Resolução dos Exercícios do Capítulo VII

1. Segundo a classificação de Cardelli e Wegner, existem quatro tipos de polimorfismo. Quais desses tipos de polimorfismo existem em C, C++ e JAVA? Mostre exemplos desses tipos de polimorfismo com trechos de código em C, C++ ou JAVA. Identifique o tipo de polimorfismo que ocorre em cada exemplo e explique porque cada um dos trechos de código é polimórfico. Indique ainda se o polimorfismo é ad-hoc ou universal e justifique.

Os quatro tipos de polimorfismo, segundo a classificação de Cardelli e Wegner, são: polimorfismo de coerção, polimorfismo de sobrecarga, polimorfismo paramétrico e polimorfismo por inclusão.
Os tipos de polimorfismo existentes em C são o de coerção, o de sobrecarga (C embute sobrecarga em seus operadores, mas os programadores não podem implementar novas sobrecargas, além disso, não existe qualquer sobrecarga de programas) e o paramétrico.
Os existentes em C++ são o de coerção, o de sobrecarga (adota postura ampla e ortogonal, realizando e permitindo que programadores realizem sobrecarga de subprogramas e operadores), o paramétrico e o por inclusão.
Os existentes em JAVA são o de coerção (só admite a realização de coerções para tipos mais amplos), o de sobrecarga (embute sobrecarga em operadores e em subprogramas de suas bibliotecas, mas somente subprogramas podem ser sobrecarregados pelo programador) e o por inclusão.

Exemplos:

1.1) Expressão com coerção em C:

```
#include <stdio.h>

main () {
   int a = 1;
   float b = 2.5, c = 3.5;
   c = c + b;                  //c = somafloat (c, b)
   printf("%.1f", c);          //imprime 6.0
   c = c + a;                  //c = somafloat (c, intToFloat (a))
   printf("\n%.1f", c);        //imprime 7.0
}
```

O trecho acima é polimórfico, porque a instrução *c = c + a* (na penúltima linha deste exemplo) sugere que o operador + corresponde a uma função capaz de realizar tanto a operação de somar dois valores do tipo *floa*t, quanto a operação de somar um *float* e um *int*, quando ela só realiza realmente a primeira dessas operações. Antes da operação soma ser chamada, ocorre um chamada implícita a uma função capaz de converter valores do tipo *int* em valores do tipo *float*.

1.2) Sobrecarga do operador + em C:

```
#include <stdio.h>
```

```
main () {
   int a = 1, b = 2;
   float c = 1.0, d = 2.0;
   a = a + b;                  //a = somaint (a, b)
   printf("%d", a);            //imprime 3
   c = c + d;                  //c = somafloat (c, d)
   printf("\n%.1f", c);        //imprime 3.0
}
```

O trecho acima é polimórfico porque sugere que o operador + representa uma função capaz de realizar tanto a operação de somar dois valores do tipo *int*, quanto a operação de somar dois valores do tipo *float*, sendo que, na realidade, cada ocorrência de + invoca operações específicas para cada uma dessas operações.

1.3) Função genérica em C++:

```
template <class T>
T identidade (T x) {
        return x;
}
class Horario {
        int h, m, s;
}
main () {
        int a;
        float b;
        Horario h1, h2;
        a = identidade (10);
        b = identidade (10.5);
        h2 = identidade (h1);
}
```

O trecho acima é polimórfico porque parametriza o subprograma *identidade* com relação ao tipo do elemento sobre o qual ele opera. A função *identidade* pode ser aplicada a valores de qualquer tipo. Esse tipo é determinado pela combinação do tipo do valor do argumento com o tipo declarado no cabeçalho da função.

