Capítulo IV – Variáveis e Constantes

"O ícone persiste porque a potência da profecia persiste."
Luis Fernando Verissimo (se referindo a Karl Marx)

Os valores dos tipos de dados devem ser armazenados em entidades de computação para que possam ser manipulados pelos programas. Variáveis e constantes são as principais entidades nas quais valores podem ser armazenados. Nesse capítulo são discutidas as características fundamentais associadas a esses conceitos.

Ênfase especial é dada na forma como é feito o gerenciamento da memória principal do computador para o armazenamento dessas entidades e também nos mecanismos disponíveis nas LPs para a programação de persistência de dados.

4.1 Variáveis

David Watt reproduz em seu livro [WATT, 1990] uma conhecida citação de Dijkstra, um famoso cientista da computação:

> *"Uma vez que o programador tenha entendido o uso de variáveis, ele entendeu a essência da programação"*.

Segundo Watt, essa frase pode ser considerada um exagero, visto que o conceito de variáveis em LPs funcionais e lógicas é diferente do conceito nas LPs imperativas. Por outro lado, ela se aplica perfeitamente ao paradigma imperativo, que é caracterizado fundamentalmente pelo uso de variáveis e pelo conceito de atribuição.

No paradigma imperativo, uma variável é uma entidade de computação que contém um valor, o qual pode ser inspecionado e atualizado sempre que necessário. Mais especificamente, uma variável é uma abstração para uma ou mais células de memória responsáveis por armazenar o estado de uma entidade de computação. Variáveis são normalmente usadas em programas para modelar entidades, animadas ou inanimadas, do mundo real ou virtual, que possuam estados.

O conceito de variável é muito importante para linguagens de programação. Para se ter uma idéia, a mudança de linguagem de máquina para `assembly` é marcada primordialmente pela substituição dos endereços numéricos de memória por nomes, tornando os programas muito mais legíveis e fáceis de escrever e manter. Além disso, essa mudança permitiu que se escapasse do problema de endereçamento absoluto de memória,

visto que, a partir daí, o tradutor se responsabiliza por converter os nomes em endereços reais.

O exemplo 4.1 apresenta um trecho de código em C onde uma variável é criada e inicializada com o valor 7. Em seguida, seu valor é incrementado de 3.

```
unsigned x;
x = 7;
x = x + 3;
```

Exemplo 4. 1 - Criação, Inicialização e Atualização de Variável

Durante a execução desse trecho de programa, um número de células suficientes (por exemplo, 2 bytes) para armazenar um valor *do tipo unsigned int* será alocado e o endereço da célula inicial dessa área será associado às referências à variável *x* no código do exemplo. O passo seguinte atribui o valor *7* a essa área de memória. O último passo da execução consiste em obter o valor armazenado nessa área, somá-lo ao valor inteiro *3* e atribuir o resultado àquela área de memória. A figura 4.1 ilustra graficamente esses passos.

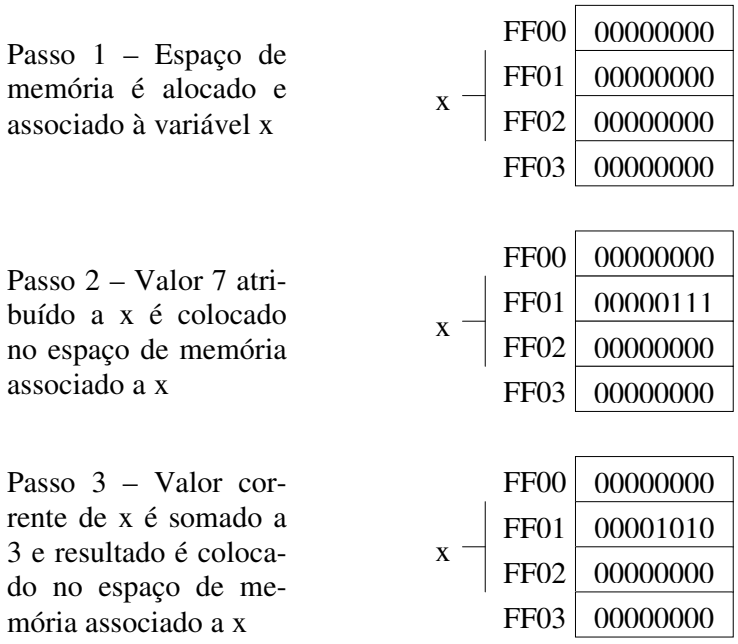

Figura 4. 1 - Operações sobre Variáveis

4.1.1 Propriedades das Variáveis

Uma variável pode ser caracterizada pelo seu nome, endereço, tipo, valor, tempo de vida e escopo de visibilidade.

Normalmente, variáveis possuem um nome definido pelo programador. Esses nomes obedecem a regra de formação de identificadores da LP. No exemplo 4.1, o nome da variável é *x*. Embora programadores frequentemente pensem em variáveis como nomes para locações de memória, existem variáveis que não possuem nomes. Neste caso, elas só podem ser referenciadas através de variáveis ponteiros (apontadores).

O endereço de uma variável é a posição na memória da primeira célula ocupada pela variável. No exemplo representado na figura 4.1, o endereço da variável *x* é FF01. Algumas linguagens, tais como C e C++, permitem que o endereço das variáveis seja acessado livremente. Embora tal prática ofereça flexibilidade ao programador, ela pode ser perigosa e comprometer a segurança e confiabilidade dos programas. O trecho de programa em C, apresentado no exemplo 4.2, mostra como um programa pode obter o endereço de uma variável e usá-lo para acessar uma outra área de memória (a qual não deveria ser acessível pelo programa).

```
char l;
char *m;
m = &l + 10;
printf("&l = %p\nm = %p\n*m = %c\n", &l, m, *m);
```

Exemplo 4. 2 - Acesso a Endereço de Variável

Diz-se que variáveis são sinônimas (`aliases`) quando existe mais de um nome referenciando o mesmo endereço em um mesmo ambiente de amarração. Isto prejudica a legibilidade de programas. O exemplo 4.3, em C++, mostra a ocorrência de sinonímia entre as variáveis *r* e *s*:

```
int r = 0;
int &s = r;
s = 10;
```

Exemplo 4. 3 - Sinonímia

No exemplo 4.3 a variável inteira *r* é criada e inicializada com o valor zero. Em seguida, é criada uma variável *s* do tipo referência associada à variável *r*. A atribuição do valor *10* a *s* também produz como efeito colateral a alteração do valor de *r* para *10*. Tal efeito colateral prejudica a legibilidade do programa porque não fica explícito no código que a variável *r* está sendo atualizada no momento da atribuição a *s*.

