INTRODUÇÃO

Este capítulo introduz estruturas de dados com base no conceito de tipo de dados abstrato e apresenta os principais recursos da linguagem C que são usados em sua implementação.

1.1 Abstração

O principal objetivo da *ciência da computação* é resolver problemas por meio da criação de programas de computador.

Quando os primeiros computadores surgiram, os problemas tratados eram relativamente simples. Mas, com o aumento da capacidade dessas máquinas, os problemas tratados e os programas criados se tornaram bem mais complexos.

Uma forma encontrada para vencer essa complexidade foi aplicar o conceito de *abstração*, que possibilita reduzir a quantidade de detalhes considerados pelo programador, em cada etapa da criação de um programa. Ao suprimir detalhes irrelevantes, e ao mesmo tempo enfatizar aqueles relevantes, a abstração pode simplificar bastante a solução de problemas complexos.

1.1.1 Abstração funcional

Abstração funcional permite que um programador use uma *operação*, sem ter que se preocupar com os detalhes de como ela é executada pelo computador.

Por exemplo, para usar a função `sqrt()`, um programador só precisa saber o *que* ela faz; *como* ela faz é irrelevante. Por outro lado, para criar essa função, um programador só precisa saber *como* ela faz; para *que* ela será usada é irrelevante. Assim, em cada etapa da criação de um programa, apenas parte dos detalhes da solução do problema tratado precisa ser considerada pelo programador.

Infelizmente, a abstração funcional é limitada: funções representam *operações abstratas*, mas não *dados abstratos*. Isso é um problema sério, pois a complexidade de um programa depende também da complexidade dos dados que ele manipula.

1.1.2 Abstração de dados

Abstração de dados permite que um programador use um *dado* sem se preocupar com os detalhes de como ele é armazenado e manipulado pelo computador.

Por exemplo, para usar um dado do tipo `float`, um programador só precisa saber *que* operações podem ser feitas com ele; detalhes de *como* ele é armazenado na memória e manipulado pelas operações aritméticas são irrelevantes.

Essa forma de abstração é baseada no conceito de *tipo de dados abstrato*, que consiste essencialmente numa descrição das *interfaces* das operações que um tipo suporta, indicando *parâmetros* e *resultados*. Para quem *usa* um tipo de dados, isso é tudo o que importa. Mas, antes de ser usado, um tipo de dados deve ser *implementado*.

A *implementação* de um tipo de dados abstrato consiste em definir como seus *dados* devem ser representados na memória e como essa representação deve ser manipulada por suas *operações*, para que os resultados esperados sejam obtidos.

1.2 Estruturas de dados

Uma *estrutura de dados* é um tipo de dados abstrato que representa uma coleção de itens inter-relacionados. Como vemos na Figura 1.1, o tipo de relacionamento entre os itens é que define a classe de estrutura de dados que eles compõem:

- Uma coleção de itens em que não há ordem nem repetição é um *conjunto*.
- Uma coleção de itens organizados linearmente é uma *lista*. Cada item em uma lista tem um único predecessor e um único sucessor; exceto o *primeiro* item, que não tem predecessor, e o *último*, que não tem sucessor.
- Uma coleção de itens organizados hierarquicamente é uma *árvore*. Cada item em uma árvore tem um único predecessor e vários sucessores; exceto a *raiz*, que não tem predecessor, e as *folhas*, que não têm sucessores.
- Uma coleção de itens organizados em rede é um *grafo*. Cada item em um grafo pode ter vários predecessores e sucessores.

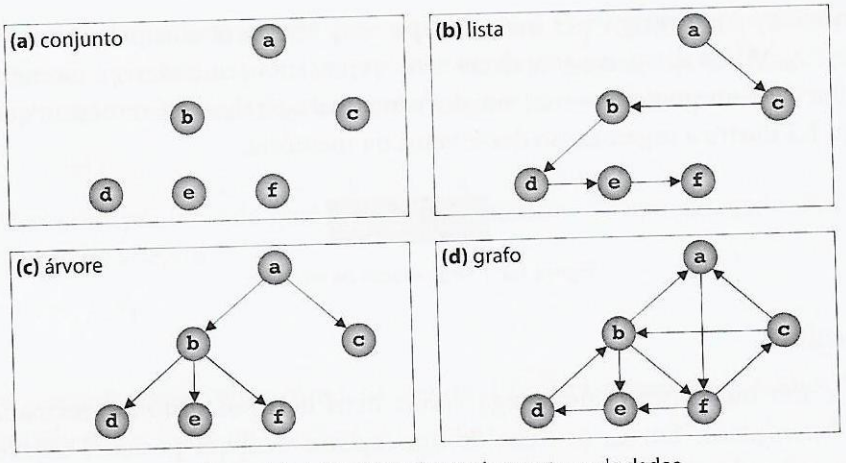

Figura 1.1 | Principais classes de estruturas de dados.

