12. Suponha a existência de uma linguagem hipotética na qual não existam vetores, mas que apresente uma pilha como um tipo de dados primitivo. Seria possível implementar os vetores a partir do tipo de dados pilha? Se sim, demonstre como fazê-lo.

13. Mostre a situação da pilha *s*, inicialmente vazia, após a execução de cada uma das operações mostradas a seguir:

1 push (s, '1')	5 push (s,pop (s))
2 push (s, '2')	6 pop (s)
3 push (s, '3')	7 push (s, '4')
4 push (s, elemTopo (s))	8 pop (s)

3 Representando pilhas na linguagem C

Antes de utilizar pilhas em programas, é necessário vermos como é possível representá-las usando estruturas de dados existentes em uma linguagem de programação. No nosso caso, a linguagem C.

Existem várias possibilidades, mas, no momento, ficaremos com a mais simples: o vetor. Posteriormente, serão apresentadas outras possibilidades, certamente consideradas mais complexas.

Representação das pilhas com vetor

É possível representar uma pilha usando um vetor, porém, é necessário compreender que vetor e pilha são estruturas diferentes. O vetor apresenta uma estrutura que pressupõe a definição prévia da quantidade de elementos que existirá no seu interior; portanto, ele possui um número fixo de elementos.

Em contrapartida, a pilha é um objeto dinâmico, porque o número de elementos aumenta ou diminui à medida que empilhamos ou desempilhamos elementos. Essa importante observação conduz a possibilidade de ocorrer o chamado estouro de capacidade de uma pilha (*'overflow'*) quando um vetor é utilizado para representar uma pilha. Veremos mais adiante que uma possibilidade de contornarmos esse comportamento será a utilização da alocação encadeada, quando o limite de elementos a serem empilhados dependerá da quantidade de memória disponível em um computador.

Para representar uma pilha utilizando-se da linguagem C, podemos construir uma estrutura chamada, por exemplo, de *stack*. Observe que a pilha sugerida para o nosso estudo poderá conter no máximo cem elementos.

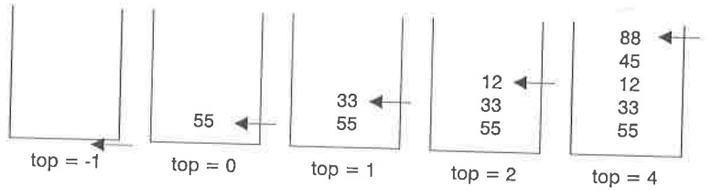
```
#define MAXPILHA 100
struct stack {
      int topo;
      int item [MAXPILHA];
```

Considerando a estrutura acima, uma pilha pode ser declarada da seguinte maneira:

```
struct stack s;
```

O campo *topo* é utilizado para sabermos qual é o elemento do topo da pilha. Como comentamos anteriormente, é o topo da pilha que se movimenta conforme são empilhados ou desempilhados os elementos. Abaixo, na Figura 3.1, temos um exemplo dos valores assumidos pelo campo *topo*, dependendo do número de elementos que estão presentes na pilha *s*. Note que, entretanto, se o conteúdo de *topo* é igual à -1, vamos assumir que a pilha está vazia. Na próxima seção, explicaremos a razão de assumirmos essa condição para uma pilha vazia.

Figura 3.1
Valores assumidos pelo campo top, de acordo com o número de pilhas.



Obviamente, não há razão para restringir uma pilha a apresentar apenas inteiros; ela pode conter qualquer tipo de dados, desde que se faça a devida configuração na estrutura. Isso pode ser ampliado mais ainda usando as uniões (*union*).

Algoritmos para manipulação de pilhas

No Capítulo 1, foram apresentadas diversas operações primitivas que podem atuar sobre uma pilha. Agora, vamos detalhar as principais em uma notação de pseudocódigo para facilitar a respectiva implementação na linguagem C.

Iniciando uma pilha

Uma pilha deve ser sempre iniciada antes de sua utilização, sob pena de presenciarmos comportamentos não desejados durante seu uso por um programa. Isso pode acontecer, se a pilha foi previamente utilizada e mantém valores residuais dessa última utilização.

