Suporte a Programação Genérica em Linguagens

Leandro M. Barros
UNISINOS – Universidade do Vale do Rio dos Sinos
Centro de Ciências Exatas e Tecnológicas
São Leopoldo, RS, Brasil
lmb@exatas.unisinos.br

RESUMO

Este artigo apresenta recursos oferecidos por linguagens de programação do ponto de vista da programação genérica. Estes recursos são discutidos com base em um conjunto de problemas para os quais soluções genéricas são particularmente desejáveis. São levados em consideração implementações genéricas baseadas em recursos de baixo nível, em orientação a objetos e em tipos e procedimentos parametrizáveis.

Palavras-chave: Programação genérica, procedimentos genéricos, coleções.

1 INTRODUÇÃO

Programas normalmente são projetados e implementados para resolver um problema específico. Porém, muitas vezes, o problema específico que se deseja resolver é apenas uma instância de uma classe maior de problemas fortemente relacionados, que podem ser solucionados por programas semelhantes. A programação genérica busca oferecer meios de projetar e implementar programas que não se limitam a resolver uma instância de um problema.

O projeto de uma solução genérica exige que o problema seja expresso através de conceitos. Cada instância do problema será diferente na forma com que os conceitos se materializam, mas os conceitos propriamente ditos permanecem os mesmos. Uma linguagem adequada para a implementação de uma solução genérica deve permitir que os programas sejam escritos em termos de conceitos e deve dar ao programador a possibilidade de configurar a forma com que estes conceitos se materializam durante a resolução de um problema real. Além disso, mantém-se o desejo de que as soluções sejam eficientes, seguras e elegantes.

Na programação não-genérica, é comum especificar, projetar e implementar algo como "um procedimento para ordenar um *array* de números inteiros em ordem crescente". Na programação genérica, esta especificação estaria muito próxima a "uma forma de ordenar uma coleção de elementos segundo algum critério de ordenação". A primeira especificação, não-genérica, é feita com termos específicos (*array*, inteiros...), enquanto que a segunda, genérica, é feita termos de conceitos (coleção, elementos...).

Este artigo mostra como as linguagens de programação atuais fornecem meios de se programar genericamente. Soluções genéricas baseadas em recursos de baixo nível, em orientação a objetos e em tipos e procedimentos parametrizáveis são analisadas com base em um conjunto de problemas básicos para os quais uma solução genérica é particularmente desejada.

A próxima seção apresenta problemas básicos que a programação genérica se propõe a solucionar e discute aspectos que estariam presentes em uma solução ideal. Em seguida, na seção três, diversas formas de suporte à programação genérica são analisadas com base nos três problemas básicos. Finalmente, a seção quatro apresenta conclusões.

2 PROBLEMAS BÁSICOS

Esta seção apresenta alguns problemas básicos em que o papel da programação genérica pode ser facilmente percebido e discutido. Além dos problemas propriamente ditos, são mostradas características desejadas para uma solução genérica ideal. Estes problemas são apenas uma pequena amostra de situações em que a utilização da programação genérica pode ser benéfica. Muitos outros exemplos interessantes são apresentados por Alexandrescu em [1].

Procedimentos Genéricos

É natural utilizar subprocedimentos para encapsular algoritmos. Os procedimentos que implementam estes algoritmos, normalmente, recebem parâmetros que permitem utilizar a mesma implementação para casos ligeiramente diferentes do mesmo problema. Na programação genérica, os parâmetros que podem ser passados são mais ricos, permitindo que uma gama maior de instâncias de um mesmo problema possa ser resolvida pelo mesmo subprocedimento.

Dois subprocedimentos muito comuns são funções que recebem dois valores como parâmetro e retornam o maior ou o menor deles. Habitualmente estas funções são chamadas de `Min` e `Max`. Implementações genéricas de `Min` e `Max` deveriam ter as seguintes características:

- *As funções devem ser independentes do tipo usado como parâmetro.* A mesma função deve ser capaz de comparar números inteiros, números de ponto-flutuante, *strings*, ou qualquer outro tipo de dados para o qual os conceitos de "maior" e "menor" façam sentido.
- *O usuário deve ter a possibilidade de definir (ou redefinir) os conceitos de "maior" e "menor".* Isto é necessário para comparar tipos para os quais estes conceitos podem variar. Por exemplo: ao comparar duas pessoas, qual delas é "a maior"? Dependendo da situação, "a maior" poderia ser a mais alta, a mais velha ou a mais pesada. A definição do "maior" ou "menor" também poderia depender de alguma computação mais complexa, que levasse várias destas características em consideração.