1.4) Herança simples em JAVA:

```
public class Liquidificador {
        protected int velocidade;
        protected int velocidadeMaxima;

        public Liquidificador () {
                velocidade = 0;
                velocidadeMaxima = 2;
        }
        public Liquidificador (int v) {
                this ();
                ajustarVelocidadeMaxima (v);
```

```
        }
        protected void ajustarVelocidadeMaxima (int v) {
                if (v > 0)
                        velocidadeMaxima = v;
        }
        protected void ajustarVelocidade (int v){
                if (v >= 0 && v <= velocidadeMaxima)
                        velocidade = v;
        }
        public int obterVelocidadeMaxima () {
                return velocidadeMaxima;
        }
        public int obterVelocidade () {
                return velocidade;
        }
}
public class LiquidificadorAnalogico extends Liquidificador {
        public LiquidificadorAnalogico () {
                velocidade = 0;
        }
        public void aumentarVelocidade () {
                ajustarVelocidade (velocidade + 1);
        }
        public void diminuirVelocidade () {
                diminuirVelocidade (velocidade – 1);
        }
}
public class LiquidificadorDigital extends Liquidificador {
        public LiquidificadorDigital () {
                velocidade = 0;
        }
        public void trocarvelocidade (int v) {
                ajustarVelocidade (v);
        }
}
public class LiquidificadorInfo {
        public void velocidadeAtual (Liquidificador l) {
                System.out.println ("Velocidade Atual: "+ l.obterVelocidade ());
        }
}
```

Esse trecho é polimórfico porque a utilização da classe *Liquidificador* permite o tratamento generalizado de todas as suas subclasses. O tratamento generalizado de classes permite escrever programas que podem ser mais facilmente extensíveis, isto é, que podem acompanhar a evolução de uma hierarquia de classe sem necessariamente serem modificados. Por exemplo, na classe *LiquidificadorInfo*, existe um método capaz de imprimir a velocidade atual de objetos liquidificador os quais podem ser tanto do tipo digital como analógico.

Nesta situação, o polimorfismo permite que um simples trecho de código seja utilizado para tratar objetos diferentes relacionados através de seu ancestral comum,

simplificando a implementação, melhorando a legibilidade do programa e também aumentando sua flexibilidade.
Os polimorfismos dos exemplos 1.1 e 1.2 são de tipo **adhoc.** O operador + é associado a diferentes trechos de código que atuam sobre diferentes tipos. Para quem lê o código, pode parecer que esse operador denota um único trecho de código polimórfico, atuando sobre elementos de tipos diferentes. Contudo, isso é apenas aparente, uma vez que para utilizar operandos de tipos diferentes numa mesma operação, o operando que não é do tipo esperado deve ser convertido para o mesmo, e para realizar a operação de adição sobre diferentes tipos, diferentes operações, específicas para cada tipo, são chamadas.
Os polimorfismos dos exemplos 1.3 e 1.4 são de tipo **universal**, pois as estruturas de dados incorporam elementos de tipos diversos e um mesmo código pode ser executado e atuar sobre elementos de diferentes tipos.

2. O que são classes abstratas? Quando devem ser usadas e quais as suas vantagens? Quais diferenças existem na definição de classes abstratas em C++ e JAVA?

As classes abstratas são classes que não possuem instâncias, mas que possuem membros (as instâncias de suas subclasses não abstratas) e que, portanto, devem ser necessariamente estendidas, ou seja, devem ser herdadas por outras, mais específicas, contendo os detalhes nela não incluídos. Elas normalmente possuem um ou mais métodos abstratos, isto é, métodos declarados, mas não implementados (a implementação dos mesmos é deixada para as suas subclasses) e também podem ter métodos definidos e atributos próprios. De fato, elas podem possuir até construtores, embora eles nunca possam ser chamados para criar instâncias dessa classe (só podem ser chamados no momento da construção das instâncias das subclasses da classe abstrata).
As classes abstratas são especialmente úteis quando uma classe, ancestral comum para um conjunto de classes, se torna tão geral a ponto de não ser possível ou razoável ter instâncias dela. As principais vantagens da sua utilização são: a melhoria da organização hierárquica de classes, pelo encapsulamento de atributos e métodos na raiz da estrutura; a promoção de uma maior disciplina na programação, visto que força o comportamento necessário nas suas subclasses (para cada método abstrato em uma classe abstrata, todas as suas subclasses devem implementar esse método ou também devem ser abstratas); e o incentivo ao uso de amarração tardia, permitindo um comportamento mais abstrato e genérico para os objetos.
Diferenças entre C++ e JAVA: Em JAVA, é possível, mas não é comum, criar classes abstratas nas quais nenhum método é abstrato. Já em C++, para uma classe ser abstrata, é obrigatório ter pelo menos um método abstrato. Em JAVA, para tornar uma classe abstrata, basta incluir a palavra *abstract* como prefixo de definição. Em C++, os métodos abstratos devem ter a terminação *=0* e devem ser declarados como virtuais, uma vez que seu comportamento terá que ser definido nas subclasses da classe abstrata.