O tipo de uma variável determina o conjunto de valores que essa variável pode assumir. A variável *x* do exemplo 4.1 é do tipo *unsigned int*. Ele pode ser definido explicitamente, determinado através de regras sintáticas ou através da semântica de uso da variável. C, por exemplo, requer que se declare o tipo da variável junto com o seu nome. Já versões iniciais de FORTRAN usavam a segunda abordagem, uma vez que determinavam o

tipo da variável pela letra inicial de seu nome. Por exemplo, variáveis iniciadas pelas letras I, J e K são do tipo inteiro. Por sua vez, ML emprega eventualmente a terceira abordagem, uma vez que é possível definir o tipo da variável através das operações que são realizadas sobre ela.

O valor de uma variável corresponde ao seu conteúdo corrente e é compatível com o seu tipo. Na medida que uma variável é atualizada, seu valor é alterado. No exemplo da figura 4.1, a variável x é criada com valor desconhecido (na verdade, o valor da variável x após a criação depende da configuração de bits existente na memória no momento da criação). Posteriormente, o valor 7 é atribuído a x. Por fim, o valor de x é modificado novamente, agora para 10.

O tempo de vida de uma variável corresponde ao período em que ela existe, ou seja, o tempo em que existem células de memória alocadas para a variável. O tempo de vida de variáveis globais corresponde a todo o período de execução do programa onde foi declarada. O tempo de vida de variáveis locais corresponde ao período em que o bloco onde foram declaradas se encontra alocado. Desde que variáveis dinâmicas podem ser criadas e destruídas em qualquer instante do programa, o seu tempo de vida não obedece a nenhuma regra padrão.

Variáveis podem ser transientes (aquelas cujo tempo de vida é limitado pelo tempo de ativação do programa que as criou), ou persistentes (aquelas cujo tempo de vida transcende ao tempo de ativação de um programa particular).

O escopo de visibilidade de uma variável determina o trecho do programa onde essa variável pode ser referenciada. É importante não confundir escopo com tempo de vida. Em algumas situações, uma variável pode estar alocada num determinado ponto de execução de um programa, mas não ser acessível naquele ponto. No exemplo 4.4, em C, todas as duas variáveis *x* existem durante a execução de *main*. Contudo, só a variável local é visível dentro de *main*. Portanto, nesse exemplo, o valor impresso é *10*.

```
int x = 15;
main() {
  int x;
  x = 10;
  printf("x = %d\n", x);
}
```

Exemplo 4. 4 - Tempo de Vida X Escopo de Visibilidade

4.1.2 Atualização de Variáveis Compostas

Variáveis compostas são variáveis cujo tipo é composto. Elas se compõem de outras variáveis.

O conteúdo de variáveis compostas pode ser inicializado, inspecionado e atualizado de forma completa ou seletiva. Enquanto na forma completa todos os componentes da variável composta são inicializados, inspecionados ou atualizados em uma única etapa, na forma seletiva essas operações são aplicadas individualmente sobre os seus componentes. Considere o exemplo 4.5 em C:

```
struct data { int d, m, a; };
struct data f = {7, 9, 1965};
struct data g;
g = f;
g.m = 17;
```

Exemplo 4. 5 - Atualizações Completa e Seletiva em C

A segunda linha de código do exemplo 4.5 mostra a inicialização completa da variável composta *f*. A penúltima linha mostra a atualização completa da variável *g*. Por sua vez, a última linha mostra a atualização seletiva do componente *m* da variável *g*.

LPs normalmente oferecem a operação de atualização seletiva em variáveis compostas. Contudo, o programador deve ter cuidado ao usar esse tipo de operação para ela não provocar problemas de violação na abstração de dados. A última linha do exemplo 4.5 ilustra uma situação na qual isso ocorre. Observe que o valor atribuído ao componente *m* (referente ao mês da data) é o número *17*, o qual não denota mês algum. Tal atribuição quebra a abstração de dados para o tipo *struct data*, uma vez que os valores desse tipo deveriam ser apenas datas.

Claramente, esse tipo de problema também poderia acontecer com a atualização completa. No entanto, como o valor atribuído é uma data completa, a tendência é que esse tipo de erro ocorra com menor frequência.

4.2 Constantes

Certos valores necessitam ser armazenados durante uma computação, contudo não devem ser modificados ao longo do programa. Constantes são entidades de programação usadas para armazenar esses valores. Tal como variáveis, elas também possuem nome, endereço, tipo, valor, tempo de vida e escopo de visibilidade.

Constantes podem ser pré-definidas na LP ou podem ser declaradas pelo usuário. Enquanto constantes pré-definidas são referenciadas através de

símbolos ou identificadores associados aos valores dos tipos de dados da LP, constantes declaradas pelo usuário devem ter um nome associado no programa.

No exemplo 4.6, em C, os valores *3, 'g', "bola"* são exemplos de constantes pré-definidas. Como os valores de constantes não devem ser modificados ao longo do programa, C não permite que se possa obter o endereço de uma constante pré-definida. A linha comentada no exemplo 4.6 provocaria um erro de compilação caso não fosse um comentário. Por outro lado, o compilador C não impede que sejam feitas atribuições para constantes apontadas por ponteiros, como na última linha do exemplo 4.6. Tal permissividade pode gerar um erro de execução ou, ainda mais grave, alterar o valor da constante pré-definida.

```
char x = 'g';
int y = 3;
char* z = "bola";
/*  int *w = &3; */
*z = 'c';
```

