

# Capítulo 13

## Vetores dinâmicos

Para representar um conjunto de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores reais. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000  
float vet[MAX];
```

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo `vet`, após a declaração ilustrada, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, a constante `vet` representa o endereço da memória em que o primeiro elemento do vetor está armazenado. De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação: `vet[i]`. Dizemos que o vetor é uma estrutura que possibilita o acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente, o que traz grandes vantagens no uso de vetores frente a outras estruturas de dados dinâmicas, como *listas*, que iremos discutir nos próximos capítulos.

No entanto, é necessário dimensionar o número máximo de elementos. Este problema é aliviado com o uso de alocação dinâmica. A dimensão do vetor pode ser determinada em tempo de execução:

```
float* vet = (float*) malloc(n * sizeof(float));
```

Neste fragmento de código, `n` representa uma variável com a dimensão do vetor, determinada em tempo de execução (podemos, por exemplo, ler o valor de `n` de um arquivo). Após a alocação dinâmica, acessamos os elementos do vetor da mesma forma que os elementos de vetores alocados estaticamente. Com alocação dinâmica, declaramos uma variável do tipo ponteiro que, posteriormente, recebe o valor do endereço do primeiro elemento do vetor. Neste caso, a área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (através da função `free`). Portanto, mesmo que um vetor seja criado dinamicamente dentro da função, podemos acessá-lo depois da função

ser finalizada, pois a área de memória ocupada por ele permanece válida, isto é, o vetor não está alocado na pilha de execução.

No entanto, a necessidade de dimensionar o número máximo de elementos persiste. Mesmo em tempo de execução, essa dimensão pode ser desconhecida. Imagine uma aplicação que precisa ler e armazenar informações presentes em um arquivo, sem que se tenha anotado o número de informações existentes. Processar o arquivo duas vezes, uma para determinar o número de informações e outra para efetivamente ler as informações (e armazenar no vetor alocado), em geral, é muito ineficiente. Em outras aplicações, não temos nem alternativas para determinar o número máximo de elementos que serão armazenados. Estimar um valor máximo pode não ser possível. A solução, para não abrir mão do uso de vetores, é trabalhar com realocação de memória.

## 13.1 Realocação de memória

A linguagem C oferece um mecanismo para reallocar 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 redimensionar um vetor, preservando os elementos existentes, que permanecem inalterados após a realocação.

O fragmento de código a seguir ilustra o uso da função `realloc`. Neste trecho, a variável `m` representa a nova dimensão do vetor:

```
...
vet = (float*) realloc(vet, m*sizeof(float));
...
```

A partir desta realocação, se bem-sucedida, `vet` aponta para uma área de memória contígua suficiente para armazenar `m` valores reais. A função `realloc` recebe dois parâmetros: o endereço de memória do vetor a ser redimensionado e o novo número, em bytes, a ser reservado para o vetor.

Conceptualmente, podemos considerar que a realocação de memória de um vetor é equivalente a uma alocação, seguida de uma cópia, seguida de uma liberação (no trecho de código a seguir, `n` representa a dimensão do vetor original):

```
float* temp = (float*)malloc(m*sizeof(float)); /* aloca novo vetor */
int min = m < n ? m : n;                      /* menor valor entre m e n */
memcpy(temp, vet, min*sizeof(float));           /* copia elementos */
free(vet);                                     /* libera vetor original */
vet = temp;                                     /* atribui novo endereço à variável original */
```

Note, portanto, que uma realocação de vetor, embora possível e muito útil, tem um alto custo computacional. A execução da função `realloc` pode ser, na verdade, um pouco mais eficiente, pois o sistema tenta preservar o vetor no espaço de memória já reservado para ele. Se a nova dimensão é maior que a original, o sistema verifica se um espaço de memória adicional à frente do vetor está livre; se sim, ele é incorporado ao espaço reservado para o vetor, e a cópia do conteúdo não é necessária. Neste caso, o endereço de memória retornado pela função `realloc` é igual ao endereço original passado como parâmetro. No entanto, nem sempre o espaço à frente do vetor está disponível,

sendo necessárias as operações de cópia e liberação. Mesmo quando a nova dimensão é menor que a original, o sistema pode optar por reposicionar o vetor para otimizar o uso da memória (por exemplo, para reduzir o número de pequenos espaços fragmentados de memória não utilizados).

## 13.2 Vetor dinâmico

Nesta seção, vamos discutir a implementação de um tipo abstrato de dados que represente um vetor dinâmico. Neste contexto, um vetor dinâmico é um vetor cuja dimensão é ajustada dinamicamente conforme novos elementos vão sendo inseridos no vetor. A estratégia comumente usada para implementar um vetor dinâmico consiste em alocar inicialmente o vetor com uma dimensão pequena e redimensionar o vetor quando o espaço se esgotar. Por exemplo, podemos usar a estratégia de dobrar a dimensão do vetor sempre que sua capacidade se esgotar. Com isso, minimizamos o número de relocações.

A implementação de um TAD representando um vetor dinâmico é muito útil em diversas aplicações. Vamos considerar a implementação das seguintes operações:

- **cria**: operação para criar um novo vetor, inicialmente vazio.
- **insere**: operação para inserir um novo elemento no final do vetor, com realocação automática, se necessário.
- **tam**: operação que retorna o número de elementos efetivamente armazenados.
- **acessa**: operação para acessar determinado elemento do vetor.
- **libera**: operação para liberar a memória ocupada pelo vetor.

Vamos considerar inicialmente um vetor de valores reais (`float`). A interface do nosso TAD pode ser dada por:

```
typedef struct vetordin VetorDin;
VetorDin* vd_cria (void);
void vd_insere (VetorDin* vd, float x);
int vd_tam (VetorDin* vd);
float vd_acessa (VetorDin* vd, int i);
void vd_libera (VetorDin* vd);
```

Vamos então discutir passo a passo a implementação deste TAD. O tipo que representa o vetor tem que armazenar, além do vetor propriamente dito, dois inteiros: a dimensão atual do vetor e o número efetivo de elementos armazenados. Portanto, o tipo do vetor pode ser:

```
struct vetordin {
    int n;          /* número de elementos armazenados */
    int nmax;       /* dimensão do vetor */
    float* v;        /* vetor dos elementos */
};
```

A função que cria o vetor, dimensiona o vetor com um número pequeno de elementos:

```
VetorDin* vd_cria (void)
{
    VetorDin* vd = (VetorDin*) malloc(sizeof(VetorDin));
    vd->n = 0;
    vd->nmax = 4;
    vd->v = (float*) malloc(vd->nmax*sizeof(float));
    return vd;
}
```

A rigor, deveríamos verificar se as alocações foram bem-sucedidas (omitimos apenas para o código ficar sucinto).

Outra opção também muito empregada é deixar a aplicação cliente escolher a dimensão inicial do vetor, a fim de minimizar a inicialização com vetores pequenos que sabidamente irão crescer. Assim, o cliente pode estimar uma dimensão, sem se preocupar com um limite máximo rígido. Neste caso, a função que cria teria que receber essa dimensão inicial como parâmetro.

A função que insere um elemento no final do vetor precisa verificar a necessidade de realocação. Vamos usar uma função interna auxiliar para a realocação:

```
static void realoca (VetorDin* vd)
{
    vd->nmax *= 2; /* dobra a dimensão */
    vd->v = (float*) realloc(vd->v, vd->nmax*sizeof(float));
}

void vd_insere (VetorDin* vd, float x)
{
    if (vd->n == vd->nmax) /* verifica se capacidade se esgotou */
        realoca(vd);
    vd->v[vd->n++] = x; /* insere elemento no final e incrementa n */
}
```

Novamente, deveríamos verificar se a realocação foi bem-sucedida.

As funções de acesso são simples. No acesso a um elemento, pode ser útil verificar a validade do índice; assim, evitamos acesso indevido à memória. Para exemplificar, faremos isso usando a macro `assert` da biblioteca `assert.h`. Se a expressão passada para a macro resultar em falso, uma mensagem é exibida na tela e o programa é abortado. Naturalmente, este teste acrescenta um custo. Por ser uma macro, a assertiva pode ser ligada ou desligada. Se o compilador for executado com a opção `-DNDEBUG`, a macro é desligada (é como se a linha da assertiva não existisse no código). Em geral, deixamos a macro ligada durante o desenvolvimento e desligamos para gerar a versão final do software já testado.