Outro exemplo de subprocedimento é uma função `Somatorio`, que calcula $\sum_{x=a}^{b} termo$, onde *termo* é uma função de x. Características desejáveis para uma implementação genérica deste subprocedimento são:

- *Os limites a e b não devem ser pré-definidos.* O usuário, obviamente, deve ser capaz de passar os limites a e b como parâmetros na chamada da função.
- *O termo não deve ser pré-definido.* O usuário deve ser capaz de utlizar a mesma implementação de `Somatorio`, não importando se o *termo* é x^2, $\frac{1}{x}$, $x^3 + 2$ ou qualquer outra função de x.

Coleções e Operações Associadas

Programas freqüentemente precisam trabalhar com coleções de dados. Em alguns casos, os recursos da própria linguagem (*arrays*, por exemplo) são suficientes. Em outros, porém, é preciso criar estruturas de dados mais complexas, como listas encadeadas, árvores ou tabelas *hash*. Além das estruturas propriamente ditas, é necessário um conjunto de operações que permitam utilizá-las. Algumas destas operações modificam o conteúdo da coleção de dados (por exemplo, operações de inserção e retirada); outras trabalham sobre a coleção como um todo (como ordenação e pesquisa); e outras permitem acessar sistematicamente os elementos contidos na coleção (permitindo, por exemplo, aplicar uma dada operação sobre cada um deles).

Uma solução genérica para estes problemas deveria ter as seguintes características:

- *Coleções devem ser independentes do tipo de dado armazenado.* A única diferença entre uma fila de números inteiros e uma fila de *strings* é o tipo de dado armazenado. A implementação de uma fila genérica deve ser capaz de armazenar elementos de qualquer tipo de dado. Isso não significa necessariamente que a coleção deva ser heterogênea[1].

[1] Uma coleção heterogênea é aquela capaz de armazenar dados de tipos diferentes ao mesmo tempo.

- *Operações devem ser independentes da coleção.* Em geral, os algoritmos que podem ser utilizados para implementar operações sobre coleções não dependem da coleção propriamente dita. Eles apenas exigem que a coleção tenha um certo conjunto de propriedades. Por exemplo: uma pesquisa binária pode ser feita em qualquer coleção, desde que ela esteja ordenada e permita acessar seu n-ésimo elemento em tempo constante. Assim, não deve ser necessário implementar os mesmos algoritmos para coleções diferentes. Por outro lado, pode ser interessante a possibilidade de utilizar algoritmos diferentes dependendo da coleção. Por exemplo: um algoritmo de ordenação que seja bom para vetores pode ser ineficiente para listas encadeadas.
- *Coleções devem ser intercambeáveis.* Diferentes coleções devem ter interfaces compatíveis, de modo que elas sejam intercambeáveis. Evidentemente cada coleção possui suas próprias características, que deverão ser decisivas na escolha da melhor estrutura para um determinado caso. Em particular, é interessante notar as características de desempenho inerentes às estruturas. Por exemplo: uma lista encadeada permite que um elemento seja inserido em uma posição arbitrária em tempo constante, mas o acesso ao n-ésimo elemento ocorre em tempo proporcional a n.

3 SUPORTE EM LINGUAGENS DE PROGRAMAÇÃO

Nesta seção são apresentadas e analisadas alternativas encontradas nas linguagens de programação atuais para solução dos problemas discutidos na seção anterior.

Recursos de Baixo Nível

Mesmo em linguagens que não oferecem nenhum recurso particularmente adequado para programação genérica, buscam-se meios de implementar soluções adaptáveis para problemas comuns. Normalmente isso é feito através da exploração de recursos de baixo nível. A linguagem C [7] é a melhor fonte de exemplos deste tipo.

Procedimentos Genéricos

Em programas escritos em C, é extremamente comum encontrar as funções `Min` e `Max` definidas como macros do pré-processador:

```
1 #define Min(a,b) (a) < (b) ? (a) : (b)
2 #define Max(a,b) (b) < (a) ? (a) : (b)
```

Esta solução é genérica no sentido de que aceita parâmetros de qualquer tipo para o qual o operador $<$ esteja definido. Por exemplo:

```
1 int i = 2;
2 int j = 3;
3 float f = 3.1415;
4 float g = 2.7182;
5 printf ("O menor entre %d e %d é %d.\n", i, j, Min(i,j));
6 printf ("O maior entre %d e %d é %d.\n", i, j, Max(i,j));
7 printf ("O menor entre %f e %f é %f.\n", f, g, Min(f,g));
8 printf ("O maior entre %f e %f é %f.\n", f, g, Max(f,g));
```

Neste exemplo, como pode se observar entre as linhas 5 e 8, as "funções" `Min()` e `Max()` são chamadas com parâmetros do tipo `int` e `float`.