1.2.1 Objetivos

O estudo de estruturas de dados tem dois objetivos principais:

- Implementar estruturas de dados, usando mecanismos de agregação de dados e alocação de memória existentes na linguagem de implementação.
- Mostrar que o uso de uma estrutura de dados adequada pode simplificar a criação de um programa que resolve um tipo de problema específico.

Para atingir esses objetivos, além de implementar as principais estruturas de dados, esse livro também apresenta aplicações práticas desenvolvidas com elas.

1.3 Mecanismos de agregação de dados

A primeira preocupação ao implementar uma estrutura de dados é decidir como agregar seus itens. Para tomar essa decisão, além de conhecer as propriedades dessa estrutura, é preciso conhecer também os *mecanismos de agregação de dados* existentes na linguagem de programação usada. A seguir, são apresentados os principais mecanismos de agregação de dados em linguagem C.

1.3.1 Vetor

Vetor é um mecanismo que agrega vários itens de um *mesmo* tipo, formando uma coleção *homogênea*. Em C, os itens de um vetor ocupam posições adjacentes de memória e são identificados por *índices*, a partir de `0`. Por exemplo, a declaração:

```
int v[3] = {9,6,7};
```

cria um vetor `v` que agrega três itens do tipo `int`, representados pelas variáveis `v[0]`, `v[1]` e `v[2]`. Além disso, o *nome* desse vetor representa seu endereço na memória, de modo que `v` é o mesmo que `&v[0]` ou, de forma mais geral, `v+i` é o mesmo que `&v[i]`. A Figura 1.2 ilustra a organização desse vetor na memória.

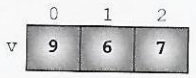

Figura 1.2 | Organização de um vetor.

1.3.2 Registro

Registro é um mecanismo que agrega vários itens de tipos *distintos*, formando uma coleção *heterogênea*. Em C, os itens de um registro ocupam posições adjacentes de memória e são identificados por *campos*. Além disso, antes de criar um registro, é preciso definir o seu *tipo* (i.e., sua *estrutura*). Por exemplo, a definição:

```
typedef struct { int a; char b; float c; } Reg;
```

define um tipo de registro denominado `Reg`. Após essa definição, a declaração

```
Reg r = {18,'a',2.5};
```

cria um registro cujos itens, representados pelas variáveis `r.a`, `r.b` e `r.c`, são de tipos distintos. A Figura 1.3 ilustra a organização desse registro na memória.

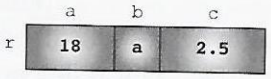

Figura 1.3 | Organização de um registro.

1.3.3 Ponteiro

Ponteiro é uma variável que guarda o endereço de outra variável. Para criar um ponteiro, basta declarar uma variável com nome prefixado por um asterisco. Por exemplo, a declaração

```
int *p;
```

cria um ponteiro `p` para guardar o endereço de uma variável do tipo `int`. Então, se `x` é uma variável `int`, para guardar seu endereço em `p`, basta executar:

```
p = &x;
```

Se um ponteiro `p` guarda o endereço de uma variável `x`, dizemos que `p` *aponta* `x` e representamos esta situação como ilustrado na Figura 1.4.

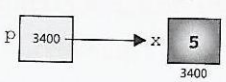

Figura 1.4 | O ponteiro p aponta a variável x, cujo endereço na memória é 3400.