No momento em que uma pilha é iniciada, ela não pode apresentar nenhum elemento. Toda vez que uma pilha não apresentar nenhum elemento, ela será considerada vazia. Para representar a pilha nesse estado vazio, faremos *s.topo = -1* antes de qualquer ação sobre a pilha. Observe que não é necessário inicializar o vetor com zeros para indicar uma pilha vazia; basta: *s.topo = -1*.

Outra alternativa é pensarmos em uma rotina chamada *inicPilha(s)*, que realiza o início de uma pilha *s*.

```
inicPilha(s)
          topo (s) ← -1


void inicPilha( struct stack *ps)
     {
        ps→topo = -1;
     };
```

A chamada da função *inicPilha(s)* seria da seguinte forma: *inicPilha(&s)*.

Verificando limites

Para desempilharmos um elemento de uma pilha, é necessário que esse elemento exista. Não é possível realizar a operação de desempilhar um elemento da pilha, caso ela esteja vazia. Da mesma forma, só é possível executar a operação de empilhamento de um elemento novo em uma pilha se esta comportar a entrada desse novo elemento. A seguir, apresentamos algoritmos e as respectivas implementações da verificação de uma pilha vazia ou de um estouro de uma pilha.

```
pilhaVazia (s)
   retorna (topo (s)= -1)


int pilhaVazia(struct stack *ps)
{
     if (ps → topo == -1)
        return ( 1 );/*Em C qualquer valor diferente de zero é verdadeiro*/
   else
        return ( 0 );/*Em C o valor zero é falso*/
```

O uso da função *pilhaVazia()* em um programa poderia ser da seguinte forma:

```
if (pilhaVazia(&s) == 1)
     /* tratamento adequado quando a pilha está vazia      */
else
     /* tratamento adequado quando a pilha não está vazia  */
     /* por exemplo, executa push ou elemTopo              */
```

Para a verificação da pilha cheia, pode-se utilizar o seguinte algoritmo com sua respectiva implementação em C:

```
pilhaCheia (s)
     retorna (topo(s) = MAXPILHA)
```

```
int pilhaCheia(struct stack *ps)
{
          if (ps → topo == MAXPILHA - 1)
               return ( 1); /*Em C qualquer valor diferente de zero é
verdadeiro*/
          else
               return ( 0 ); /* Em C o valor zero é falso*/
```

Um programa poderia usar a rotina *pilhaCheia()* da seguinte forma:

```
if (pilhaCheia(&s) == 1)
     /* tratamento adequado quando a pilha está cheia      */
else
     /* tratamento adequado quando a pilha não está cheia  */
     /* por exemplo, executa push                          */
```

Empilhando um elemento

Levando em consideração que fizemos uma representação na linguagem C da estrutura de uma pilha, da sua declaração e da estruturação das funções que verificam se uma pilha está vazia ou se nela ocorreu o estouro, vamos apresentar uma possibilidade de implementação da operação de empilhamento de um novo elemento em uma pilha.

Podemos observar que a operação de empilhamento faz a verificação se ocorre ou não a possibilidade de estouro de capacidade da pilha (*overflow*). Só poderá ocorrer o empilhamento de um novo elemento quando o estouro de capacidade da pilha não for concretizado, ou seja, existe espaço para o novo elemento. Optamos por implementar a operação de empilhamento como uma função chamada *push*.

```
empilha (s,x)
    se (pilhaCheia (s)="V" )então
                              imprima "Ocorreu overflow na pilha"
                           senão
                              topo (s) ← topo (s) + 1
                              conteúdo (s) [topo (s) ] ← x
    fim se
```



```
int push (struct stack *ps, int x)
{
     if (pilhaCheia(ps)==1 ){
          printf ("%s","Ocorreu overflow na pilha!\n");
          exit(1);}
     else
          return (ps → item [++(ps → top) ]= x);
}
```

Analisando a rotina *push* sugerida, observamos que o campo *topo*, possuindo um valor que é o próprio índice do vetor *item*, será incrementado para determinarmos a posição do vetor *item* que receberá o valor presente em x. Por exemplo, se a pilha está vazia, *topo* será igual a -1. Ao ser empilhado o primeiro elemento na pilha *s*, *topo* irá para 0. Um ponto importante é que, na chamada da função *pilhaCheia(ps)*, o operador de endereço "&" não está presente. Isso acontece porque "ps" já é um ponteiro.