Mas esta solução apresenta um grande problema: ela está permanentemente atrelada ao operador $<$. Como conseqüência, é impossível utilizar `Min` e `Max` para tipos definidos pelo usuário, que não aceitam este operador. Tampouco é possível redefinir os conceitos de "maior" e "menor". Um problema adicional desta solução é a utilização do pré-processador da linguagem C: em certas circustâncias a utilização de `Min` e `Max` pode apresentar efeitos colaterais indesejados[2].

[2] A chamada `Min(i--,j--)`, por exemplo, não se comporta como o esperado: a variável com o menor valor é decrementada duas vezes.

A biblioteca padrão da linguagem C possui duas funções interessantes do ponto de vista da programação genérica: `qsort()` e `bsearch()`, que implementam operações de ordenação e pesquisa sobre *arrays*. Como se desejava que estas funções fossem capazes de operar sobre *arrays* de qualquer tipo, elas foram projetadas de modo que um dos parâmetros que devem ser passados a elas é um ponteiro para uma "função de comparação". Esta função é usada sempre que for necessário comparar dois elementos do *array*. Este mesmo artifício pode ser usado para implementar a função `Somatorio`:

```
1  typedef float (*Termo)(int);
2
3  float Somatorio (int a, int b, Termo termo)
4  {
5     int i;
6     float acum = 0.0;
7
8     for (i = a; i <= b; ++i)
9        acum += termo(i);
10
11    return acum;
12 }
```

A primeira linha define o que é um `Termo`: uma função que recebe um inteiro como parâmetro e retorna um valor de ponto flutuante. A função `Somatorio`, que aparece a partir da linha 3, é bastante simples: ela simplesmente acumula os valores retornados por uma série de chamadas a `termo()`. A utilização da função `Somatorio` é mostrada a seguir:

```
1  float UmSobreX (int x)
2  {
3     return 1.0 / x;
4  }
5
6  float XAoQuadrado (int x)
7  {
8     return x * x;
9  }
10
11 int main()
12 {
13    printf ("Primeiro somatório = %f.\n", Somatorio (2, 5, UmSobreX));
14    printf ("Segundo somatório =  %f.\n", Somatorio (5, 9, XAoQuadrado));
15 }
```

Inicialmente são definidas duas funções, `UmSobreX()` e `XAoQuadrado()`, que podem ser usadas como termos porque seguem o padrão de receber um parâmetro inteiro e retornar um valor de ponto flutuante. Na função `main()`, a partir da linha 11, são feitas duas chamadas à função `Somatorio()`, cada uma usando um termo e um intervalo diferente.

Esta implementação de `Somatorio` satisfaz aos dois requisitos citados na seção 2 e não possui nenhum problema significativo. Ela permite, inclusive, que o compilador faça a verificação de tipos, evitando erros de tempo de execução.

Coleções

Coleções genéricas podem ser escritas em C se elas forem feitas para armazenar ponteiros para `void`. Como estes ponteiros podem apontar para qualquer tipo de dado, uma coleção baseada nesta idéia pode armazenar elementos de qualquer tipo. A seguir é mostrado um exemplo desta técnica: uma pilha genérica.

```
1  #define TAM_PILHA 10
2
3  typedef struct _Pilha
4  {
5      int topo;
6      void* dados[TAM_PILHA];
7  } Pilha;
8
9  void ConstroiPilha (Pilha* pilha)
10 {
11     pilha->topo = 0;
12 }
13
14 void PushPilha (Pilha* pilha, void* dado)
15 {
16     if (pilha->topo == TAM_PILHA)
17         abort(); // Overflow
18     else
19         pilha->dados[pilha->topo++] = dado;
20 }
21
22 void* PopPilha (Pilha* pilha)
23 {
24     if (pilha->topo == 0)
25         abort(); // Underflow
26     else
27         return pilha->dados[--pilha->topo];
28 }
```

Para facilitar a compreensão, a pilha apresentada é bastante simples. Sua capacidade é estática, conforme o valor definido logo na primeira linha. A pilha é representada por uma pequena estrutura, definida entre as linhas 3 e 7. Além disto, são definidas três operações, que permitem inicializar, inserir e retirar dados de uma pilha. Um exemplo de uso desta pilha é mostrado a seguir:

```
1  int main()
2  {
3      int *i1, *i2;
4      Pilha pilha;
5
6      // Aloca e inicializa dados
7      i1 = (int*)malloc (sizeof(int));
8      i2 = (int*)malloc (sizeof(int));
9      *i1 = 111;
10     *i2 = 222;
11
12     // Inicializa a pilha
13     ConstroiPilha (&pilha);
14
15     // Insere dados na pilha
16     PushPilha (&pilha, i1);
17     PushPilha (&pilha, i2);
18
19     // Remove dados da pilha
20     i1 = (int*)PopPilha (&pilha);
21     i2 = (int*)PopPilha (&pilha);
22
23     // Libera a memória dos dados
24     free (i1);
25     free (i2);
26 }
```

