CAPÍTULO

2

PILHAS

Este capítulo introduz a estrutura de dados pilha, descrevendo as principais operações que esse tipo de dados suporta, e mostra como implementá-la com alocação dinâmica sequencial.

2.1 Fundamentos

Pilha é uma lista em que todas as operações de inserção, remoção e acesso são feitas num mesmo extremo, denominado *topo*.

Quando um item é inserido numa pilha, ele é colocado em seu topo e, em qualquer instante, apenas o item no topo da pilha pode ser removido. Devido a essa política de acesso, os itens são removidos da pilha na *ordem inversa* àquela em que foram inseridos, ou seja, o último a entrar é o primeiro a sair (Figura 2.1). Por isso, pilhas são também denominadas listas LIFO (*Last-In/First-Out*).

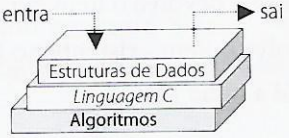

Figura 2.1 | Uma pilha de livros: o último livro empilhado é o primeiro a ser desempilhado.

A principal propriedade de uma pilha é a sua capacidade de *inverter a ordem* de uma sequência. Essa propriedade é útil em várias aplicações em computação.

Por exemplo, num navegador *web*, conforme as páginas vão sendo acessadas, seus endereços vão sendo inseridos numa pilha. Em qualquer instante durante a navegação, o endereço da última página acessada está no topo da pilha. Quando o botão *voltar* é clicado, o navegador remove um endereço da pilha e recarrega a página correspondente. Então, à medida que o botão *voltar* é clicado, as páginas acessadas são reapresentadas na ordem inversa àquela em que foram visitadas.

Controle do fluxo de execução é outro exemplo interessante do uso de pilha. Durante a execução de um programa, sempre que uma função é chamada, antes de passar o controle a ela, um endereço de retorno correspondente é inserido numa pilha. Quando a função termina sua execução, o endereço no topo da pilha é removido e a execução do programa continua a partir dele. Assim, a última função que passa o controle é a primeira a recebê-lo de volta (Figura 2.2).

```
#include <stdio.h>
void b(void) { puts("b"); }
void a(void) { b(); puts("a"); }
int main(void) {
   a();
   puts("main");
   b();
   return 0;
}
```

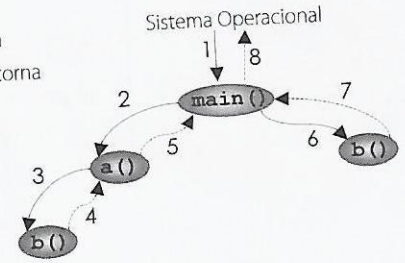

Figura 2.2 | Fluxo de execução: a ordem de retornos é inversa à ordem de chamadas.

2.2 Operações em pilhas

Uma pilha `P` suporta as seguintes operações:

- `pilha(m)`: cria e devolve uma pilha vazia `P`, com capacidade máxima `m`;
- `vaziap(P)`: devolve `1` (*verdade*) se `P` está vazia; senão, devolve `0` (*falso*);
- `cheiap(P)`: devolve `1` (*verdade*) se `P` está cheia; senão, devolve `0` (*falso*);
- `empilha(x,P)`: insere o item `x` no topo da pilha `P`;
- `desempilha(P)`: remove e devolve o item existente no topo da pilha `P`;
- `topo(P)`: acessa e devolve o item existente no topo da pilha `P`;
- `destroip(&P)`: destrói a pilha `P`.

A Figura 2.3 mostra os efeitos e resultados dessas operações numa pilha `P`. Nessa figura, a pilha é representada por uma lista com topo no extremo direito.

Operação	Pilha P	Resultado
P = pilha(3)	[]	-
vaziap(P)	[]	1
cheiap(P)	[]	0
empilha(1,P)	[1]	-
empilha(2,P)	[1,2]	-
empilha(3,P)	[1,2,3]	-
vaziap(P)	[1,2,3]	0
cheiap(P)	[1,2,3]	1
desempilha(P)	[1,2]	3
desempilha(P)	[1]	2
empilha(desempilha(P),P)	[1]	-
empilha(topo(P),P)	[1,1]	-
vaziap(P)	[1,1]	0
cheiap(P)	[1,1]	0
destroip(&P)	inexistente	-

Figura 2.3 | Efeitos e resultados das operações em pilha.

