Capítulo VI – Modularização

"Isto
é só a
ponta de um
iceberg submerso.
O verdadeiro poema
está embaixo deste verso."

Luis Fernando Verissimo

Na época em que surgiram os primeiros computadores, os programadores eram forçados a buscar o máximo de eficiência nos seus programas por causa das limitações dos recursos computacionais. Os primeiros programas consistiam de um único bloco monolítico de código pois a divisão de programas em vários blocos exige um sistema de gerenciamento de memória e consome mais recursos. O programador devia utilizar um pequeno número de variáveis, as quais cumpriam diferentes papéis ao longo do programa, com o intuito de economizar memória. Ao invés de se buscar ordenação no fluxo de controle dos programas, os programadores tinham de definir o fluxo mais eficiente, o que normalmente exigia o uso extensivo e indiscriminado de comandos de desvio incondicional (tal como o *goto*).

Enquanto as aplicações desenvolvidas eram de pequeno porte, isto é, envolviam um número relativamente pequeno de linhas de código e realizavam poucas tarefas (se comparadas às aplicações atuais), essa forma de desenvolver programas era satisfatória. Contudo, na medida que novos recursos computacionais se tornavam disponíveis e que se vislumbrava o grande potencial dos computadores para a construção de aplicações mais complexas, foi se constatando que a programação em bloco monolítico tornava pouco viável a construção de grandes sistemas de programação.

Tipicamente, os programas eram escritos por um único programador, pois não havia como dividir o programa em trechos relativamente independentes que pudessem ser construídos por programadores diferentes. Além de impedir a divisão do trabalho, o fato das mesmas variáveis serem usadas em todo o programa e o fluxo de controle poder ir de um ponto a qualquer outro do programa provocava a ocorrência de um grande número de erros na programação, atrasando o desenvolvimento do sistema.

Com a maior disponibilidade de recursos computacionais, o principal fator inibidor do desenvolvimento e disseminação de aplicações de grande porte passa a ser a eficiência de programação, isto é, o tempo de trabalho

dos programadores. Era preciso avançar um pouco mais no processo de desenvolvimento de programas.

Para tornar mais eficiente o trabalho dos programadores, foi identificada a necessidade de se apoiar o processo de resolução de problemas complexos. Apoiar esse processo envolve tanto a elaboração de técnicas para permitir o programador resolver um problema complexo totalmente novo (criação de soluções) quanto para permitir reusar soluções criadas para um problema em um novo problema semelhante (reuso de trabalho).

Em geral, a técnica fundamental utilizada para resolver problemas complexos consiste no uso da estratégia de "dividir para conquistar". Usar essa estratégia implica resolver um grande problema dividindo-o em vários subproblemas mais gerenciáveis e resolvê-los independentemente. Essa estratégia aumenta ainda a potencialidade de reuso de trabalho, uma vez que se pode disponibilizar para reuso as soluções dos subproblemas ao invés de disponibilizar apenas uma solução completa do problema.

A implementação da estratégia de "dividir para conquistar" em LPs é realizada através de técnicas de modularização. Além de apoiar a resolução de problemas complexos, essas técnicas tornam mais fácil o entendimento dos programas e viabilizam o reuso de código. Basicamente, as técnicas de modularização promovem a segmentação do programa em módulos e o encapsulamento de dados. Encapsular dados significa agrupar dados e processos logicamente relacionados em uma mesma entidade de computação.

Abstração é um conceito fundamental para se atingir uma boa modularização. Abstração é o processo no qual se seleciona, do contexto do problema a resolver, o que é importante de ser representado para resolver o problema. A abstração possibilita o trabalho em níveis no desenvolvimento de programas. Em um nível inferior, identificam-se as abstrações importantes no contexto do problema e implementam-se módulos correspondentes a essas abstrações. Em um nível superior, utiliza-se as abstrações do nível inferior para resolver um problema sem que se necessite preocupar com os detalhes da implementação dessas abstrações.

Inicialmente, esse capítulo discute em maior detalhe o conceito de abstração e o seu papel em LPs. Atenção especial é dada a relação desse conceito com a idéia de modularização. Em seguida, apresenta-se e discute-se técnicas de modularização oferecidas por LPs para permitir a implementação de abstrações. Por fim, discute-se algumas formas de operacionalização dessas técnicas em programas cujo código fonte é dividido em múltiplos arquivos.

6.1 Abstrações

Abstração é um processo importante em todas as áreas do conhecimento. Abstração é fundamental para o raciocínio e resolução de problemas em geral porque é impossível representar qualquer fenômeno da realidade em toda a sua riqueza de detalhes. Em outras palavras, devemos sempre nos concentrar nos aspectos essenciais do problema e ignorar os aspectos irrelevantes para a resolução do problema em questão. Caso contrário, a tarefa de representar a realidade se torna impraticável e intratável.

Como não poderia deixar de ser, o conceito de abstração é amplamente disseminado dentro da área de computação. Ele tem sido empregado frequentemente no sentido de facilitar a resolução de problemas por computador através do provimento de mecanismos que tornem mais simples a sua operação e programação. Alguns exemplos do uso do conceito de abstração em computação são:

- Ao se utilizar os comandos de um sistema operacional, está-se abstraindo do uso de instruções específicas de hardware para controlar e manipular o computador.
- Ao se usar instruções de uma linguagem de baixo nível (`assembly`), está-se abstraindo do uso de instruções binárias para a programação de computadores.
- Ao se usar instruções de uma linguagem de programação de alto nível, como PASCAL ou C, está-se abstraindo do uso de instruções em linguagem `assembly`.
- Ao se usar um programa de reservas de passagens, está-se abstraindo do conjunto de instruções que descrevem como se realiza o processo de reserva.

Em LPs, especificamente, o conceito de abstração é utilizado segundo duas perspectivas:

a) A LP funciona como uma abstração sobre o hardware, isto é, os programadores podem entender o computador como sendo uma nova máquina capaz de entender os comandos da LP.

b) A LP fornece um conjunto de mecanismos para o programador criar e representar suas abstrações sobre o problema. A partir do momento que o programador cria uma abstração, ela se incorpora aos elementos da LP que podem ser usados pelo programador ou por outros programadores para criar novas abstrações e programas.

É nessa última perspectiva que o conceito de abstração serve como base para a modularização. Em programas bem modularizados, cada módulo

corresponde a uma abstração existente no contexto do problema. Diferentes LPs fornecem mecanismos distintos para suportar o processo de abstração dos programadores. Seções subsequentes desse capítulo discutem como LPs possibilitam ao programador criar e representar as abstrações do programa. Particular destaque é dado nas formas como esses mecanismos são usados para permitir a distinção entre:

- o que uma parte do programa faz (foco do programador que usa a abstração)
- como isso é implementado (foco do programador que implementa a abstração)

6.1.1 Tipos de Abstrações

Programas são conjuntos de instruções descrevendo como realizar processos para manipular, alterar e produzir dados. Para se poder criar programas, o conjunto de instruções da LP deve ser capaz de descrever dados e descrever como realizar processos.

Nos capítulos 3 e 4, foi visto que LPs fornecem valores, tipos, variáveis e constantes para a representação de estruturas de dados. No capítulo 5, também foi visto que elas fornecem diversos tipos de expressões e comandos de controle de fluxo de execução para a descrição de processos. Contudo, esses mecanismos são insuficientes para suportar as necessidades de abstração do programador, principalmente quando levamos em conta as demandas por encapsulamento e reuso de código.

Para atender essas necessidades, uma LP deve fornecer mecanismos para permitir ao programador a criação de novas formas de expressões, comandos ou representações de dados. Nesse sentido, os mecanismos de abstração fornecidos por LPs podem ser classificados em:

a) **Abstrações de Processos**: são abstrações sobre o fluxo de controle do programa.

b) **Abstrações de Dados**: são abstrações sobre as estruturas de dados do programa.

O mecanismo mais comum em LPs para provimento de abstrações de processos são os subprogramas. Subprogramas definem trechos limitados de programa onde podem ser definidas entidades como variáveis, constantes e tipos, as quais são utilizadas apenas localmente dentro do código do subprograma. Subprogramas podem ser chamados várias vezes em um mesmo programa ou até mesmo em programas diferentes, o que possibilita o reuso de código.

Subprogramas podem ser do tipo função ou procedimento. Enquanto um subprograma do tipo função é uma abstração de uma expressão, um subprograma do tipo procedimento é uma abstração de um comando. Isso significa que, ao construir um subprograma, o programador cria um novo tipo de fluxo de controle dos programas, aumentando assim o conjunto de instruções fornecidos pela LP.

Por exemplo, a função *sqrt* da biblioteca padrão de C é uma abstração sobre um conjunto de instruções que produz a raiz quadrada de um número como resultado. Na chamada dessa função, o programador se abstrai de todas as instruções que compõem essa função. Para ele, *sqrt* funciona como uma instrução pré-existente da LP.

Análogo às abstrações de processos, que permitem ver uma determinada combinação de comandos e expressões como sendo um novo comando ou expressão, abstrações de dados permitem ver uma determinada combinação de dados como sendo um novo dado ou tipo de dados.

Quando se vê uma coleção de bits como sendo uma célula de memória e passa-se a tratar essa coleção de bits como unidade básica de armazenamento, está-se construindo uma abstração de dados. Pode-se agora enxergar a memória como composta por células de memória (em vez de bits) e realizar operações sobre essas células.

Abstrações a nível de célula de memória ainda estão muito relacionadas com a máquina física, sendo frequentemente usadas apenas em linguagens de programação chamadas de baixo nível. Desde o aparecimento das linguagens de programação de alto nível, uma de suas características mais importantes tem sido a inclusão de abstrações de dados num nível mais conceitual. Por exemplo, a inclusão de tipos de dados inteiro, ponto flutuante e vetorial na própria LP permite ao programador utilizar dados desses tipos sem que se tenha de preocupar significativamente em como esses dados estão sendo efetivamente armazenados e manipulados. Assim, os programadores podem construir mais facilmente novas abstrações de dados cada vez mais complexas. Por exemplo, construir uma estrutura de dados para a representação de uma agenda é muito mais fácil se utilizamos uma LP cujos conceitos de números inteiros e matrizes sejam pré-definidos do que em uma LP que manipule apenas dados binários e células de memória.

Já se constatou que o fornecimento de tipos de dados pré-definidos na LP como único mecanismo de abstração de dados não é suficiente para atender às necessidades de programação com qualidade. Muitas vezes, o programador precisa criar seus próprios tipos de dados (como por exemplo, o tipo pilha) para tornar o código mais legível, redigível, flexível, confiável

e reusável. LPs têm fornecido vários mecanismos para permitir ao programador criar suas abstrações de dados.

Os principais mecanismos fornecidos por LPs para a implementação de abstrações de processos e de dados são discutidos na próxima seção.

6.2 Técnicas de Modularização

Técnicas de modularização foram desenvolvidas com o propósito principal de dar apoio à programação de sistemas de grande porte, embora também sejam úteis à programação de sistemas de pequeno porte.

Em contraste aos sistemas de pequeno porte, sistemas de grande porte se caracterizam por envolverem um número grande de entidades de computação, por normalmente serem implementados por uma equipe de programadores e por serem compostos por um número grande de linhas de código, geralmente distribuídas em muitos arquivos fontes.

Outra distinção importante entre sistemas de pequeno e grande porte pode ser feita ao se considerar o processo de compilação desses tipos de sistemas. Enquanto em sistemas de pequeno porte, compilar e recompilar o programa por completo após uma modificação não é muito dispendioso, o mesmo não ocorre em sistemas de grande porte. Nesses últimos é conveniente evitar a recompilação das partes não alteradas do programa.

No estudo de LPs costuma-se considerar um módulo como uma unidade de programa a qual pode ser compilada separadamente. Um módulo bem projetado tem um único propósito e uma boa interface com outros módulos. Módulos são reutilizáveis (podem ser incorporados em vários programas) e modificáveis (podem ser revisados sem forçar mudanças nos outros módulos).

Um módulo bem projetado deve identificar claramente:

- Qual o seu objetivo? (preocupação do usuário)
- Como ele atinge seu objetivo? (preocupação do implementador)

Um módulo pode ser composto por um único tipo, variável, constante, procedimento ou função, ou mesmo um conjunto deles. É mais comum, porém, que um módulo seja formado por um grupo composto por vários componentes distintos (tipos, constantes, variáveis, procedimentos e funções) declarados com um objetivo comum.

Técnicas de modularização permitem transformar trechos do programa em unidades lógicas relacionadas. Para isso, a LP deve oferecer mecanismos a partir dos quais se possa encapsular trechos contendo várias entidades de programação correlacionadas em uma única entidade de programação.

6.2.1 Subprogramas

O primeiro avanço em direção à modularização de programas foi a criação do conceito de subprogramas. Eles permitem segmentar um programa em vários blocos logicamente relacionados. Subprogramas também servem para evitar que trechos de código muito semelhantes tenham de ser completamente reescritos simplesmente por que operam sobre dados diferenciados. Isso pode ser feito através dos mecanismos de parametrização dos subprogramas.

Não faz sentido dividir um programa em vários subprogramas levando em conta, exclusivamente, o tamanho do código de cada subprograma. Modularizações efetuadas dessa maneira possuem baixa qualidade. De fato, um subprograma deve ser responsável por realizar uma determinada funcionalidade, mantendo sempre correspondência com uma abstração de processo.

Essa postura aumenta a legibilidade dos programas e facilita a depuração, manutenção e o reuso de código. Quando um subprograma possui um propósito único e claro, a leitura do programa fica muito mais fácil pois não há necessidade de se analisar o código do subprograma para saber o que ele faz. Também fica mais fácil identificar onde é preciso modificar o código durante a depuração ou alteração do programa. Isso ocorre porque as correções no código ficam concentradas no subprograma cuja funcionalidade necessita ser depurada ou alterada. Além disso, se torna muito mais simples e efetivo reusar código pois se pode usar o subprograma sempre que sua funcionalidade for necessária.

6.2.1.1 Perspectivas do Usuário e do Implementador de Subprogramas

É interessante analisar funções e procedimentos segundo as perspectivas do usuário da abstração e do implementador da abstração [WATT, 1990].

Uma função abstrai uma expressão a ser avaliada. Ela produz um valor como resultado. O usuário da função se importa apenas com o resultado, não se interessando com o modo como ele foi obtido. A função do exemplo 6.1, em C, calcula o fatorial de um número inteiro não negativo n[6.1].

```
int fatorial(int n) {
  if (n<2) {
    return 1;
  } else {
```

[6.1] Assume-se que essa função só é aplicada a números inteiros não negativos

```
    return n * fatorial (n – 1);
  }
}
```

Exemplo 6. 1 – Visões do Usuário e do Implementador de Função em C

A visão do usuário em uma chamada de função é a de que ela mapeia os argumentos usados na chamada em um resultado. Assim, a função *fatorial* do exemplo 6.1 mapeia um inteiro *n* para o resultado *n!*.

