Resolução dos Exercícios do Capítulo IV

1. Sinonímia ocorre quando uma mesma variável ou constante pode ser referenciada por mais de um nome em um mesmo ambiente de amarração. Mostre exemplos de situações nas quais isso pode ocorrer em C e JAVA.

Exemplo de ocorrência de sinonímia em C

```
int a;
int * b;
a = 0;
b = & a;                                  // b aponta para o endereço da
variável a
*b = 1;                                   // a é modificado para 1
```

Exemplo de ocorrência de sinonímia em Java

```
int [] a = new int[3];
a[0] = 3;
a[1] = 6;
a[2] = 9;
int [] b = a;                  // b passa a ser um sinônimo de a
b[1] = 5;                                 // a[1] passa a valer 5
```

2. Mostre situações nas quais a permissão de acesso ao endereço de variáveis pode ser benéfica ao programador. Mostre também quando isso pode ser prejudicial a confiabilidade dos programas.

Situação em que a permissão de acesso a endereços de variáveis é benéfica ao programador

Em algumas situações pode-se conseguir mais eficiência. Considere, o seguinte trecho de código:

```
main (){
float matrx [50][50];
int i,j;
for (i=0;i<50;i++)
  for (j=0;j<50;j++)
    matrx[i][j]=0.0;
}
```

Ele pode ser reescrito de uma maneira muito mais eficiente. Isso é possível em virtude de se poder obter o endereço da posição inicial onde a matriz está alocada:

```
main (){
  float matrx [50][50];
  float *p;
   int count;
  p= &matrx[0][0];
  for (count=0;count<2500;count++){
     *p=0.0;
      p++;
```

```
        }
}
```

Situação em que a permissão de acesso a endereços de variáveis é prejudicial a confiabilidade dos programas

A permissão de acesso a endereços de variáveis em C possibilita que uma variável de determinado tipo seja acessada ou alterada por um ponteiro para outro tipo. Isso é uma violação do sistema de tipos da linguagem. Muitas vezes isso é feito por engano. Nesse sentido é reduzida a confiabilidade dos programas

3. Edite o programa seguinte, compile-o e o execute. Relate o que ocorreu na compilação e durante a execução.

```
main() {
  char* z = "bola";
  *z = 'c';
  printf("%s\n", z);
  printf("bola");
}
```

O compilador usado para resolver a questão foi o gcc. Durante a compilacão, o compilador não apresentou qualquer problema. Ao executar o programa, o mesmo foi interrompido devido ao uma falha de segmentacão de memória. A linha que causou o problema é a que contém a atribuição

```
*z = 'c';
```

Esse problema ocorre por causa da tentativa de alterar o valor de uma constante.

4. Faça um programa com as linhas de código do exemplo 4.8, retirando os comentários das linhas de código comentadas. Compile o programa usando seu compilador C e seu compilador C++. Relate o que aconteceu.

Os compiladores usados na resolucão da questão foram o gcc e o g++. O gcc apenas avisou que estava sendo feito um incremento e uma atribuicão a uma variável exclusiva de leitura (*y*) e que a atribuição *x = &y;* descarta os qualificadores do tipo alvo do ponteiro. O programa é compilado com sucesso.
O g++ também informou que estava sendo feito um incremento e uma atribuicão a uma variável exclusiva de leitura (*y*) e que a atribuicao *x = &y;* tenta fazer uma conversão inválida de um *const int** para um *int**. O programa não é compilado.

5. Explique as vantagens de se utilizar um modelo de gerenciamento de memória principal com regiões de pilha e monte em relação aos modelos que só utilizam a pilha ou só utilizam o monte ou que alocam variáveis apenas em tempo de carga do programa.

