espaço para os parâmetros e variáveis locais. No monte são feitas as alocações explícitas. O espaço entre o monte e a pilha é a memória livre disponível, que pode ser usada tanto para um fim quanto para outro. Quando há o encontro da pilha com o monte, esgota-se a memória disponível, de forma que não se pode haver mais chamadas de função e também as solicitações de alocação explícitas falham.

3 Alocação dinâmica de memória

Para se entender o mecanismo de alocação dinâmica, as descrições feitas neste texto serão baseadas na linguagem C. Os conceitos, porém, são mais amplos do que isso e valem para, basicamente, qualquer linguagem que utilize alocação e ponteiros.

Se uma variável é declarada explicitamente, uma área de memória é reservada automaticamente para ela. Por exemplo, considere a declaração:

```
double d;
```

Uma variável do tipo **double** em C ocupa 8 bytes na memória. Durante a escrita do código, é possível usar essa área de memória simplesmente referindo-se a ela pelo nome da variável. A atribuição seguinte armazena nos 64 bits reservados para a variável uma representação do valor atribuído:

```
d = 12.3;
```

O conhecimento da localização dessa área de memória não é necessária para que seja usada, uma vez que o uso do identificador da variável já dá subsídios suficientes para que, durante a execução, o programa saiba onde o dado deve ser armazenado. E isso é válido tanto para as variáveis globais (estáticas) quanto para as locais e parâmetros (dinâmicas).

Uma vantagem dessa abstração é que saber o tamanho da variável e onde ela está na memória é perfeitamente dispensável em situações usuais.

Por outro lado, quando o programador deseja fazer a alocação de dados explicitamente, ele tem que manter controle dessa alocação e não há como abstrair o local dessa área de dados. Nessa nova forma de usar a memória, saber onde a memória foi alocada é essencial para conseguir usá-la.

Todo o controle da memória alocada dinamicamente pelo programador é feita registrando-se o endereço dessa área, o que é feito por meio de ponteiros. Como ponteiros são variáveis que guardam endereços de memória, são eles próprios que devem ser usado para guardar a localização da memória alocada. Na prática, é guardado o endereço do primeiro byte dessa área, apenas. Cabe a programador, que sabe quanto foi alocado, cuidar para que não seja usada memória fora dessa área.

Na linguagem C, os protótipos para as funções de alocação e desalocação de memória estão definidos no arquivo de cabeçalho **malloc.h** (***m**emory **alloc**ation*). Assim, é preciso acrescentar, nos programas em C, a inclusão desse arquivo.

```
#include <malloc.h>
```

3.1 Exemplos simples

Os conceitos de alocação de memória serão tratados por dois exemplos. Ambos são apenas ilustrativos e não possuem uma utilidade prática relevante. Essa abordagem, porém ajuda com a compreensão dos conceitos.

O primeiro exemplo, dado no Programa 1, mostra como reservar espaço explicitamente para guardar um valor inteiro.

Em primeiro lugar, observa-se a inclusão do arquivo **malloc.h** na linha 6. Essa inclusão permite o uso adequado das funções de alocação e desalocação de memória.

A variável **p** é a que será usada para guardar o endereço da memória alocada para guardar um valor inteiro. Essa variável é um ponteiro para inteiro e está declarada na linha 9.

A alocação é feita na linha 12, usando-se a função **malloc()**. Essa função é uma chamada do sistema e é gerenciada pelo sistema operacional. Ela possui um único parâmetro, que é a quantidade de bytes que deve ser reservada no monte para esta solicitação em particular. O sistema verifica se existe um pedaço **contínuo** de memória no monte correspondente ao número de bytes. Se houver, esse espaço é reservado e futuras chamadas a **malloc()** não considerarão tais bytes como disponíveis. A função retorna, finalmente, o endereço do primeiro byte da área alocada.