A visão do implementador de uma função é a de que ela executa o corpo da função tendo recebido previamente os valores dos seus argumentos. Ao implementador interessa o algoritmo da função. Na função *fatorial*, a visão do implementador é que ela calcula seu resultado através de um algoritmo recursivo.

Um procedimento abstrai um comando a ser executado. Quando um procedimento é chamado, ele atualiza variáveis. O usuário dessa abstração observa apenas essas atualizações, não se importando em como elas foram efetivadas. O exemplo 6.2, em C, ordena crescentemente um vetor de números inteiros.

```
void ordena (int numeros[50]) {
        int j, k, aux ;
        for (k = 0; k < 50; k++) {
                for (j = 0; j < 49; j++) {
                        if (numeros[j] > numeros[j+1]) {
                                aux = numeros[j];
                                numeros[j] = numeros[j+1];
                                numeros[j+1] = aux;
                        }
                }
        }
}
```

Exemplo 6. 2 - Perspectivas do Usuário e do Implementador de um Procedimento em C[6.2]

O implementador enxerga essa abstração de procedimento como a implementação de um algoritmo de ordenação pelo método da bolha. A visão do usuário é a de que existe um comando chamado *ordena(v)*, o qual produz como efeito a ordenação de *v*. Se a implementação for melhorada, o procedimento executará mais eficientemente, mas o usuário observará o mesmo efeito.

[6.2] A título de ilustração desse conceito, funções em C que retornam *void* são consideradas como procedimentos.

6.2.1.2 Parâmetros

Parâmetros facilitam o processo de aplicação de dados diferenciados a chamadas distintas de um subprograma. Sem o uso de parâmetros, a utilidade de subprogramas se concentraria na segmentação de código em trechos menores. O reuso de subprogramas seria muito mais difícil pois haveria redução de redigibilidade, legibilidade e confiabilidade. O exemplo 6.3 ilustra o uso de um subprograma em C implementado sem o uso de parâmetros.

```
int altura, largura, comprimento;
int volume () { return altura * largura * comprimento; }
main() {
  int a1 = 1, l1 = 2, c1 = 3, a2 = 4, l2 = 5, c2 = 6;
  int v1, v2;
  altura = a1;
  largura = l1;
  comprimento = c1;
  v1 = volume();
  altura = a2;
  largura = l2;
  comprimento = c2;
  v2 = volume();
  printf ("v1: %d\nv2: %d\n", v1, v2);
}
```

Exemplo 6. 3 - Função Sem Parâmetros em C

Observe no exemplo 6.3 que a função *volume* é usada para calcular o volume de um paralelepípedo. Como ela não possui parâmetros, é necessário utilizar três variáveis globais *altura*, *largura* e *comprimento* para lhe conferir generalidade e possibilitar o seu reuso em duas linhas de *main*.

Essa postura compromete a redigibilidade do código porque antes de chamar *volume* é sempre necessário incluir operações para atribuir os valores desejados às variáveis globais. Ela também reduz a legibilidade do código pois na chamada de *volume* não existe qualquer menção à necessidade de uso dos valores das variáveis *altura*, *largura* e *comprimento*. Ela pode ainda diminuir a confiabilidade do código por não exigir que sejam atribuídos valores a todas as variáveis globais utilizadas em *volume*. Assim, um programador inadvertido poderia deixar de atribuir um valor a uma das variáveis *altura*, *largura* e *comprimento* e ainda assim o programa seria compilado e executaria normalmente.

Todos esses problemas se resolvem com a utilização de parâmetros. Compare a implementação do exemplo 6.3 com a implementação do e-

xemplo 6.4, o qual realiza a mesma funcionalidade do exemplo 6.3, mas agora usando parâmetros.

```
int volume (int altura, int largura, int comprimento) {
        return altura * largura * comprimento;
}
main() {
  int a1 = 1, l1 = 2, c1 = 3, a2 = 4, c2 = 5, l2 = 6;
  int v1, v2;
  v1 = volume(a1, l1, c1);
  v2 = volume(a2, l2, c2);
  printf ("v1: %d\nv2: %d\n", v1, v2);
}
```

Exemplo 6. 4 - Função em C com Parâmetros

É fácil observar no exemplo 6.4 a melhoria da redigibilidade em virtude da não necessidade de fazer operações de atribuição a variáveis globais. A legibilidade também aumenta pois as chamadas de *volume* explicitam os valores utilizados em cada chamada. Além disso, a confiabilidade é incrementada porque qualquer tentativa de chamar *volume* sem a especificação de algum dos parâmetros produz um erro de compilação.

Habitualmente se utiliza o termo parâmetro para designar conceitos distintos. Em alguns contextos, o termo parâmetro se refere aos identificadores listados no cabeçalho do subprograma e usados no seu corpo. No estudo de LPs se convencionou usar o termo parâmetro formal para designar esse conceito. É importante lembrar que parâmetros formais são, necessariamente, identificadores de variáveis ou constantes, isto é, não podem ser valores nem tampouco expressões.

Em outros contextos, o termo parâmetro se refere aos valores, identificadores ou expressões utilizados na chamada do subprograma. Em LPs se usa o termo parâmetro real para denotar esse conceito. Ainda em outros contextos, o termo argumento é usado como sinônimo de parâmetro. Em LPs esse termo é usado para designar o valor passado do parâmetro real para o parâmetro formal.

O exemplo 6.5 ilustra esses conceitos através de um programa em C que calcula a área de um círculo. O parâmetro formal de *area* é o identificador *r*. O parâmetro real e o argumento da chamada de *area* em *main* são respectivamente a expressão *diametro/2* e o valor *1.4*.

```
float area (float r) {
  return 3.1416 * r * r;
}
main() {
  float diametro, resultado;
```

```
  diametro = 2.8;
  resultado = area (diametro/2);
}
```

Exemplo 6. 5 - Parâmetro Formal, Parâmetro Real e Argumento de Subprograma

6.2.1.3 Correspondência entre Parâmetros Formais e Parâmetros Reais

Existem duas formas de correspondência entre a lista de parâmetros formais na declaração do subprograma e a lista de parâmetros reais na chamada do subprograma. A forma mais comum de correspondência é a posicional, adotada por C, C++ e JAVA e pela imensa maioria de LPs. Nesse tipo de correspondência a sequência na qual os parâmetros são escritos determina a correspondência entre os parâmetros reais e formais. A outra forma de correspondência, conhecida como por palavras chave, envolve a listagem explícita dos parâmetros reais e seus correspondentes formais na chamada do subprograma. Ressalta-se a necessidade, em ambos tipos de correspondência, de haver compatibilidade de tipos entre o parâmetro real e seu correspondente parâmetro formal. O exemplo 6.6 ilustra o uso de correspondência por palavras chave em ADA.

```
procedure palavrasChave is
  a: integer := 2;
  b: integer := 3;
  c: integer := 5;
  res: integer;
  function multiplica(x, y, z: integer) return integer is
  begin
    return x * y * z;
  end multiplica;
begin
  res := multiplica(z=>b, x=>c, y=>a);
end palavrasChave;
```

Exemplo 6. 6 - Correspondência por Palavras Chave em ADA

A correspondência por palavras chave é útil principalmente quando a lista de parâmetros é longa, uma vez que não obriga o programador a lembrar a sequência na qual os parâmetros formais foram declarados. Em contrapartida, o programador deve saber o nome dado aos parâmetros formais na chamada do subprograma.

Algumas LPs, como ADA, permitem o uso das duas formas de correspondência de parâmetros. De fato, em uma chamada de subprograma pode-se usar isoladamente qualquer uma das formas ou mesmo usá-las de modo combinado.

Em algumas LPs, tais como C++ e ADA, é possível definir valores `default` para os parâmetros formais na declaração do subprograma. Valores `default` são valores atribuídos aos parâmetros formais caso os parâmetros reais correspondentes sejam omitidos na chamada do subprograma. O exemplo 6.7 ilustra o uso desses valores em C++.

```
int soma (int a[], int inicio = 0, int fim = 7, int incr = 1){
  int soma = 0;
  for (int i = inicio; i < fim; i+=incr) soma+=a[i];
  return soma;
}
main() {
  int [] pontuacao = { 9, 4, 8, 9, 5, 6, 2};
  int ptotal, pQuaSab, pTerQui, pSegQuaSex;
  ptotal = soma(pontuacao);
  pQuaSab = soma(pontuacao, 3);
  pTerQui = soma(pontuacao, 2, 5);
  pSegQuaSex = soma(pontuacao, 1, 6, 2);
}
```

Exemplo 6. 7 - Valores Default de Parâmetros em C++

No exemplo 6.7 foi criada a função *soma* para calcular a pontuação acumulada por um jogador em diferentes dias de uma semana de competição. Observe que três parâmetros formais de *soma* possuem valores `default`. Assim, é possível calcular a pontuação total, obtida pelo jogador na semana, omitindo, na chamada de *soma*, todos parâmetros reais cujos formais possuem valor `default`. Já para a pontuação obtida entre quarta e sábado, deve-se especificar na chamada o parâmetro real correspondente ao formal *inicio*. Para se obter a pontuação acumulada de terça a quinta é preciso especificar os parâmetros *inicio* e *fim*. Por fim, para obter a pontuação acumulada na segunda, quarta e sexta é necessário especificar todos os parâmetros reais possíveis.

Em C++ os parâmetros `default` devem ser sempre os últimos da lista. Isso significa que uma tentativa de retirar o valor `default` dos parâmetros formais *fim* ou *incr* na função *soma* do exemplo 6.7 produziria um erro de compilação. Essa restrição existe em C++ para evitar ambigüidade na correspondência posicional entre parâmetros formais e reais.

Normalmente LPs requerem na definição da função a especificação de um número de parâmetros reais igual ao número de parâmetros formais. Elas também requerem que os tipos dos parâmetros reais sejam compatíveis com os dos parâmetros formais correspondentes. C e C++ são exceções pois permitem a criação de funções cujo número de parâmetros reais pode variar de uma chamada para outra. Uma forma de fazer isso em C++

é através do uso de parâmetros `default`. Contudo, as variações permitidas no número de parâmetros são relativamente pequenas pois são limitadas pelo número de parâmetros `default` utilizados. Além disso, os tipos dos parâmetros não podem variar de uma chamada para outra. Portanto, esse mecanismo não é suficientemente genérico para criar subprogramas nos quais não se possa antecipar o número e o tipo dos parâmetros.

Um exemplo dessa categoria de subprogramas é a função *printf* de C. Como essa função é usada pelos mais variados programas nos mais variados contextos, qualquer limitação no seu número e tipo de parâmetros a tornaria insuficiente para atender todas suas possíveis demandas. C e C++ possibilitam a criação de funções dessa categoria através do uso do símbolo ... (reticências) encerrando a lista de parâmetros formais na definição do cabeçalho da função.

Ao se definir uma função com tipo e número de parâmetros variável é necessário ter algum modo do corpo da função obter o valores e tipos dos parâmetros de uma chamada. Sem isso não haveria utilidade em se permitir parâmetros variáveis, uma vez que o corpo do subprograma não poderia operar sobre eles. Para resolver esse problema, C e C++ oferecem um conjunto de macros na biblioteca padrão *stdarg.h* e exigem que exista pelo menos um parâmetro nomeado e não variável na definição do cabeçalho da função. Esse parâmetro deve fornecer as informações necessárias sobre os tipos e sobre o número dos parâmetros variáveis. O protótipo da função *printf* de C é mostrado a seguir.

```
int printf (char* fmt, ... );
```

Observe o uso da string de formatação *fmt* como parâmetro nomeado. Esse parâmetro oferece as informações sobre os tipos e número dos parâmetros reais de uma chamada do subprograma. Observe ainda o uso das reticências colocadas obrigatoriamente ao final da lista de parâmetros. O exemplo 6.8 mostra a definição completa de uma função dessa categoria e a sua utilização por um programa.

```
#include <stdarg.h>
int ou (int n, …) {
  va_list vl;
  int i, r = 0;
  va_start (vl, n);
  for (i = 0; i < n; i++)
    if (va_arg (vl, int)) r = 1;
  va_end (vl);
  return r;
}
```

```
main() {
  printf ("%d\n", ou (1, 3 < 2));
  printf ("%d\n", ou (2, 3 > 2, 7 > 5));
  printf ("%d\n", ou (3, 1 != 1, 2 != 2, 3 != 3));
  printf ("%d\n", ou (3, 1 != 1, 2 != 2, 3 == 3));
}
```

Exemplo 6. 8 - Função com Número de Parâmetros Variável em C

A função *ou* do exemplo 6.8 calcula a operação de conjunção lógica sobre todos os parâmetros reais de uma chamada (com exceção do parâmetro nomeado *n*). A função *ou* espera parâmetros reais inteiros na lista variável. Assim, o parâmetro nomeado é usado apenas para especificar o número de parâmetros adicionais de cada chamada de *ou*. Observe ainda o uso das macros *va_list*, *va_start*, *va_arg* e *va_end* de *stdarg.h*. O tipo *va_list* é usado para definir uma variável *vl* a qual se refere a cada argumento por sua vez. A macro v*a_start* inicializa *vl* para designar o primeiro argumento não nomeado. Note que *va_start* recebe como argumento o identificador do último parâmetro nomeado da lista de parâmetros. Já *va_arg* retorna um dos argumentos da lista de parâmetros variáveis cada vez que é chamada. É preciso informar na chamada de *va_arg* o tipo do valor a ser retornado. Ao final é preciso chamar *va_end* antes do retorno da função.

Se, por um lado, a existência dessa categoria de funções oferece uma maior flexibilidade às LPs, por outro lado, isso reduz a confiabilidade dos programas uma vez que não é possível verificar os tipos dos parâmetros em tempo de compilação. Nesses subprogramas, é tarefa exclusiva dos programadores, que implementam e usam essas funções, garantir o correto funcionamento do subprograma.

6.2.1.4 Passagem de Parâmetros

Passagem de parâmetros é o processo no qual os parâmetros formais assumem seus respectivos valores durante a execução de um subprograma. Também faz parte do processo de passagem de parâmetros a eventual atualização de valores dos parâmetros reais durante a execução do subprograma. LPs fornecem modos distintos de passagem de parâmetros. Esses modos podem ser diferenciados pela direção da passagem, pelo mecanismo de implementação e pelo momento no qual a passagem é realizada.

6.2.1.4.1 Direção da Passagem dos Parâmetros

A direção da passagem de parâmetros pode ser unidirecional de entrada, unidirecional de saída e bidirecional de entrada e saída. Na unidirecional

de entrada o valor do parâmetro formal assume o valor passado pelo parâmetro real, mas os valores eventualmente atribuídos ao parâmetro formal no subprograma não são repassados ao parâmetro real. Na unidirecional de saída os valores atribuídos ao parâmetro formal são passados ao parâmetro real, mas o parâmetro formal não assume inicialmente qualquer valor. Na bidirecional de entrada e saída tanto o parâmetro formal assume o valor passado pelo parâmetro real quanto os valores atribuídos ao parâmetro formal são repassados para o parâmetro real.