3. Enquanto em C++ somente os métodos precedidos pela palavra *virtual* utilizam o mecanismo de amarração tardia de tipos, em JAVA todos os métodos empregam este mecanismo. Justifique esta decisão dos criadores dessas linguagens.

O mecanismo de amarração tardia de tipos oferece maior flexibilidade para a escrita de código reutilizável, já que possibilita a criação de código usuário com polimorfismo universal, isto é, código usuário capaz de operar uniformemente sobre

objetos de tipos diferentes. Por darem preferência à versatilidade, os criadores de JAVA optaram por sempre adotar esse mecanismo. Contudo, a amarração tardia de tipos reduz a eficiência computacional quando comparada com a amarração estática, pois na amarração tardia é necessário manter sempre a cadeia de ponteiros e segui-la em todas as chamadas de métodos, enquanto na estática nada disso é necessário já que o subprograma a ser chamado é definido em tempo de compilação. Assim, os criadores de C++ decidiram adotar uma postura diferente, em que o programador pode decidir se deseja o uso da amarração tardia, identificada pela declaração do método precedida da palavra *virtual*, ou da amarração estática, identificada pela omissão da palavra *virtual*, possibilitando ao programador escolher entre versatilidade e eficiência, respectivamente.

4. Linguagens de programação orientadas a objetos podem adotar herança simples ou múltipla. C++, por exemplo, adota herança múltipla. Quais os dois problemas que podem ocorrer quando se adota herança múltipla? Explique-os usando exemplos em C++. Mostre de que forma C++ permite contornar esses problemas. Apresente um exemplo de situação na qual os mecanismos de C++ são inadequados para tratá-los.

Os dois problemas que podem ocorrer quando se adota herança múltipla são o conflito entre nomes de classes bases diferentes e a herança repetida.
Os conflitos entre nomes ocorrem quando duas ou mais classes bases possuem nomes idênticos (homônimos) de atributos ou métodos. Por exemplo:

```
class Desempenho {
        int n;
        float t;
public:
        int numero () { return n; }
        float tempo() { return t; }
};
class Corrida: public Desempenho {
        float quilometragem;
public:
        void imprime ();
};
class Natacao: public Desempenho {
        float metragem;
public:
        void imprime ()
};
class Duatlo : public Corrida, public Natação {
        char prova;
}
main () {
        Duatlo atleta;
        //atleta.imprime;
}
```

Caso a linha comentada fosse compilada, seria detectado um erro de ambigüidade, pois não foi especificado de qual classe herdada éo método referenciado pelo objeto da classe herdeira. Em C++ esse problema pode ser resolvido pela sobrescrição da operação de impressão na classe *Duatlo* e pelo uso do operador de resolução de escopo *::*, como mostrado abaixo, para identificar qual operação de impressão está sendo chamada.

```
class Duatlo: public Corrida, public Natação {
        char prova;
public:
        void imprime ();
};
void Duatlo :: imprime () {
        if (prova == 'C')
                Corrida :: imprime ();
        else
                Natacao :: imprime ();
}
main () {
        Duatlo atleta;
        atleta.imprime;
}
```

O problema da herança repetida ocorre quando uma classe faz herança múltipla de classes descendentes de uma mesma classe. Os atributos dessa classe comum são repetidos na classe na qual é feita herança múltipla. C++ fornece um mecanismo especial, utilizando a palavra *virtual* para resolver esse problema. Se a especificação da classe herdada é precedida por *virtual*, somente um objeto daquela classe comporá o objeto das classes herdeiras, independentemente de a classe ser herdada múltiplas vezes ou não.
No exemplo anterior, a palavra *virtual* não foi utilizada na especificação das classes *Corrida* e *Natação*, o que faz com que os atributos numero (*n*) e tempo (*t*) de *Desempenho* sejam repetidos na classe *Duatlo*. Isso não ocorre na implementação mostrada a seguir:

```
class Desempenho {
         int n;
         float t;
public:
        int numero () { return n; }
        float tempo() { return t; }
};
class Corrida: virtual public Desempenho {
        float quilometragem;
public:
        void imprime ();
};
class Natacao: virtual public Desempenho {
        float metragem;
public:
```

```
        void imprime ()
};
```

Contudo, esses mecanismos adotados por C++ para solucionar esses problemas apresentam dois inconvenientes. Primeiro, a especificação da palavra *virtual* não é feita na classe na qual ocorre a herança repetida. Inicialmente, ao se criar as classes *Corrida* e *Natacao*, não se sabe se elas serão herdadas futuramente por uma mesma subclasse e muito menos se esta subclasse precisará ou não compartilhar os atributos comuns.
O segundo problema é que não é possível especificar que um atributo deva ser repetido e outro não. No exemplo acima, o número do atleta é o mesmo, mas os tempos para as provas de corrida e natação são tempos distintos.