```
int vd_tam (VetorDin* vd)
{
    return vd->n;
}
```

```

float vd_acessa (VetorDin* vd, int i)
{
    assert(i>=0 && i<vd->n);
    return vd->v[i];
}

```

Por fim, a função que libera a memória alocada:

```

void vd_libera (VetorDin* vd)
{
    free(vd->v); /* libera vetor de float */
    free(vd); /* libera estrutura */
}

```

Em algumas aplicações, a estratégia de dobrar o espaço a cada realocação pode significar um uso excessivo de memória. Outros fatores de incremento também funcionam. Por exemplo, podemos aumentar a capacidade do vetor em 25% a cada realocação, ou podemos acrescentar determinado montante de espaço fixo a cada realocação. Usar um fator multiplicador, em geral, reduz o número de realocações (ao preço de um eventual uso excessivo de memória).

Também é válido ajustar para baixo a dimensão do vetor. Por exemplo, se a taxa de ocupação da memória do vetor chegar a 25%, podemos reduzir sua dimensão à metade e ainda ter folga. No entanto, na prática, a estratégia de redução não é muito empregada, pois o vetor não é uma estrutura de dados adequada para retirar elementos. Retirar um elemento do meio do vetor requer deslocar os elementos para preencher o espaço vazio. Na prática, é mais comum o número de elementos do vetor sempre crescer, até que o conjunto de informações esteja todo armazenado.

Para exemplificar o uso de vetor dinâmico, vamos considerar uma aplicação que precisa processar um conjunto de valores armazenados em um arquivo. Como estamos usando vetor dinâmico, não precisamos conhecer o número de valores que serão armazenados. Portanto, o formato do arquivo pode ser simplesmente uma lista de números:

34.32
4.3
23.56
9.33
...

Podemos então escrever uma função que recebe o nome do arquivo como parâmetro e retorna um vetor preenchido com os valores do arquivo:

```

#include "vetordin.h"
#include <stdio.h>
...
VetorDin* le_valores (char* arquivo)
{
    FILE* f = fopen(arquivo, "rt");
    if (f==NULL)
        return NULL;

```

```

    VetorDin* vd = vd_cria();
    float x;
    while (fscanf(f, "%f", &x) == 1)
        vd_insere(vd, x);
    fclose(f);
    return vd;
}

```

Como podemos observar, o uso de vetores dinâmicos traz muita flexibilidade. Como dissemos, essa estrutura é muito útil em diferentes aplicações.

### 13.3 Cadeia de caracteres dinâmica

A função `realloc` é especialmente importante para a implementação de vetores dinâmicos que envolvem grandes quantidades de informações. Contudo, podemos adotar uma estratégia semelhante para implementar cadeias de caracteres dinâmicas. Aqui, o principal objetivo é abstrair o gerenciamento da memória e sempre representar a cadeia com a dimensão necessária.

Uma das maiores fontes de erro em programação em C consiste no mau uso da memória. Em especial, muitos programadores cometem erros na alocação de cadeias de caracteres: tentar armazenar mais caracteres do que a dimensão alocada do vetor. Um tipo abstrato de dados que gerencia automaticamente a memória de cadeias de caracteres pode facilitar o desenvolvimento de programas corretos.

Para exemplificar uma implementação de cadeia dinâmica, vamos considerar as seguintes operações:

- `criavazia`: cria uma cadeia dinâmica inicialmente vazia.
- `criacopia`: cria uma cadeia dinâmica inicializada com uma cadeia fornecida.
- `atribui`: reatribui (redefine) o valor da cadeia dinâmica.
- `concatena`: concatena a cadeia fornecida na cadeia dinâmica.
- `acessa`: retorna o ponteiro da cadeia efetivamente armazenada.
- `libera`: libera memória da cadeia dinâmica.

A interface desse tipo abstrato pode ser:

```

typedef struct strdin StrDin;

StrDin* sd_criavazia (void);
StrDin* sd_criacopia (const char* s);
void sd_atribui (StrDin* sd, const char* s);
void sd_concatena (StrDin* sd, const char* s);
const char* sd_acessa (StrDin* sd);
void sd_libera (StrDin* sd);

```

Nesta implementação, faremos uso do modificador de tipo `const` para ilustrar seu uso. Em especial, note que a função de acesso retorna uma cadeia constante, indicando que a

cadeia não pode ser alterada pelo cliente. Isto é fundamental, pois assim a implementação do tipo abstrato tem controle assegurado do conteúdo da cadeia.

Talvez, a principal operação seja a de atribuição. A possibilidade de reatribuir um valor a uma cadeia de caracteres nos permite, conceitualmente, tratar cadeias de caracteres por valor. Da mesma forma que atribuímos valores numéricos a variáveis, podemos atribuir valores cadeias de caracteres a cadeias dinâmicas. Naturalmente, o custo de gerenciamento de memória existe, mas está escondido dentro do tipo abstrato:

```
StrDin* s = sd_criavazia();
...
sd_atribui(s, "Rio de Janeiro");
...
sd_atribui(s, "Recife");
...
```

Internamente, nosso tipo sempre usa apenas o espaço de memória para armazenar o “valor” corrente da cadeia.

Vamos começar pela implementação das funções de criação. Note que vamos reusar muitas funções da própria interface nesta implementação.