Enquanto na passagem unidirecional de entrada o parâmetro real pode ser uma variável, constante ou expressão do tipo definido para o parâmetro formal, nas passagens unidirecional de saída e bidirecional o parâmetro real deve ser necessariamente uma variável do tipo definido para o parâmetro formal, uma vez que o parâmetro real poderá ter seu valor alterado na execução do subprograma.

Existem duas variações possíveis no tratamento do parâmetro formal na passagem unidirecional de entrada. O parâmetro formal pode ser considerado como uma variável ou como uma constante. Quando considerado como variável esse parâmetro pode ter o seu valor alterado no corpo do subprograma, embora essas modificações só tenham efeito interno, não sendo repassadas para o parâmetro real correspondente. Quando considerado como constante, qualquer tentativa de alteração do valor do parâmetro formal no corpo do subprograma causa a ocorrência de erro (geralmente identificado na própria compilação).

A tabela 6.1 resume as informações importantes a respeito da direção da passagem dos parâmetros. A coluna **Direção da Passagem** indica as formas de direção da passagem: unidirecional de entrada com parâmetro formal variável, unidirecional de entrada com parâmetro formal constante, unidirecional de saída e bidirecional de entrada e saída. A coluna **Forma do Parâmetro Real** destaca que, nas passagens unidirecionais de entrada, o parâmetro real pode ser uma variável, uma constante ou mesmo uma expressão e, nas outras passagens, ele deve ser necessariamente uma variável. A coluna **Atribuição do Parâmetro Formal** ressalta que somene na passagem unidirecional de entrada constante não é permitido fazer atribuições ao parâmetro formal. Finalmente, a coluna **Fluxo** indica o sentido da passagem de valor entre parâmetros reais e formais. Nas passagens unidirecionais de entrada o fluxo de valor vai do parâmetro real para o parâmetro formal. Na passagem unidirecional de saída o fluxo vai do parâmetro formal para o parâmetro real. Já na passagem bidirecional de entrada e saída o fluxo ocorre nos dois sentidos.

Direção da Passagem	Forma do Parâmetro Real (R)	Atrib. do Parâm. Formal (F)	Fluxo
Entrada Variável	Variável, Constante ou Expressão	Sim	R → F
Entrada Constante	Variável Constante ou Expressão	Não	R →F
Saída	Variável	Sim	R ← F
Entrada e Saída	Variável	Sim	R ↔F

Tabela 6. 1 - Direção da Passagem dos Parâmetros

C só oferece passagem unidirecional de entrada. Embora se possa usar a palavra *const* na definição de um parâmetro, a inclusão dessa palavra não torna esse parâmetro necessariamente constante, uma vez que os compiladores podem simplesmente ignorá-la. Portanto, pode-se considerar que C somente ofereça passagem unidirecional de entrada variável. Essa opção em C confere simplicidade a LP, mas traz algumas dificuldades em subprogramas nos quais é necessário alterar os valores dos parâmetros. A única forma de se fazer isso é passando ponteiros como parâmetros. O exemplo 6.9 ilustra a passagem de parâmetros em C através de dois subprogramas. Um desses subprogramas aparentemente tenta trocar os valores de duas variáveis inteiras do programa, mas não consegue produzir esse efeito. O outro utiliza ponteiros para efetuar a troca dos valores.

```
void naoTroca (int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
void troca (int* x, int* y) {
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}
main() {
  int a = 10, b = 20;
  naoTroca (a, b);
  troca (&a, &b);
}
```

Exemplo 6. 9 - Passagem Unidirecional de Entrada Variável em C

Note na função *naoTroca* do exemplo 6.9 as atribuições feitas aos parâmetros *x* e *y*. Embora sejam permitidas e troquem os valores de *x* e *y*, seu efeito é apenas interno, não efetivando a troca dos valores das variáveis *a* e *b* de *main*. Já a função *troca* realiza a troca dos valores de *a* e *b*. Ela consegue fazer isso porque *x* e *y* são ponteiros. Observe que o corpo do subprograma não altera o valor dos parâmetros e sim os valores das variáveis apontadas por eles. Contudo, isso força usar como parâmetros reais expressões que obtém os endereços das variáveis *a* e *b*, além de obrigar o uso de operações de derreferenciamento no corpo do subprograma. Isso acaba reduzindo a redigibilidade e legibilidade dos programas.

Além da passagem unidirecional de entrada variável, C++ também oferece as passagens unidirecional de entrada constante e bidirecional. O exemplo 6.10 mostra o uso dessas formas de passagem de parâmetros em C++.

```
int triplica (const int x) {
  // x = 23;
  return 3*x;
}
void troca (int& x, int& y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
main() {
  int a = 10, b = 20;
  b = triplica (a);
  troca (a, b);
  // troca (a, a + b);
}
```

Exemplo 6. 10 - Passagem Unidirecional de Entrada Constante e Bidirecional em C++

No exemplo 6.10, a função *triplica* utiliza a passagem de parâmetros unidirecional de entrada constante e a função *troca* usa a passagem bidirecional. A execução de *troca* faz com que os valores de *a* e *b* sejam trocados. Caso a linha comentada da função *triplica* tivesse sido compilada haveria um erro, uma vez que não se pode realizar atribuições sobre o parâmetro formal na passagem de entrada constante. Também haveria erro de compilação caso a linha com a última chamada de *troca* não fosse comentada em *main*. O erro seria ocasionado pelo uso da expressão *a+b* como parâmetro real na chamada, o que não é permitido na passagem bidirecional.

JAVA oferece passagem unidirecional de entrada variável ou constante para tipos primitivos. Para tipos não primitivos, a passagem pode ser considerada unidirecional de entrada variável ou constante, pois atribuições de valores completos do tipo não primitivo ao parâmetro formal não produzem efeito no parâmetro real, ou bidirecional, pois atribuições aos componentes do parâmetro formal têm efeito sobre os componentes do parâmetro real.

O exemplo 6.11 mostra uma função implementada em JAVA, denominada *preencheVet*, a qual é utilizada para atribuir valores a um conjunto de elementos de um vetor (correspondente ao parâmetro formal *a*). O trecho a ser preenchido no vetor é especificado pelos valores assumidos pelos parâmetros formais *i* e *j*. Observe a variação do valor do parâmetro formal *i* nessa função. Embora o valor de *i* seja alterado, o parâmetro real correspondente não é modificado, uma vez que a passagem de tipos primitivos é unidirecional de entrada.

```
void preencheVet (final int[] a, int i, final int j) {
  while (i <= j) a[i] = i++;
  // j = 15;
  // a = new int [j];
}
```

Exemplo 6. 11 - Passagem de Parâmetros em JAVA

A palavra *final* antes da declaração do vetor *a* e do inteiro *j* indica que esses parâmetros são constantes. Assim, as linhas comentadas em *preencheVet*, caso fossem compiladas, provocariam erros.

É importante notar que essa função realmente modifica os valores dos elementos do parâmetro real correspondente ao vetor *a*. Isso ocorre independentemente do fato do parâmetro *a* ter sido declarado (ou não) como constante. De fato, a declaração *final* para tipos não primitivos simplesmente impede a atribuição ao parâmetro como um todo (tal como na última linha comentada de *preencheVet*), não proibindo a atribuição aos seus componentes. Caso o parâmetro *a* não fosse declarado como constante, a última linha comentada também seria legal, mas não produziria qualquer efeito sobre o parâmetro real correspondente, só produzindo efeitos internos.

ADA oferece passagem de parâmetros unidirecional de entrada constante, unidirecional de saída e bidirecional de entrada e saída. O exemplo 6.12 mostra essas três formas de passagem de parâmetros em ADA através da declaração de dois subprogramas.

function triplica (x: in integer; out erro: integer) return integer;
procedure incrementa (x: in out integer; out erro: integer);

Exemplo 6. 12 - Passagem de Parâmetros em ADA

Poder-se-ia omitir a palavra *in* na declaração do parâmetro *x* da função *triplica* no exemplo 6.12, uma vez que a omissão do especificador de direção da passagem determina a passagem unidirecional de entrada constante. Em ambos subprogramas desse exemplo existe um parâmetro de saída *erro* usado para retornar um código indicador da ocorrência de algum problema na realização do subprograma.

Deve-se ter cuidado com o uso das passagens de parâmetros unidirecional de saída e bidirecional de entrada e saída para evitar possíveis colisões no retorno dos resultados. Por exemplo, a chamada

incrementa (i, i);

do subprograma *incrementa* declarado no exemplo 6.12 provoca esse tipo de colisão, podendo gerar indeterminismo.

6.2.1.4.2 Mecanismos de Implementação da Passagem de Parâmetros

Existem dois mecanismos para a implementação da passagem de parâmetros. O mecanismo de passagem por cópia envolve a criação de uma cópia do parâmetro real no ambiente local do subprograma. O parâmetro formal designa essa cópia durante a execução do subprograma. O mecanismo de passagem por referência envolve a criação de uma referência ao parâmetro real no ambiente local do subprograma. O parâmetro formal designa essa referência durante a execução do subprograma. A figura 6.1 ilustra a diferença entre os mecanismos de passagem de parâmetros por cópia e por referência.

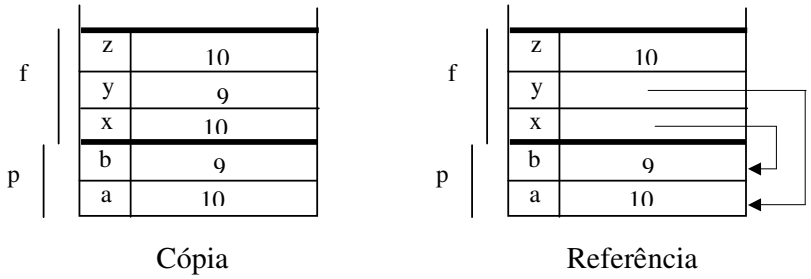

Figura 6. 1 - Passagem por Cópia e por Referência

A figura 6.1 retrata o momento em que a função *f*, a qual possui dois parâmetros *x* e *y*, está sendo executada, após ser chamada por *p*. Quando o mecanismo de cópia é utilizado, os parâmetros reais *a* e *b* têm seu tipo e valor copiados para o ambiente de *f*. Qualquer alteração feita por *f* nos parâmetros formais implica em modificação nos valores das cópias *x* e *y*. Se a direção da passagem de parâmetros envolver o repasse dos valores

dos parâmetros formais para os parâmetros reais, isso só será realizado ao final da execução de *f*.

Quando o mecanismo de referência é utilizado, os endereços dos parâmetros reais *a* e *b* são copiados para o ambiente de *f*. Os parâmetros formais *x* e *y* são, na realidade, referências para os parâmetros reais *a* e *b*. Qualquer alteração feita por *f* nos parâmetros formais implica na modificação imediata dos valores dos parâmetros reais.

Uma contribuição do mecanismo de cópia é viabilizar a passagem unidirecional de entrada variável de parâmetros. Essa direção de passagem de parâmetros só pode ser feita através do mecanismo de cópia, uma vez que o uso do mecanismo de referência implicaria na modificação do parâmetro real.

Adicionalmente, o mecanismo de cópia pode facilitar a recuperação do estado prévio do programa quando um determinado subprograma termina inesperadamente. Uma vez que as atualizações executadas pelo subprograma são efetuadas sobre o parâmetro formal, e só são repassadas para o parâmetro real no final da execução, uma interrupção inesperada do subprograma não afeta, em princípio, o estado das variáveis do programa.

O mecanismo de referência proporciona uma semântica simples e uniforme para passagem de parâmetros, adequada a todos os tipos de valores da LP e não somente para tipos nos quais é possível realizar atribuição (como é o caso da passagem por cópia). Por exemplo, o mecanismo de referência torna possível passar subprogramas como parâmetros para outros subprogramas. Nesse caso, durante a ativação do subprograma, o parâmetro formal recebe o endereço inicial do trecho de memória onde um determinado subprograma está alocado. Referências ao parâmetro formal no corpo do subprograma equivalem a chamadas indiretas ao subprograma para o qual esse parâmetro referencia. Passagem de parâmetros subprogramas aumentam as possibilidades de reuso de subprogramas, pois um mesmo subprograma passa a poder ser usado para atingir diferentes funcionalidades. Assim, além de se poder alterar os dados sobre os quais o subprograma atua, também se pode alterar a sua própria funcionalidade. Veja no exemplo 3.8, do capítulo 3, uma situação na qual esse recurso é utilizado em C para alterar a funcionalidade de um subprograma. Observe nesse exemplo que C usa ponteiros para função para viabilizar a passagem de parâmetros subprogramas.

Em geral, a passagem de um parâmetro que contenha um grande volume de dados, tal como um vetor, é mais eficiente através do mecanismo de referência do que no de cópia. Ao se usar o mecanismo de cópia é necessário alocar espaço adicional suficiente para repetir todo o volume de dados do parâmetro real. Além disso, é necessário realizar a cópia dos valo-

res do parâmetro real para o formal e vice-versa. Isso é contrastado com a passagem por referência, a qual demanda apenas a alocação de espaço adicional para armazenar um ponteiro. Além disso, não é necessário copiar quaisquer dados.

Por outro lado, como o mecanismo de referência baseia-se no acesso indireto aos dados do parâmetro real, em uma implementação distribuída, o corpo do subprograma pode estar sendo executado num processador remoto, distante dos dados. Nessa situação o acesso indireto pode ser menos eficiente que o mecanismo de cópia seguido de acesso local.

Uma outra desvantagem do mecanismo de passagem por referência é a possibilidade de ocorrência de sinonímia, o que tende a dificultar o entendimento dos programas. Isso pode ocorrer na passagem por referência quando dois parâmetros formais são associados ao mesmo parâmetro real ou quando o subprograma faz uso de variáveis globais e um parâmetro formal referencia uma dessas variáveis globais. O exemplo 6.13 mostra uma função em C++ usada para incrementar de *1* os valores dos parâmetros *k* e *l*.

```
void incr (int& k, int& l) {
        k = k + 1;
        l = l + 1;
}
```

Exemplo 6. 13 - Sinonímia em Passagem por Referência em C++

Se *a* vale *10* e b vale *3*, após a chamada *incr (a, b)*, *a* e *b* são incrementados de 1 e passam a valer *11* e *4*, respectivamente. Contudo, se a chamada fosse *incr (a[i], a[j])* com o valor de *i* igual ao valor de *j,* o valor de *a[i]* (ou *a[j]*) seria incrementado duas vezes após a execução do subprograma. Isso pode ser difícil de identificar, em caso de engano, e entender, quando for essa a intenção do implementador da função.

C oferece apenas o mecanismo de passagem por cópia. C++, ADA e PASCAL adotam tanto o mecanismo de passagem por cópia quanto por referência. C++ usa o operador *&* para designar um parâmetro passado por referência (ver a função *troca* do exemplo 6.10), diferenciando-o assim dos parâmetros que usam passagem por cópia.