5. C++ oferece três mecanismos distintos para permitir a realização de estreitamento. Mostre exemplos do uso desses três mecanismos e os compare em termos de confiabilidade e eficiência.

As três maneiras distintas oferecidas por C++ para permitir a realização de estreitamento são: o mecanismo usual de conversão explícita (*cast*), o mecanismo de conversão explícita estática (*static_cast*) e o mecanismo de conversão explícita dinâmica (*dynamic_cast*). Por exemplo:

```
class UmaClasse {
public:
        virtual void temVirtual () {};
}
class UmaSubclasse: public UmaClasse {}
class OutraSubclasse: public UmaClasse {}
class OutraClasse {}
main () {
        //primeira parte do exemplo
        UmaClasse *pc = new UmaSubclasse;
        OutraSubclasse *pos = dynamic_cast <OutraSubclasse *> (pc);
        UmaSubclasse *ps = dynamic_cast <UmaSubclasse *> (pc);
        //segunda parte do exemplo
        UmaSubclasse us;
        pc = &us;
        pc = static_cast <UmaClasse *> (&us);
        OutraClasse *poc = (OutraClasse *) pc;
}
```

Na primeira parte do exemplo acima é mostrado o uso do mecanismo de *dynamic_cast*. Essa é uma forma de fazer estreitamento de modo seguro. Ao usar *dynamic_cast*, o que se está tentando fazer é um estreitamento para um tipo particular. O valor de retorno dessa operação será um ponteiro para o tipo desejado, no caso de o estreitamento ser apropriado. De outra forma, o valor retornado será zero (*null*) para indicar que o tipo não era o esperado.
Somente podemos usar o *dynamic_cast* em classes com funções virtuais. Isso ocorre porque o *dynamic_cast* usa a informação armazenada em uma tabela de métodos virtuais para determinar o tipo atual. No exemplo, o ponteiro *pos* receberá *zero*, pois o

estreitamento para *OutraSubclasse* * é incorreto. É responsabilidade do programador, verificar se o resultado do estreitamento por *dynamic_cast* é diferente de zero.
A operação de *dynamic_cast* sobrecarrega um pouco a execução do programa, portanto, se um programa usa muito *dynamic_cast*, isso poderá diminuir a eficiência de execução.
O mecanismo de *static_cast* deve ser utilizado quando é possível saber durante a própria redação do programa com qual tipo estamos lidando no local do estreitamento, pois assim a verificação é feita em tempo de compilação. Usar *static_cast* para realizar estreitamento é melhor do que o mecanismo de conversão explícita formal (*cast*), pois o primeiro não permite fazer conversões fora da hierarquia de classes, o que é permitido pelo segundo. Como a eficiência é a mesma para códigos gerados usando ambos os mecanismos, *static_cast* deve ser preferido, pois é mais seguro.
A segunda parte do exemplo mostra o uso desse mecanismo. Nessa parte, um objeto (*us*) de *UmaSubclasse* é criado e é feita uma ampliação a um ponteiro para *UmaClasse*. Essa mesma operação é repetida usando *static_cast*. Como se trata de uma ampliação, não existe obrigatoriedade de usar *static_cast*, mas seu uso pode ser conveniente para tornar mais explícita a ampliação e para evitar a realização equivocada de conversões fora da hierarquia de classes.
Após as operações de ampliação, o exemplo mostra a diferença entre se fazer estreitamento com *static_cast* e o mecanismo tradicional. Com o mecanismo tradicional, é possível fazer a conversão fora da hierarquia de classes entre os ponteiros para *UmaClasse* e para *OutraClasse*. Isso não é permitido no caso do *static_cast*. Caso a linha de código na qual essa operação é feita não estivesse comentada, ocorreria um erro de compilação.
Resumindo, é boa prática usar preferencialmente os mecanismos do *dynamic_cast*, pois embora seja mais rápido fazer estreitamento estaticamente, a conversão dinâmica é mais segura, pois os mecanismos de conversão estática podem produzir conversões inapropriadas.