Caso a função **malloc()** não seja bem sucedida na alocação, ela retorna um ponteiro nulo, ou seja, um ponteiro para a posição zero da memória. Como boa prática de

Programa 1

```
1         /*
2              Exemplo de alocacao para armazenar um valor inteiro
3         */
4
5         #include <stdio.h>
6         #include <malloc.h>
7
8         int main(){
9              int *p; // ponteiro para o inteiro alocado
10
11             // allocacao
12             p = malloc(sizeof(int));
13             if (!p)
14                   printf("Erro na alocacao\n");
15             else{
16                   // uso da area alocada
17                   *p = 10;
18                   printf("valor = %d\n", *p);
19
20                   printf("digite um valor inteiro: ");
21                   scanf("%d", p);
22                   printf("valor digitado = %d\n", *p);
23
24                   // liberacao da memoria alocada
25                   free(p);
26             }
27
28             return 0;
29        }
```

programação, é importante verificar se uma tentativa de alocação foi bem sucedida. Esse tratamento é feito no comando **if** da linha 13. No caso, a expressão **!p** é verdadeira se **p** for igual a zero, situação em que apenas a mensagem de erro é apresentada na tela. A mesma verificação poderia ser feita de formas diferentes, porém equivalentes:

```
if(p == 0)
```

ou

```
if(p == NULL)
```

sendo que **NULL** é um valor simbólico igual a zero.

Deve-se observar que o operador **sizeof** foi utilizado na linha 12 para indicar a quantidade de bytes necessária. Mesmo que o programador saiba que um inteiro ocupa 4 bytes, o uso de **sizeof** é importante, pois torna o código imune a mudanças nas representações[a].

Após a execução do comando da linha 12, portanto, temos um espaço para um inteiro reservado no monte e o endereço desse espaço guardado na variável **p**.

De forma similar ao uso de ponteiros como recurso para passagem de parâmetros por referência, o ponteiro **p** pode ser usado para manipular o conteúdo da área de memória para a qual ele aponta. Na linha 17, é atribuído o valor 10 para essa área de memória. Como **p** é do tipo **int***, então ***p** usa os bits da área apontada para guardar um valor do tipo **int**. Na linha 18 o valor armazenado é apresentado na tela.

As linhas de 20 a 22 mostram como fazer a leitura de um inteiro e usar a área alocada para armazenamento. Em particular, é importante lembrar que o comando **scanf()** usa como argumento o endereço de uma variável e, nesse exemplo, o endereço passado é o da área alocada, que é o que está armazenado na variável **p**. Assim, dispensa-se o uso do **&** na leitura.

Quando a memória alocada não for mais necessária, então ela pode ser desalocada. Isso significa, apenas, que a memória alocada não está mais reservada e que, em outras chamadas a **malloc()**, ela poderá ser usada. A função **free()** é usada para fazer a desalocação. Ela tem como argumento o endereço da área alocada, o qual, no programa exemplificado, está guardado na variável **p**. A quantidade de bytes liberados é feita sempre de forma total, com o controle do número de bytes sendo automático. Assim, não é possível[b] alocar 10 bytes e liberar apenas 5; todos os bytes alocados são sempre liberados.

O segundo exemplo é similar ao anterior, porém são usados bytes suficientes na alocação para armazenar uma quantidade dados maior. O Programa 2 usa a alocação dinâmica para reservar espaço para um registro de dados, definido na forma **struct**.

A declaração do tipo se inicia na linha 9 do Programa 2, definindo um registro para armazenar dados, incluindo nome, telefone e a data de nascimento.

Um ponteiro, chamado **amigo**, é a única variável necessária e ela guarda, como todo ponteiro, um endereço de memória.

Na linha 19 é feita a chamada para **malloc()**, a qual pode retornar o ponteiro nulo, em caso de insucesso, ou o endereço da área alocada, que corresponde a um bloco de

(a) A título de exemplo, há alguns anos o tipo **int** correspondia a uma variável de 2 bytes; atualmente são usados 4 bytes. Compilar o programa usando **sizeof** garante que o número correto seja sempre usado.
(b) A função **realloc()** faz esse tipo de manipulação, mas não será abordada neste texto.