Neste exemplo a pilha foi usada para armazenar ponteiros para inteiros, mas ela poderia ter sido usada para armazenar ponteiros para qualquer tipo de dados que se desejasse. Neste aspecto, a pilha pode ser considerada genérica. Contudo, por ser baseada em recursos de baixo nível, a utilização desta solução não pode ser considerada segura. Há muitos detalhes que não podem ser verificados pelo compilador,

aumentando a possibilidade de erros em tempo de execução. O caso mais evidente é a perda de toda a checagem de tipos, pois a coleção trabalha com ponteiros para `void`. Como pode se ver nas linhas 20 e 21, *casts* são necessários para restaurar o tipo original dos elementos na sua retirada.

Outras oportunidades para erros são conseqüência da coleção armazenar apenas ponteiros, e não "elementos completos". As chances de ocorrerem "vazamentos de memória" aumentam[3], pois a alocação (linhas 7 e 8) e liberação (linhas 24 e 25) de memória deve ser manualmente controlada pelo programador. Todos estes são problemas que irão se manifestar apenas em tempo de execução e podem ser muito difíceis de corrigir.

Orientação a Objetos

Os conceitos introduzidos pela orientação a objetos trouxeram algumas novas possibilidades para a programação genérica. Nesta seção são discutidas algumas formas de utilizar a orientação a objetos para construir soluções adaptáveis.

Procedimentos Genéricos

A orientação a objetos pouco pode fazer para auxiliar na criação de funções `Min` e `Max` genéricas. Entretanto, uma boa solução para o problema do `Somatorio` pode ser desenvolvida se uma classe abstrata for utilizada para modelar um termo. O exemplo a seguir mostra como isto seria feito em Java:

```
1  abstract class Termo
2  {
3     abstract public float calcula (int x);
4  };
5
6  class XAoQuadrado extends Termo
7  {
8     public float calcula (int x) { return x * x; }
9  };
10
11 class Somatorio
12 {
13    public float faz (int a, int b, Termo termo)
14    {
15       float acc = 0.0f;
16       for (int i = a; i <= b; ++i)
17          acc += termo.calcula (i);
18       return acc;
19    }
20 };
21
22 public class TesteSomatorio
23 {
24       static public void main (String[] argv)
25       {
26          XAoQuadrado termo = new XAoQuadrado();
27          Somatorio somatorio = new Somatorio();
28          float res = somatorio.faz (0, 10, termo);
29          System.out.println (res);
30       }
31 };
```

A partir da linha 1 é definida uma classe abstrata `Termo`. Esta classe possui apenas um método, `calcula()`, que deve ser implementado nas subclasses de modo a retornar o valor do termo em função do parâmetro passado. A classe `XAoQuadrado`, mostrada a partir da linha 6, mostra como pode ser implementado um termo que calcula x^2. O somatório propriamente dito é computado por um método de outra classe, como mostrado a partir da linha 11. A princípio, o somatório poderia ser resolvido por um método da própria classe `Termo`, mas isto estaria indo contra os princípos da programação

[3]Assumindo que se esteja usando uma linguagem como o C, na qual o gerenciamento do *heap* é totalmente feito pelo programador.

genérica: dois conceitos distintos (termo e somatório), estariam sendo fortemente acoplados. Idealmente os conceitos devem ser fracamente acoplados e deve ser possível combiná-los de forma ortogonal.

Coleções

O mecanismo de herança permite a criação de coleções genéricas. Se todos os objetos que se deseja armazenar são de uma classe `A` ou de alguma classe derivada de `A`, então uma coleção de `A`s é capaz de armazenar todos os objetos desejados. Esta técnica é utilizada pela bilioteca padrão da linguagem Java [6]: as suas coleções, representadas por classes como `Vector` e `LinkedList`, foram projetadas para armazenar objetos da classe `Object`. Como todos os objetos em Java são `Object`s, as coleções Java podem armazenar qualquer objeto instanciado em um programa.