Exemplo 4. 6 - Constantes Pré-Definidas em C

Constantes declaradas pelo usuário são úteis para aumentar a legibilidade e a modificabilidade de programas. No exemplo 4.7, em C++, o trecho de programa se torna muito mais legível ao usar o nome *pi* ao invés da constante *3.1416*. Esse mesmo caso ilustra como o uso de constantes declaradas pelo usuário pode aumentar a modificabilidade de programas. Caso se queira aumentar a precisão do valor de *pi* para *3.14159*, basta alterar a linha de declaração da constante e todo o programa será atualizado.

```
const float pi = 3.1416;
float raio, area, perimetro;
raio = 10.32;
area = pi * raio * raio;
perimetro = 2 * pi * raio;
```

Exemplo 4. 7 – Uso de Constantes Declaradas pelo Usuário

Pode-se argumentar que, com os recursos de substituição de cadeias de caracteres oferecidos pelos editores de texto, também seria fácil modificar o programa sem o uso de constantes declaradas pelo usuário. Esse argumento não é totalmente verdadeiro porque poderia ser trabalhoso ter de alterar os vários arquivos que poderiam compor o programa. Além disso, poderiam ocorrer situações onde o valor *3.1416* não fosse referente a constante *pi*. Nesse caso, uma alteração completa no texto do programa poderia introduzir erros difíceis de serem encontrados.

Tentativas de alterar o valor de constantes declaradas pelo usuário em C++ produzem erros de compilação. O mesmo ocorre quando se tenta fazer um ponteiro comum apontar para uma constante. Isso só é permitido quando se indica na declaração do ponteiro que ele apontará para uma constante (assim, o compilador pode impedir a alteração de qualquer valor apontado por esse ponteiro). No exemplo 4.8, as linhas de código comentadas gerariam erros de compilação, uma vez que são tentativas de alterar o valor de uma constante ou de fazer um ponteiro comum apontar para ela. Observe que a última linha do trecho é válida porque *z* é declarado como um ponteiro para uma constante.

```
int* x;
const int y = 3;
const int* z;
//  y = 4;
//  y++;
//  x = &y;
z = &y;
```

Exemplo 4. 8 - Constantes Declaradas em C++

ANSI C também adota a palavra reservada *const* introduzida por C++, embora com um intuito diferenciado. A finalidade de *const* em C é anunciar objetos que **possam** ser colocados em memória somente de leitura, e talvez aumentar oportunidades de otimização [KERNIGHAN & RITCHIE, 1989]. Segundo Kernighan e Ritchie, exceto por ter que diagnosticar tentativas explícitas de alterar objetos *const*, um compilador pode ignorar este qualificador. Talvez seja por isso que alguns compiladores C não tratem as linhas comentadas do exemplo 4.8 como erros, apresentando apenas mensagens de aviso a respeito da tentativa de alteração de constantes.

4.3 Armazenamento de Variáveis e Constantes

Enquanto variáveis e constantes transientes apenas necessitam ser armazenadas na memória principal do computador durante a execução do programa que as utiliza, as persistentes também necessitam ser armazenadas em memória secundária para serem utilizadas por outros programas ou por execuções futuras do programa que as criou.

Entender como variáveis e constantes são armazenadas na memória principal e secundária pode ser de grande valia para um programador. Nessa seção, apresenta-se uma visão geral sobre como o armazenamento em memória principal e em memória secundária pode ser realizado.

4.3.1 Armazenamento na Memória Principal

A memória principal de um computador consiste de uma enorme sequência contígua e finita de bits. Contudo, a menor unidade de memória que pode ser diretamente endereçada corresponde normalmente ao tamanho da palavra do computador (8 bits, 16 bits, 32 bits, etc.). Pode-se imaginar, então, a memória como sendo um vetor de tamanho finito cujos elementos correspondem ao tamanho da palavra do computador, o qual chamaremos aqui de vetor de memória.

Quando da execução de um programa, além do código executável, deve ser reservado um espaço na memória para armazenar os dados manipulados pelo programa. Uma forma de se fazer isso seria identificar quais variáveis são utilizadas no programa, qual o seu tamanho e reservar um espaço na memória correspondente a soma dos tamanhos de todas as variáveis. A primeira linguagem de programação de alto nível, FORTRAN, adotou essa abordagem. Contudo, essa alternativa apresenta alguns problemas.

Muitas vezes, a memória disponível acaba sendo mal utilizada. Isso ocorre porque o tamanho das variáveis necessárias pode variar de execução para execução de programa. Tipicamente, a solução adotada por programadores nessas linguagens é usar o maior tamanho possível da variável. Contudo, falhas eventuais nessas estimativas podem ocasionar erros inconvenientes em tempo de execução. Além disso, essa solução provoca desperdício de memória por se reservar mais espaço para as variáveis do que o necessário no caso geral.

Outra causa de desperdício de memória nessa abordagem é a necessidade de alocar espaço para as variáveis locais de todos os subprogramas. Como na execução de um programa é muito freqüente se invocar apenas um subconjunto dos subprogramas, todo o espaço ocupado pelas variáveis locais dos subprogramas não utilizados também é reservada desnecessariamente.

Por fim, essa abordagem impede a implementação de subprogramas recursivos, isto é, subprogramas que invocam a si mesmos durante sua própria execução. É importante perceber a necessidade de existir um conjunto de variáveis locais para cada subprograma ativo na execução do programa. Como não se pode saber, antecipadamente, quantas chamadas de subprogramas recursivos serão feitas durante a execução de um programa, a alocação de memória para as variáveis locais desses subprogramas só pode ser realizada em tempo de execução.

A solução para lidar com esses problemas é permitir a alocação dinâmica de memória, isto é, somente reservar espaço para as variáveis quando o

tamanho necessário for conhecido e quando elas forem efetivamente necessárias. Isto pode ocorrer somente durante a execução do programa. Para realizar alocação dinâmica, os programas devem implementar ou usar um mecanismo de gerenciamento dinâmico de memória.

Uma possibilidade para implementação desse tipo de mecanismo é alocar, na medida que se constate a necessidade e se identifique o tamanho de uma variável, o número de palavras correspondentes ao seu tamanho no vetor de memória. Para uma melhor utilização da memória, as diferentes variáveis seriam alocadas contiguamente no vetor.

Contudo, essa solução apresenta alguns problemas. Pode ser necessário aumentar o tamanho das variáveis durante uma mesma execução de um programa. Além disso, variáveis podem ser úteis em apenas um intervalo da execução do programa.