Programa 2

```
1.        /*
2.             Exemplo de alocacao para armazenar um registro
3.        */
4.
5.        #include <stdio.h>
6.        #include <string.h>
7.        #include <malloc.h>
8.
9.        typedef struct{
10.                char nome[50];     // nome
11.                char telefone[15]; // numero do telefone
12.                int dia, mes, ano; // data de nascimento
13.       } tAgenda;
14.
15.       int main(){
16.            tAgenda *amigo;
17.
18.            // allocacao
19.            amigo = malloc(sizeof(tAgenda));
20.            if (!amigo)
21.                 printf("Erro na alocacao\n");
22.            else{
23.                 // uso da area alocada
24.                 strcpy(amigo->nome, "Fulano");
25.                 strcpy(amigo->telefone, "(99) 1122-3344");
26.                 amigo->dia = 1;
27.                 amigo->mes = 10;
28.                 amigo->ano = 1990;
29.
30.                 printf("Dados:\n");
31.                 printf("%s atende no numero %s\n",
32.                      amigo->nome, amigo->telefone);
33.                 printf("Aniversario %d/%d, desde %d\n\n",
34.                      amigo->dia, amigo->mes, amigo->ano);
35.
36.                 // liberacao da memoria alocada
37.                 free(amigo);
38.            }
39.
40.            return 0;
41.       }
```

memória contínuo com **sizeof(tAgenda)** bytes[a]. Caso essa alocação seja efetuada, **amigo** recebe o endereço do espaço alocado.

As linhas de 24 a 30 ilustram uma manipulação a área alocada como um registro. Como **amigo** é um ponteiro para o registro, o acesso aos campos é feito usando-se a notação **->**. A linha 25, por exemplo, usa a função **strcpy()** para atribuir o texto correspondente ao número do telefone ao campo **telefone**.

Na linha 37 encontra-se o comando que indica a liberação do bloco de memória.

A utilidade de se alocar espaço para uma única variável é bastante limitada e, em geral, usar usar uma variável comum é mais adequado.

(a) Reforça-se aqui o uso de **sizeof**. Usando-se o compilador **gcc**, descobriu-se que o registro usa exatamente 80 bytes.

3.2 Alocação de vetores

Uma aplicação interessante para a alocação dinâmica é reservar espaço no monte para um conjunto de dados de mesmo tipo. A função **malloc()** permite especificar qualquer quantidade de bytes e, assim, no lugar de alocar espaço para um único valor inteiro, é possível alocar espaço para 300 valores, ou então 10.000, 300.000 etc.

A alocação de espaço para 100 valores inteiros pode ser especificada por:

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

Este comando, reserva no monte espaço para 100 valores inteiros, em um único bloco contínuo de memória.

O Programa 3 mostra a utilização de alocação de conjunto de valores.

Programa 3

```
1.  /*
2.      Alocacao de vetor de reais
3.  */
4.
5.  #include <stdio.h>
6.  #include <malloc.h>
7.
8.  int main(){
9.      int i;
10.     int elementos; // numero de itens
11.     float *vetor;  // ponteiro para o inicio do vetor
12.     float *p;      // ponteiro para percorrer o vetor
13.
14.     // obtencao do numero de itens
15.     printf("Numero de itens: ");
16.     scanf("%d", &elementos);
17.
18.     // alocacao
19.     vetor = malloc(elementos * sizeof(float));
20.     if(!vetor)
21.         printf("Erro de alocacao.\n");
22.     else{
23.         // leitura dos dados
24.         p = vetor;  // ambos apontam para o inicio
25.         for(i = 0; i < elementos; i++){
26.             printf("item[%d]: ", i);
27.             scanf("%f", p);
28.             p++;
29.         }
30.
31.         // escrita
32.         p = vetor;  // ambos apontam para o inicio
33.         for(i = 0; i < elementos; i++)
34.             printf("%d) %.2f\n", i, *p++);
35.
36.         // desalocacao
37.         free(vetor); // liberacao de todo o bloco
38.     }
39.
40.     return 0;
41. }
```

A alocação é feita na linha 19, na qual se destaca que o valor lido para a variável **elementos** é usado para dimensionar o número bytes que se deseja reservar. A verificação de sucesso na alocação é feita na linha 20.