Observe que, no momento em que *pilhaCheia(ps)* retornar o valor 1 (verdadeiro), a rotina *push* exibirá uma mensagem de erro, e o programa será encerrado com *exit (1)*. Acreditamos que não seja a melhor ação a ser tomada, porque um programa deve ser finalizado na ocorrência de qualquer erro, de forma mais elegante. Deixaremos que o leitor encontre uma boa forma para contornar essa situação.

A chamada da função *push* poderia ser:

```
push (&s, x);
```

Desempilhando um elemento

Essa operação realiza o inverso da operação de empilhamento, ou seja, se existir um elemento na pilha, irá ocorrer o desempilhamento do elemento que estiver no topo da pilha. Isso significa que o elemento presente no topo da pilha será retirado dessa pilha. Observamos que é necessário haver a verificação do estado da pilha antes de executar a operação de desempilhamento. Se a pilha estiver vazia, não há o que ser retirado; fazer isso causaria o chamado '*underflow*'. Optou-se em implementar a função de desempilhamento com o nome *pop*.

```
desempilha (s)
     se (pilhaVazia (s)= "V") então
                              imprima "Ocorreu underflow na pilha"
                           senão
                              desempilha ← conteúdo (s) [topo (s)]
                              topo (s) ← topo (s) - 1
     fim se
```

```
int pop ( struct stack *ps)
{
      if (pilhaVazia(ps)==1){
            printf("%s", "Ocorreu underflow na pilha!\n");
            exit (1)};
      else
            return (ps → item [ps → topo --]);
}
```

A presença de *exit (1)* na função *pop* tem as mesmas restrições realizadas na função *push*, no que tange o encerramento do programa sem um tratamento adequado dessa situação de erro. A função *pop* poderia ser usada da seguinte forma:

```
x = pop (&s);
```

Evidentemente, o programa poderia assegurar a ocorrência de um *underflow* antes de solicitar *pop*:

```
if (pilhaVazia(&s)==0)
            x = pop (&p);
      else
            /* tratamento adequado quando a pilha está vazia*/;
```

Consultando o elemento do topo da pilha

Algumas vezes, não queremos retirar o elemento que está no topo de uma pilha, mas apenas realizar uma consulta nesse elemento. Essa operação de consulta também deve considerar a possibilidade do *underflow*. Essa rotina é muito semelhante à rotina de desempilhamento, com exceção da ausência de um passo que provoca a diminuição do campo *topo* em uma unidade no momento que ocorre o desempilhamento dos elementos. Mais uma vez, note as restrições do uso de *exit (1)* como forma de abandonar a execução de *elemTopo( )* na ocorrência de um *underflow*.

```
elemento_topo (s)
      se (pilhaVazia (s)="V") então
            imprima "Ocorreu underflow na pilha"
      senão
            desempilha ← conteúdo (s) [topo (s)]
      fim se
```

```
int elemTopo( struct stack *pos)
{
      if (pilhaVazia(ps)==1){
            printf("%s", "Ocorreu underflow na pilha!\n");
            exit (1)};
      else
            return (ps → item [ps → topo ]);
}
```


A chamada da função *elemTopo( )* poderia ocorrer da seguinte forma:

```
x = elemTopo(&s);
```

Outra possibilidade interessante para representar a consulta de um elemento presente no topo de uma pilha é a realização de uma combinação de *pop* e *push*, da seguinte forma:

```
x = pop(&s);
push (&s,x);
```

A Figura 3.2 mostra a evolução da execução das seguintes instruções: *push (s, a), push (s,f), elemTopo(s), pop (s), push (s, pop(s)), push(s, z), pop(s)* e *pop(s)*. Acompanhe o conteúdo da pilha *s* conforme são aplicadas as instruções sugeridas:

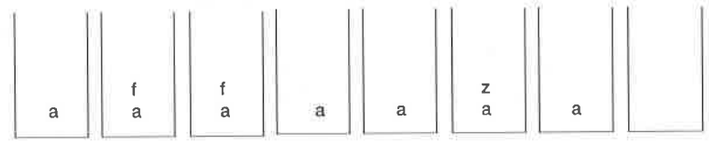


Figura 3.2
Evolução da aplicação de push, pop e stacktop sobre uma pilha *s*.