A memória se esgotaria rapidamente ao se usar a política de alocar as novas áreas necessárias sempre após o último bloco alocado e não reutilizar as áreas desalocadas. Por outro lado, utilizar as áreas desalocadas demandaria exclusões e movimentações de elementos no vetor. Tais operações demandam bastante esforço computacional e diminuem a eficiência da execução. Além disso, elas requeririam que as referências às variáveis no código executável fossem atualizadas com o seu endereço absoluto sempre que elas fossem alocadas ou realocadas. Essas operações também prejudicariam a execução eficiente do programa.

A forma mais comum de implementação dos mecanismos de gerenciamento dinâmico de memória de execução de programas subdivide a memória de dados do programa em duas áreas: a área de pilha e a área de monte (ou `heap`). LPs que implementam esse tipo de gerenciamento de memória são conhecidas como ALGOL`-like`.

A pilha é utilizada para resolver os problemas de alocação de subprogramas recursivos e de alocação dinâmica dos dados locais de blocos e subprogramas. Toda vez que ocorre uma entrada em um bloco ou uma invocação de um subprograma, reserva-se no topo da pilha espaço suficiente para armazenar as variáveis locais (e os parâmetros) do bloco ou do subprograma. Ao se encerrar o bloco ou subprograma, o espaço que lhe foi destinado no topo da pilha é liberado. Tal procedimento só é possível porque o último bloco ou subprograma ativado é sempre o primeiro a ser encerrado.

Com essa política, as variáveis locais e parâmetros só ficam alocados enquanto podem ser necessárias, isto é, durante a execução do bloco ou subprograma. Por se tratar de uma pilha, a liberação da área de memória do subprograma envolve apenas o deslocamento do marcador do topo da

pilha, não requerendo movimentações de elementos no vetor de memória. Isso torna esse mecanismo bastante eficiente.

A figura 4.2 apresenta um exemplo de uso da pilha. Nesse exemplo, o programa de nome *p* possui duas variáveis inteiras chamadas *a* e *b*, e o subprograma *f* possui dois parâmetros *x* e *y* e uma variável local *z*, todos inteiros. O momento da execução retratado na figura ocorre durante a chamada de *f* pelo programa *p*.

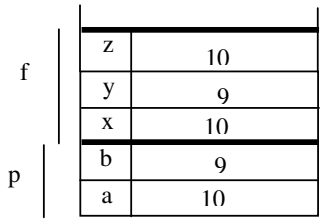

Figura 4. 2 - Estado da Área de Pilha

Contudo, o problema das variáveis cujo tamanho se modifica em tempo de execução ainda permanece. Alocar uma variável desse tipo na pilha provocaria um grande problema quando fosse necessário aumentar o seu tamanho. Seria preciso retirar todas as variáveis localizadas acima dela na pilha, copiando-as em outra área de memória, aumentar o espaço reservado para a variável e, posteriormente, empilhar as variáveis retiradas na mesma ordem em que haviam sido alocadas originalmente. Isto seria claramente ineficiente.

A solução para o problema do aumento dinâmico de variáveis é o uso da área de monte. O monte é uma área de memória que não obedece a uma regra bem estabelecida como a pilha. Se existe necessidade de armazenar uma variável, reserva-se um espaço no monte onde exista espaço contíguo suficiente para alocar a variável.

Se, em algum momento, esse tamanho se mostra insuficiente, pode-se adotar duas alternativas. A primeira alternativa consiste em alocar um novo espaço de memória, agora de tamanho suficiente, e copiar os dados da área anteriormente alocada para esse novo espaço. A segunda alternativa é alocar um novo espaço de memória suficiente apenas para completar o tamanho requerido e fazer uma ligação da área previamente existente com a nova área alocada.

A figura 4.3 ilustra essas duas alternativas. A primeira alternativa é ilustrada pela variável *x*. Observe que *x* inicialmente é uma variável alocada no monte com valor *10*. Quando há necessidade de aumentar essa variável, um novo espaço no monte é alocado, o valor original de *x* é copiado, *x* passa a referenciar essa nova área e o espaço ocupado anteriormente por ela é desalocado. A segunda alternativa é ilustrada na figura 4.3 pela variável *y*. Essa variável é aumentada alocando-se uma nova área de memória

e ligando o final da área corrente ao início dessa nova área através do uso de um ponteiro.

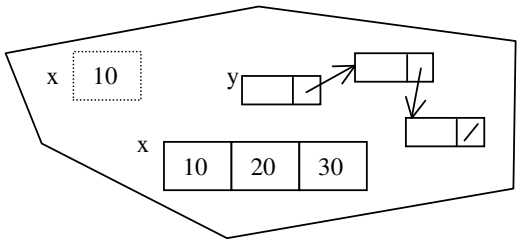

Figura 4. 3 - Área de Monte

Note que as duas alternativas apresentam vantagens e desvantagens. Enquanto a alternativa de cópia torna o redimensionamento da variável uma operação de alto custo, ela não causa qualquer prejuízo no acesso aos componentes da variável. Já a alternativa de completar a variável não causa maiores custos para redimensionar a variável, mas torna mais lento o acesso aos seus componentes.

Você pode estar se perguntando o porquê de dividir a memória em pilha e monte. Por que não usar toda a memória apenas como se fosse um monte? A resposta para essa pergunta é que isso não seria tão eficiente.

A alocação dinâmica de memória no monte demanda que subáreas da memória sejam alocadas e desalocadas ao longo da execução do programa. Como não existe uma regra bem estabelecida sobre quando se deve alocar ou desalocar uma variável, é necessário possuir estruturas de dados adicionais responsáveis por controlar as áreas livres e já alocadas. Sempre que uma nova área de memória é necessária, percorre-se a lista de áreas livres procurando uma área com espaço suficiente para armazenar a variável. Toda vez que uma área de memória é alocada, ela é retirada da lista de áreas livres e colocada na lista de áreas alocadas. Toda vez que uma área é desalocada, a movimentação inversa é realizada.

Esse tipo de controle demanda um esforço computacional muito maior que o necessário para a alocação dinâmica via pilha. Portanto, a combinação de pilha com monte visa obter o melhor dos dois modelos. Quando uma variável não necessita mudar seu tamanho dinamicamente ela é alocada na pilha para se obter mais eficiência nas operações de alocação e desalocação. Por outro lado, quando isso é necessário, utiliza-se o monte.