É interessante notar que este método baseia-se exatamente no mesmo princípio que as "coleções de ponteiros para `void`" da linguagem C: armazenar dados de um tipo para o qual qualquer outro tipo possa ser convertido. Como conseqüência, alguns dos mesmos problemas discutidos anteriormente se manifestam com a utilização desta técnica. O exemplo a seguir mostra como uma coleção Java do tipo `Vector` pode ser usada e como um erro pode acontecer devido à sua má utilização.

```
1  public class TesteColecoesJava
2  {
3          static public void main (String[] argv)
4          {
5              // Cria um vetor e dois inteiros
6              Vector vetor = new Vector();
7              Integer i1 = new Integer (1234);
8              Integer i2 = new Integer (4321);
9
10             // Insere os inteiros no vetor
11             vetor.add (i1);
12             vetor.add (i2);
13
14             // Retira os inteiros do vetor
15             Integer certo = (Integer)vetor.get (0);    // ok
16             Float errado = (Float)vetor.get (1);       // erro!
17         }
18 };
```

O grande problema desta solução pode ser visto nas linha 15 e 16: a retirada de um elemento da coleção exige um *cast* para "restaurar" o tipo do elemento armazenado. Isto impossibilita que o compilador faça uma verificação de tipos completa, o que cria a possibilidade de erros em tempo de execução. A linha 16 mostra uma situação em que um erro ocorreria em tempo de execução.

A implementação das operações sobre as coleções é um problema à parte. Uma idéia inicial seria colocar todas as operações como métodos da classe `Colecao`. Como cada coleção possui características muito particulares, a princípio esta solução exigiria que todas as operações fossem implementadas para cada uma das coleções. Do ponto de vista da programação genérica isto não é aceitável.

Uma alternativa seria oferecer acesso aos elementos de todas as coleções através de uma interface comum, a partir da qual todas operações possam ser implementadas. As operações seriam implementadas na própria classe `Colecao`, utilizando apenas os métodos desta interface comum. Assim, automaticamente, todas coleções que implementassem a interface comum corretamente iriam suportar todas as operações definidas na classe `Colecao`. E nada impediria que, quando fosse desejado, alguma classe derivada de `Colecao` "reimplementasse" uma ou mais operações. Isso pode ser interessante, por exemplo, quando uma determinada coleção permite que alguma operação seja implementada de forma mais eficiente que a implementação padrão.

Há, contudo, um problema nesta solução. As operações estão fortemente acopladas à classe `Colecao`, de modo que a criação de novas operações exigiria que ela fosse alterada. Claramente soluções genéricas exigem que as coleções e as operações sobre as coleções sejam desacopladas. Por isso, implementações genéricas de operações sobre coleções costumam ser feitas em procedimentos independentes da classe `Colecao`.

Tipos e Procedimentos Parametrizáveis

Como mostrado nas seções anteriores, nem as "técnicas tradicionais" e nem a orientação a objetos são capazes de resolver satisfatoriamente alguns problemas básicos relacionados à programação genérica. Em particular, as implementações genéricas mostradas tendem a enfraquecer o mecanismo de tipos da linguagem. Isso fica muito claro para as coleções que exigem *casts* na retirada de elementos, mas também está presente em outros casos, como por exemplo no uso do pré-processador da linguagem C para implementar as funções `Min` e `Max`: basta lembrar que o pré-processador é simplesmente um expansor de macros, que não tem nenhum conhecimento a respeito de tipos.

Recursos que permitem a criação de tipos e procedimentos parametrizáveis foram projetados especificamente para dar suporte à programação genérica. Mais importante, eles permitem que algoritmos e estruturas de dados genéricas sejam fortemente tipadas.

A idéia por trás dos tipos e procedimentos parametrizáveis é a generalização de um conceito muito simples: a passagem de parâmteros. Um procedimento é parametrizado toda vez que recebe um parâmetro na sua chamada, ou seja, o resultado da execução do procedimento vai ser diferente, dependendo do parâmetro passado. Em uma linguagem orientada a objetos, a parametrização de um objeto é feita toda vez que um parâmetro é passado para o seu construtor durante a sua instanciação.

A maioria das linguagens de programação admite apenas que valores (ou refrências para valores) sejam passados como parâmetros. Na implementação de estruturas de dados e algoritmos genéricos, muitas vezes, é desejável passar tipos ou subprocedimentos como parâmetros.

Desta forma, "procedimentos e tipos parametrizáveis" nada mais são do que procedimentos e tipos para os quais podem se passar parâmetros mais interessantes do ponto de vista da programação genérica. Nesta seção são mostradas soluções para os problemas básicos, utilizando estes recursos. Serão dados exemplos em C++ [5] e Ada [3], as duas linguagens mais representativas que possuem este recurso.