6. Amarração tardia de tipos é o processo de identificação em tempo de execução do tipo real de um objeto. Esse processo pode ser utilizado para a identificação dinâmica do método a ser executado (quando ele é sobrescrito) e para a verificação das operações de estreitamento. Explique como pode ser implementado o mecanismo de amarração tardia de tipos em linguagens como C++ e JAVA e como ele é usado para realizar as operações mencionadas na frase anterior.

Em JAVA, na utilização desse mecanismo, quando um método é invocado por uma variável (essa variável aponta para um objeto), uma cadeia de ponteiros é seguida. Essa cadeia começa pela variável que chamou o método e que está na base da pilha de registros de ativação e que aponta para um objeto. Esse objeto apontado pela variável possui, além dos atributos de sua classe, uma referência à tabela de métodos de sua classe. Essa tabela, por sua vez, também possui uma referência à tabela de métodos de sua superclasse. Assim, essa cadeia é seguida até chegar à primeira tabela de métodos da classe na qual esse método foi definido e o método a ser utilizado é identificado.
Com relação à verificação das operações de estreitamento, a cadeia de ponteiros citada acima é seguida e, caso a classe para a qual se deve fazer a conversão seja encontrada em algum momento, a operação é validada.

7. Tanto C++ quanto JAVA oferecem bibliotecas de classes que disponibilizam estruturas de dados genéricas, tais como listas, árvores e tabelas `hash`. Ambas utilizam polimorfismo para a implementação dessas estruturas, embora sejam formas diferentes de polimorfismo. Explique como essas linguagens usam o polimorfismo para a implementação dessas estruturas. Discuta as vantagens e desvantagens de cada abordagem. Justifique também porque os criadores dessas linguagens adotaram essa postura diferenciada.

Estruturas de dados genéricas são capazes de armazenar e operar sobre elementos de tipos diferentes. Estruturas genéricas podem ser preenchidas com elementos de um mesmo tipo (nesse caso são chamadas de estruturas homogêneas) ou de tipos diferentes (nesse caso são chamadas de estruturas heterogêneas).
C++ utiliza o polimorfismo paramétrico proporcionado pelo mecanismo de *template* para a criação de estruturas de dados genéricas homogêneas, e o mecanismo de polimorfismo por inclusão para a criação de estruturas de dados heterogêneas. É possível ainda combinar o mecanismo de *template* com o polimorfismo de inclusão para criar estruturas de dados genéricas heterogêneas.
JAVA utiliza o polimorfismo por inclusão para permitir a criação de estruturas de dados genéricas heterogêneas. Para isso, JAVA considera todas as classes existentes como subclasses (diretas ou indiretas) da classe *Object*.
Assim, estruturas de dados cujos elementos são do tipo *Object* podem abrigar elementos de qualquer classe em JAVA. Para ter uma estrutura de dados homogênea, o programador deve garantir que os elementos inseridos sejam sempre de um mesmo tipo. Isso é pior do que a solução de C++, na qual o compilador garante a homogeneidade da estrutura. Além disso, a solução de JAVA obriga a realização de estreitamento sempre que um elemento deve ser acessado a partir da lista. Por outro lado, a solução de JAVA simplifica a linguagem, pois não é necessário incluir o polimorfismo paramétrico, imprescindível em C++.