Na linha 24, a variável **p**, que é ponteiro para **float**, recebe cópia do endereço em **vetor**. Como ambas as variáveis contém o endereço do primeiro byte do bloco alocado, ambas apontam para o mesmo ponto.

Na repetição iniciada na linha 25, o ponteiro **p** é usado para percorrer cada **float** a partir do endereço inicial, fazendo a leitura. Na prática, o primeiro valor do conjunto é lido e, quando se usa **p++**, passa-se para o próximo. O uso da **aritmética de ponteiros** é o recurso usado se ter acesso a cada valor individual. Ao final da repetição, **p** aponta para o primeiro byte **fora** da área alocada, pois depois da leitura do último valor, ainda é feito um incremento do apontador.

A repetição seguinte usa o mesmo recurso para escrever os valores lidos. Na escrita, um destaque pode ser dado ao uso simultâneo dos operadores ***** e **++** junto à variável **p**. Como o operador **++** é posterior ao operando[a], **p++** resulta no endereço do ponteiro atual e incrementa para o próximo; ao se aplicar ***** para acesso ao conteúdo apontado, isso é feito sobre o valor resultante do **p++**, que é a posição atual. O efeito obtido é a escrita do valor atual e o incremento do ponteiro para a próxima posição.

Em seguida é feita a desalocação do bloco completo, na linha 37.

Um dos pontos interessantes, próprio da linguagem C, é que a notação de acesso a vetores com uso do índice é válida, mesmo quando a variável é um ponteiro. O Programa 4 mostra uma versão do Programa 3 que não usa o ponteiro auxiliar. Nas linhas 25 e 30 é possível notar que o ponteiro **vetor** é usado como um vetor regular.

3.3 Alocação de matrizes

A alocação de matrizes segue uma ideia similar à alocação de vetores, alocando todo o bloco de memória necessário de uma única vez, de forma a se ter um segmento de memória contínuo e se poder usar a aritmética de ponteiros para percorrer os dados. Além do bloco de dados, é interessante também se ter uma estrutura auxiliar, que dá acesso às linhas. Assim, a especificação de um item individual da matriz pode ser feito indicando sua linha e coluna, o que é natural para acesso aos dados de uma matriz.

O Programa 5 ilustra essa estruturação. Na linha 20 é alocado um vetor de ponteiros, existindo um ponteiro para cada linha. A função desse vetor é indicar o início de cada linha na área de dados a ser alocada. É importante notar que cada posição guarda uma variável do tipo **float***. Desse modo, depois dos ajustes dos ponteiros, **mat[0]** apontará para a área de dados da primeira linha, da mesma forma que cada **mat[i]** apontará para a área de dados da linha **i**.

Na linha 22 é feita a alocação destinada aos dados da matriz, o que corresponde a um total de **linhas*colunas** elementos do tipo **float**. O endereço dessa área, retornado pela função **malloc()**, é armazenado em **mat[0]**, de forma que essa posição já aponta para o início da matriz, o que coincide com a linha 0.

Havendo sucesso na alocação e sabendo-se que **mat[0]** aponta para o início da linha 0, é preciso ajustar os ponteiros das demais linhas. A repetição da linha 27 faz essa tarefa. Para cada linha restante da matriz, ajusta-se seu ponteiro para o início da área correspondente a seu “sub-bloco” de dados. A Figura 2 mostra estes ajustes para uma matriz de dimensão 3X4.

(a) Uma revisão sobre as diferenças entre **i++** e **++i** pode ajudar a entender os resultados obtidos.