2.2.1 Conversão em binário

Para exemplificar o uso de pilhas em programação, vamos criar um programa que converte um número natural em binário. Para realizar essa conversão, o programa precisa efetuar sucessivas divisões por 2, a partir do número dado pelo usuário, até que o quociente 0 seja obtido. Depois, para mostrar o binário correspondente, basta que ele exiba os restos das divisões efetuadas, na ordem inversa àquela em que eles foram obtidos. Por exemplo, como mostra a Figura 2.4, a conversão do número 13 em binário resulta em 1101. O programa criado com base nessa ideia é apresentado na Figura 2.5.

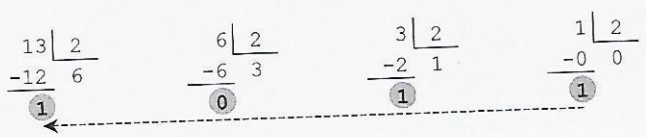

Figura 2.4 | Conversão do número 13 em binário.

```
// binario.c - conversao em binario

#include <stdio.h>
#include "../ed/pilha.h" // pilha de int

int main(void) {
   int n;
   Pilha P = pilha(32);
   printf("Decimal? ");
   scanf("%d",&n);
   do { empilha(n%2,P); n /= 2; } while( n!=0 );
   printf("Binario: ");
   while( !vaziap(P) ) printf("%d",desempilha(P));
   destroip(&P);
   return 0;
}
```

Figura 2.5 | Programa para conversão em binário.

Nesse programa, a linha `#include "../ed/pilha.h"` inclui o arquivo com a implementação do tipo `Pilha`, que será desenvolvida na próxima seção. A lógica do programa consiste basicamente de duas repetições: a primeira delas empilha o resto da divisão de `n` por `2` em `P`, e atualiza `n`, até `n` se tornar `0`; a segunda, enquanto a pilha `P` não estiver vazia, exibe no vídeo um item desempilhado de `P`.

2.2.2 Inversão de cadeia

Outro exemplo do uso de pilhas é apresentado na Figura 2.6. O objetivo desse programa é exibir a cadeia inversa àquela digitada pelo usuário. Por exemplo, se o usuário digitar a cadeia `roma`, o programa deverá exibir a cadeia `amor`.

Em C, uma *cadeia* é uma sequência de caracteres que termina com `'\0'`. Como o código ASCII do caractere `'\0'` é `0`, usando a expressão `c[i]` como condição do comando `for`, garantimos que a repetição só terminará quando o final da cadeia `c` for alcançado, ou seja, quando `c[i]` for igual a `'\0'` (isto é, *falso*). Se os caracteres distintos de `'\0'` forem empilhados durante essa repetição, no final, para ver a cadeia invertida, basta desempilhar e exibir os itens da pilha.

```
// inverte.c - inversao de cadeia

#include <stdio.h>
#include "../ed/pilha.h" // pilha de char

int main(void) {
   char  c[81];
   Pilha P = pilha(81);
   printf("Cadeia? ");
   gets(c);
   for(int i=0; c[i]; i++) empilha(c[i],P);
   printf("Inverso: ");
   while( !vaziap(P) ) printf("%c", desempilha(P));
   destroip(&P);
   return 0;
}
```

Figura 2.6 | Programa para inversão de cadeia.

2.3 Implementação de pilha

Em C, uma pilha pode ser definida como mostra a Figura 2.7.

```
typedef char Itemp;                    // tipo dos itens da pilha

typedef struct pilha {
   int    max;                         // capacidade da pilha
   int    topo;                        // posicao do topo
   Itemp *item;                        // itens da pilha
} *Pilha;
```

Figura 2.7 | Definição da estrutura de pilha.

Nessa figura, a primeira linha define o tipo `Itemp` como `char`, indicando que os itens da pilha são caracteres. As demais linhas definem `Pilha` como um tipo de ponteiro que aponta uma estrutura (`struct pilha`) com três campos: `max`, que indica a *capacidade* máxima da pilha; `topo`, que indica a posição de *topo* da pilha; e `item`, que aponta um vetor dinâmico que guarda os *itens* da pilha. Por exemplo, a Figura 2.8 mostra uma pilha criada a partir dessas definições.