8. Implemente o tipo abstrato de dados lista genérica em C, C++ e JAVA. É suficiente apresentar a definição do tipo e o cabeçalho dos métodos de construção, ordenação, destruição, verificação de lista vazia, inclusão e exclusão de elemento (não é preciso codificar os métodos da lista). Atente para o fato de que a operação de ordenação na lista deve ser única, mas deve permitir que a lista seja ordenada por critérios distintos. Justifique a sua implementação, enfocando o modo como se obtém a generalidade da lista e o funcionamento da operação de ordenação.

```
//C
typedef struct no {
        void *info;
        struct no *prox;
}No;
typedef struct Lista {
        No *prim, *ult;
        int tam;
}Lista;
Lista InicLista ();
Lista OrdenaLista (Lista lst, int (*compara) (void*, void*));
void DestroiLista (Lista lst);
int VaziaLista (Lista lst);
```

```
Lista InsereLista (Lista lst, int pos, void *elem);
Lista ElimLista (Lista lst, int pos);

//C++
template <class T>
class Lista {
        struct no {
                T info;
                struct no *prox;
        }No;
        No *prim, *ult;
        int tam;
public:
        Lista ();
        Lista OrdenaLista (int (*compara) (T, T));
        int VaziaLista ();
        Lista InsereLista (int pos, T elem);
        Lista ElimLista (int pos);
        void DesalocaInfos();
        ~Lista ();
}

//JAVA
class Lista {
        private Object [] lst;
        private int tam = 0;
        private int marcador;
        Lista ();
        public int VaziaLista ();
        public void OrdenaLista (Comparator c, int ordem);
        public void InsereLista (int pos, Object o);
        public void ElimLista (int pos);
        public void finaliza ();
}
```

A generalidade da lista é obtida pela utilização de *void** em C, *template* em C++ e *Object* em JAVA, que permitem generalizar os elementos que compõem a lista. No caso de C, o tipo *void** permite que o elemento da lista seja de qualquer tipo ponteiro, o que confere generalidade a lista. No caso de C++, o mecanismo *template* possibilita que o tipo do elemento da lista seja definido no momento de criação das variáveis lista. No caso de JAVA, como toda classe em JAVA é subclasse de *Object*, é possível colocar qualquer tipo de objeto como elemento da lista.
A operação de ordenação pode ser realizada em diferentes contextos, isto é, diferentes características dos objetos podem ser comparadas ou pode ser escolhido se o desejado é a ordenação crescente ou decrescente, por exemplo. Isso é feito em C e C++ pela utilização do ponteiro para função *compara* passado como parâmetro para a função *OrdenaLista* e pelo parâmetro *c* da interface *Comparator* passado para a função *OrdenaLista* em JAVA. No caso de C e C++, o que se precisa fazer é criar uma função de comparação que compare dois elementos do tipo da lista e passar seu endereço como argumento para o parâmetro formal *compara* da função *OrdenaLista*.

No caso de JAVA é preciso criar uma classe que implemente a interface *Comparator* e que defina como deve ser feita a comparação em seu único método. Depois, basta passar um objeto dessa classe para o método *OrdenaLista*.

9. Em alguns problemas pode ser conveniente permitir a um mesmo objeto participar de duas ou mais listas cujos elementos são de tipos diferentes. Por exemplo, em um problema no qual é necessário armazenar em listas os diversos tipos de dependências de um apartamento haveria uma lista para quartos e outra para salas. Contudo, é possível haver uma mesma dependência usada como quarto e como sala. Nesse caso, essa dependência participaria tanto da lista de quartos quanto da lista de salas.

   a) Como você resolveria esse problema em uma linguagem que não possui mecanismos para a realização de subtipagem múltipla, ou seja, que não permita a um mesmo objeto fazer parte de listas de dados cujos elementos sejam de tipos distintos. Existe alguma desvantagem na sua solução?

   b) Sabendo que C também não permite a realização de subtipagem múltipla, responda se é possível resolver esse mesmo problema de outra maneira? Se a resposta for positiva, explique como seria essa solução e faça uma análise de suas vantagens e desvantagens.

   c) Mostre através da implementação desse exemplo como C++ e JAVA permitem a realização de subtipagem múltipla.

   Compare ainda os mecanismos oferecidos por C++ e JAVA para realização de subtipagem múltipla apresentando vantagens e desvantagens de cada um.

a) Uma forma seria através da existência de um objeto do tipo sala e outro do tipo quarto referindo-se ao mesmo cômodo. O problema com essa abordagem é que qualquer alteração feita em um atributo comum dos objetos deve implicar na atualização de um ou outro objeto, sob pena de se gerar uma inconsistência de dados caso isso não seja feito apropriadamente.

b) Em C isso poderia ser feito através da criação de uma nova estrutura representando objetos do tipo sala e quarto juntamente com a especificação do elemento das duas listas como ponteiro para void. Um problema dessa abordagem é que se torna necessário manter junto com o elemento uma indicação de seu tipo para que toda vez que for utilizado se possa verificar o tipo real do elemento. Outro problema é que as listas passam a ser genéricas, podendo assim receber elementos de outros tipos e não apenas salas e quartos.