De forma geral, lembramos que é muito importante que um programa não seja interrompido de forma abrupta quando ocorrer um *overflow* ou um *underflow*. É recomendado que, em vez disso, sejam realizadas atitudes corretivas para essas duas situações, que podem ser inclusive mensagens esclarecedoras. Apresentamos anteriormente algumas possibilidades para que possa ser feita a verificação do estado da pilha, em relação ao fato dessa pilha estar cheia ou vazia, para que possa ser executada uma operação de *push* ou de *pop*.

Uso prático das pilhas

Para mostrar como as pilhas podem ser usadas para a construção de soluções elegantes, apresentaremos alguns problemas simples — mas suficientes — para exercitarmos as operações básicas apresentadas até agora.

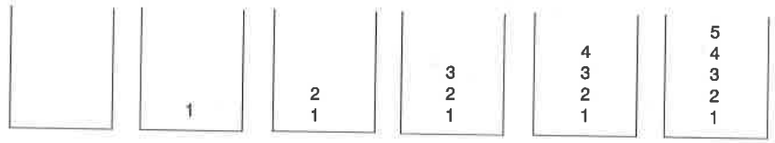
Uma aplicação imediata das pilhas é na reordenação de um conjunto de dados, tal que o primeiro e o último elementos devem ser trocados de posição e todos os elementos entre o primeiro e o último devem também ser relativamente trocados.

Por exemplo, se inserirmos {1 2 3 4 5}, a saída será {5 4 3 2 1}. O algoritmo mostrado a seguir ilustra uma possibilidade para reverter uma certa seqüência de valores numéricos inseridos até que seja digitado o valor 999.

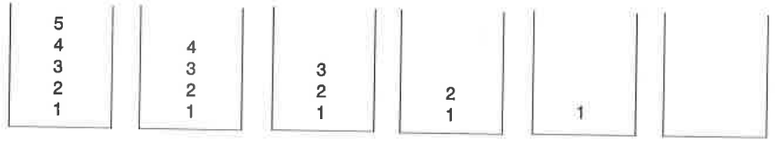
```
início
    inicPilha(s) // faz a pilha s ser vazia
    leia num
    enquanto ((num≠999) e (pilhaCheia(s)≠"V") faça  //empilha valores de entrada
        push (s,num)
        leia num
    fim enquanto                                     //desempilha os valores na
    enquanto (pilhaVazia(s) ≠ "V") faça              //ordem inversa
        X ← pop (s)
        imprima x
    fim enquanto
fim.
```

Para testarmos o algoritmo apresentado anteriormente, suponhamos as seguintes entradas: 1, 2, 3, 4, 5, 999. Durante a execução da primeira estrutura repetitiva, podemos ilustrar a evolução da pilha conforme exibe a Figura 3.3 (a). No momento da digitação do valor 999, finaliza-se a entrada de dados. Em seguida, deve-se começar o processo de desempilhamento da pilha *s* (Figura 3.3 (b)). Assim, percebemos que obtemos uma inversão dos dados de entrada, ou seja, obtemos como saída 5, 4, 3, 2, 1.

Figura 3.3
Evolução da reversão dos dados de entrada.



(a) Empilhamento. A entrada é 1, 2, 3, 4, 5.

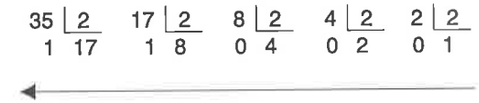


(b) Desempilhamento. A saída é 5, 4, 3, 2, 1.