Procedimentos Genéricos

As funções `Min` e `Max`, para serem realmente genéricas, precisam receber quatro parâmetros: um tipo, uma função que compare dois elementos deste tipo, e dois elementos deste mesmo tipo. A seguir é mostrado como a função `Min` poderia ser implementada em Ada:

```
1  -- Declaração da função genérica
2  generic
3     type Tipo is private;
4     with function Menor_Que (A, B: Tipo) return Boolean;
5  function Min_Generico (A, B: Tipo) return Tipo;
6
7  -- Definição da função genérica
8  function Min_Generico (A, B: Tipo) return Tipo is
9  begin
10
11    if MenorQue (A, B) then
12       return A;
13    else
14       return B;
15    end if;
16 end Min_Generico;
17
18 -- Definição de uma função de comparação para inteiros
19 function Menor_Int (A, B: Integer) return Boolean is
20 begin
21    return A < B;
22 end Menor_Int;
23
24 -- Instanciação da função genérica
25 function Min_Int is new Min_Generico (Integer, MenorInt);
26
27 -- Chamada da função genérica
```

```
28 procedure Teste_Min is
29    Menor: Integer;
30 begin
31    Menor := Min_Int (3, 2);
32    Ada.Integer_Text_IO.Put (Menor);
33 end Teste_Min;
```

A palavra reservada `generic` é usada para dar suporte a tipos e procedimentos genéricos em Ada. Entre as linhas 2 e 5 esta palavra reservada é usada para declarar a função genérica `Min_Generico`. Dois "parâmetros genéricos" devem ser passados para `Min_Generico`: um tipo (`Tipo`) e uma função (`Menor_Que`). Entre as linhas 8 e 16 a função genérica é implementada. Os parâmetros genéricos `Tipo` e `Menor_Que` são usados na implementação como se fossem um tipo ou uma função qualquer. A partir da linha 19 é definida uma função não-genérica, `Menor_Int`, que é usada para comparar dois valores inteiros.

Para usar um procedimento genérico em Ada, é preciso instanciá-lo explicitamente. Isso é feito na linha 25. Ali, a função `Min_Generico` é instanciada com o nome `Min_Int`, e como parâmetros são passados o tipo `Integer` e função `Menor_Int`. A partir deste ponto a função `Min_Int` pode ser usada para verificar qual é o menor de dois inteiros. Um exemplo de chamada é mostrado na linha 31.

A função `Min_Genérico` atende aos dois requisitos citados na seção 2: para que esta função possa trabalhar com outros tipos ou com outras definições do que é "menor", basta instanciá-la outras vezes passando parâmetros genéricos diferentes.

Em C++ a programação genérica é suportada através da palavra chave `template`. Uma implementação da função `Min` em C++ é mostrada a seguir:

```
1  // Definição da função genérica
2  template <typename Tipo, typename Compara>
3  Tipo Min (Tipo a, Tipo b, Compara menorQue)
4  {
5     if (menorQue (a, b))
6        return a;
7     else
8        return b;
9  }
10
11 // Definição de uma função de comparação
12 bool MenorQue (int a, int b)
13 {
14    return a < b;
15 }
16
17 // Exemplo de uso
18 int main()
19 {
20    cout << "O menor entre 2 e 1 é " << Min (2, 1, MenorQue) << endl;
21 }
```

A função genérica `Min` é definida entre as linhas 2 e 9. A linha 2, que utiliza a palavra reservada `template`, informa ao compilador que a função declarada a seguir recebe dois parâmetros genéricos: `Tipo` e `Compara`. `Tipo` representa o tipo dos elementos que serão utilizados por `Min` e `Compara` representa a função de comparação.

Em seguida, a partir da linha 3, a função `Min` é definida, utilizando os parâmetros genéricos `Tipo` e `Compara`. A função `MenorQue` (a partir da linha 12) é um exemplo de função de comparação. Na linha 20 é mostrada uma chamada à função genérica. Em C++ uma função genérica é automaticamente instanciada na sua chamada; não é necessária uma instanciação explícita, como em Ada.

Convém observar que os dois parâmetros genéricos são declarados com a palavra reservada `typename`. Em C++, ao contrário de Ada, os tipos dos parâmetros genéricos não são declarados. A validade ou não dos parâmetros passados é sendo verificada em tempo de compilação, de acordo com o seu uso na função ou tipo genérico que está sendo criado. Por exemplo: na função `Min`, o parâmetro genérico `Compara`

não precisa ser necessariamente uma função; ele pode ser qualquer expressão da linguagem que possa ser chamada com uma sintaxe igual à de uma chamada de função[4]. Isso é interessante porque permite que um conceito (ou seja, um parâmetro genérico) possa ser implementado com o recurso da linguagem mais adequado para cada caso. A conseqüência negativa é a dificuldade para o compilador gerar boas mensagens de erro, que ajudem a identificar e corrigir problemas. As mensagens de erro geradas pelos compiladores C++ para código baseado em `templates` são, normalmente, confusas.