A figura 4.4 mostra um exemplo do uso de gerenciamento de memória dinâmica com áreas de pilha e monte. Note que, para se ter acesso às variáveis na área do monte, é necessário existir uma variável ponteiro ou referência armazenada na pilha. Na figura 4.4, a variável *p* exerce esse papel.

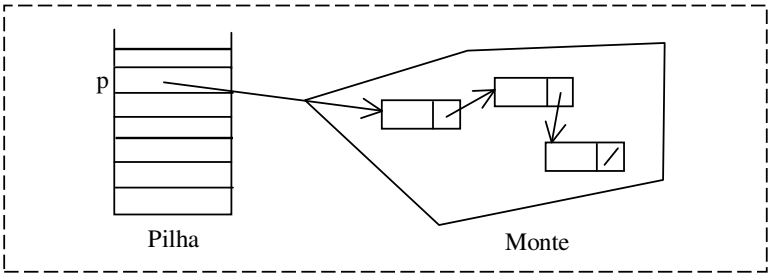

Figura 4. 4 - Áreas de Pilha e Monte

4.3.1.1 Pilha de Registros de Ativação

Nessa seção apresenta-se um modelo simplificado de como podem ser implementadas as áreas de pilha de linguagens ALGOL-`like`. Nesse modelo, a pilha é composta por registros de ativação (RA) de subprogramas, os quais armazenam vários tipos de informações e dados relativos a uma ativação de um subprograma. A figura 4.5 apresenta um esquema de um registro de ativação:

constantes locais
variáveis locais
parâmetros
link dinâmico
link estático
endereço de retorno

Figura 4. 5 - Esquema de Registro de Ativação de Subprogramas

O campo **constantes locais** armazena os valores das constantes declaradas localmente. O campo **variáveis locais** armazena as variáveis declaradas localmente. O campo **parâmetros** armazena os parâmetros do subprograma. O campo **link dinâmico** é uma referência para a base do registro de ativação anterior. Ele indica o endereço base a ser usado para acessar as informações do RA após o fim do subprograma corrente. O campo **link estático** é uma referência para a base do RA do subprograma imediatamente externo ao subprograma do registro de ativação corrente, isto é, o do subprograma onde ele foi definido. O campo **endereço de retorno** armazena o endereço da próxima instrução a ser executada ao final da execução do subprograma.

Considere o exemplo 4.9 na linguagem ADA:

```
procedure Principal is
    x: integer;
    procedure Sub1 is
        a, b, c: integer;
        procedure Sub2 is
            a, d: integer;
            begin  --Sub2
                a := b + c;
                d := 8;                    -- 1
                if a < d then
                  b := 6;
                  Sub2;                    -- 2
                end if;
            end Sub2;
        procedure Sub3 (x: integer) is
            b, e: integer;
            procedure Sub4 is
                c, e: integer;
                begin --Sub4
                    c := 6; e:= 7;
                    Sub2;                  -- 3
                    e := b + a;
                end Sub4;
            begin  --Sub3
                b := 4; e := 5;
                Sub4;                      -- 4
                b := a + b + e;
            end Sub3;
        begin --Sub1
            a := 1; b:= 2; c:= 3;
            Sub3(19);                      -- 5
        end Sub1;
    begin --Principal
        x := 0;
        Sub1;                              -- 6
    end Principal;
```

Exemplo 4. 9 - Registros de Ativação e Chamadas de Subprogramas em ADA

No exemplo 4.9 o programa *Principal* chama o subprograma *Sub1*, que chama o subprograma *Sub3*, que chama o subprograma *Sub4*, que chama o subprograma *Sub2*, o qual se chama recursivamente. A figura 4.6 mostra o estado da pilha de registros de ativação quando o programa passa pela linha marcada pelo número comentado -- *1*. Os demais números co-

mentados ao final de algumas linhas (por exemplo, -- *4*) representam endereços de retorno dos subprogramas.