ADA adota passagem por cópia para tipos primitivos e para tipos derivados dos primitivos. A especificação de ADA define uma lista de tipos cuja passagem deve ser por referência. Para os tipos não especificados, a passagem pode ser por referência ou por cópia. Isso é especificado pela implementação do compilador.

JAVA adota o mecanismo de passagem por cópia para passar valores dos tipos primitivos. Existem duas perspectivas alternativas sobre a passagem

de parâmetros de tipos não primitivos. A primeira perspectiva considera a passagem como sendo por cópia. De fato, são feitas cópias no parâmetro formal das referências para os valores dos tipos não primitivos. Contudo, referências não são tipos de dados que podem ser manipulados explicitamente em JAVA. Portanto, não existe uma passagem através de cópia efetiva de um tipo de dados de JAVA.

A segunda perspectiva considera a passagem como sendo por referência. Como os dados do tipo não primitivo não são copiados para o ambiente do subprograma, mas podem ser alterados por ele, essa perspectiva considera tal postura como caracterizadora de passagem por referência. No entanto, a passagem de tipos não primitivos em JAVA, em certas situações, tem um comportamento um pouco diferenciado do que se costuma considerar passagem por referência. Somente atribuições aos componentes do parâmetro formal produzem efeito no valor referenciado pelo parâmetro real. Atribuições de valores completos do tipo não primitivo feitas ao parâmetro formal não produzem efeito sobre o parâmetro real correspondente. De fato, essas atribuições impedem até que os componentes do parâmetro real possam ser modificados a partir dali. O exemplo 6.14 ilustra essa diferença apresentando implementações de uma função *f* (que recebe dois parâmetros do tipo não primitivo *T*) em JAVA e em C++ (a versão em C++ usa passagem por referência).

```
void f (T t1, T t2) {
  t1 = t2;
}
```

```
void f(T& t1, T& t2) {
  t1 = t2;
}
```

Exemplo 6. 14 - Passagem de Tipos Não Primitivos em JAVA e C++

Enquanto na implementação de *f* em C++ o parâmetro real correspondente a *t1* teria seu valor modificado, isso não ocorre na implentação de *f* em JAVA. De fato, após a atribuição em JAVA, *t1* passa a designar o objeto referenciado por *t2* e perde qualquer associação com o valor passado para ele na chamada do subprograma.

Agora fica claro porque JAVA oferece passagem unidirecional de entrada para os tipos primitivos e unidirecional de entrada e bidirecional para os tipos não primitivos. Quando a passagem é de tipos primitivos, ocorre cópia. Qualquer atribuição ao parâmetro formal não produz efeito no parâmetro real. Quando a atribuição é de tipos não primitivos passa-se uma cópia da referência. Atribuições aos componentes do parâmetro formal produzem efeito no parâmetro real, caracterizando a passagem bidirecional. Por sua vez, atribuições de valores completos não produzem qualquer efeito no parâmetro real, caracterizando a passagem unidirecional de entrada.

Cabe frisar que a passagem unidirecional de entrada implementada através do mecanismo de cópia é mais conhecida no estudo de LPs pelo termo de passagem por valor. A existência da passagem unidirecional de entrada constante implementada através do mecanismo de referência oferece um poder expressivo similar ao da passagem por valor com a vantagem de não demandar cópias de grandes volumes de dados.

6.2.1.4.3 Momento da Passagem de Parâmetros

LPs podem definir diferentes momentos nos quais se deve avaliar o parâmetro real para uso no subprograma. No modo **normal** (`eager`) a avaliação ocorre no momento da chamada do subprograma. No modo **por nome** (`by name`) a avaliação ocorre em todos os momentos em que o parâmetro formal é usado. No modo **preguiçoso** (`lazy`) a avaliação ocorre no primeiro momento em que o parâmetro formal é usado.

Considere, no exemplo 6.15, a implementação da função *caso* em C. Essa função retorna o valor de *w* caso *x* seja negativo, o valor de *y* se *x* é positivo e o valor de *z* quando *x* é zero.

```
int caso (int x, int w, int y, int z) {
  if (x < 0) return w;
  if (x > 0) return y;
  return z;
}
```

Exemplo 6. 15 - Momento da Passagem de Parâmetros em C

C adota o modo normal para o momento da passagem dos parâmetros. Assim, a chamada *caso(p(), q(), r(), s())* implica na imediata avaliação das funções *p*, *q*, *r* e *s* e na imediata passagem dos seus resultados para *x*, *w*, *y* e *z*. O problema com o modo normal nessa situação é que somente o parâmetro *x* e um dos outros parâmetros são realmente necessários em cada chamada de *caso*. Se alguma das funções *q*, *r* e *s* não podem ser executadas em certos contextos, a função *caso* também não pode ser executada nesses contextos. Por exemplo, se em um determinado contexto, *s* executasse uma divisão por zero, a função *caso* não poderia ser executada nesse contexto também. Além disso, se *q*, *r* e *s* realizam um processamento computacional intensivo, o modo normal acaba reduzindo a eficiência do processamento de *caso*, uma vez que todas essas funções seriam sempre executadas em cada chamada.

Admitindo-se hipoteticamente que C adotasse o modo por nome para o momento da passagem de parâmetros, a execução das funções *p*, *q*, *r* ou *s* só seria realizada quando, na execução do corpo do subprograma, fosse necessário utilizar os valores dos parâmetros formais correspondentes.

Dessa maneira os problemas que ocorrem no modo normal seriam minimizados. Por exemplo, no contexto no qual *s* executasse uma divisão por zero, a função *caso* poderia ainda ser executada desde que *p* não resultasse no valor zero. Como *q*, *r* ou *s* só são executadas no momento em que a execução de *caso* referencia *w, y* ou *z*, também não haveria realização de computações desnecessárias, uma vez que o subprograma só se refere a um desses parâmetros em cada chamada.

Por outro lado, o modo por nome provocaria a execução repetida de *p* quando o resultado dessa função fosse não negativo (uma execução na comparação $x < 0$ e outra na comparação $x > 0$). Esse problema se agravaria ainda mais se o subprograma usasse *x* em uma repetição. Nesse caso, haveria uma execução de *p* a cada repetição da referência a *x*. Pode-se concluir, portanto, que o modo por nome é claramente ineficiente.

Outro problema com o modo por nome pode ocorrer em LPs que permitem efeitos colaterais. Considere, por exemplo, que a função *p* usada na chamada de *caso* atuasse como um iterador de uma lista, retornando sempre o elemento seguinte. Nessa situação, o valor de *x* utilizado na primeira comparação seria diferente do valor usado na segunda. Isso certamente contradiz o propósito da função *caso* e facilita o uso equivocado desse subprograma.

Caso C adotasse o modo preguiçoso, *p* só seria executada na primeira aparição de *x* (isto é, na comparação $x < 0$) e o valor usado na segunda aparição seria o mesmo usado na primeira, mesmo que a função *p* produzisse efeitos colaterais. Além disso, somente uma das funções *q*, *r* ou *s* seria executada em cada chamada, eliminando os problemas relacionados com a avaliação de parâmetros reais desnecessários.

Portanto, o modo preguiçoso apresenta boas soluções para os problemas de flexibilidade (caso o parâmetro formal não seja necessário o parâmetro real não será avaliado), eficiência (a avaliação do parâmetro real é realizada no máximo uma única vez) e efeitos colaterais (o parâmetro formal será associado a um único valor, mesmo que haja efeitos colaterais).

A maior parte das LPs, como C, C++, JAVA, ADA, ML e PASCAL, fornecem apenas o modo normal. ALGOL-60 permite ao programador escolher entre o modo normal e o por nome. SML permite ao programador escolher entre o modo normal e o preguiçoso. HASKELL e MIRANDA utilizam o modo preguiçoso.

6.2.1.5 Verificação de Tipos dos Parâmetros

Algumas LPs não verificam se o número e tipo dos parâmetros na chamada do subprograma são compatíveis com o número e tipo dos parâme-

tros declarados no cabeçalho do subprograma e utilizados no seu corpo. Nessas LPs só se pode verificar se uma determinada operação do subprograma pode ser executada no momento em que ela for realizada. Tal postura retarda a descoberta de erros e tende a produzir programas menos robustos.

Uma propriedade interessante proporcionada por uma LP é permitir aos compiladores garantir a não ocorrência de erros de tipos no uso dos parâmetros durante a ativação dos subprogramas. Para isso, é necessário fazer verificação de tipos dos parâmetros. A maioria das LPs ditas ALGOL-`like`, tais como, PASCAL, MODULA-2, ADA e JAVA fazem verificação de tipos.

Versões prévias de algumas LPs não requeriam a realização de verificação de tipos. Contudo, hoje já incorporaram mecanismos para possibilitar essa verificação. Na versão original de C não se requeria aos compiladores a verificação do número e tipos dos parâmetros. Embora pudessem existir programas a parte para realizar a verificação, o programador poderia não utilizá-los.

Versões atuais, tal como ANSI C, já permitem ao programador definir se os compiladores devem ou não realizar a verificação de tipos na chamada dos subprogramas. O exemplo 6.16 mostra um programa com uma função, chamada *origem*, a qual usa a forma prévia de declaração de C, e outra função, chamada *distancia*, a qual usa a nova forma. A função *origem* é usada para dizer se um ponto se encontra ou não na origem do eixo de coordenadas. A função *distancia* calcula a distância entre a origem e um determinado ponto.

```
#include math.h
typedef struct coordenadas {
  int x, y, z;
} coord;
int origem (c)
  coord c;
{
  return c.x*c.y*c.z;
}
float distancia (coord c) {
  return sqrt(c.x*c.x + c.y*c.y +c.z*c.z);
}
main() {
  coord c = { 1, 2, 3 };
  printf("%d\n", origem(2));
  printf("%d\n", origem(1, 2, 3, 4));
```

```
  printf("%d\n", origem(c));
  // printf("%f\n", distancia(2));
  // printf("%f\n", distancia (1, 2, 3));
  printf("%f\n", distancia (c));
}
```

Exemplo 6. 16 - Verificação de Tipos dos Parâmetros em C

Como a função *origem* usa a forma prévia de declaração de parâmetros, todas as chamadas a essa função são válidas em *main*, embora as duas primeiras não façam muito (ou qualquer) sentido. Por outro lado, como *distancia* usa a nova forma de declaração, caso as linhas com as duas primeiras chamadas dessa função não estivessem comentadas, haveriam erros de compilação.

É importante saber a diferença entre a declaração de uma função na qual a lista de parâmetros é omitida *f()* e a declaração de uma função sem parâmetros *f(void)*. A declaração *f()* segue o padrão prévio de C, ou seja, não requer a verificação de tipos. Isso significa que chamadas a *f* podem ser feitas com qualquer número ou tipo de parâmetros reais ou mesmo sem parâmetro real algum. A declaração *f(void)* segue o novo padrão e estabelece que *f* só pode ser chamada sem parâmetros.

Em ANSI C e C++ a verificação de tipos também pode ser evitada através do uso do operador elipse (…), cujo uso foi ilustrado no exemplo 6.8.

6.2.2 Tipos de Dados

A criação de novos tipos de dados é uma forma de modularização usada para implementar abstrações de dados. Tipos de dados permitem agrupar dados correlacionados em uma mesma entidade computacional. Usuários dessa nova entidade computacional passam a enxergar o grupo de dados como um todo pré-definido e não precisam se preocupar em como essa entidade foi implementada ou em como seus dados são armazenados.

LPs oferecem diferentes mecanismos para a definição de novos tipos de dados, tais como tipos anônimos, tipos simples e tipos abstratos de dados.

6.2.2.1 Tipos de Dados Anônimos

Tipos anônimos são definidos exclusivamente durante a criação de variáveis e definição de parâmetros. O exemplo 6.17 ilustra o uso de um tipo anônimo em C para criar uma variável chamada *pilhaNumeros*.

```
struct {
  int elem[100];
```

```
  int topo;
} pilhaNumeros;
```

Exemplo 6. 17 - Tipo de Dados Anônimo

Observe que a definição do tipo só pode ser usada uma única vez em virtude do tipo não possuir um nome. Se uma outra variável necessitar ter os mesmos dados de *pilhaNumeros*, toda a definição terá de ser repetida, diminuindo a redigibilidade do programa e impedindo o reuso desse trecho de código.

6.2.2.2 Tipos de Dados Simples

A idéia fundamental relacionada a essa técnica de modularização é combinar um grupo de dados relacionados em uma única entidade nomeada, a qual permite manipulá-los como um todo. O exemplo 6.18, em C, ilustra a definição e uso de um tipo de dados simples *struct pilha* (declarado com o nome *tPilha*).

```
#define max 100
typedef struct pilha {
  int elem[max];
  int topo;
} tPilha;
tPilha global;
void preenche (tPilha *p, int n) {
    for (p->topo=0; p->topo < n && p->topo < max; p->topo++)
      p->elem[p->topo] = 0;
    p->topo—;
}
main( ) {
  tPilha a, b;
  preenche(&a, 17);
  preenche(&b, 29);
  preenche(&global, 23);
}
```

Exemplo 6. 18 - Tipo de Dados em C

Note no exemplo 6.18 que a definição do tipo de dados *tPilha* possibilita tratar um vetor e uma variável inteira (representantes dos elementos e do topo da pilha) das variáveis *global, a* e *b* como uma entidade única.

O exemplo 6.18 destaca algumas das vantagens dessa técnica de modularização. Em primeiro lugar, ela torna o código mais reusável, uma vez que a definição de *tPilha* pode ser usada em diversos pontos do programa para definir variáveis e parâmetros desse tipo.

Ela também torna o código mais redigível. Caso não fosse possível definir o tipo de dados *tPilha* seria necessário criar variáveis e parâmetros do tipo vetor de inteiros e inteiro para representar isoladamente os elementos e os topos de cada variável ou parâmetro que representa uma pilha no programa. Isso certamente tornaria a escrita do código do programa muito mais trabalhosa.

Outra vantagem dessa técnica é dar maior legibilidade ao código explicitando que as variáveis e parâmetros criados com a estrutura *tPilha* devem armazenar valores relacionados ao conceito abstrato de pilha.

Em algumas LPs, a definição de tipos de dados também aumenta a confiabilidade da programação, assegurando que só possam ser atribuídos para as variáveis e parâmetros do tipo definido, váriaveis e valores desse mesmo tipo. Isto é, o compilador impede a atribuição de valores de outros tipos (mesmo com uma estrutura equivalente) a variáveis e parâmetros do tipo criado.

O maior problema da definição de tipos de dados simples é não possibilitar o ocultamento da informação, fornecendo acesso livre aos dados internos do tipo. Quando um tipo simples é criado, suas operações já estão pré-definidas pela própria LP. Normalmente, essas operações fornecem acesso indiscriminado à implementação do tipo, permitindo ao usuário alterar os seus dados sem garantir a manutenção de sua consistência.