c)

C++:

```
class Quarto {
public:
        Quarto() {}
        int numeroCamas () {}
};
class Sala {
public:
        Sala() {}
        int numeroMesas () {}
};
```

```
class QuartoSala: public Quarto, public Sala { };
```

JAVA:

```
interface Quarto {
        int numeroCamas ();
}
interface Sala {
        int numeroMesas ();
}
class QuartoSala implements Quarto, Sala {
        public int numeroCamas() {}
        public int numeroMesas() {}
}
```

C++ permite herança múltipla de classes, o que torna muito natural o desenvolvimento de heterarquia de classes e de programas que as utilizam. Por outro lado, o mecanismo de herança de C++ admite a ocorrência dos problemas de conflito de nomes e herança repetida.

JAVA só permite herança múltipla de interfaces (isto é, classes abstratas puras). Enquanto isso garante a inexistência de problemas de conflito de nomes e herança repetida, a reutilização de código por herança não é possível (uma vez que interfaces não implementam métodos nem possuem atributos) ou se torna mais complexa (há uma forma de reutilizar código por composição).

10. Considere as seguintes definições de funções e classes em C++:

```
template <class T >
T xpto (T x, T y) {
  return y;
}
template <class T, class U>
U  ypto (T x, U y) {
  return y;
}
template <class T, class U>
T zpto (T x, U y) {
  return ((T) y);
}
class tdata {
  int d, m, a;
};
class thorario {
  int h, m, s;
};
class tdimensao {
  int h, l, w;
};
```

Indique quais das linhas de código de *main* são legais e explique as que não são.

```
main () {
  tdata a;
```

```
  thorario b;
  tdimensao c;
  a = xpto (a, a);
  b = xpto (a, a);
  c = xpto (a, b);
  a = ypto (a, a);
  b = ypto (a, a);
  b = ypto (a, b);
  c = ypto (a, b);
  a = zpto (a, a);
  b = zpto (a, a);
  a = zpto (a, b);
  c = zpto (a, b);
}
```

Explique ainda como o compilador C++ implementa o mecanismo de polimorfismo paramétrico. Discuta essa solução em termos de reusabilidade de código.

Linhas legais:

a = xpto (a, a);
a = ypto (a, a);
b = ypto (a, b);
a = zpto (a, a);

Linhas que não são legais:

b = xpto (a, a); → Uma vez que não foi feita uma sobrecarga do operador de atribuição da classe *thorario* que receba um parâmetro do tipo *tdata*, a chamada de *xpto* geraria erro de compilação porque ela retorna um valor do tipo *tdata*, mas a atribuição demanda um *thorario*.

c = xpto (a, b); → Uma vez que não foi feita uma sobrecarga do operador de atribuição da classe *tdimensao* que receba um parâmetro do tipo *thorario*, a chamada de *xpto* geraria erro de compilação porque ela retorna um valor do tipo *thorario*, mas a atribuição demanda um *tdimensao*.

b = ypto (a, a); → Uma vez que não foi feita uma sobrecarga do operador de atribuição da classe *thorario* que receba um parâmetro do tipo *tdata*, a chamada da função *ypto* geraria erro de compilação porque ela retorna um valor do tipo *tdata*, mas a atribuição demanda um *thorario*.

c = ypto (a, b); → Uma vez que não foi feita uma sobrecarga do operador de atribuição da classe *tdimensao* que receba um parâmetro do tipo *thorario*, a chamada da função *ypto* geraria erro de compilação porque ela retorna um valor do tipo t*horario*, mas a atribuição demanda um *tdimensao*.

b = zpto (a, a); → Uma vez que não foi feita uma sobrecarga do operador de atribuição da classe *thorario* que receba um parâmetro do tipo *tdata*, a chamada da função *zpto* geraria erro de

compilação porque ela retorna um valor do tipo *tdata*, mas a atribuição espera um *thorario*.

a = zpto (a, b); → A chamada da função *zpto* geraria erro de compilação porque não existe um construtor definido na classe *tdata* que receba como parâmetro um objeto *thorario*.

c = zpto (a, b) → A chamada da função *zpto* geraria erro de compilação porque não existe um construtor definido na classe *tdata* que receba como parâmetro um objeto *thorario* e porque ela retorna um valor do tipo *tdata*, mas a atribuição espera um *tdimensao*.