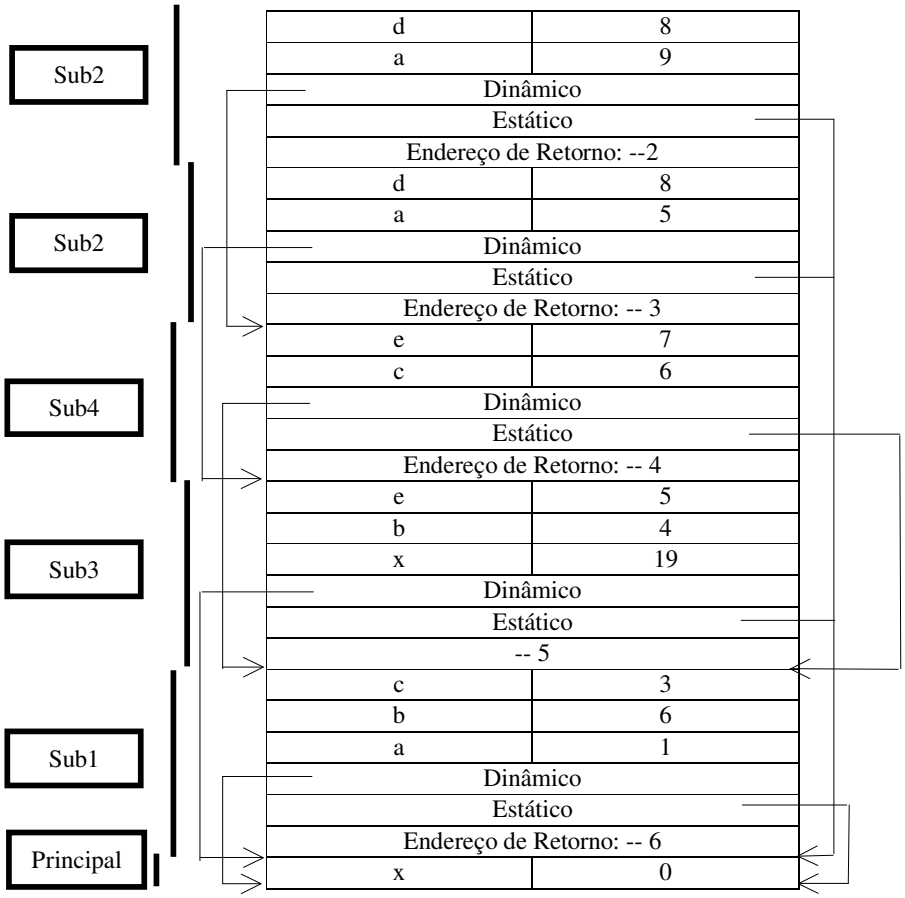

Figura 4. 6 - Estado de Pilha de Registros de Ativação

Note que o link estático de *Sub2* e *Sub3* aponta para a base do registro de ativação de *Sub1*, o link estático de *Sub4* aponta para a base do registro de *Sub3* e que o link estático de *Sub1* aponta para a base do registro de *Principal*. Perceba ainda que, para obter o valor de *a* no código de *Sub4* é necessário seguir dois passos na cadeia de links estáticos até encontrá-lo em *Sub1*. Observe por fim que o link dinâmico sempre aponta para a base do registro de ativação anterior. Isso ocorre até quando existe uma chamada recursiva. Por exemplo, o link dinâmico do RA correspondente a chamada recursiva de *Sub2* é o RA da chamada anterior de *Sub2*.

4.3.1.2 Gerenciamento do Monte

O controle da área de memória no monte é um pouco mais complexo (e computacionalmente menos eficiente) que na pilha, uma vez que não existe uma regra geral definindo onde a área de memória requisitada deve ser alocada nem em qual momento essa área deve ser desalocada. Portan-

to, implementações diferentes de LPs podem seguir políticas significativamente distintas para a realização dessas operações.

Uma forma possível de gerenciar a memória do monte envolve a utilização de duas listas de controle contendo informações a respeito dos espaços de memória livres e ocupados. A LED (Lista de Espaços Disponíveis) contém, em cada um de seus elementos, informações (local de início e tamanho) sobre cada bloco de memória que se encontra livre para ser alocado. Por sua vez, a LEO (Lista de Espaços Ocupados), contém, em cada um de seus elementos, informações (local de início e tamanho) sobre cada bloco de memória que se encontra ocupado. Quando o programa faz uma requisição para alocação de memória, percorre-se a LED em busca de um elemento cuja área de memória atenda essa requisição. Quando é encontrado, ele é retirado da LED e transferido para a LEO. Quando o programa faz uma requisição para desalocação de memória, busca-se o elemento correspondente na LEO. Quando encontrado, ele é removido da LEO e transferido para a LED.

Diversas políticas de alocação de memória podem ser adotadas para escolher qual elemento da LED será alocado dentre todos que satisfazem a requisição de memória. Pode-se escolher o primeiro encontrado, ou o de tamanho mais próximo, ou o de maior tamanho. Essa escolha determinará a ocorrência mais rápida ou não do problema de fragmentação da memória, isto é, a situação na qual os espaços de memória disponíveis na LED se tornam tão pequenos que não mais atendem às requisições. Nesses casos, haverá necessidade de alocação de memória adicional para o programa ou da execução de um processo de desfragmentação.

Enquanto o momento de alocação de memória é sempre bem definido pelo programa (isto é, o momento no qual é feita uma operação de criação de variável ou constante dinâmica), o momento de desalocação depende da linguagem de programação. Em algumas LPs, o controle do momento de desalocação é feito pelo programador (por exemplo, em C, C++ e PASCAL). Nesses casos, o programa deve conter instruções explícitas para chamar a operação de desalocação. Em outras LPs, o próprio sistema de implementação da LP define o momento de desalocação. Nesse caso, o programa não contém operações explícitas de desalocação e utiliza-se um mecanismo chamado de coletor de lixo. Esse é o caso de JAVA.

A primeira solução permite a construção de programas mais eficientes (o construtor do programa é quem diz o momento no qual se deve desalocar a memória). Contudo, essa solução deixa sob responsabilidade do programador o processo de desalocação de memória, sendo menos confiável e muito mais trabalhosa. Erros frequentes de programação associados a essa solução são a falta de desalocação, criando objetos pendentes e pro-

vocando vazamentos de memória, e a desalocação indevida de memória, deixando-se referências pendentes e provocando erros de acesso indevido.

Por sua vez, a segunda solução envolve a utilização de um coletor de lixo que, de tempos em tempos, entra em ação transformando as células de memória não mais utilizadas em células livres disponíveis para alocação. Se isso torna a vida do programador muito mais fácil (ele normalmente não precisa se preocupar com a gerência da memória), essa solução faz com que programas sejam normalmente executados menos eficientemente, além de tornar mais complexa a implementação da LP.

A perda de eficiência computacional ocorre porque um gerenciador genérico de memória não pode tirar vantagem de situações específicas de modo a tornar o programa mais eficiente. Normalmente, o coletor de lixo tem de percorrer todos os ponteiros das estruturas de dados do programa marcando qual a memória utilizada. Isso demora muito mais do que desalocar explicitamente a memória na medida que ela não é mais necessária.

Outro problema com o uso de coletor de lixo é a falta de controle sobre o momento em que ele vai atuar (frequentemente, isso ocorre quando falta memória livre). Tal fato pode ocorrer em um momento no qual a eficiência de execução seja mais crítica. O risco disso ocorrer pode ser reduzido quando o gerenciamento é deixado na mão do programador.

O uso do coletor de lixo em JAVA permite a adoção de uma política de alocação de memória no monte quase tão eficiente quanto a alocação de memória na pilha. Sempre que ocorre uma requisição de alocação de memória dinâmica no programa, aloca-se uma área de memória do tamanho requisitado imediatamente após o final da última área de memória alocada pelo programa. Assim, não é necessário manter listas de espaços disponíveis e ocupados, embora se chegue ao final da área de monte mais rapidamente. Nesse momento, o coletor de lixo entra em ação removendo basicamente as áreas não mais utilizadas e realocando de maneira contígua as áreas ocupadas de memória para o início da área de monte.

4.3.2 Armazenamento na Memória Secundária

Variáveis e constantes necessitam ser armazenadas na memória secundária e também precisam ser recuperadas de lá. O termo “persistência de dados” é usado para designar e agrupar temas relacionados com o armazenamento e recuperação de dados em memória secundária.