As vantagens de se utilizar gerenciamento de memória principal com regiões de pilha e monte decorrem do fato de se poder beneficiar das melhores características de cada um desses modelos. A pilha tem por característica principal a alocação de espaço suficiente para armazenar as variáveis locais de um bloco ou subprograma quando estes são executados. No momento que o bloco ou subprograma é encerrado, essas variáveis são retiradas da pilha, liberando este espaço de memória. Já o monte resolve

o problema do aumento dinâmico das variáveis, ou seja, caso haja necessidade de aumentar o espaço para uma variável já existente, esse espaço adicional será reservado no monte, mais especificamente em local onde exista espaço contíguo suficiente para alocar a nova variável.
O problema de só utilizar um dos tipos (pilha ou monte) é que sozinhos eles não são muito eficientes. Ao usar apenas a pilha, teríamos problemas com variáveis nas quais o tamanho se modifica em tempo de execução. Em relação ao uso exclusivo do monte, a alocação não seria muito eficiente visto que a alocação dinâmica de memória no monte demanda o uso de estruturas de dados para gerenciamento da memória do programa, e por conseguinte a sua atualização a cada momento de alocação e desalocação de variáveis.
Outra possibilidade seria alocar todas as variáveis em tempo de carga. Isso se torna ineficaz uma vez que frequentemente haverá desperdício ou insuficiência de memória. Além disso, essa abordagem impede a implementação de programas recursivos, pois só se sabe quantas vezes o mesmo será chamado durante cada execução do programa.

6. Enquanto implementações de linguagens de programação que incluem o conceito de ponteiros (por exemplo, C e C++) tipicamente deixam parte da alocação e desalocação de memória sob a responsabilidade do programador, implementações de linguagens que não possuem ponteiros (por exemplo, JAVA) devem necessariamente incluir um sistema de gerenciamento de memória que controle e libere o espaço de memória utilizado. Compare estas diferentes abordagens em termos de eficiência, redigibilidade e confiabilidade.

Eficiência
As LPs que incluem ponteiros (e, consequentemente parte da alocação e desalocação de memória para o programa) são melhores que as que não possuem ponteiros. A explicação é que, como o programador assume a tarefa de alocação e desalocação, este pode fazê-lo nos momentos mais oportunos, nos exatos pontos em que as áreas de memória alocadas deixam de ser necessárias, e também possibilita ao programador desalocar a memória em pontos do programa que não interfiram no seu desempenho, problemas estes que podem ocorrer com o sistema de coletores de lixo.

Redigibilidade
A opção de não se utilizar ponteiros (como JAVA) deixa a programação mais fácil pois o programador não precisa se preocupar com os detalhes da programação com ponteiros. Mais especificamente, não é necessário qualquer esforço para desalocar memória alocada dinamicamente. Isso é executado automaticamente sem a interferência do programador, poupando-o de se preocupar com esses detalhes.

Confiabilidade
A programação com ponteiros é necessariamente menos confiável pois, como foi dito antes, a redigibilidade se reduz, o que aumenta a possibilidade de provocar erros por parte do programador, o qual tem de cuidar de detalhes que a programação com ponteiros exige. Um importante aspecto é que, na programação com ponteiros, o programador detém a tarefa de efetuar a desalocação dos espaços de memória dinâmicos, o que faz com que diversos erros possam ocorrer, tais como: esquecer de desalocar memória desnecessária, manutenção de referências para áreas já desalocadas, desalocação de áreas de memória ainda úteis. Todos esses erros não

ocorrem com o *coletor de lixo* que atua nas LPs sem ponteiros pois estes processos são realizados sem a interferência do programador.

7. Identifique inicialmente a quais subprogramas pertencem as variáveis referenciadas nas linhas de código do programa do exemplo 4.9. Verifique se sua avaliação está correta seguindo a cadeia de elos estáticos na figura 4.6.

Em *Principal* => *x* pertence a *Principal*
Em *Sub1* => *a*, *b* e *c* pertencem a *Sub1*
Em *Sub2* => *a* e *d* pertencem a *Sub2* e *b* e *c* pertencem a *Sub1*
Em *Sub3* => *b* e *e* pertencem a *Sub3* e a pertence a *Sub1*
Em *Sub4* => *c* e *e* pertencem a *Sub4*, *b* pertence a *Sub3* e *a* pertence a *Sub1*

8. Desenhe o estado da pilha de registros de ativação após cada chamada e encerramento de subprograma ao longo da execução do programa do exemplo 4.9.