Isso provoca dificuldades na legibilidade, confiabilidade e modificabilidade dos programas. O exemplo 6.19 mostra um programa que utiliza o tipo *tPilha*, definido no exemplo 6.18, para realizar operações sobre uma variável desse tipo e ilustrar a ocorrência desses problemas.

```
main( ) {
  tPilha a;
  preenche(&a, 10);
  a.elem[++a.topo] = 11;
  a.topo= 321;
}
```

Exemplo 6. 19 - Problemas no Uso de Tipos de Dados Simples

A legibilidade do exemplo 6.19 é sacrificada porque o usuário do tipo *tPilha* acessa e modifica seu conteúdo sem o uso de operações especiais. Dessa maneira, ele mistura o código relacionado com a implementação do tipo com o código relacionado ao seu uso, tornando os programas menos legíveis. Caso o tipo *tPilha* tivesse uma operação para empilhar elementos, o exemplo 6.19 se tornaria muito mais legível pois o comando de inclusão do valor *11* na pilha seria substituído por uma chamada a essa operação e o usuário não teria de implementá-la no seu próprio código.

A confiabilidade também é afetada porque o usuário tem acesso indiscriminado a estrutura interna do tipo. Assim, ele pode alterar inadvertidamente e de maneira inconsistente algum dos elementos do tipo. No exemplo 6.19, a última linha de código atribui um valor inconsistente ao topo da pilha. Tal alteração inviabiliza a utilização correta da variável *a*.

Por fim, a modificabilidade é reduzida porque a alteração da estrutura interna do tipo geralmente implica na necessidade de alterações nos trechos de código que usam o tipo. O exemplo 6.19 necessitaria ser totalmente alterado caso o implementador do tipo *tPilha* resolvesse implementá-lo usando uma lista encadeada.

6.2.2.3 Tipos Abstratos de Dados

Conceitualmente, tipos abstratos de dados (TADs) são conjuntos de valores que apresentam um comportamento uniforme definido por um grupo de operações (geralmente, um grupo de constantes iniciais e um conjunto de funções e procedimentos). O conjunto de valores é definido indiretamente através da aplicação sucessiva das operações, começando pelas constantes.

Em Linguagens de Programação, TADs [GUTTAG, 1977] são novos tipos de dados cuja representação e operações associadas são especificadas pelo programador que implementa o TAD. O implementador do TAD escolhe uma representação para os valores do tipo abstrato e implementa as operações em termos da representação escolhida. As implementações de TADs são usadas por programadores usuários para criar estruturas de dados desse novo tipo e para realizar operações sobre elas. O usuário do TAD só pode utilizar as operações definidas pelo implementador do TAD para manipular os dados daquele tipo. Assim, o usuário do TAD utiliza sua representação e operações, como uma caixa preta, para resolver seu problema.

Para permitir a implementação de TADs é essencial que a LP ofereça meios para o ocultamento da informação. Isto é, faz-se necessário tornar a implementação interna do TAD invisível para o usuário. Isso normalmente é feito através da especificação da interface do TAD. Na interface são incluídos todos os componentes do TAD (tipicamente, operações) que devem ser públicos. Componentes públicos são os que podem ser usados diretamente pelo código usuário.

A figura 6.2 apresenta um esquema de uso de um TAD qualquer. As operações do TAD atuam como um invólucro protetor dos dados do TAD. O código usuário só acessa ou modifica os dados do TAD através dessas operações.

Existem quatro tipos diferentes de operações que podem ser realizadas sobre um TAD. Operações construtoras são usadas para inicializar o TAD. Uma dessas operações deve ser usada antes de qualquer outra para garantir que o TAD foi inicializado corretamente. Dessa maneira, pode-se ter certeza que as demais operações serão realizadas apropriadamente. Operações consultoras são usadas para obter informações relacionadas com os valores do TAD. Já operações atualizadoras permitem a alteração dos valores do TAD. Por fim, operações destrutoras são responsáveis por realizar qualquer atividade de finalização quando o TAD não é mais necessário, tal como desalocar memória.

Todos os problemas mencionados com o uso de tipos simples são resolvidos com o uso de TADs, uma vez que o programador pode especificar e restringir as operações (um conjunto de subprogramas) a serem realizadas sobre o tipo. Programadores usuários desse tipo somente acessam os dados internos através do uso desses subprogramas. Assim, o código fica legível pois só inclui chamadas a essas operações, não necessitando incluir código de implementação do tipo. O código também fica confiável pois o usuário não mais efetua livremente mudanças nos dados. Isso só é feito através das operações oferecidas pelo implementador do tipo. Por fim, o código usuário geralmente não precisa ser alterado quando a implementação do TAD é modificada. Para isso ocorrer basta não haver alterações na interface do TAD.

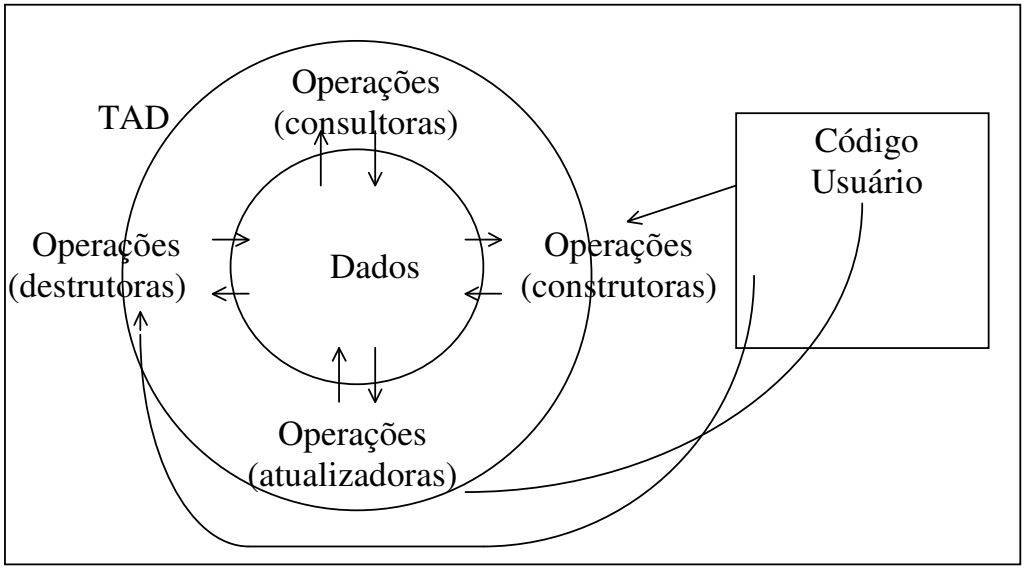

Figura 6. 2 - Representação Esquemática de um TAD

6.2.2.3.1 Simulação de TADS em C

C e PASCAL não oferecem mecanismos para a implementação de TADs. Mesmo assim, simular o uso de TADS nessas LPs é uma boa prática de

programação. Para isso, é necessário definir um tipo de dados simples e um conjunto de operações (subprogramas) que se aplicam sobre valores desse tipo. Tanto o tipo quanto suas operações devem ser disponibilizadas para os programadores usuários. O exemplo 6.20 ilustra a simulação do TAD pilha de números naturais em C.

```
#define max 100
typedef struct pilha {
  int elem[max];
  int topo;
} tPilha;
tPilha cria () {
  tPilha p;
  p.topo = –1;
  return p;
}
int vazia (tPilha p) {
  return p.topo == –1;
}
tPilha empilha (tPilha p, int el) {
  if (p.topo < max–1 && el >= 0)
     p.elem[++p.topo] = el;
  return p;
}
tPilha desempilha (tPilha p) {
  if (!vazia(p)) p.topo—;
  return p;
}
int obtemTopo (tPilha p) {
  if (!vazia(p)) return p.elem[p.topo];
  return –1;
}
```

Exemplo 6. 20 - Simulação de TAD Pilha em C

A implementação da simulação do TAD *tPilha* do exemplo 6.20 utiliza um vetor de inteiros para armazenar números naturais na pilha e um inteiro indicador do topo da pilha. Já as operações dessa simulação possuem as seguintes assinaturas:

cria: *void* → *tPilha*
vazia: *tPilha* → *int*
empilha: *tPilha x int* → *tPilha*
desempilha: *tPilha* → *tPilha*
obtemTopo: *tPilha* → *int*

A operação construtora *cria* retorna uma pilha inicializada. A operação consultora *vazia* recebe uma pilha e retorna um inteiro indicando se a pilha contém elementos. A operação atualizadora *empilha* recebe uma pilha e um inteiro e retorna a pilha com o inteiro no topo, caso seja um número natural. Se o inteiro não for um número natural, retorna a pilha tal como recebida. A operação atualizadora *desempilha* recebe uma pilha e a retorna sem o elemento do topo. Por fim, a operação consultora *obtemTopo* retorna o elemento do topo da pilha caso ela não esteja vazia. Caso contrário, é retornado um número inteiro não natural. Como não existe alocação dinâmica na implementação dessa simulação de TAD, não há necessidade de de uma operação destrutora.

O exemplo 6.21 mostra o uso da simulação do TAD do exemplo 6.20. Observe dessa vez que as funções *preenche* e *main* do código usuário não acessam a estrutura interna do tipo *tPilha*, realizando suas funcionalidades apenas através das operações do TAD.

```
tPilha global;
void preenche (tPilha *p, int n) {
  int i;
  for (i = 0; i < n; i++) *p = empilha (*p, 0);
}
main( ) {
  tPilha a, b;
  global = cria();
  a = cria();
  b = cria();
  preenche(&a, 17);
  preenche(&b, 29);
  preenche(&global, 23);
  a = empilha(a, 11);
  // a.elem[++a.topo] = 11;
  // a.topo= 321;
  // global = a;
}
```

Exemplo 6. 21 - Uso de Simulação de TAD em C

O uso disciplinado da simulação do TAD garante uma série de propriedades interessantes ao código usuário. Primeiramente, o código fica mais legível. Isso pode ser observado comparando a implementação da função *preenche* com a implementação da mesma função no exemplo *6.18*. Além dessa melhoria, o código usuário se torna muito mais fácil de ser redigido, uma vez que o programador usuário não mais necessita implementar o código das operações do TAD. Por fim, o código usuário não precisa ser alterado caso seja necessário realizar uma alteração na implementação do

tipo *tPilha* ou nas suas operações (isso é verdadeiro somente quando os cabeçalhos das funções não são modificados).

Contudo, além de não promover o encapsulamento das operações e dados em uma única unidade sintática, essa abordagem não impede o uso indisciplinado do TAD. Caso o programador esqueça ou não chame a operação *cria* antes de qualquer outra operação sobre uma variável do tipo *tPilha*, o uso correto do TAD ficará comprometido. Além disso, é importante observar que o programador usuário pode realizar operações adicionais sobre o TAD além das especificadas pelos subprogramas. Por exemplo, o programador pode acessar diretamente a estrutura de dados interna do TAD, o que acaba com todas as vantagens mencionadas no parágrafo anterior, ou mesmo realizar atribuição entre duas pilhas (isso poderia gerar problemas de compartilhamento de dados se a pilha fosse implementada como uma lista encadeada). Para se ter clareza sobre esse aspecto, note que as linhas comentadas no exemplo 6.21 seriam sentenças válidas do programa C caso não fossem comentários.

6.2.2.3.2 Uso de Interface e Implementação nos TADS em ADA

Uma abordagem usada em LPs para a implementação de TADs envolve a divisão da definição do TAD em duas unidades sintáticas do programa. Em uma unidade é definida a interface do TAD enquanto na outra é definida a sua implementação. Somente o que é definido na unidade de interface é exportado. Isso significa que os programadores usuários do TAD só terão acesso às entidades definidas nessa unidade. A unidade de implementação contém detalhes a respeito de como as entidades da interface são implementadas e também contém outras entidades utilizadas para auxiliar a implementação das entidades da interface. MODULA-2 e ADA são exemplos de linguagens que adotam essa abordagem. O exemplo 6.22 mostra a definição e uso do TAD *tPilha* em ADA.

```
package pilha_naturais is
  type tPilha is limited private;
  procedure cria (p: out tPilha);
  function vazia (p: in tPilha) return boolean;
  procedure empilha (p: in out tPilha; el: in integer);
  procedure desempilha (p: in out tPilha);
  function obtemTopo (p: in tPilha) return integer;
private
  max: constant integer := 100;
  type tPilha is record
    elem: array (1 .. max) of integer;
    topo: integer;
```

```
    -- topo: integer := 0;
  end record;
end pilha_naturais;

package body pilha_naturais is
  procedure cria (p: out tPilha) is
  begin
    p.topo := 0;
  end cria;
  function vazia (p: in tPilha) return boolean is
  begin
    return (p.topo = 0);
  end vazia;
  procedure empilha (p: in out tPilha; el: in integer) is
  begin
    if p.topo < max and then el >= 0 then
      p.topo := p.topo + 1;
      p.elem(p.topo) := el;
    end if;
  end empilha;
  procedure desempilha (p: in out tPilha) is
  begin
    if not vazia(p) then
      p.topo = p.topo – 1;
    end if;
  end desempilha;
  function obtemTopo (p: in tPilha) return integer is
  begin
    if not vazia(p) then
      return p.elem(p.topo);
    return –1;
  end topo;
end pilha_naturais;

use pilha_naturais;
procedure main is
  pilha: tPilha;
  numero: integer;
  cria (pilha);
  empilha (pilha, 1);
  empilha (pilha, 2);
  while not vazia(pilha) loop
    numero := obtemTopo(pilha);
    desempilha(pilha);
```

```
   end loop;
end main;
```

Exemplo 6. 22- TAD Pilha em ADA

O primeiro bloco do exemplo 6.22, formado pela unidade *package pilha_naturais*, contém a definição da interface do TAD *tPilha*. Nessa unidade são declarados o TAD *tPilha* e os procedimentos e funções correspondentes às operações do TAD. Embora a implementação[6.3] do tipo *tPilha* também seja colocada nessa unidade, somente o nome *tPilha* se torna acessível para o usuário, uma vez que a implementação de *tPilha* é feita na parte privada (*private*) da unidade. Observe ainda que somente os protótipos dos procedimentos e funções são colocados nessa unidade.

O segundo bloco do exemplo 6.22, formado pela unidade *package body pilha_naturais*, contém a definição da implementação do TAD *tPilha*. No caso específico do TAD *tPilha* essa unidade contém as implementações dos procedimentos e funções cujos protótipos foram declarados na unidade de interface.

O terceiro bloco do exemplo 6.22 contém um programa que usa o TAD *tPilha*. Note que o usuário do TAD tem de criar variáveis do tipo *tPilha* e só pode aplicar sobre essas variáveis as operações definidas na unidade de interface. De fato, ao se declarar o tipo *tPilha* como *limited* consegue-se garantir que as operações de atribuição e comparação também não possam ser aplicadas. Essa característica garante o uso disciplinado do tipo *tPilha*, conferindo uma maior legibilidade, redigibilidade, confiabilidade e modificabilidade (desde que os protótipos na interface não sejam alterados) ao código usuário.