Essa idéia de reversão de dados apresentada anteriormente pode ser usada na resolução de um problema clássico, como o que faz a conversão de um número na base decimal para a base binária. Suponha que se deseja converter o número 35 da base decimal para seu equivalente na base binária.

Uma solução bastante usada é realizar divisões sucessivas por dois até que o quociente apresente o valor um.

Figura 3.4
Conversão decimal para binário.

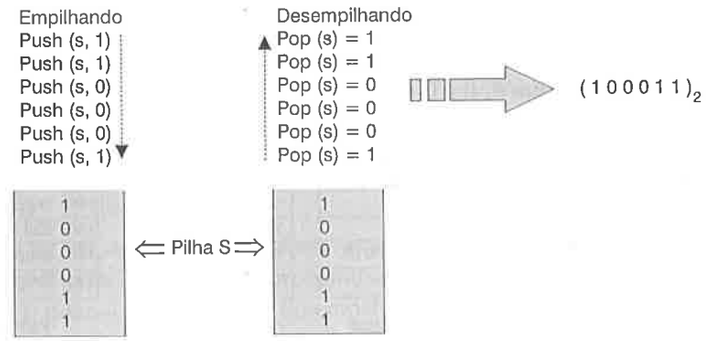


O valor equivalente em binário pode ser obtido pegando-se o último quociente e, da direita para a esquerda, todos os restos das divisões (Figura 3.4), exibindo, assim, o seguinte resultado:

$$(1\,0\,0\,0\,1\,1)_2 = (35)_{10}$$

Analisando o comportamento das divisões efetuadas, percebemos que os valores dos restos e o valor do último quociente podem ser empilhados em uma pilha e, em seguida, desempilhados para a obtenção do valor numérico em binário (Figura 3.5). Esse comportamento verificado nas divisões é do tipo *LIFO (Last-in-First-Out)*; o último a entrar, é o primeiro a sair.

Figura 3.5
Uso da pilha na conversão decimal para binário.



O algoritmo na notação de pseudocódigo, para a situação visualizada anteriormente, pode ficar da seguinte maneira:

```
início
  leia numerador
  enquanto ((numerador ≠ 0) e (pilhaCheia(s) ≠ "V")) faça
     resto ←numerador mod 2
     denominador ← numerador div 2
     numerador ← denominador
     push (s, resto)
  fim enquanto

  enquanto (pilhaVazia(s) ≠ "V") faça
     x ←pop (s)
     imprima x
  fim enquanto

fim.
```

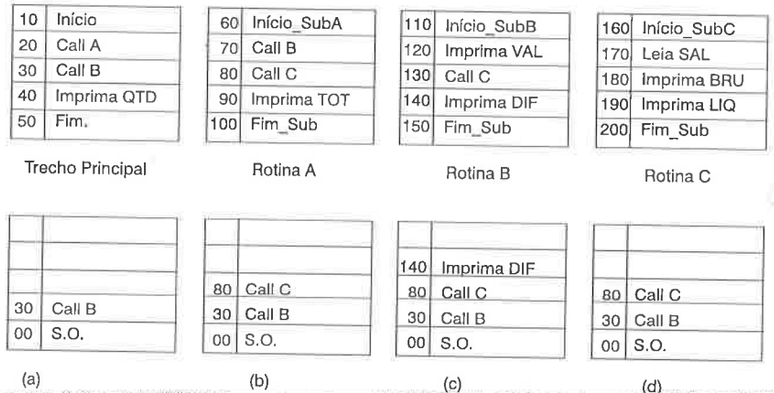
A observação especial fica para *mod*, que retorna o resto da divisão de dois números inteiros, e para *div*, que retorna o quociente da divisão de dois inteiros.