Para acessar a variável apontada por um ponteiro, basta prefixar o ponteiro com asterisco. Então, se `p` é um ponteiro, `*p` é a variável que ele aponta. Por exemplo, supondo a situação na Figura 1.4, a instrução a seguir exibe o valor `5`.

```
printf("%d",*p);
```

É possível criar ponteiro de qualquer tipo, até mesmo de tipo agregado. Por exemplo, as declarações a seguir

```
int v[3] = {9,6,7};
int *p = v;
```

criam um ponteiro para vetor, como na Figura 1.5. Para acessar o `i`-ésimo item do vetor apontado por `p`, basta escrever `*(p+i)` ou `p[i]`.

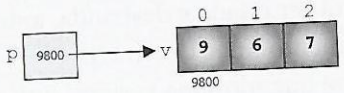

Figura 1.5 | O ponteiro p aponta o vetor v, cujo endereço na memória é 9800.

Por outro lado, as declarações a seguir

```
typedef struct { int a; char b; float c; } Reg;
Reg r = {18,'a',2.5};
Reg *p = &r;
```

criam um ponteiro para registro, como na Figura 1.6. Para acessar o campo `c` do registro apontado por `p`, basta escrever `(*p).c` ou `p->c`. Note, porém, que o operador `->` só pode ser usado com ponteiros para registros.

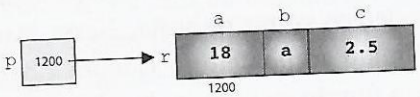

Figura 1.6 | O ponteiro p aponta o registro r, cujo endereço na memória é 1200.

Para indicar que um ponteiro não está apontando uma variável, basta usar o endereço constante `NULL`, definido em `stdio.h`. Por exemplo, após a atribuição

```
p = NULL;
```

qualquer tentativa de acesso à memória via `p` causará um *erro fatal*, que abortará a execução do programa. Um ponteiro *nulo* é representado como na Figura 1.7.

Figura 1.7 | Um ponteiro p nulo não aponta variável alguma.

1.4 Formas de alocação de memória

Depois de decidir como agregar os itens que compõem uma estrutura de dados, o próximo passo é escolher a *forma de alocação de memória* para armazená-los. A seguir, são apresentadas as principais formas de alocação de memória em C.

1.4.1 Alocações estática e dinâmica

Há duas formas básicas de alocação de memória:

- A *alocação estática* ocorre quando uma variável é declarada num programa. Nesse caso, com base no tipo da variável e no local onde ela foi declarada no programa, o compilador pode determinar a quantidade de memória necessária para armazená-la e decidir quando ela deverá ser criada e destruída, automaticamente (não confundir com a classe de armazenamento `static` em C).
- A *alocação dinâmica* ocorre quando uma variável que *não* foi declarada em um programa é necessária durante sua execução. Nesse caso, o próprio programador deve determinar a quantidade de memória necessária para armazená-la, bem como decidir quando ela deverá ser criada e destruída.

Em C, variáveis dinâmicas podem ser criadas pela função `malloc()`, declarada em `stdlib.h`. Por exemplo, a instrução

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

cria uma variável dinâmica do tipo `int`. Nessa instrução, `sizeof(int)` indica a quantidade de memória necessária (em *bytes*) para armazenar uma variável do tipo `int`. Quando a função `malloc()` é chamada, ela aloca uma área de memória do tamanho indicado e devolve seu endereço como resposta. Nessa instrução, o endereço devolvido por `malloc()` é atribuído ao ponteiro `p` (Figura 1.8a). Caso não haja memória suficiente para ser alocada, a função `malloc()` devolve `NULL`.

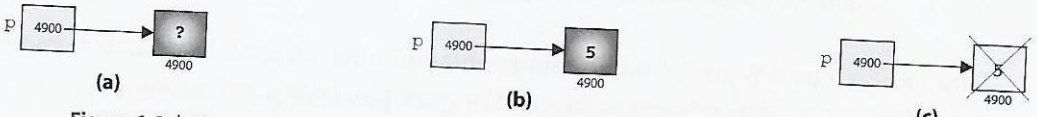

Figura 1.8 | Um ponteiro p apontando uma variável dinâmica (ou anônima), cujo endereço é 4900.

Embora uma variável dinâmica não tenha nome (isto é, seja *anônima*), seu conteúdo pode ser acessado por meio de seu ponteiro. Por exemplo, a atribuição `*p = 5` produz como efeito a situação ilustrada na Figura 1.8b.