A necessidade de chamar a operação *criar* antes de qualquer outra da pilha permanece nessa versão do TAD em ADA. Se o programador usuário não realizar essa operação, o uso das demais operações será incorreto. Isso reduz a confiabilidade do TAD. ADA oferece alguns mecanismos para contornar essa dificuldade. No exemplo 6.22, uma opção seria eliminar a necessidade da existência da operação *criar* fazendo a inicialização do *topo* da pilha no momento de sua declaração. Para isso, basta substituir a linha onde *topo* é declarado na unidade de interface pela linha comentada imediatamente subsequente.

[6.3] Em princípio, a implementação de *tPilha* deveria ser feita na unidade de implementação. No entanto, existe uma razão operacional para ser colocada na unidade de interface, a qual será explicada posteriormente ainda nesse capítulo.

6.2.2.3.3 TADS como Classes em C++

Com o advento da programação orientada a objetos, um novo tipo de entidade de computação, chamado de classe, foi introduzido nas Linguagens de Programação. C++ e JAVA utilizam esse conceito para implementar TADs.

Classes permitem ao programador criar um novo tipo de dados, incluindo de uma forma encapsulada tanto a representação do novo tipo, quanto as operações associadas ao tipo. Classes oferecem proteção dos dados do tipo através do uso de especificadores de acesso. Por exemplo, em C++ e JAVA, os dados da classe e as suas operações podem ser privados (só são visíveis para a implementação das operações do TAD) ou públicos (são visíveis para qualquer trecho do programa)[6.4].

Ao proporcionar os mecanismos de encapsulamento de dados e ocultamento da informação, necessários para a implementação de TADs, classes conferem ao código usuário as vantagens de maior legibilidade, redigibilidade, confiabilidade e modificabilidade. Além disso, classes tornam especiais as operações construtoras (resolvendo o problema da inicialização de TADs) e destrutoras, e dão suporte aos conceitos de herança e polimorfismo, fundamental para a orientação a objetos[6.5]. O exemplo 6.23 ilustra a implementação e uso do TAD *tPilha* em C++.

```
class tPilha {
  static const int max = 100;
  int elem[max];
  int topo;
public:
  tPilha () {
    topo = –1;
  }
  int vazia () {
    return topo == –1;
  }
  void empilha (int el);
  void desempilha (void);
  int obtemTopo (void);
}
tPilha::empilha (int el) {
  if (topo < max–1 && el >= 0)
```

[6.4] Existem outros tipos de especificadores de acesso em C++ e JAVA, os quais serão discutidos posteriormente.
[6.5] Esses conceitos serão estudados no capítulo 7.

```
      elem[++topo] = el;
}
void tPilha::desempilha (void) {
  if (!vazia()) topo—;
}
void tPilha::int obtemTopo (void) {
  if (!this->vazia()) return elem[topo];
  return –1;
}
main () {
  tPilha p;
  int n;
  p.empilha (1);
  p.empilha (2);
  while (! p.vazia ()) {
    n = p.obtemTopo ();
    p.desempilha ();
  }
}
```

Exemplo 6. 23 - TAD Pilha como Classe em C++

O exemplo 6.23 contém inicialmente a definição da classe *tPilha*. A estrutura de dados do tipo *tPilha* é privada[6.6] e só pode ser acessada pelo código usuário através das operações públicas declaradas na classe (funções declaradas após a palavra *public*). Além da implementação da estrutura de dados e da declaração das operações da interface de *tPilha*, essa unidade também contém a implementação de algumas das operações da interface da classe (por exemplo, a função *vazia*).

Observe que somente os protótipos das operações *empilha*, *desempilha* e *obtemTopo* são colocados na definição da classe. Suas implementações são externas à definição da classe. C++ requer o uso do operador de resolução de escopo *::* para relacionar a implementação dessas funções com as operações da classe *tPilha*.

Outro aspecto importante a ser observado na implementação de classes, em contraste à simulação de TADS em C e à implementação de TADS em ADA, é a ausência de necessidade de declaração do parâmetro *tPilha* nas operações da classe. LPs orientadas a objeto assumem que as operações de uma determinada classe são aplicadas a um objeto dessa classe, cujos membros podem ser acessados diretamente na implementação das suas operações. Note isso ocorrendo nas implementações das operações

[6.6] C++ considera os membros da classe privados quando o especificador de acesso é omitido.

de *tPilha*. As referências a *topo* e *elem* são feitas livremente sem qualquer associação a uma pilha, aumentando assim a redigibilidade desse código.

A chamada dessas funções no código usuário em *main* também é feita com uma sintaxe diferenciada. Por exemplo, a função *obtemTopo* é chamada como *p.obtemTopo()*, indicando que a operação *topo* é aplicada sobre a pilha *p*. Isso significa que as referências a *elem* e *topo* dentro da implementação da operação *obtemTopo* são referências aos atributos *elem* e *topo* do objeto *p*.

Em algumas situações, é necessário referenciar o próprio objeto sobre o qual se aplica a operação dentro da implementação dessa operação. Uma dessas situações ocorre quando o objeto deve ser passado como parâmetro em uma chamada de função. Para que isso seja feito, linguagens orientadas a objeto incluem implicitamente um novo membro na classe, o qual é usado especificamente para referenciar o próprio objeto dentro das operações. No caso de C++, esse membro é um ponteiro e se chama *this*. A operação *obtemTopo* do exemplo 6.23 usa *this* para chamar a operação *vazia*. Note que esse uso é meramente ilustrativo do conceito, uma vez que não é preciso usar *this* para chamar a função *vazia*, tal como ocorre na operação *desempilha*.

Em linguagens orientadas a objeto, as operações construtoras possuem características especiais. Além de não se especificar um tipo de retorno para essas funções, elas também são sempre chamadas no momento em que um objeto da classe é criado. Dessa maneira, não existe risco do usuário chamar uma outra operação antes de chamar a operação construtora. Em C++ e JAVA, as operações construtoras são identificadas pelo seu nome, o qual deve ser o mesmo da sua classe.

Em geral, é possível criar várias funções construtoras distintas para uma mesma classe através do mecanismo de sobrecarga[6.7]. Isso permite a aplicação de diferentes funções construtoras de acordo com o contexto de criação do objeto. No exemplo 6.23 a função *tPilha* é a única função construtora da classe. Observe que essa função é chamada implicitamente em *main* no momento da declaração da variável *p*. Funções construtoras também são chamadas em C++ no momento em que um objeto é criado dinamicamente através do operador *new*. Essa última é a única forma de chamada de funções construtoras em JAVA. A construção dinâmica de um objeto da classe *tPilha* em C++ pode ser vista na seguinte linha de código:

```
tPilha* i = new tPilha;
```

[6.7] O mecanismo de sobrecarga será estudado no capítulo 7.

Objetos em C++ precisam sempre ser criados através da chamada de um construtor. Caso não seja especificado qualquer construtor para uma classe, um construtor `default` é criado implicitamente pelo compilador. Um construtor `default` é chamado sem a passagem de parâmetros reais, tal como o construtor de *tPilha*. O construtor `default` inicializa os membros objetos da classe chamando os seus construtores `default`. Já os membros de tipos primitivos não são inicializados. Construtores `default` são especialmente importantes em C++ porque são os únicos que podem ser usados durante a criação de um vetor de objetos da classe. JAVA também cria um construtor `default` quando a classe não tem um. Esse construtor não realiza nada além das inicializações dos membros da classe tais como foram declaradas.

Objetos em C++ sempre podem ser copiados na sua inicialização ou através de atribuição. Para permitir a cópia na inicialização, C++ cria um construtor de cópia para a classe quando não foi especificado um. Para permitir a cópia na atribuição, o operador de atribuição de C++ possui uma forma de atuação padrão aplicável a todas as classes. A atuação do construtor de cópia criado pelo compilador e do operador de atribuição são equivalentes. Eles fazem a cópia membro a membro do objeto copiado para o objeto cópia. O exemplo 6.24 mostra o uso do construtor de cópia e do operador de atribuição em objetos da classe *tPilha*.

```
main() {
  tPilha p1, p2;
  tPilha p3 = p1;        // construtor de copia
  p3 = p2;               // operador de atribuicao
}
```

Exemplo 6. 24 - Construtor de Cópia e Operador de Atribuição em C++

A terceira linha do código do exemplo 6.24 mostra o uso do construtor de cópia para criar uma nova pilha *p3* com conteúdo idêntico à pilha *p1*. Já a última linha mostra o uso do operador de atribuição para fazer *p3* assumir um conteúdo idêntico à pilha *p2*.

Esse comportamento do construtor de cópia e do operador de atribuição pode não ser adequado em todas as situações. Em particular, quando existem membros da classe que são ponteiros, esse comportamento tende a gerar compartilhamento de dados no monte ao invés de cópia. Nesses casos, a classe deve ter a sua própria definição do construtor de cópia e do operador de atribuição, especificando como deve ser feita a cópia em cada situação.

O construtor de cópia é muito importante em C++ porque ele é chamado implicitamente nas passagens de parâmetros por cópia e nos retornos de

objetos em funções. Como JAVA copia referências na atribuição de objetos, nas passagens de parâmetros objeto e nos retornos de objetos em funções, não é necessário definir construtores de cópia e operadores de atribuição para as classes nessa linguagem.

Quando um objeto deixa de ser usado em C++, é necessário desalocá-lo. Se o objeto foi declarado localmente em um bloco, sua desalocação será feita ao final do bloco. Se o objeto foi alocado no monte, ele terá de ser desalocado explicitamente através do uso do operador *delete*. Contudo, a simples desalocação dos objetos em ambos casos pode não ser satisfatória. Por exemplo, se os objetos contiverem membros alocados no monte, esses membros não serão desalocados e provocarão vazamento de memória.

Para evitar isso, é preciso definir uma operação destrutora única para a classe especificando o que precisa ser feito antes de desalocar o objeto. Essa operação destrutora, quando definida na classe, sempre é chamada implicitamente quando um objeto dessa classe é desalocado, seja por fim de bloco ou por uso de *delete*. A operação destrutora em C++ possui uma sintaxe especial, consistindo do nome da classe prefixado pelo símbolo ~, tal como em *~tPilha*. Funções destrutoras em C++ não podem possuir parâmetros.

Como a desalocação de memória em JAVA é feita automaticamente pelo coletor de lixo, não existe necessidade de uma operação destrutora específica para esse fim nas classes dessa linguagem. Quando é necessário algum outro tipo de finalização, que não seja desalocação de memória, a classe deve possuir uma função específica para esse propósito e o código usuário deve chamá-la explicitamente quando o objeto não for mais necessário.

Em alguns contextos, pode ser necessário compartilhar alguns membros de uma classe por todos os objetos dessa classe, isto é, ao invés de se ter um membro para cada objeto da classe, tem-se apenas um membro usado por todos os objetos da classe. Esses membros são conhecidos como membros de classe e são caracterizados em C++ e JAVA pela ocorrência da palavra reservada *static* precedendo a definição desse membro.

Quando o membro de classe é uma variável, esse membro permanecerá alocado em memória durante toda a execução do programa, independentemente da existência ou não de objetos da classe. Quando o membro da classe é uma função, ela poderá ser chamada independentemente da existência ou não de um objeto da sua classe.

6.2.3 Pacotes

A modularização através de subprogramas e tipos ainda não é totalmente suficiente para atender as demandas existentes na construção de grandes sistemas. Sistemas construídos com uso exclusivo dessas técnicas acabam apresentando baixa granularidade, deixando as entidades do programa espalhadas pelo código, o que dificulta sua identificação.

Quando se considera a reutilização de código em grandes sistemas, também se constata que parte substancial dos componentes de código desses sistemas provém de fontes diversas, muitas vezes utilizados como uma caixa preta, isto é, o programa usuário desconhece como o código foi implementado pela sua fonte. Um problema com essa forma de reuso é a possibilidade de ocorrência de conflitos entre os nomes das entidades das diferentes fontes de código utilizadas e os nomes das entidades do sistema sendo criado.

Existem diferentes tipos de fontes de código, as quais são coleções de entidades reusáveis de computação. Elas se diferenciam pelo forma como são usadas. Bibliotecas são a fonte mais conhecida e comum. Elas agrupam tipos, variáveis, constantes e subprogramas usados para realizar funcionalidades similares. Por exemplo, a biblioteca padrão *stdio* de C oferece um conjunto de tipos, funções e macros para a realização de operações de entrada e saída.

Outros tipos de fontes de código são aplicações utilitárias, `frameworks` e aplicações completas. Aplicações utilitárias são sistemas construídos para serem usados como caixa preta por outros sistemas. O sistema usuário simplesmente invoca a aplicação utilitária quando necessita realizar a funcionalidade para a qual ela é destinada. `Frameworks` são implementações parciais de um determinado sistema. Para se criar uma aplicação específica completa, é necessário que o construtor da aplicação complemente o código fornecido pelo `framework`. Aplicações completas são sistemas prontos para serem usados pelo usuário final.

Pacotes são unidades sintáticas que podem agrupar diversas entidades de computação, das quais algumas são exportáveis (isto é, visíveis para o programa usuário do pacote) e outras não. LPs usam o conceito de pacotes para permitir a organização das entidades de computação em módulos funcionais como bibliotecas, aplicações utilitárias, frameworks e aplicações completas. De fato, pacotes também são usados internamente em cada uma dessas fontes de código para organizar as entidades de computação do sistema segundo sua arquitetura funcional.

Por possuírem um nome próprio, pacotes podem ser usados para resolver conflito de nomes de entidades provenientes de diferentes fontes. Quando

isso ocorre, basta usar a especificação completa do nome da entidade, cuja composição é formada pelo nome do pacote e pelo nome da própria entidade.

O exemplo 6.22 usa os pacotes de ADA para a implementação de um tipo abstrato de dados. Contudo, o conceito de pacotes em ADA é muito mais amplo, admitindo a definição e o agrupamento de várias entidades além de tipos e subprogramas.