A existência de uma área da ciência da computação (Banco de Dados) dedicada primordialmente ao estudo e investigação de técnicas para modelagem e implementação dos processos de persistência de dados ilustra a sua relevância nos sistemas de computação.

As Linguagens de Programação também têm se interessado em incorporar mecanismos para facilitar a implementação de programas nos quais ocorram esses processos. A pretensão maior e final é oferecer LPs nas quais não exista qualquer distinção entre entidades transientes e persistentes.

4.3.2.1 Arquivos

Um exemplo comum de variáveis persistentes em LPs são os arquivos, os quais são responsáveis por armazenar valores em dispositivos de memória secundária (discos, fitas, etc.). Desse modo, esses valores podem continuar existindo independentemente da execução de um programa, o que caracteriza essas variáveis como persistentes.

Normalmente, os valores dos arquivos são armazenados contigüamente na memória secundária. A maior parte das LPs restringe os modos de acesso e atualização de arquivos. Em geral, é proibida a atribuição de arquivos completos.

Arquivos podem ser seriais ou diretos. A distinção entre arquivos seriais e diretos está fundamentalmente na forma como os componentes desses arquivos são acessados. Os componentes dos arquivos seriais são acessados sequencialmente, isto é, para acessar o componente de ordem **n+1** é estritamente necessário acessar previamente o componente de ordem **n**. Componentes de arquivos diretos não precisam obedecer esta regra, podendo ser acessados a qualquer momento. A implementação de arquivos diretos pode envolver o tratamento do arquivo como um vetor de componentes ou pode ser feita através do uso de uma tabela indexadora dos componentes do arquivo.

LPs normalmente requerem que programas realizem uma operação de abertura do arquivo antes de acessar seus componentes e uma operação de fechamento do arquivo após o seu uso. Além disso, por questões de eficiência de acesso, esses programas necessitam converter os dados em memória para uma sequência de bytes na memória secundária. Considere o exemplo 4.10, em C:

```
#include <stdio.h>
struct data {
  int d, m, a;
};
struct data d = {7, 9, 1999};
struct data e;
main() {
  FILE *p;
  char str[30];
  printf("\n\n Entre com o nome do arquivo:\n");
```

```
  gets(str);
  if (!(p = fopen(str,"w"))) {
    printf ("Erro! Impossivel abrir o arquivo!\n"); exit(1);
  }
  fwrite (&d, sizeof(struct data), 1, p);
  fclose(p);
  p = fopen(str,"r");
  fread (&e, sizeof(struct data), 1, p);
  fclose (p);
  printf ("%d/%d/%d\n", e.a, e.m, e.d);
}
```

Exemplo 4. 10 - Persistência através de Arquivos em C

Observe que a operação *fwrite* converte a variável transiente *d* numa cadeia de bytes para salvá-la num arquivo binário e a operação *fread* converte uma cadeia de bytes do arquivo na variável transiente *e*. Observe também que para acessar individualmente o valor do campo *m*, armazenado na variável persistente, de maneira independente dos demais campos, seria necessário realizar operações de posicionamento de cursor no arquivo. Isto contrasta com a forma direta de acesso fornecida para as variáveis transientes *d* e *e*.

Outra forma comum de se obter persistência de dados é através de interfaces com gerenciadores de bancos de dados. Nesse caso, os valores de variáveis transientes precisam ser transformados em registros de tabelas em um banco de dados através do uso de comandos em linguagens como SQL. Nesse caso, a linguagem de programação não emprega ou oferece um tipo de dados persistente. O programador deve escrever sentenças compostas por termos da linguagem de gerenciamento de banco de dados e por valores das variáveis transientes. O gerenciador executa essas sentenças para armazenar e recuperar os valores das variáveis persistentes.

O exemplo 4.11 mostra um trecho de um programa em JAVA no qual se utiliza um comando em SQL para recuperar informações de pessoas (nome, idade e data de aniversário) armazenadas em um banco de dados.

```
int idadeMaxima = 50;
Statement comando = conexao.createStatement();
ResultSet resultados = comando.executeQuery (
        "SELECT nome, idade, nascimento FROM pessoa " +
        "WHERE idade < " + idadeMaxima);
);
while (resultados.next()) {
   String nome = resultados.getString("nome");
   int idade = resultados.getInt("idade");
```

```
    Date nascimento = resultados.getDate("nascimento");
}
```

Exemplo 4. 11 - Recuperação de Dados Persistentes Através de Gerenciador de Banco de Dados

A terceira linha do trecho de código do exemplo 4.11 é onde se envia um comando em SQL para o gerenciador de banco de dados. Observe a utilização da variável *idadeMaxima* no comando SQL para somente selecionar pessoas de idade inferior ao valor desta variável. Os valores retornados pela consulta ao banco de dados são armazenados na variável *resultados*.

4.3.2.2 Persistência Ortogonal

A maioria das LPs fornece diferentes tipos para variáveis persistentes e transientes. Em C, por exemplo, uma variável transiente pode ser de qualquer tipo, mas uma variável persistente deve ser do tipo arquivo binário ou texto.

Porque não permitir que variáveis persistentes sejam de tipos primitivos, registros, vetores ou variáveis anônimas pertencentes a estruturas dinâmicas? Idealmente, todos os tipos deveriam ser permitidos tanto para variáveis transientes quanto para persistentes. Além disso, não deveria haver qualquer distinção no código que lida com variáveis persistentes daquele que lida com variáveis transientes. A identificação de persistência se daria através da percepção da continuidade da necessidade de utilização da variável.

Uma LP aplicando esses princípios se qualifica como ortogonalmente persistente [ATKINSON & MORRISON, 1995]. Ela seria simplificada por não ter tipos especiais e comandos ou procedimentos para entrada e saída. O programador estaria liberado de ficar convertendo dados de um tipo de dados persistente para um transiente, ao necessitar usar os dados, ou vice-versa, ao necessitar armazenar os dados.

Acredita-se que cerca de 30% do código de uma aplicação é destinado a essas conversões [JORDAN & ATKINSON, 1998]. Isso oferece uma idéia do quão impactante podem ser esses processos de conversão entre variáveis persistentes e transientes no projeto, programação e desempenho das aplicações.