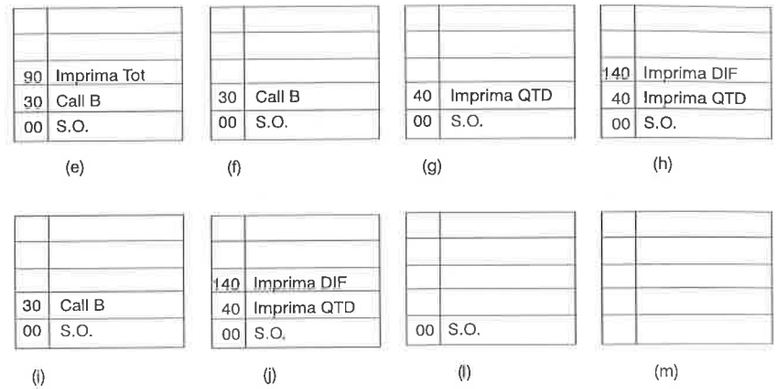
Outra possível utilização das pilhas pode ser no controle do ponto de retorno de sub-rotinas. Como sabemos, quando uma sub-rotina é finalizada, o retorno do controle do fluxo de execução deve ocorrer no ponto seguinte de onde ocorreu a chamada da sub-rotina.

Para construir um exemplo com o intuito de ilustrar esse uso, vamos considerar uma linguagem hipotética que possui somente o seguinte conjunto de comandos: *Leia*, *Imprima* e *Call* para chamar sub-rotinas.

Suponha ainda que temos um programa composto de um trecho principal, que faz a chamada de três sub-rotinas A, B e C descritas, como é exibido abaixo:

Figura 3.6
Uso de pilhas no controle de retorno de sub-rotinas.





Vamos agora demonstrar como o uso de uma pilha pode ser uma boa forma para gerenciar o fluxo de controle de execução, à medida que as sub-rotinas são chamadas e finalizadas. Suponha que o retorno do fluxo de controle para o sistema operacional, após o término de um programa, seja feito no endereço *00*.

Considere, ainda, que a chamada de execução de um programa é realizada a partir do sistema operacional. Dessa forma, espera-se que, no término da execução do programa, o sistema operacional retome o fluxo de controle de execução. Este é o motivo de o endereço *00.S.O.* ser colocado na pilha de controle de execução no momento em que o programa é chamado para ser executado. Durante a execução do trecho principal do programa, é realizada a chamada da sub-rotina *Call A* em *20 Call A*. Considerando que, no momento da finalização da sub-rotina A, o controle do fluxo de execução deve retornar para *30 Call B*, esse endereço é empilhado na pilha que controla a chamada das sub-rotinas (Figura 3.6 (a)). Note que, sempre na chamada de uma sub-rotina, é o endereço seguinte da sub-rotina que deve ser empilhado, e não o endereço da sub-rotina que foi chamada.

Quando executamos a chamada da sub-rotina B no endereço *70 Call B*, empilha-se o endereço de onde deve continuar o fluxo de execução após o término da sub-rotina B. Então, *80 Call C* é colocado na pilha (Figura 3.6 (b)).

A sub-rotina B começa a ser executada até que ocorra a chamada da sub-rotina C, e *140 Imprima DIF* é empilhado (Figura 3.6 (c)). Quando ocorrer o término da sub-rotina C no endereço *200 Fim_Sub*, o sistema deve realizar o desempilhamento do endereço que está no topo da pilha, para determinar a próxima ação que será realizada. De acordo com o nosso exemplo, o endereço que está no topo é *140 Imprima* (Figura 3.6 (d)). Então, o sistema sabe que o fluxo de execução deve ir para *140 Imprima*, que está na sub-rotina B.

Após a execução de *140 Imprima DIF*, a sub-rotina B é finalizada em *150 Fim_Sub*, novamente é desempilhada a informação que está no topo da pilha; no caso, *80 Call C*. Na chamada dessa sub-rotina, empilha-se o endereço de retorno após seu término; no caso, *90 Imprima TOT*

O processo descrito prossegue até o momento em que o próprio trecho principal do programa é finalizado em *50 Fim*, ocorrendo, então, o desempilhamento de *00 S. O.*, que é a chamada de retorno ao sistema operacional. Nesse momento, a pilha usada para controlar a chamada e o retorno das sub-rotinas, como é perceptível, torna-se vazia.

De uma forma geral, podemos dizer que as pilhas são utilizadas em situações nas quais temos a necessidade de realizar algum trabalho de inversão dos dados de entrada, ou seja, quando necessitamos de um controle do tipo *LIFO (Last-in-First-Out)*. Observamos também, por meio dos exemplos apresentados, que as pilhas podem colaborar na obtenção de algoritmos consistentes e elegantes.