Quando uma variável dinâmica não é mais necessária, ela pode ser destruída com a função `free()`, declarada em `stdlib.h`. Após a execução de `free(p)`, o ponteiro `p` continua apontando o mesmo endereço; porém, a variável dinâmica que ocupava esse

(vetor +1) → mat

endereço não existe mais (Figura 1.8c). De qualquer forma, quando a execução de um programa termina, todas as suas variáveis dinâmicas existentes na memória são automaticamente destruídas pelo sistema operacional.

1.4.2 Alocações sequencial e encadeada

Com relação ao modo como os itens de uma coleção são distribuídos na memória, há duas formas de alocação:

- Na *alocação sequencial*, os itens ocupam posições *adjacentes* de memória.
- Na *alocação encadeada*, os itens ocupam posições *arbitrárias* de memória.

O esquema de alocação sequencial é ilustrado na Figura 1.9. Nesse esquema, a partir do endereço do primeiro item, pode-se obter *diretamente* o endereço de qualquer outro item da coleção. Por exemplo, se o endereço do primeiro item for μ, e cada item ocupar k bytes, então o endereço do `i`-ésimo item será $\mu + i.k$.

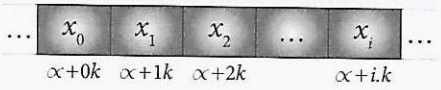

Figura 1.9 | Esquema de alocação sequencial.

Em C, a alocação sequencial é feita com *vetor*. Então, como o endereço do primeiro item num vetor é representado pelo próprio nome do vetor, o endereço do `i`-ésimo item num `v` vetor é `v+i`. Durante a compilação, a expressão `v+i` é transformada em `u+i*sizeof(*v)`, na qual `u` indica o valor `int` correspondente ao endereço `v` e `sizeof(*v)` indica o tamanho (em *bytes*) dos itens no vetor `v`.

A alocação sequencial pode ser estática ou dinâmica. Por exemplo, as declarações a seguir criam um *vetor estático* `v`, com tamanho *constante*, e um *vetor dinâmico* `w`, com tamanho *variável* (indicado pela variável `n`).

```
int v[9];                          // o vetor tera 9 posicoes
int *w = malloc(n*sizeof(int));    // o vetor tera n posicoes
```

Um item num vetor dinâmico é acessado do mesmo modo que um item num vetor estático. Assim, para acessar o `i`-ésimo item do vetor `w`, escrevemos `w[i]`.

A vantagem da alocação sequencial é a rapidez de *acesso* aos itens, pois cada um deles pode ser acessado *diretamente* (i.e., $\mu + i.k$ é um ponteiro constante para o `i`-ésimo item da coleção). A desvantagem dessa forma de alocação aparece quando é preciso *inserir* ou *remover* itens. Nesse caso, um grande tempo pode ser gasto para deslocar os demais itens da coleção, de modo a abrir espaço para o item a ser inserido ou a ocupar o espaço liberado pelo item que foi removido.

O esquema de alocação encadeada, ilustrado na Figura 1.10, é baseado no conceito de nó. Um *nó* é um registro que guarda um item e um ponteiro para um próximo nó.

A estrutura formada pelo encadeamento de nós é chamada *lista encadeada*. O endereço do primeiro nó de uma lista encadeada é guardado num ponteiro inicial (`I`), a partir do qual se pode obter *indiretamente* o endereço de cada um dos demais nós da lista. O último nó da lista encadeada tem um ponteiro `NULL` (indicando que não existe próximo nó). Não há nenhuma garantia de que os nós de uma lista encadeada estejam fisicamente ordenados na memória.

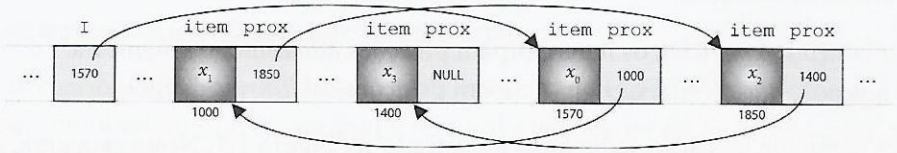

Figura 1.10 | Esquema de alocação encadeada.