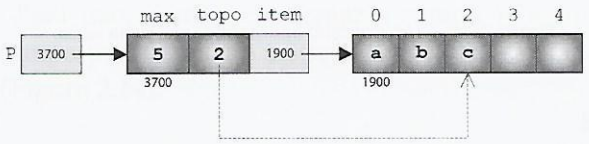

Figura 2.8 | Ponteiro para uma estrutura de pilha, que armazena os itens a, b e c.

Os campos de uma pilha apontada por um ponteiro `P` são representados por `P->max`, `P->topo` e `P->item`. Porém, esses campos nunca devem ser acessados diretamente por um programa que *usa* a pilha. Toda manipulação de pilha deve ser feita *exclusivamente* pelas funções que implementam as operações em pilha.

2.3.1 Criação de pilha

A função para criação de pilha é definida na Figura 2.9.

```
Pilha pilha(int m) {
   Pilha P = malloc(sizeof(struct pilha));
   P->max  = m;
   P->topo = -1;
   P->item = malloc(m*sizeof(Itemp));
   return P;
}
```

Figura 2.9 | Função para criação de pilha.

Quando chamada, a função `pilha()` executa os seguintes passos:

- Chama a função `malloc()` para alocar a área de memória onde a estrutura de pilha será criada, cujo tamanho em *bytes* é `sizeof(struct pilha)`. Caso haja memória suficiente, a função `malloc()` aloca o espaço solicitado e devolve o seu endereço como resposta; caso contrário, ela devolve `NULL`. O endereço devolvido pela função `malloc()` é atribuído ao ponteiro `P`.
- Acessa o campo `max` apontado por `P` e atribui a ele o valor `m`.
- Acessa o campo `topo` apontado por `P` e atribui a ele o valor `-1` (esse valor indica que não há item no topo da pilha, ou seja, que a pilha está *vazia*).
- Acessa o campo `item` apontado por `P` e atribui a ele o endereço de um vetor dinâmico, com capacidade para armazenar `m` valores do tipo `Itemp`.
- Devolve como resposta o endereço da estrutura de pilha que foi criada.

Por exemplo, a pilha na Figura 2.10 pode ser criada da seguinte forma:

```
Pilha P = pilha(5);
```

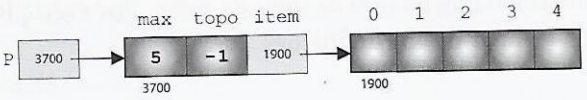

Figura 2.10 | Um ponteiro apontando uma pilha vazia.

2.3.2 Teste em pilha

Numa pilha apontada por um ponteiro `P`, o campo `P->topo` indica a posição do vetor `P->item` em que foi guardado o último item inserido na pilha. Como as posições desse vetor são indexadas de `0` até `P->max-1`, quando a pilha apontada por `P` está *vazia*, o campo `P->topo` tem valor `-1` (Figura 2.10); inversamente, quando essa pilha está *cheia*, o campo `P->topo` tem valor `P->max-1` (Figura 2.11).

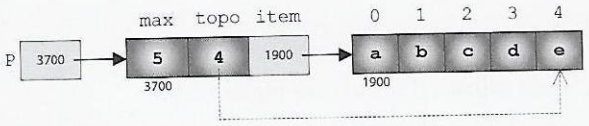

Figura 2.11 | Um ponteiro apontando uma pilha cheia.

A função que verifica se uma pilha está vazia é definida na Figura 2.12. Essa função avalia a expressão `P->topo == -1` e, se ela for verdadeira, devolve `1` (*verdade*) como resposta; senão, devolve `0` (*falso*) como resposta.

```
int vaziap(Pilha P) {
   if( P->topo == -1 ) return 1;
   else return 0;
}
```

Figura 2.12 | Função para teste de pilha vazia.

A função que verifica se uma pilha está cheia é definida na Figura 2.13. Essa função avalia a expressão `P->topo == P->max-1` e, se ela for verdadeira, devolve `1` (*verdade*) como resposta; senão, devolve `0` (*falso*) como resposta.

```
int cheiap(Pilha P) {
   if( P->topo == P->max-1 ) return 1;
   else return 0;
}
```

Figura 2.13 | Função para teste de pilha cheia.