Programa 4

```
1.        /*
2.             Alocacao de vetor de reais
3.        */
4.
5.        #include <stdio.h>
6.        #include <malloc.h>
7.
8.        int main(){
9.             int i;
10.            int elementos; // numero de itens
11.            float *vetor;  // ponteiro para o inicio do vetor
12.
13.            // obtencao do numero de itens
14.            printf("Numero de itens: ");
15.            scanf("%d", &elementos);
16.
17.            // alocacao
18.            vetor = malloc(elementos * sizeof(float));
19.            if(!vetor)
20.                 printf("Erro de alocacao.\n");
21.            else{
22.                 // leitura dos dados
23.                 for(i = 0; i < elementos; i++){
24.                      printf("item[%d]: ", i);
25.                      scanf("%f", &vetor[i]);
26.                 }
27.
28.                 // escrita
29.                 for(i = 0; i < elementos; i++)
30.                      printf("%d) %.2f\n", i, vetor[i]);
31.
32.                 // desalocacao
33.                 free(vetor); // liberacao de todo o bloco
34.            }
35.
36.            return 0;
37.       }
```

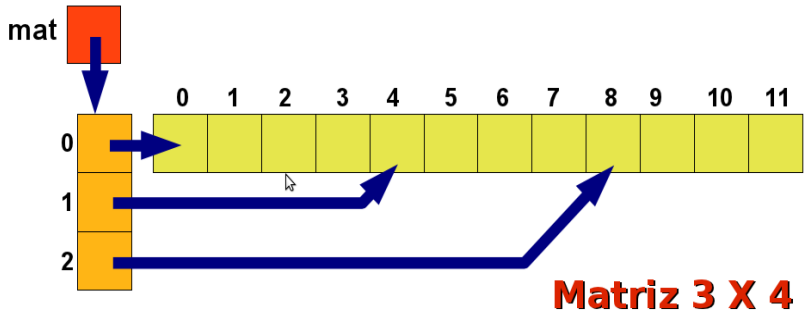

*Figura 2. Esquema de organização para acesso ao bloco de dados alocado para uma matriz 3X4. Em amarelo está representado o bloco de dados (tipo **float**); em laranja, o vetor de ponteiros para as linhas (tipo **float***); em vermelho, o ponteiro inicial para toda a estrutura (tipo **float****).

Programa 5

```
1. /*
2.       Alocacao de dados para uma matriz de reais
3. */
4.
5. #include <stdio.h>
6. #include <malloc.h>
7.
8. int main(){
9.       int i, j;
10.           int linhas, colunas;  // dimensoes da matriz
11.           float **mat;  // ponteiro para a matriz
12.
13.           printf("Numero de linhas: ");
14.           scanf("%d", &linhas);
15.           printf("Numero de colunas: ");
16.           scanf("%d", &colunas);
17.
18.           // alocacao
19.           // 1) vetor de ponteiros
20.           mat = malloc(linhas * sizeof(float *));
21.           // 2) bloco para os dados
22.           mat[0] = malloc(linhas*colunas * sizeof(float));
23.           if(!mat || !mat[0])
24.                 printf("Erro na alocacao.\n");
25.           else{
26.                 // ajuste dos demais ponteiros
27.                 for(i = 1; i < linhas; i++)
28.                       mat[i] = &mat[0][i * colunas];
29.
30.                 // uso da matriz
31.                 for(i = 0; i < linhas; i++)
32.                       for(j = 0; j < colunas; j++)
33.                             mat[i][j] = i + j;
34.
35.                 for(i = 0; i < linhas; i++){
36.                       for(j = 0; j < colunas; j++)
37.                             printf("%6.1f", mat[i][j]);
38.                       printf("\n");
39.                 }
40.
41.                 // desalocacao
42.                 free(mat[0]); // bloco de dados
43.                 free(mat);    // vetor das linhas
44.           }
45.
46.           return 0;
47.     }
```

As linhas 33 e 37 mostram como a notação usual de matriz pode ser empregada pela variável **mat**. Embora **mat** seja um ponteiro duplo para **float**, é viável usar os colchetes para indicar cada posição da matriz. Da mesma forma, cada linha da matriz, que é um vetor unidimensional, pode ser especificada usando-se apenas o primeiro índice. Assim, **mat[i]** é o vetor da linha **i**, como usado nas linhas 22 e 28, por exemplo.