EXERCÍCIOS DE APROFUNDAMENTO

1. Considere que existam duas pilhas vazias de inteiros, s1 e s2. Desenhe uma ilustração de cada uma das pilhas e das instruções presentes no seguinte trecho de programa C:

```
push(s1,3);
push(s1,5);
push(s1,7);
push(s1,9);
push(s1,11);
push(s1,13);
while (!pilhaVazia(s1)) {
      pop(s1,x);
      push(s2,x);
}
```

2. Suponha que nós temos duas pilhas vazias de inteiros, s1 e s2. Faça a ilustração dessas pilhas e das seguintes instruções presentes em um certo trecho de programa C:

```
push(s1,3);
push(s1,5);
push(s1,7);
push(s1,9);
push(s1,11);
push(s1,13);
while (!pilhaVazia(s1)) {
      pop(s1,x);
      pop(s1,x);
      push(s2,x);
}
```


3. Codifique um programa em C para o algoritmo que reverte uma série de números apresentados neste capítulo. Teste seu programa com a série 1, 3, 5, 7, 9, 2, 4, 6, 8.

4. Codifique um programa em C para o algoritmo que converte um número decimal para binário. Teste seu programa com os números 19, 127 e 255.

5. Codifique uma função que transforma um número decimal em um número octal. Veja o algoritmo apresentado neste capítulo.

6. Escreva uma função que transforma um número decimal em um número hexadecimal. Dica: Se o resto for 10, 11, 12, 13, 14 ou 15, imprima, respectivamente, A, B, C, D, E ou F.

7. Uma string é considerada palíndroma se ela pode ser lida da esquerda para a direita ou da direita para a esquerda com o mesmo significado. Neste caso, não podemos considerar as acentuações, as letras maiúsculas ou minúsculas, os espaços e os caracteres especiais. A seguir, estão alguns exemplos:

   Subi no Onibus **Able was I ere I saw Elba**
   Radar **Go dog** **Madam, I'm Adam**

   Solicita-se que você faça um algoritmo que determine se uma expressão é palíndroma ou não.

8. Faça um esquema, conforme ilustrado neste capítulo, da evolução do conteúdo da pilha que controla o fluxo de execução dos programas apresentados logo a seguir, conforme é realizada a chamada das sub-rotinas. Dica: Considere a possibilidade de *overflow*.

a.

Inicio	Inicio_SubA	Inicio_SubB	Inicio_SubC
1 imprima LADO	4 call B	7 imprima AREA	10 imprima TOT
2 call A	5 call C	8 call C	11 leia ALTURA
3 fim	6 fim_sub	9 fim_sub	12 fim_sub

b.

Inicio	Inicio_SubA	Inicio_SubB
1 imprima LADO	4 imprima AREA	7 imprima TOT
2 call A	5 call B	8 call A
3 fim	6 fim_sub	9 fim_sub

9. Codifique um programa para ler uma frase e imprimi-la com as palavras invertidas. Exemplo: A frase 'um por todos e todos por um', deverá ficar 'mu rop sodot e sodot rop um'.

10. Escreva um algoritmo que empilhe uma seqüência de valores numéricos inteiros positivos até o momento em que o valor 999 for digitado. Neste momento, o conteúdo da pilha deverá ser distribuído em outras duas pilhas. Uma delas conterá apenas os valores ímpares, e a outra conterá apenas os valores pares.

11. Implemente um programa em C, o qual localize um determinado elemento em uma pilha. Em seguida, faça a remoção do mesmo. Para tanto, se o elemento não estiver no topo, os elementos anteriores ao elemento procurado deverão ser desempilhados e, após a remoção, empilhados novamente. O programa deve prever a possibilidade de o elemento que está sendo procurado não existir na pilha.

12. Considere uma pilha com valores numéricos inteiros positivos. Faça um programa que remova todos os múltiplos de 3, porém, sem a utilização de uma estrutura auxiliar.