2.3.3 Inserção em pilha

Para inserir um item numa pilha, primeiro temos que verificar se há espaço. Caso a pilha esteja cheia, a função de inserção causa um erro de *pilha cheia* (ou *stack overflow*) e a execução do programa é abortada. Caso contrário, o item deve ser inserido no topo da pilha. Para isso, basta incrementar o campo `P->topo` e, depois, usar o novo valor deste campo para acessar a posição do vetor `P->item`, onde o novo item será armazenado (Figura 2.14).

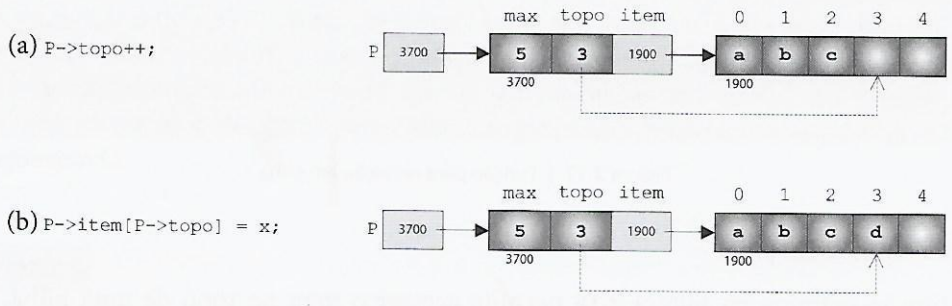

Figura 2.14 | Passos para inserir um item numa pilha.

A função para inserção em pilha é definida na Figura 2.15. Nessa função, a função `abort()`, declarada em `stdlib.h`, é usada para abortar a execução do programa.

```
void empilha(Itemp x, Pilha P) {
   if( cheiap(P) ) { puts("pilha cheia!"); abort(); }
   P->topo++;
   P->item[P->topo] = x;
}
```

Figura 2.15 | Função para inserção em pilha.

2.3.4 Remoção em pilha

Para remover um item de uma pilha, primeiro temos que verificar se há itens na pilha. Caso a pilha esteja vazia, a função de remoção causa um erro de *pilha vazia* (ou *stack underflow*) e a execução do programa é abortada. Caso contrário, o item no topo da pilha deve ser removido e devolvido. Para isso, basta copiar o item `P->item[P->topo]` numa variável a ser devolvida como resposta e, depois, decrementar o campo `P->topo`, para que esse item seja removido (Figura 2.16).

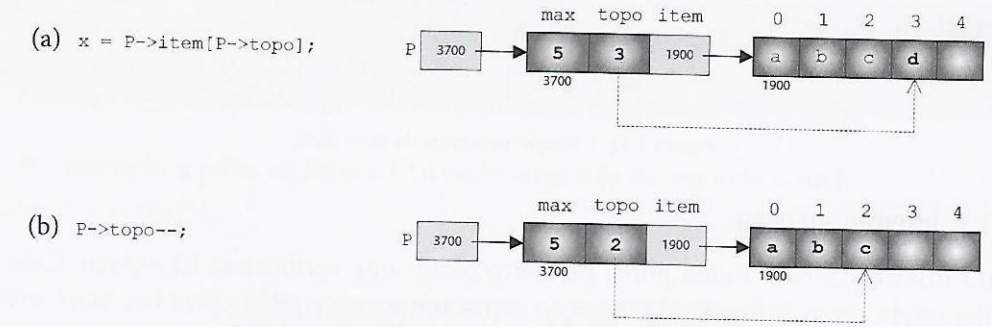

Figura 2.16 | Passos para remover um item de uma pilha.

A função para remoção em pilha é definida na Figura 2.17.

```
Itemp desempilha(Pilha P) {
   if( vaziap(P) ) { puts("pilha vazia!"); abort(); }
   Itemp x = P->item[P->topo];
   P->topo--;
   return x;
}
```

Figura 2.17 | Função para remoção em pilha.

2.3.5 Acesso em pilha

A função definida na Figura 2.18 permite acessar o item no topo de uma pilha, sem removê-lo. Essa operação causa um erro fatal quando a pilha está vazia.

```
Itemp topo(Pilha P) {
   if( vaziap(P) ) { puts("pilha vazia!"); abort(); }
   return P->item[P->topo];
}
```

Figura 2.18 | Função para acesso em pilha.