`generics` e `templates` também são capazes de solucionar o problema do somatório. Uma implementação em Ada é mostrada abaixo.

```
1  generic with function Termo (X: Integer) return Float;
2  function Somatorio_Generico (A, B: Integer) return Float;
3
4  function Somatorio_Generico (A, B: Integer) return Float is
5     Acum: Float := 0.0;
6  begin
7     for I in A..B loop
8        Acum := Acum + Termo(I);
9     end loop;
10
11    return Acum;
12 end Somatorio_Generico;
13
14 function Quadrado (X: Integer) return Float is
15 begin
16    return Float(X**2);
17 end Quadrado;
18
19 function Somatorio_De_Quadrados is new Somatorio_Generico (Quadrado);
20
21 procedure Teste_Soma is
22 begin
23    Ada.Float_Text_IO.Put (Somatorio_De_Quadrados (1, 10));
24 end;
```

A linha 1 avisa ao compilador que, na seqüência, aparecerá uma declaração genérica e que a função genérica que será declarada irá receber como parâmetro uma função que recebe um parâmetro do tipo `Integer`, retorna um `Float` e será conhecida por `Termo`. Na segunda linha a "entidade genérica" é declarada: trata-se de uma função, `Somatorio_Generico`, que recebe como parâmetros dois números inteiros e retorna um número real (além do "parâmetro generico" `Termo`, declarado na linha anterior). Em seguida, da linha 4 à linha 9, a função genérica `Somatorio_Generico` é definida. Note que a função `Termo` (o parâmetro genérico) é usada na linha 8 como uma função normal.

Da linha 14 até a linha 17 é definida uma função não-genérica, `Quadrado`, que será usada neste exemplo como parâmetro para `SomatorioGenerico`. Note que ela é compatível com o parâmetro genérico `Termo` da função `SomatorioGenerico`, ou seja, recebe um parâmetro do tipo `Integer` e retorna um `Float`.

Na linha 19 a função genérica é instanciada. A partir deste ponto para a existir uma nova função chamada `Somatorio_De_Quadrados`, que é simplesmente a função genérica `Somatorio_Generico` recebendo como o parâmetro genérico `Termo` a função `Quadrado`.

Finalmente, o procedimento `Teste_Soma` definido a partir da linha 21 utiliza a instância de procedimento genérico `Somatorio_De_Quadrados` para calcular o somatório dos quadrados dos números inteiros de 1 a 10.

Coleções

Tipos parametrizáveis são o melhor recurso disponíveis nas linguagens atuais para a criação de coleções genéricas. Com eles é possível criar coleções (homogêneas) fortemente tipadas que armazenam elementos de qualquer tipo. A seguir é mostrada a implementação de uma pilha genérica em C++.

[4]Em C++, através da sobrecarga de operadores, é possível criar "objetos função", que são objetos que admitem a sintaxe de chamada de função

```
1  template <typename T, int N>
2  class Pilha
3  {
4      public:
5          Pilha()
6              : topo_ (0)
7          { }
8
9          void push (const T& item)
10         {
11             if (topo_ == N)
12                 throw "Overflow";
13             else
14                 dados_[topo_++] = item;
15         }
16
17         T pop()
18         {
19             if (topo_ == 0)
20                 throw "Underflow";
21             else
22                 return dados_[--topo_];
23         }
24
25     private:
26         T dados_[N];
27         int topo_;
28 };
```

A única diferença fundamental entre esta classe de pilha genérica e uma classe não-genérica é a primeira linha. Ela declara que a classe a seguir possui dois parâmetros genéricos: um tipo (`T`) e um valor inteiro `N`. Estes parâmetros genéricos são utilizados na definição da classe genérica para representar, respectivamente, o tipo armazenado na pilha e a sua capacidade máxima. A utilização de `T` pode ser vista como parâmetro do método `push()` (linha 9), como valor de retorno de `pop()` (linha 17). Na linha 26 é possível ver que, internamente, os dados ficam armazenados em um "*array* de `T`s" de tamanho `N`. Um exemplo de uso desta pilha pode ser visto a seguir:

```
1  int main()
2  {
3      Pilha <float, 10> pilha;
4
5      pilha.push (3.1415);
6      pilha.push (1.2345);
7
8      float f1 = pilha.pop();
9      float f2 = pilha.pop();
10 }
```