Contudo, a implementação de sistemas ortogonalmente persistentes é um grande desafio. Dentre as principais dificuldades podem ser citadas a perda eventual de eficiência computacional, a necessidade de recuperação de dados quando a execução do programa é interrompida inesperadamente e a demanda por integração dos múltiplos e diversos ambientes envolvidos nas aplicações, tais como sistemas operacionais, bancos de dados e lin-

guagens de programação. Por isso, sistemas ortogonalmente persistentes ainda são raros.

4.3.2.3 Serialização

Alguns aspectos importantes necessitam ser considerados ao se implementar a persistência de variáveis e constantes:

1. A variável (ou constante) transiente deve ser convertida de sua representação na memória primária para uma sequência de bytes na memória secundária.
2. Como os valores absolutos dos ponteiros na memória não terão significado na próxima vez que o programa for executado, esses ponteiros devem ser relativizados quando armazenados.
3. As variáveis (ou constantes) anônimas apontadas por valores ponteiros também devem ser armazenadas e recuperadas.
4. Ao restaurar uma variável (ou constante) da memória secundária, os ponteiros devem ser ajustados de modo a respeitar as relações existentes anteriormente entre as variáveis anônimas.

Como uma variável tem de ser convertida do seu `layout` na memória para uma representação serial em memória secundária e vice-versa, esses processos são chamados de serialização (escrita da variável no disco) e desserialização (leitura da variável no disco).

Embora seja extremamente conveniente suportar esses processos diretamente, eles podem sobrecarregar a LP e sua implementação. C++ é uma LP que opta por não fornecê-los diretamente. Ela deixa para o programador implementar funções membro especiais nas suas classes para suportar a serialização e desserialização. Assim, o criador da classe, que é quem melhor conhece como os componentes da classe devem ser lidos ou escritos, é responsável por programar esses processos. Contudo, isso requer um esforço muito maior de programação.

JAVA possui um mecanismo de serialização de objetos que possibilita transformar qualquer objeto que implemente a interface *Serializable* numa sequência de bytes que pode ser posteriormente restaurada no objeto original. Isto é verdade inclusive em um ambiente de rede de computadores, o que significa que o mecanismo de serialização compensa automaticamente as diferenças entre sistemas operacionais, ou seja, é possível criar um objeto em uma máquina Windows, serializá-lo e enviá-lo pela rede para uma máquina Unix onde ele será reconstruído corretamente. O programador não precisa se preocupar com a forma de representação nas diferentes máquinas, nem com a ordenação de bytes e outros detalhes.

Embora o mecanismo de serialização de objetos ainda não seja o ideal para persistência (existem esforços no sentido de tornar JAVA ortogonalmente persistente [JORDAN & ATKINSON, 1998]), ele permite não apenas salvar a imagem do objeto mas também seguir e salvar todos os objetos referenciados nesse objeto e nos objetos referenciados por ele. Essa característica remove do programador todo o trabalho necessário para manter o esquema de serialização seguindo todas as conexões de objetos, enxugando significativamente o código dos programas.

O exemplo 4.12 ilustra o uso do mecanismo de serialização em JAVA. Nesse exemplo é criada uma lista *w* contendo os seis primeiros números ímpares positivos, a qual é armazenada no arquivo *impares.dat*. Posteriormente, essa lista é lida do arquivo na variável *z*.

```
import java.io.*;
public class Impares implements Serializable {
  private static int j = 1;
  private int i = j;
  private  Impares prox;
 Impares(int n) {
     j = j + 2;
     if(--n > 0)
         prox = new Impares(n);
  }
 Impares() {
     j = j + 2;
     prox = new Impares(9);
  }
 public String toString() {
     String s = ""  +  i;
     if(prox != null)
        s += " : " + prox.toString();
     return s;
 }
 public static void main(String[] args) {
     Impares w = new Impares(6);
     System.out.println("w = " + w);
     try {
        ObjectOutputStream out =
            new ObjectOutputStream(
               new FileOutputStream("impares.dat"));
        out.writeObject("Armazena Impares");
        out.writeObject(w);
        out.close();
```

```
            ObjectInputStream in =
                new ObjectInputStream(
                    new FileInputStream("impares.dat"));
            String s = (String)in.readObject();
            Impares z = (Impares)in.readObject();
            System.out.println(s + ", z = " + z);
            in.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Exemplo 4. 12 - Serialização em JAVA

É importante observar que a operação

out.writeObject(w)

é responsável por gravar a lista *w* no arquivo. Note que toda a lista de ímpares é gravada sem que seja necessário criar uma operação para percorrer seus elementos e gravá-los um a um. De modo similar, a operação

Impares z = (Impares)in.readObject()

é responsável por resgatar o conteúdo da lista no arquivo e criar a lista *z* sem que seja necessário criar separadamente cada elemento dessa lista. Observe ainda que, para conseguir esse tipo de comportamento, basta declarar que a classe *Impares* implementa a interface *Serializable*.

4.4 Considerações Finais

Nesse capítulo destacou-se a importância do papel de variáveis e constantes em LPs. Discutiu-se, em particular, como o armazenamento de variáveis e constantes pode ser feito na memória principal e secundária do computador.

Algumas questões chave sobre esse tema devem ser esclarecidas ao se estudar uma nova LP. Por exemplo, é importante saber se ela permite o acesso direto ao endereço das variáveis e constantes, se permite ao programador definir suas próprias constantes, se as constantes definidas se comportam efetivamente como uma constante pré-existente na LP.

Especial atenção deve ser tomada no estudo do gerenciamento de memória e persistência de dados. É necessário saber se a responsabilidade pela desalocação de memória é do programador ou se a LP possui coletor de lixo. Em caso do programador ser o responsável, deve-se conhecer de que forma e em que escopo do programa essa operação deve ser realizada.

Com relação à persistência, é fundamental conhecer o poder de persistência oferecido pela LP. Assim, deve-se saber quais os tipos de dados que admitem persistência, como se pode denotar e manipular um tipo persistente, e como é a implementação do mecanismo de persistência.

4.5 Exercícios

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.

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.

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");
}
```

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.

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.

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.

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 links estáticos na figura 4.6.

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.

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.

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 imple-

mentação 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?

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.