Após chamada de Principal:

x	0

Após chamada de Sub1:

c	3
b	2
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após chamada de Sub3:

e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	2
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após chamada de Sub4:

e	7
c	6
Dinâmico: Sub3	
Estático: Sub3	
Endereço de Retorno: --4	
e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	2
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após a 1ª chamada de Sub2:

d	8
a	5
Dinâmico: Sub4	
Estático: Sub1	
Endereço de Retorno: --3	
e	7
c	6
Dinâmico: Sub3	
Estático: Sub3	
Endereço de Retorno: --4	
e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após a 2ª chamada de Sub2:

d	8
a	9
Dinâmico: Sub2	
Estático: Sub1	
Endereço de Retorno: --2	
d	8
a	5
Dinâmico: Sub4	
Estático: Sub1	
Endereço de Retorno: --3	
e	7
c	6
Dinâmico: Sub3	
Estático: Sub3	
Endereço de Retorno: --4	
e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após o encerramento da 2ª chamada de Sub2:

d	8
a	5
Dinâmico: Sub4	
Estático: Sub1	
Endereço de Retorno: --3	
e	7
c	6
Dinâmico: Sub3	
Estático: Sub3	
Endereço de Retorno: --4	
e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após o encerramento da 1ª chamada de Sub2:

e	7
c	6
Dinâmico: Sub3	
Estático: Sub3	
Endereço de Retorno: --4	
e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após encerramento de Sub4:

e	5
b	4
x	19
Dinâmico: Sub1	
Estático: Sub1	
Endereço de Retorno: --5	
c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

Após encerramento de Sub3:

c	3
b	6
a	1
Dinâmico: Principal	
Estático: Principal	
Endereço de Retorno: --6	
x	0

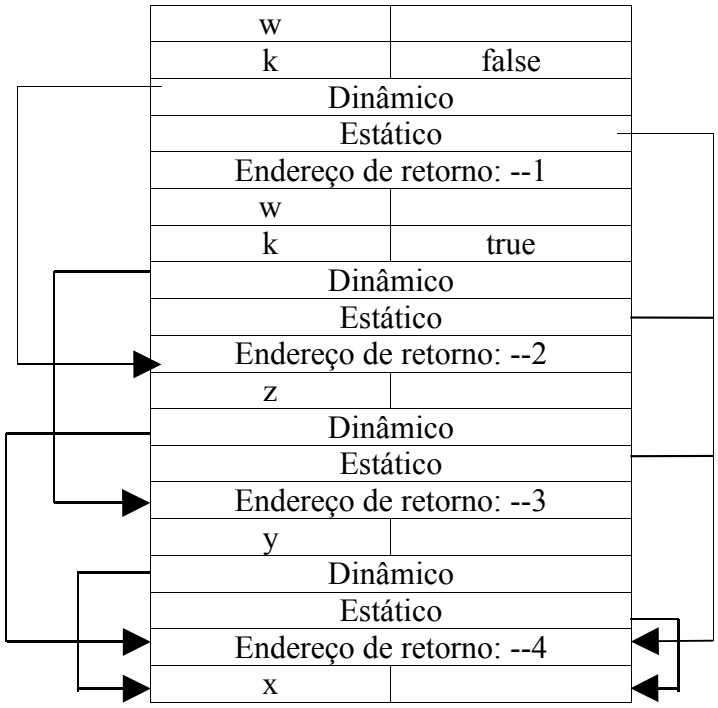
Após encerramento de Sub1:

x	0

Após encerramento de Principal: pilha vazia

9. Desenhe a pilha de registros de ativação, incluindo o endereço de retorno e as cadeias estática e dinâmica, quando a execução atinge o ponto marcado com # no seguinte esqueleto de programa ADA.

```
procedure Main is
  x: integer;
   procedure A;
     y: integer;
      procedure B (k: boolean);
         w: integer;
         begin -- B
                     if k then
                       B (false);
                    else -- #;
         end B;
      procedure C;
         z: integer;
         begin -- C
             ... B (true);
         end C;
      begin  -- A
         ... C; ...
      end A;
  begin -- Main
    ... A;
  end Main;
```



10. Em JAVA, todos os objetos (variáveis compostas) são alocados no monte. Explique por que isso não é tão ineficiente em relação a LPs que alocam essas variáveis na pilha.