6.2.3.1 Pacotes em C++

C++ usa a palavra *namespace* para definir um pacote. Cada conjunto de definições em uma biblioteca ou programa pode ser embutido em uma *namespace* de C++, e se alguma outra definição tem um identificador idêntico, mas em uma outra *namespace*, então é possível resolver o conflito de nomes. A criação de uma *namespace* é muito similar a criação de uma classe e é ilustrada no exemplo 6.25.

```
namespace umaBiblioteca {
  int x = 10;
  void f() {}
  class tC {}
}
```

Exemplo 6. 25 - Pacote em C++

Contudo, em contraste a uma definição de classe, uma nova definição de uma *namespace* não implica na redefinição da *namespace* e sim em uma continuação da definição anterior. O exemplo 6.26 continua a *namespace umaBiblioteca* inicialmente definida no exemplo 6.25.

```
// Adiciona mais entidades a umaBiblioteca
namespace umaBiblioteca {      // nao eh redefinicao!
  int y = 15;
  void g(){}
  // int x = 13;
}
```

Exemplo 6. 26 - Continuação de Pacote em C++

Após a definição do exemplo 6.26 a *namespace umaBiblioteca* é composta pelas variáveis *x* e *y*, pelas funções *f* e *g* e pela classe *tC*. Note que não seria possível redefinir a variável *x*, tal como na linha comentada no exemplo 6.26, pois isso geraria um conflito de definições interno à *namespace*. Tal fato não seria problema caso a nova definição de *x* fosse realizada em uma nova *namespace*, como ilustrado no exemplo 6.27.

```
namespace outraBiblioteca {
  int x = 13;
```

```
  void h(){}
}
```

Exemplo 6. 27 - Outro Pacote em C++

C++ usa o operador de resolução de escopo para especificar de qual *namespace* é a entidade de computação referenciada no programa. Dessa maneira, não existe conflito entre nomes coincidentes usados em diferentes pacotes. O exemplo 6.28 ilustra como isso é feito.

```
main() {
  umaBiblioteca::y = 20;
  umaBiblioteca::f();
  umaBiblioteca::x = 5;
  outraBiblioteca::x = 5;
  outraBiblioteca::h();
}
```

Exemplo 6. 28 - Usando Entidades Empacotadas em C++

C++ permite associar uma namespace a um outro nome, possibilitando ao programador não usar nomes grandes ou esquisitos dados à *namespace*. O exemplo 6.29 mostra isso sendo feito.

```
namespace bib1 = umaBiblioteca;
namespace bib2 = outraBiblioteca;
main() {
  bib1::y = 20;
  bib1::x = 5;
  bib2::x = 5;
  bib2::h(){};
}
```

Exemplo 6. 29 - Renomeando Pacotes em C++

Ter sempre de usar o nome da *namespace* junto com o operador de resolução de escopo para referenciar as entidades da *namespace* reduz significativamente a redigibilidade dos programas, tornando enfadonho o processo de escrita de programas. Para contornar esse problema, C++ usa a palavra reservada *using*, a qual possibilita usar declarações e definições de uma certa namespace sem o uso do operador de resolução de escopo. O exemplo 6.30 ilustra a utilização de *using*.

```
using namespace umaBiblioteca;
using namespace outraBiblioteca;
main() {
  y = 20;
  f();
  h(){};
```

```
  // x = 5;
  umaBiblioteca::x = 5;
  outraBiblioteca::x = 5;
}
```

Exemplo 6. 30 - A Palavra Reservada *using* em C++

Com a utilização de *using* não é mais necessário usar o nome da *namespace* para referenciar as entidades dos pacotes. Essa regra somente não é válida quando existem entidades de mesmo nome definidas em diferentes pacotes. No exemplo 6.30, caso não estivesse comentada, a linha na qual a variável *x* é atribuída produziria um erro na compilação. Nessas situações, o conflito se resolve através do uso do operador de resolução de escopo, como ilustrado nas duas últimas linhas desse exemplo.

Toda a biblioteca padrão de C++ está embutida na *namespace std*. Portanto, a inclusão da frase *using namespace std;* em um programa possibilita o uso de qualquer entidade pertencente a biblioteca padrão.

6.2.3.2 Pacotes em JAVA

JAVA usa a palavra reservada *package* para definir um pacote. Pacotes em JAVA contém um conjunto de classes relacionadas. O exemplo 6.31 mostra como definir duas classes chamadas *umaClasse* e *outraClasse* pertencentes ao pacote *umPacote*.

```
package umPacote;
public class umaClasse {}
class outraClasse {}
```

Exemplo 6. 31- Definição de Pacote em JAVA

Existem duas maneiras de usar as classes de um pacote na implementação de uma classe externa. Pode-se especificar o nome do pacote em toda referência à classe do pacote ou usar o comando *import* seguido do nome do pacote. O exemplo 6.32 ilustra essas duas maneiras

```
// usando diretamente o nome do pacote
umPacote.umaClasse m = new umPacote.umaClasse();
// usando import
import umPacote.*;
umaClasse m = new umaClasse();
```

Exemplo 6. 32 - Uso de Pacote em JAVA

Note que o uso de *import* torna o código mais redigível. A linha

```
import umPacote.*;
```

faz com que todas as classes de *umPacote* possam ser usadas pelo código usuário. Se o uso for apenas da classe *umaClasse*, é também possível especificar unicamente o nome dessa classe, tal como a linha

import umPacote.umaClasse;

Nesse caso, somente *umaClasse* poderá ser usada pelo código usuário.

Um aspecto interessante a respeito do conceito de pacote em JAVA é sua relação com a organização de arquivos do programa em diretórios do sistema de arquivo. Cada pacote deve necessariamente ter um diretório correspondente com o mesmo nome no qual são colocadas todas as classes do pacote. Além disso, tal como se pode criar subdiretórios no sistema de arquivos, também é possível organizar os pacotes em níveis hierárquicos. Por exemplo, todos os pacotes padrões da linguagem JAVA fazem parte do pacote *java*. Para se ter acesso às classes do pacote padrão *util* de JAVA é necessário usar o comando

import java.util.;*

JAVA faz uso da sequência de nomes de pacotes especificada no *import* para determinar em qual diretório deve encontrar as classes usadas pelo código usuário.

Pacotes em JAVA também são usados para definir um novo tipo de especificador de acesso para os membros das classes. Quando o especificador de acesso de um membro é omitido, JAVA considera esse membro como acessível pelos métodos de todas as classes do pacote. Embora isso possa parecer uma quebra na proteção dos dados, esse novo especificador foi criado para oferecer mais uma opção para os implementadores, liberando os construtores da classe de ter de disponibilizar funções públicas usadas exclusivamente pelos construtores do pacote para acessar e modificar os membros privados dessa classe. De fato, nessas situações, a solução apresentada por JAVA oferece maior proteção aos dados da classe, uma vez que os métodos de classes não pertencentes ao pacote não terão como acessar ou modificar esses dados. Além disso, caso não seja desejável tornar os membros da classe acessíveis para as outras classes do pacote, sempre se pode definir esses membros como privados. O exemplo 6.33 mostra o uso desse tipo de especificador de acesso em JAVA.

```
package umPacote;
public class umaClasse {
  int x;
  private int y;
  public int z;
}
class outraClasse {
```

```
  void f() {
    umaClasse a = new umaClasse();
    // a.y = 10;
    a.z = 15;
    a.x = 20;
  }
}
```

Exemplo 6. 33 - Especificador de Acesso Baseado em Pacotes JAVA

O membro *x* de *umaClasse* não possui especificador de acesso no exemplo 6.33. Como *outraClasse* pertence ao mesmo pacote, a sua operação *f* pode acessar e alterar o valor de *x* de *a* (um objeto de *umaClasse*). Como o membro *y* de *umaClasse* é privado, ele não pode ser acessado diretamente em *f*. Já o membro *z*, por ser público, pode ser acessado por qualquer método de qualquer classe.

6.3 Modularização, Arquivos e Compilação Separada

A maioria das técnicas de modularização vistas até agora podem ser aplicadas para a modularização de programas contidos em um único arquivo. Contudo, à medida que o tamanho dos programas cresce, alguns problemas práticos surgem com essa abordagem e acabam reduzindo a produtividade dos programadores.

Em primeiro lugar, a redação e modificação de programas de tamanho razoável em um único arquivo se torna mais difícil, uma vez que o programador necessita vasculhar todo o programa para encontrar partes a serem modificadas ou usadas. Após encontrar a parte procurada, e entender como usá-la ou modificá-la, esse processo deve ser repetido, agora para voltar ao ponto onde ele estava no programa. Todo esse processo de busca no arquivo se repete inúmeras vezes ao longo da construção ou modificação dos programas, tornando o processo de programação bem mais lento.

Outro problema com essa abordagem envolve o fato de que algumas entidades de programação (subprogramas, tipos, variáveis e constantes) podem ser reusadas em vários programas. Quando os programas são escritos em arquivos únicos, a única forma de reusar essas entidades é através da sua busca nos arquivos existentes e do processo de cópia do trecho de código que as implementa para o arquivo do programa que as usará.

A solução para resolver esses dois problemas é permitir a divisão do programa em vários arquivos separados. Cada arquivo seria responsável por definir uma ou mais entidades de programação relacionadas lógica e funcionalmente. Assim, os diversos arquivos servem como indexadores para

o programador encontrar a parte desejada do programa mais rapidamente. Além disso, quando um novo programa é construído, basta incluir os arquivos que implementam as entidades usadas pelo programa. Um exemplo dessa abordagem, em C, seria criar uma biblioteca de arquivos com código de implementação de entidades de programação (costuma-se usar a terminação **.c** no nome desses arquivos) e incluir os arquivos necessários no programa a ser criado.

No entanto, essa forma de modularização preserva outro problema também existente na abordagem com arquivo único. No caso de um programa pequeno, compilar e recompilar o programa por completo após uma modificação não demanda muito tempo e esforço. Mas, quando o programa cresce, o custo da compilação e recompilação também cresce substancialmente. Isso acaba atrasando o trabalho do programador e demandando muito mais esforço computacional do que o realmente necessário, uma vez que não se pode restringir a compilação ou recompilação às partes alteradas ou ainda não compiladas do programa.

Para resolver esse problema se torna necessário permitir a compilação separada dos vários arquivos que fazem parte do código fonte de um programa. A compilação de cada um desses arquivos gera arquivos objetos com código em linguagem de máquina. Depois da compilação de todos os arquivos fonte é preciso utilizar um programa especial, chamado ligador (`linker`), para coletar os arquivos objeto gerados e ligá-los em um único arquivo executável.

Dessa maneira, se alguma modificação é necessária, basta recompilar os arquivos fonte modificados e chamar novamente o ligador para montar o arquivo executável, sem que seja necessário recompilar todos os arquivos do programa.

Contudo, para permitir a compilação separada dos diferentes módulos, os compiladores tinham de relaxar certos tipos de verificação de erros. Por exemplo, ao se compilar um arquivo que fizesse chamadas a funções definidas em outro arquivo não era possível verificar se a função existia de fato, nem se os argumentos passados e valores retornados na chamada eram do tipo e número apropriado. A mesma dificuldade ocorria no caso do arquivo referenciar variáveis definidas em outro arquivo. Nesse caso, não era possível verificar se o tipo da variável era adequado às operações a qual essa variável era submetida.

Para contornar esse problema, C permite que variáveis e funções definidas em um arquivo sejam novamente declaradas no arquivo onde são usadas. Isso é feito através do uso dos protótipos das funções e da palavra *extern* precedendo a declaração da variável (ver capítulo 2 para maiores informações sobre declarações de variáveis e funções). No entanto, essa

abordagem não é suficientemente genérica pois pode ser necessário criar variáveis em um arquivo de tipos de definidos em outros arquivos.

Outra alternativa, mais geral, para permitir a compilação separada de arquivos, mantendo a possibilidade de verificação de erros, consiste basicamente em dividir os arquivos fontes em um arquivo de interface e outro de implementação.

No arquivo de interface são declaradas ou definidas as entidades de computação a serem exportadas, isto é, as entidades que serão usadas por outros arquivos. No arquivo de implementação são definidas as entidades de computação declaradas no arquivo de interface e as entidades usadas internamente, isto é, as que não são exportadas. Quando um arquivo, que usa entidades definidas em outros arquivos, necessita ser compilado separadamente, basta importar os arquivos de interface, os quais contém a informação necessária para fazer a verificação de tipos adequadamente.

É importante notar que normalmente os arquivos de interface contém definições de variáveis, constantes e tipos e apenas declarações de subprogramas. Assim, a parte mais trabalhosa e pesada da compilação se encontra nos arquivos de implementação, onde se encontram as definições de todos os subprogramas.

Além de permitir a compilação separada com verificação de erros, essa abordagem oferece mais uma maneira para a realização de ocultamento de informação. Isso é realizado definindo a entidade a ser ocultada unicamente no arquivo de implementação.

Exemplos dessa abordagem são os arquivos de definição (*DEFINITION MODULE*) e de implementação (*IMPLEMENTATION MODULE*) de MODULA-2, os arquivos .h e .c de C, .h e .cpp de C++, e o *package* e o *package body* de ADA.

Por possibilitar a compilação separada com verificação de erros e permitir a realização de ocultamento de informação, a separação dos arquivos fonte em interface e implementação se tornaram o principal instrumento usado por programadores para construir TADs em boa parte das LPs. No arquivo de interface se define o tipo da estrutura de dados e são declarados os protótipos dos subprogramas correspondentes às operações do TAD. No arquivo de implementação, são definidas as operações do TAD e quaisquer outras entidades de computação necessárias para a implementação das operações do TAD.

Um problema com o uso dessa abordagem é a necessidade de definição da estrutura de dados do TAD no arquivo de interface. Tal necessidade ocorre em LPs como C, ADA e C++ porque na compilação dos arquivos

usuários do TAD é necessário saber o tamanho a ser alocado para os valores, variáveis, constantes e parâmetros desse tipo[6.8].

Como visto na seção 6.2.2.3.1, esse problema em C é mais grave pois os membros do TAD se tornam visíveis para os programadores usuários, os quais podem acessar diretamente a estrutura interna do tipo sem usar as operações definidas no arquivo de interface. Isso, além de quebrar o ocultamento de informação e poder provocar inconsistências no uso do TAD, também diminui a modificabilidade do código, visto que uma alteração na implementação da estrutura de dados do TAD pode implicar na necessidade de reescrever o código usuário.

ADA e C++ minimizam os problemas de quebra de ocultamento de informação e necessidade de reescrita de código usuário permitindo ao programador declarar a definição da estrutura de dados do tipo como privada no próprio arquivo de interface. Contudo, caso seja necessário alterar a estrutura interna do TAD (mesmo mantendo inalterados os protótipos das operações da interface do TAD), além de ser necessário recompilar o próprio TAD, também é preciso recompilar os arquivos usuários. Isso pode ser especialmente inconveniente em situações em que o TAD é um tipo utilitário muito usado em várias aplicações.

MODULA-2 introduz o conceito de tipo opaco para contornar esses problemas com a implementação de TADs. Tipos opacos são ponteiros especiais utilizados no arquivo de interface para apontar para um tipo definido no arquivo de implementação. Os protótipos das operações do TAD se referem apenas ao tipo opaco. Assim, os programadores usuários somente podem realizar sobre o TAD as operações definidas no arquivo de interface, visto que os usuários não sabem para onde o tipo opaco aponta. Isso impede o uso inconsistente do TAD, visto que a sua estrutura interna não pode ser acessada nos arquivos usuários, e limita a necessidade de alteração do código usuário apenas às situações nas quais as operações declaradas na interface do TAD são alteradas.

Por sua vez, no arquivo de implementação, define-se a representação do tipo apontado pelo tipo opaco e implementa-se as operações do TAD levando-se em conta o fato do tipo opaco ser um ponteiro para esse tipo. Com o uso do tipo opaco só é necessário recompilar os arquivos usuários quando os protótipos das operações do TAD são alterados. Quando apenas a estrutura interna do TAD ou a implementação das operações é alterada, não é preciso recompilar os arquivos usuários. Isso ocorre porque os compiladores de MODULA-2 podem verificar o uso apropriado do TAD nos arquivos usuários (somente podem ser usados nas operações definidas

[6.8] Essa é a razão para a colocação da definição do tipo *tPilha* na unidade de interface no exemplo 6.22 em ADA.

no arquivo de interface) e também sabem quanto de memória é necessário alocar para criar valores, variáveis, constantes e parâmetros do tipo do TAD (é necessário alocar o espaço para um ponteiro).