A desalocação dos blocos de dados obtidos dinamicamente é feita nas linhas 42 e 43. A ordem desses dois comandos é importante, pois se o vetor de ponteiros for liberado antes, não se tem mais acesso a **mat[0]** e, portanto, perde-se o ponteiro para o bloco de

dados.

3.4 Outras aplicações de alocação dinâmica de memória

A obtenção de espaço de memória para os dados conforme a necessidade é um recurso essencial para a programação. Ele dá liberdade para que o programa não fique restrito a limites estabelecidos na hora da compilação, como o tamanho máximo de um vetor.

Existem, ainda, inúmeras outras aplicações para alocação dinâmica, as quais são abordadas fora do contexto desta disciplina.

A título de exemplo, listas encadeadas formam uma classe de estrutura de dados de aplicação versátil. Imagine-se, como exemplo, que cada item que se deseja ter seja alocado individualmente. Para se ter acesso a esse item, é preciso manter um ponteiro para ele. Não se deseja, porém, um conjunto enorme de ponteiros se houver um conjunto enorme de itens. As listas encadeadas são uma estrutura que exige apenas um ponteiro, que indica o primeiro item da lista. Para acesso aos demais elementos, coloca-se em cada item um ponteiro para o próximo da sequência.

A Figura 3 apresenta um esquema para visualizar este encadeamento. Cada elemento verde indica uma unidade alocada, que é composta por uma informação (indicada pelas letras) e um ponteiro. A partir do ponteiro **inicio** se tem acesso ao primeiro item e, também, ao ponteiro para o segundo; a partir de cada item se tem acesso ao próximo e, em consequência, a todos. Na ilustração, o último elemento possui um ponteiro nulo (igual a NULL ou zero), que é usado como critério para saber que é o último item da lista.

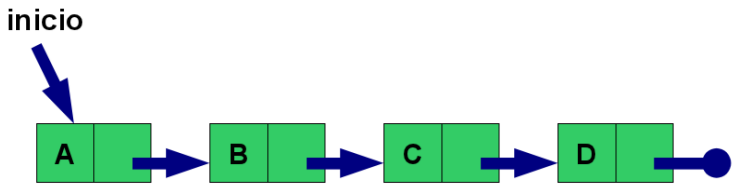

Figura 3. Ilustração de uma lista encadeada.

Cada item da lista encadeada pode ser alocado quando necessário e, para se ter acesso a ele, basta que haja os ajustes de ponteiros para que ele pertença ao encadeamento.

4 Considerações finais

A alocação dinâmica explícita de elementos é um artifício de grande importância para a programação. Não é, entretanto, um conceito fácil e o entendimento de seu funcionamento depende de compreender desde o conceito de ponteiros e endereços de memória até o uso de aritmética de ponteiros.

Alguns pontos relativos a alocação ainda devem ser destacados. A liberação de memória que foi alocada por meio do **malloc()** ocorre automaticamente quando o programa termina, pois toda a área de memória do programa é retomada pelo sistema operacional. Isso, porém, não exime o programador de fazer essa desalocação explicitamente. O esquecimento do código no programa que faz a desalocação é considerado como desleixo do programador, tornando o código escrito pobre e sujeito a erros futuros, no caso de modificações do programa fonte feitas posteriormente. Para o

bom programador, o uso correto do **free()** é, assim, obrigatório.

Um cuidado também deve ser tomado para que se possa liberar a memória. Como, para a liberação é necessário passar para **free()** o endereço inicial do bloco alocado, esse valor não pode ser perdido. Por exemplo, no Programa 3, foi usado um ponteiro auxiliar, **p**, para percorrer o vetor. Caso o próprio ponteiro **vetor** fosse usado, seria perdida a localização do início do bloco, de forma que nem seria possível retornar ao início para escrever os dados nem liberar a memória alocada, uma vez que não se tem mais o endereço inicial.