Os parâmetros genéricos são passados na instanciação da pilha, como se vê na linha 3. Neste caso, está sendo criada uma pilha capaz de armazenar até 10 elementos do tipo `float`. Depois da instanciação não há diferença alguma entre um objeto de uma classe genérica e um objeto de uma classe não-genérica. Isso pode ser visto nas chamada dos métodos `push()` e `pop()` da linha 5 à linha 9. Também é interessante observar que nas chamadas ao método `pop()` não é necessário um *cast*, pois o tipo retornado já é um `float`.

Mas isto ainda não soluciona o problema das operações sobre as coleções. Implementações genéricas de coleções costumam utilizar o conceito de *iteradores* para este fim. Um iterador, como o nome sugere, é algo que permite iterar pelos elementos de uma coleção. Ele pode ser visto como um ponteiro para um elemento da coleção que admite uma série de operações, como "avançar para o próximo elemento", "retroceder n elementos" ou "apontar para o n-ésimo elemento da coleção". Se cada coleção disponibilizar iteradores que possam ser usados com ela, as operações sobre as coleções podem ser implementadas em termos dos iteradores.

4 CONCLUSÕES

A utilização de recursos de baixo nível para criar soluções genéricas é questionável. De um modo geral, as soluções que podem ser criadas desta forma não atendem a todos os quesitos desejáveis. Pior do que isso, eles dão origem a uma série de riscos. Para que a programação genérica possa ser sistematicamente utilizada com alguma segurança, ela precisa ser implementada através de recursos de alto nível.

A orientação a objetos ajuda a resolver alguns problemas, mas ainda não é a solução ideal. Boas soluções exigem hierarquias de classes complexas. Muitas vezes a utilização de herança múltipla (ou, pelo menos, de "herança de interfaces") é necessária. E, ainda assim, há casos que não podem ser resolvidos.

Tipos e procedimentos parametrizáveis foram criados especialmente para atender às necessidades da programação genérica e são o melhor recurso disponível atualmente para criar soluções adaptáveis. Estão presentes em Ada e C++, mas existem grupos de pesquisa trabalhando na inclusão deste recurso na linguagem Java [4] [8]. Tipos e procedimentos parametrizáveis permitem criar soluções notáveis para diversos problemas, sem perder em termos de segurança. Soluções baseadas em tipos e procedimentos parametrizáveis também podem ser muito eficientes. Como exemplo, vale citar a biblioteca Blitz++ [2]. Ela é uma biblioteca C++ voltada à computação científica, totalmente baseada em `templates`. Medições de desempenho mostraram que programas baseados em Blitz++ são tão ou mais eficientes que programas equivalentes escritos em Fortran.

Os `generics` da linguagem Ada obrigam que os parâmetros genéricos sejam declarados em detalhes. Isto permite que o compilador saiba exatamente o que deve ser passado como parâmetro. Os `templates` de C++ não são tão exigentes na declaração dos parâmetros genéricos. Isso permite que conceitos sejam implementados com o recurso da linguagem mais adequado para cada caso, mas dificulta a exibição de boas mensagens de erro pelo compilador ou na criação de ferramentas CASE que trabalhem sobre código C++ genérico.

Possivelmente o recurso ideal para a programação genérica seja algo entre os `generics` de Ada e os `templates` de C++: ele não deveria obrigar os programadores a implementar um conceito com um determinado recurso da linguagem, mas permitiria que um conceito fosse explicitamente declarado. Assim não haveria dúvidas quanto ao papel de cada parâmetro genérico nem um acoplamento forte entre conceitos e recursos da linguagem.

REFERÊNCIAS

[1] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Reading: Addison-Wesley. 2001.

[2] *Blitz++*. `http://www.oonumerics.org/blitz/`.

[3] Cohen, N.H. *Ada as a Second Language*. Second edition. New York: McGraw-Hill. 1996.

[4] *GJ: A Generic Java Language Extension*. `http://www.cis.unisa.edu.au/ pizza/gj/`.

[5] Stroustrup, B. *The C++ Programming Language*. Special Edition. Reading: Addison-Wesley. 2000.

[6] *Java Technology Homepage*. `http://www.java.sun.com`.

[7] Kernigham, B. and Ritchie, D. *The C Programming Language*. Second Edition. Englewood Cliffs: Prentice-Hall. 1988.

[8] *PolyJ: Java with Parameterized Types*. `http://www.pmg.lcs.mit.edu/polyj/`.