2.3.6 Destruição de pilha

A função para destruição de pilha é definida na Figura 2.19. Para destruir uma pilha apontada por `P`, basta chamar `destroip(&P)`. Essa é a única função de pilha cujo parâmetro `P` é passado por *referência* (isto é, `Q` é um *ponteiro de ponteiro*).

```
void destroip(Pilha *Q) {
   free((*Q)->item);
   free(*Q);
   *Q = NULL;
}
```

Figura 2.19 | Função para destruição de pilha.

Quando a chamada `destroip(&P)` é feita, o endereço do ponteiro `P` é copiado para o ponteiro `Q`, usado como parâmetro da função. Então, a notação `*Q` permite acessar o ponteiro `P` e, consequentemente, a notação `(*Q)->item` permite acessar o campo `item` da estrutura apontada por `P`. Assim, quando a chamada `free((*Q)->item)` é feita, o vetor `item` apontado por `P` é destruído. Depois, quando a chamada `free(*Q)` é feita, a estrutura de pilha apontada por `P` também é destruída. Finalmente, quando a atribuição `*Q=NULL` é feita, o ponteiro `P` passa a ter valor `NULL` (isto é, a pilha que era apontada por ele não existe mais).

2.3.7 O arquivo pilha.h

Daqui em diante, assumimos que as definições de tipos e funções para pilhas estão no arquivo `pilha.h`, na pasta `Pelles C Projects/ed` (veja mais detalhes no Apêndice C). Então, para usar o tipo `Pilha` num programa, basta adicionar a diretiva `#include "../ed/pilha.h"`. Assim, durante a sua compilação, todas as definições no arquivo `pilha.h` serão usadas automaticamente. Note que o uso de aspas em `#include "../ed/pilha.h"`, em vez de < e >, serve para enfatizar que `pilha.h` não é um arquivo padrão da linguagem C.

Exercícios

2.1 Qual a saída exibida pelo programa a seguir? Por quê?

```
#include <stdio.h>
#include "../ed/pilha.h" // pilha de int
int main(void) {
   Pilha P = pilha(3);
   empilha(1,P);
   empilha(2,P);
   printf("%d\n",desempilha(P));
   printf("%d\n",desempilha(P));
   printf("%d\n",desempilha(P));
   return 0;
}
```

2.2 Qual a saída exibida pelo programa a seguir? Por quê?

```
#include <stdio.h>
#include "../ed/pilha.h" // pilha de float
int main(void) {
   Pilha P = pilha(100);
   empilha(8,P);
   while( topo(P)>0 ) empilha(topo(P)/2,P);
   while( !vaziap(P) ) printf("%f\n",desempilha(P));
   return 0;
}
```

2.3 Usando pilha, crie um programa para inverter a ordem das letras nas palavras de uma frase digitada pelo usuário. Por exemplo, se for digitada a frase `"apenas um teste"`, o programa deverá exibir a frase `"sanepa mu etset"`.

2.4 Crie um programa que usa duas pilhas `A` e `B` para ordenar uma sequência de `n` números reais dados pelo usuário. A ideia é organizar a pilha `A` de modo que nenhum item seja empilhado sobre outro menor (use a pilha `B` apenas como espaço de manobra), depois, descarregue e exiba os itens da pilha `A`.

2.5 Usando pilha, crie uma função para verificar se uma expressão composta apenas por chaves, colchetes e parênteses, representada por uma cadeia, está ou não *balanceada*. Por exemplo, as expressões `"[{()()}{}]"` e `"{[([{}])]}"` estão balanceadas, mas as expressões `"{[(}])"` e `"{[)()(]}"` não estão.

2.6 Supondo que o usuário digite as cadeias `"um"`, `"dois"` e `"tres"`, qual será a saída exibida pelo programa a seguir? Por quê?

```
#include <stdio.h>
#include "../ed/pilha.h" // pilha de char *
int main(void) {
   Pilha P = pilha(5);
   char s[11];
   for(int i=1; i<=3; i++) { printf("? "); gets(s); empilha(s,P); }
   while( !vaziap(P) ) puts(desempilha(P));
   return 0;
}
```

2.7 Usando a função `_strdup(s)`, declarada em `string.h`, corrija o programa do exercício anterior. Essa função copia a cadeia `s` para uma área de memória, alocada *dinamicamente* pela função `malloc()`, e devolve o endereço dessa área. Depois de usada, essa cópia pode ser destruída com a função `free()`.