Porque em Java não são utilizadas listas de espaços disponíveis e ocupados. A máquina virtual faz uma alocação logo após a última alocação feita, só tendo de alterar o marcador de final do monte. Esse processo é equivalente ao feito na pilha para alocar memória. Nesse caso, só é preciso alterar o "stack pointer". Por outro lado, o processo de desalocação é mais ineficiente pois ocorre no momento que o monte fica com uma alta taxa de ocupação. Nesse caso o coletor de lixo é chamado para liberar todo o espaço não utilizado.

11. Explique como a persistência de dados é implementada em C. Compare essa abordagem com a de JAVA em termos de redigibilidade e legibilidade. Justifique sua resposta enfocando especialmente a implementação de persistência em estruturas de dados que utilizam ponteiros. Na sua opinião haveria algum efeito adverso em deixar sob o controle da linguagem de programação todo o processo de persistência de dados?

A persistência de dados em C é realizada através da leitura e gravação de seqüências de bytes em arquivos. Para que o programador possa fazer isso, a linguagem oferece em sua biblioteca padrão um conjunto de funções cujo objetivo é permitir a abertura, o fechamento, a gravação e a leitura de um conjunto de bytes dos arquivos. É responsabilidade do programador definir como as estruturas de dados do programa devem ser convertidas em uma seqüência de bytes na gravação dos dados e vice-versa na leitura. Tal processo se torna complexo e bastante trabalhoso quando a estrutura de dados envolve vários ponteiros, pois a informação desses ponteiros necessita ser

relativizada quando armazenada em memória secundária para que a estrutura possa ser recuperada tal como existia originalmente.
JAVA também realiza persistência de dados através da leitura e gravação de seqüências de bytes em arquivos. Contudo, ela oferece um mecanismo de serialização automática que se responsabiliza por fazer as conversões necessárias da estrutura de dados para uma seqüência de bytes e vice-versa. Para isso, é necessário apenas que o programador declare que a classe como implementa a interface *Serializable*. Isso facilita muito a vida do programador e aumenta a redigibilidade e legibilidade dos programas nessa LP.
Existem alguns inconvenientes em deixar sob total controle da LP todo o processo de persistência de dados (observe que JAVA não faz isso pois o programador é quem diz quais classes devem ser serializadas e quais as variáveis devem ser lidas ou gravadas e quando). Pode-se querer armazenar os dados em formatos diferenciados (por exemplo, com apenas alguns campos, com uma ordem diferente desses campos ou de uma maneira codificada). Pode-se querer também realizar a persistência em alguns momentos específicos do programa quando o esforço computacional requerido é pequeno.

12. Considere um programa que armazena em um grafo (usando encadeamento dinâmico) as distâncias terrestres entre grandes cidades brasileiras e grava essa estrutura em memória secundária para recuperá-la posteriormente em uma outra variável. Compare como seriam os códigos das implementações em C e JAVA desse programa em termos de redigibilidade, legibilidade e confiabilidade.

O código em JAVA é mais redigível porque o programador não necessita implementar código para dizer como converter o grafo (implementado através de encadeamento dinâmico) em uma seqüência de bytes e vice-versa. Isso também torna o código mais legível, uma vez que todo o programa pode se concentrar fundamentalmente na realização de operações sobre o grafo, sem que esse código fique misturado com as operações de conversão de dados. Por fim, o código se torna mais confiável porque o programador não precisa implementar essas operações de conversão. O mecanismo de conversão utilizado já é fornecido pela LP, o que fornece uma maior garantia de correção.