Em C, a alocação encadeada é feita com *nós*. Por exemplo, a estrutura de um nó que guarda um item do tipo `int` é definida como segue:

```
typedef struct no { int item; struct no *prox; } No;
```

Essa definição cria um tipo de registro, chamado `No`, com dois campos:

- O campo `item` é uma variável inteira para guardar um item.
- O campo `prox` é um ponteiro para guardar o endereço do próximo nó.

Assim, por exemplo, para acessar o item x_0 na lista da Figura 1.10, usamos `I->item`; para acessar o item x_1, usamos `I->prox->item`; para acessar o item x_2, usamos `I->prox->prox->item`; e, para acessar o item x_3, usamos `I->prox->prox->prox->item`.

A alocação encadeada pode ser estática ou dinâmica. Por exemplo, a seguir temos uma declaração que cria nove *nós estáticos*, armazenados em um vetor `v`, e uma declaração que cria um *nó dinâmico*, apontado por um ponteiro `p`:

```
No v[9];                          // a lista podera ter no maximo 9 nos
No *p = malloc(sizeof(struct no)); // a lista podera ter varios nos
```

A vantagem da alocação encadeada é a rapidez de *inserção* e *remoção* de itens. Como os itens ocupam posições arbitrárias, a inserção ou remoção de um item nunca requer o deslocamento de outros itens da coleção (basta mudar o valor do ponteiro que aponta o nó com o item inserido ou removido). A desvantagem dessa forma de alocação aparece quando um item arbitrário da coleção precisa ser *acessado*. Nesse caso, um grande tempo pode ser necessário para percorrer a lista, inspecionando seus nós, até que o nó com o item desejado seja alcançado.

Exercícios

1.1 Analise a estrutura de dados esquematizada a seguir:

Explique por que essa estrutura:
(a) Não é um conjunto.
(b) Não é uma lista.
(c) Não é uma árvore.

1.2 Indique a estrutura de dados mais adequada para ser usada num programa que precisa representar:
(a) Estradas entre cidades de um mapa.
(b) A estrutura organizacional de uma empresa.
(c) A fila de impressão em um sistema operacional.
(d) A estrutura de diretórios em um sistema operacional.

1.3 Ao ser executado, o programa a seguir exibe a palavra `diferentes`. Explique por que isso acontece.

```
#include <stdio.h>
int main(void) {
   char s[3] = "um";
   char t[3] = "um";
   if( s == t ) puts("iguais");
   else puts("diferentes");
   return 0;
}
```

1.4 Explique o que acontece quando o programa a seguir é executado.

```
#include <stdio.h>
typedef struct { char valor[10]; } Str;
int main(void) {
   Str x = { "um" };
   Str y = { "dois" };
   puts(x.valor);
   x = y;
   puts(x.valor);
   return 0;
}
```

1.5 Explique o que acontece quando o programa a seguir é executado.

```
#include <stdio.h>
int main(void) {
   int a = 3, b = 5;
   int *p = &a, *q = &b;
   *p = *p + *q;
   *q = *p - *q;
   *p = *p - *q;
   printf("%d,%d\n", a, b);
   return 0;
}
```

1.6 Indique a forma de alocação de memória (*estática sequencial, estática encadeada, dinâmica sequencial* ou *dinâmica encadeada*) usada para armazenar os itens 1, 2, 3 e 4, em cada um dos programas a seguir.

(a)
```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int i, *v = malloc(4*sizeof(int));
   for(i=0; i<4; i++) v[i] = i+1;
   for(i=0; i<4; i++) printf("%d\n",v[i]);
   return 0;
}
```

(b)
```
#include <stdio.h>
typedef struct no {
   int item;
   struct no *prox;
} No;
int main(void) {
   No v[4] = {{3,v+2},{1,v+3},{4,NULL},{2,v+0}};
   for(No *p = v+1; p != NULL; p = p->prox)
      printf("%d\n",p->item);
   return 0;
}
```

(c)
```
#include <stdio.h>
#include <stdlib.h>
typedef struct no {
   int item;
   struct no *prox;
} *Lst;
Lst no(int x, Lst p) {
   Lst n = malloc(sizeof(struct no));
   n->item = x;
   n->prox = p;
   return n;
}
int main(void) {
   Lst p = no(1,no(2,no(3,no(4,NULL))));
   while( p ) {
      printf("%d\n",p->item);
      p = p->prox;
   }
   return 0;
}
```