Um grande inconveniente do uso do tipo opaco em MODULA-2 é obrigar ao programador a utilizar ponteiros e alocação dinâmica de memória na implementação das operações do tipo opaco. Isso reduz significativamente a redigibilidade e legibilidade do código, além de causar perda de eficiência, uma vez que se torna necessário fazer endereçamento indireto para acessar os valores desse tipo.

Os problemas de redigibilidade e legibilidade poderiam ser resolvidos caso os próprios compiladores de MODULA-2 se incumbissem de gerar código para as tarefas de derreferenciar os ponteiros e gerenciar a alocação e desalocação dinâmica de memória. Contudo, isso demandaria um sistema de gerenciamento de memória para a linguagem (o que tornaria muito mais complexa a implementação de MODULA-2) e reduziria ainda mais a eficiência do código. Essas são possíveis razões para os projetistas de ADA e C++ não oferecerem um mecanismo equivalente para a implementação de TADs nessas LPs.

Arquivos com código usuário em JAVA não necessitam ser recompilados quando a estrutura interna ou a implementação das operações da classe são alteradas. Isso é possível porque JAVA sempre aloca objetos no monte e possui um coletor de lixo. Assim, o código usuário só necessita alocar espaço para uma referência para o objeto (sempre do tamanho de um ponteiro). A criação de um objeto no código usuário é feita através da chamada de uma operação construtora da classe, a qual tem a responsabilidade de definir como o objeto será alocado no monte. Portanto, é importante atentar para o fato do código de alocação ser colocado no arquivo de implementação da classe. Por sua vez, a desalocação de memória é responsabilidade do coletor de lixo.

Algumas LPs só requerem a escrita do arquivo de implementação. Ao escrever esse arquivo, o programador especifica quais são as entidades exportáveis e quais não são. Em algumas LPs, o compilador gera automaticamente o arquivo de interface, o qual é incluído em uma biblioteca de unidades de interface para ser usado pelos outros módulos. Em outras LPs (por exemplo, JAVA), a informação sobre as entidades exportáveis é mantida no próprio arquivo compilado (os arquivos .class).

6.4 Considerações Finais

Nesse capítulo foi apresentada uma visão abrangente dos mecanismos oferecidos por linguagens de programação para apoiar a modularização de programas. Vários são os benefícios obtidos com a modularização,

com destaque para o aumento da legibilidade, redigibilidade, modificabilidade, reusabilidade, confiabilidade e eficiência de programação.

A legibilidade dos programas aumenta significativamente em consequência da divisão lógica do programa em unidades funcionais e da separação do código relacionado com a implementação do código relacionado ao uso de uma abstração. Muitas vezes, para entender genericamente todo um programa, basta analisar um pequeno trecho do código. Outras vezes, quando o objetivo é entender detalhamente uma funcionalidade específica do programa, isso só requer uma análise detalhada do módulo no qual essa funcionalidade foi implementada, sem demandar a análise de todo o código do programa.

A redigibilidade também é aprimorada porque um mesmo módulo pode ser usado em vários pontos do programa, não requerendo assim que o código de sua implementação tenha de ser reescrito várias vezes.

A modificabilidade dos programas é aumentada porque, em um grande número de vezes, a alteração da implementação de um módulo não requer a modificação dos seus programas e códigos usuários.

A reusabilidade é incrementada porque ao se criar módulos que cumprem uma certa funcionalidade, esse mesmo módulo pode ser reutilizado sempre que essa funcionalidade for necessária.

A eficiência da programação também é aumentada porque o programador tem mais facilidade para construir o programa dividindo-o em módulos menores. Assim, ele pode se dedicar à programação de cada um desses módulos ao invés de tentar escrever o programa como um todo, o que dificultaria sua implementação. Além disso, é possível compilar os módulos separadamente, evitando dessa maneira que qualquer modificação em um módulo implique em um atraso provocado pela necessidade de recompilação de todo o sistema computacional. Por último, o desenvolvimento do sistema pode ser dividido entre vários programadores, os quais podem codificar, compilar e testar os módulos paralelamente.

A modularização ainda aumenta a confiabilidade do código visto que cada módulo criado pode ser verificado independentemente e extensivamente antes de ser usado pelos outros módulos, permitindo assim que o uso e a reutilização desse código sejam feitos com mais garantias.

Técnicas de modularização são apropriadas para apoiar o processo de desenvolvimento de programas `top-down` (orientado a funcionalidades) ou `bottom-up` (orientado a dados).

O processo `top-down` (também conhecido pelo termo de refinamentos sucessivos) propõe um método de desenvolvimento hierárquico-funcional dos programas. Nessa perspectiva, um programa é visto como uma des-

crição de um processo para realização de uma determinada funcionalidade. Para atingir essa funcionalidade, ele é dividido em subprogramas, os quais são responsáveis por cumprir partes da funcionalidade geral do programa. Por sua vez, cada um desses subprogramas pode ser subdividido em novos subprogramas em um processo recorrente.

O processo `bottom-up` propõe um método de desenvolvimento baseado na identificação das entidades (objetos) reais existentes no domínio do problema no qual o programa atuará. Além de selecionar as entidades do domínio, é necessário identificar as características e comportamento dessas entidades. Cada uma das entidades identificadas é representada por uma estrutura de dados específica, normalmente através da definição de um tipo de dados.

É importante ressaltar a complementaridade dos métodos `top-down` e `bottom-up` ao invés da sua alternância. O processo `top-down` requer a representação de estruturas de dados usadas na comunicação entre os subprogramas. Essa representação é mais adequada quando a estrutura de dados possui um mapeamento claro para as entidades do domínio, conforme advoga o método `bottom-up`. Por sua vez, o processo `bottom-up` requer a representação do comportamento das entidades, os quais são implementados através de subprogramas. Essa representação é mais apropriada quando o subprograma é desenvolvido usando refinamentos sucessivos, isto é, o método `top-down`.

Finalmente, cabe lembrar que esse capítulo apresenta uma rápida pincelado sobre o conceito de classes e sua implementação em C++ e JAVA. Isso claramente não é suficiente para dirimir a maior parte das questões relacionadas com esse tema. O leitor interessado pode obter informações bastante completas sobre isso nos livros de C++ de Bjarne Stroustrup [STROUSTRUP, 2000] e Bruce Eckel [ECKEL, 2000] [ECKEL & ALLISON, 2003] e também no livro de JAVA de Bruce Eckel [ECKEL, 2002].

6.5 Exercícios

1. Implemente uma função sem parâmetros em C na qual se efetue a troca de dois valores. Utilize-a em um programa executor de trocas de valores entre diversos pares de variáveis. Explique porque os problemas de redigibilidade, legibilidade e confiabilidade seriam ainda mais graves nesse caso do que no exemplo 6.3.

2. É possível implementar, para cada tipo primitivo, funções em JAVA nas quais sejam trocados os valores dos seus parâmetros formais? Caso sua resposta seja afirmativa, implemente uma dessas funções e ex-

plique como funciona, destacando como a troca é feita. Em caso de resposta negativa, justifique. Existiria alguma diferença na sua resposta caso a troca fosse realizada entre parâmetros de um mesmo tipo objeto? Justifique.

3. Um TAD (tipo abstrato de dados) é definido pelo comportamento uniforme de um conjunto de valores. Embora a linguagem C não suporte a implementação do conceito de TADs, o programador pode simular o seu uso. Explique como isto pode ser feito. Descreva os problemas com essa aproximação.

4. Considere uma função em JAVA recebendo um objeto como único parâmetro e simplesmente realizando a atribuição de *null* ao seu parâmetro formal. Qual o efeito dessa atribuição no parâmetro real? Justifique.

5. JAVA não permite a criação de funções com lista de parâmetros variável, isto é, funções nas quais o número e o tipo dos parâmetros possam variar, tal como a função *printf* de C. Como JAVA faz para possibilitar a criação da função *System.out.println* com funcionalidade similar à função *printf* de C? Como o problema da falta de lista de parâmetros variável pode ser contornado de maneira geral pelo programador JAVA em situações nas quais esse tipo de característica pode ser útil? Compare essa abordagem geral de JAVA com a adotada por C e C++ em termos de redigibilidade e legibilidade.

6. Uma das vantagens de se programar usando a técnica de tipos abstratos de dados (TADs) é aumentar a modificabilidade dos programas. Isso ocorre porque a maior parte das alterações no código do TAD não implicam em necessidade de modificação do código usuário. Indique em quais tipos de alterações do código do TAD essa vantagem não pode ser aproveitada.

7. O uso de parâmetros em um subprograma visa aumentar as possibilidades de reuso desse subprograma. Normalmente, os valores dos parâmetros correspondem a dados que serão manipulados pelo subprograma. Contudo, os parâmetros podem servir também para alterar a funcionalidade do subprograma, tornando sua aplicação mais abrangente e aumentando sua possibilidade de reuso. Mostre, através de um exemplo em C, como valores do tipo ponteiro para função podem ser utilizados como parâmetros para tornar um determinado código mais reusável. Discuta como esse problema seria resolvido sem o uso do parâmetro ponteiro para função. Analise e compare as duas soluções

propostas em termos de redigibilidade, legibilidade, eficiência e reusabilidade.

8. Contrastando com a maioria das LPs imperativas, em C é possível criar funções cuja lista de parâmetros é variável (tome como exemplo, a função *printf*). Analise a abordagem adotada por C em comparação a:
   - abordagem adotada por MODULA-2, que não permite a existência de subprogramas com lista de parâmetros variável (enfoque a comparação nos conceitos de redigibilidade e confiabilidade)
   - abordagem adotada por PASCAL, que permite a existência de lista de parâmetros variável para funções pré-definidas da linguagem, tais como *read* e *readln*, mas não permite ao programador criar subprogramas com lista de parâmetros variável (enfoque a comparação nos conceitos de reusabilidade e ortogonalidade)

9. Tipos Abstratos de Dados (TADs) são uma ferramenta poderosa de projeto e programação. Descreva, de uma forma geral, como a programação com TADs pode ser feita em C, ADA e C++. Exemplifique com a descrição do tipo abstrato de dados *fila* de elementos inteiros (não é necessário implementar as operações da fila). Compare as três abordagens em termos de encapsulamento, ocultamento de informação, confiabilidade do uso e necessidade de alteração do código fonte usuário quando ocorrem alterações no código do TAD.

10. Considere o seguinte programa escrito na sintaxe de C:

```
void calculoMaluco (int a, int b) {
    a = a + b;
    b = a + b;
}
void main() {
        int valor = 0;
        int lista [5] = { 1, 3, 5, 7, 9 };
        calculoMaluco ( valor, lista [valor] );
}
```

Determine qual o valor das variáveis *valor* e *lista* após a execução do programa, supondo que:

a) A direção da passagem de parâmetros é unidirecional de entrada variável, o mecanismo é por cópia e o momento de passagem é definido pelo modo normal.

b) A direção é bidirecional de entrada e saída, o mecanismo é por referência e o momento é normal.
c) A direção é bidirecional de entrada e saída, o mecanismo é por referência e o momento é por nome.

Explique os resultados alcançados.

11. Os dois trechos de código seguintes apresentam definições (em arquivos .h) do tipo abstrato de dados *BigInt* em C e C++, respectivamente. *BigInt* é um tipo de dados que permite a criação de números inteiros maiores que *long*.

```
// C
struct BigInt {
    char* digitos;
    unsigned ndigitos;
}
struct BigInt criaBigIntC (char*);          // cria a partir de string
struct BigInt criaBigIntN (unsigned n);     // cria a partir de unsigned
struct BigInt criaBigIntB (struct BigInt);  // cria a partir de outro BigInt
void atribui (struct BigInt*, struct BigInt*);
struct BigInt soma (struct BigInt, struct BigInt);
void imprime (FILE* f, struct BigInt);
void destroi (struct BigInt);

// C++
class BigInt {
    char* digitos;
    unsigned ndigitos;
public:
    BigInt (const char *);
    BigInt (unsigned n = 0);
    BigInt (const BigInt&);
    void atribui (const BigInt&);
    BigInt soma (const BigInt&) const;
    void imprime (FILE* f = stdout) const;
    ~ BigInt();
};
```

Compare essas definições em termos de encapsulamento, proteção dos dados e confiabilidade das operações de inicialização e terminação das instâncias desse TAD. Justifique sua resposta.

A operação *atribui* da classe *BigInt* também poderia ser definida através do seguinte protótipo:

```
void atribui(BigInt);
```

Compare essa definição com a usada na classe *BigInt* em termos de eficiência de execução e confiabilidade na proteção dos dados do parâmetro formal. Explique sua resposta.

12. Ao se modificar a estrutura de dados de uma classe em C++, ainda que mantida a mesma interface (isto é, as assinaturas das operações públicas da classe continuam idênticas às existentes antes da alteração), é necessário recompilar não apenas o código da própria classe, mas também os programas usuários dessa classe. Explique porque isso ocorre levando em conta que não há alterações no código fonte dos programas usuários. Explique como JAVA evita a necessidade de recompilação nesses casos. Apresente razões para justificar a não incorporação dessa característica em C++?

13. Execute o seguinte trecho de código em C++, mostrando o seu resultado.

```
void incrementa (int& x, int& y) {
    x = x + y;
    y++;
}
main ( ) {
  int a [ ] = { 1, 2, 3 };
  for ( int i = 0; i < 3; i++ )  {
    incrementa ( a [ i ], a [ 1 ] );
    cout << a [ i ]  << "\n" ;
  }
}
```

Explique como o resultado foi produzido. A execução desse código produz algum efeito estranho prejudicial a legibilidade? Justifique sua resposta.

14. Em uma LP, o momento da avaliação dos parâmetros reais, durante a passagem de parâmetros, pode ser por definido pelo modo normal (`eager`), por nome (`by name`) ou preguiçoso (`lazy`). Explique o significado de cada um desses modos. Execute três vezes o programa C seguinte, supondo que a linguagem adotasse, em cada execução, um tipo diferente de modo de avaliação. Explique os resultados alcançados.

```
void avalia (int c) {
   int i;
   for (i = 0; i < 3; i++) {
```

```
        printf ("%d\n", c);
    }
}

void main ( ) {
    int j = 0;
    avalia (j++);
}
```

15. Descreva como deve ser feita a operação de desalocação de memória em um tipo abstrato de dados lista de inteiros em C, C++ e JAVA. Compare as diferentes abordagens adotadas por essas LPs na implementação e uso dessa operação em termos de redigibilidade, confiabilidade e eficiência.

16. Qualifique os tipos de passagem de parâmetros oferecidos por C, C++ e JAVA em termos da direção da passagem e do mecanismo de passagem. Compare-os em termos de confiabilidade e eficiência.