```
StrDin* sd_criavazia (void)
{
    return sd_criacopia("");
}

StrDin* sd_criacopia (const char* s)
{
    StrDin* sd = (StrDin*) malloc(sizeof(StrDin));
    sd->v = NULL;
    sd_atribui(sd,s);
    return sd;
}
```

As funções de atribuição e concatenação reservam o espaço de memória e fazem uso das funções da biblioteca padrão. Vamos criar uma função auxiliar interna para cuidar da alocação e da preocupação de sempre ter o espaço adicional para o caractere '\0':

```
static void realoca (StrDin* sd, int n)
{
    sd->v = (char*) realloc(sd->v,n+1);
}

void sd_atribui (StrDin* sd, const char* s)
{
    realoca(sd, strlen(s));
    strcpy(sd->v,s);
}

void sd_concatena (StrDin* sd, const char* s)
{
    realoca(sd, strlen(sd->v)+strlen(s));
    strcat(sd->v,s);
}
```

Note que podemos passar o valor `NULL` como endereço inicial de um vetor a ser realocado: a função `sd_criacopia` inicializa o ponteiro da cadeia interna como sendo `NULL` e chama a função `sd_atribui` que, por sua vez, faz a realocação da cadeia interna.

As funções de acesso e liberação têm implementações diretas:

```
const char* sd_acessa (StrDin* sd)
{
    return sd->v;
}

void sd_libera (StrDin* sd)
{
    free(sd->v);
    free(sd);
}
```

Para ilustrar o uso da cadeia dinâmica, podemos considerar um exemplo no qual desejamos ler o conteúdo inteiro de um arquivo texto para dentro de uma cadeia de caracteres:

```
StrDin* le_texto (char* arquivo)
{
    FILE* f = fopen(arquivo, "rt");
    if (f == NULL)
        return NULL;

    StrDin* s = sd_criavazia();

    char buffer[120];
    while (fgets(buffer, 120, f) != NULL)
        sd_concatena(s, buffer);

    fclose(f);
    return s;
}
```

Um função `main` para testar essa implementação pode ser:

```
int main (void)
{
    StrDin* s = le_texto("strdin.c");
    printf("%s\n", sd_acessa(s));
    sd_libera(s);
    return 0;
}
```

## Exercícios

1. Considere uma estrutura que armazena dados de um aluno:

```
typedef struct aluno Aluno;
struct aluno {
    char nome[81];      /* nome */
    float p1, p2, p3;   /* notas */
};
```

Pede-se:

- (a) Implemente um tipo abstrato de dados para representar vetores dinâmicos que armazenem dados de aluno. O vetor deve ser de ponteiros para `Aluno` e deve alojar uma estrutura que representa os dados de um aluno apenas para as posições efetivamente usadas pelo vetor.
  - (b) Escreva um programa para testar seu tipo abstrato. Neste programa, deve-se ler um arquivo texto de entrada com o seguinte formato, para cada aluno listado: nome em uma linha e as três notas na linha seguinte. Considere que o número total de alunos listados no arquivo não é informado (isto é, deve-se ler os dados para descobrir o total). Os dados lidos devem ser armazenados no vetor.
2. Altere a implementação da cadeia de caracteres dinâmica apresentada. A fim de minimizar o número de realocações, implemente a seguinte estratégia:
- Se for necessário aumentar a dimensão, isto é, se  $n \geq n_{max}$ , redimensione o vetor para o tamanho  $\max(n, 2n_{max})$ .
  - Se for necessário diminuir a dimensão, caracterizado quando  $n < 0.5n_{max}$ , redimensione o vetor para  $0.5n_{max}$ .
  - Acrescente a função `redimensiona` na interface; esta função permite ao cliente ajustar a dimensão do vetor para apenas o espaço efetivamente usado para representar a cadeia.