C++ usa o mecanismo de *template* para incorporar o polimorfismo paramétrico. A forma de implementação do mecanismo *template* é curiosa. Ao contrário do que seria mais desejado (a reutilização de código-fonte e objeto), esse mecanismo só possibilita a reutilização de código-fonte. Isso significa que não é possível compilar o código usuário das funções ou classes definidas com polimorfismo paramétrico separadamente do código de implementação dessas funções ou classes. De fato, para compilar funções ou classes paramétricas, o compilador C++ necessita saber quais tipos serão associados a elas. A partir de uma varredura do código usuário, o compilador identifica os tipos associados a essas funções e classes e replica todo o código de implementação para cada tipo utilizado, criando assim um código objeto específico para cada tipo diferente utilizado.

11. Cada um dos programas seguintes, escritos em C++, utiliza um tipo de polimorfismo visto nesse capítulo. Defina o que é polimorfismo. Descreva as características desses tipos de polimorfismo, indicando o tipo empregado por cada programa. Execute os programas passo a passo, mostrando o resultado apresentado, indicando onde ocorre polimorfismo e explicando sua execução.

```
// programa 1
#include <iostream>

class base {
   public:
      virtual void mostra1() {
         cout << "base 1\n";
      }
      void mostra2 () {
         cout << "base 2 \n";
      }
};
class derivada1: public base {
   public:
      void mostra1() {
         cout << "derivada 1\n";
      }
};
class derivada2: public base {
   public:
      void mostra2 () {
         cout << "derivada 2 \n";
      }
};
void prt(base *q) {
   q->mostra1();
   q->mostra2();
}
void main( ) {
   base b;
   base *p;
   derivada1 dv1;
   derivada2 dv2;

   p = &b;
   prt(p);

   dv1.mostra1();
   p = &dv1;
   prt(p);

   dv2.mostra2();
   p = &dv2;
   prt(p);
}
```

```
// programa 2
#include <iostream>

class teste {
    int d;
 public:
    teste () {
       d = 0;
       cout << "default \n";
    }
    teste (int p, int q = 0) {
       d = p + q;
       cout << "soma \n";
    }
    teste (teste & p) {
       d = p.d;
       cout << "copia \n";
    }
    teste & operator = (teste & p)
    {
       cout << "atribuicao 1\n";
       d = p.d; return *this;
    }
    teste & operator = (int i)  {
       cout << "atribuicao 2\n";
       d = i; return *this;
    }
    void mostra ( ) {
       cout <<  d << " \n";
    }
};
void main ( ) {
   teste e1 (2, 6);  e1.mostra ( );
   teste e2;           e2.mostra ( );
   teste e3 (73);    e3.mostra ( );
   teste e4;
   e4 = e1;  e4.mostra();
   teste e5 (e2);    e5.mostra ( );
   e5 = 55;           e5.mostra ( );
   teste e6 = e3;   e6.mostra ( );
     teste  e7  (21,3);    e7.mostra
( );
   e1 = e3 = e5 = e7;
   e1.mostra();     e5.mostra();
}
```

```
// programa 3
#include <iostream>

template <class T>
class pilha {
    T* v;
    T* p;
public:
    pilha (int i) {
       cout << "cria "<< i << "\n";
       v = p = new T[i];
    }
    ~pilha () {
       delete[ ] v;
        cout << "tchau \n";
    }
    void empilha (T a) {
       cout << "emp "<< a << "\n";
       *p++ = a;
    }
    T desempilha () {
       return *--p;
    }
    int vazia() {
      if (v == p) return 1;
      else  return 0;
    }
};
void main ( ) {
   pilha<int> p(40);
   pilha<char> q(30);
   p.empilha(11);
   q.empilha('x');
   p.empilha(22);
   q.empilha('y');
   p.empilha(33);
   do {
      cout << p.desempilha() <<
      "\n";
   } while (!p.vazia());
   do {
      cout << q.desempilha() <<
      "\n";
   } while (!q.vazia());
}
```

O polimorfismo em LPs se refere à possibilidade de criar código capaz de operar (ou, pelo menos, aparentar operar) sobre valores de tipos distintos.

Programa 1

Tipo de polimorfismo: Por inclusão. Baseia-se na noção de hierarquia de classes para tratar objetos de um determinado tipo (da superclasse) como sendo de outro (da subclasse) através da amarração dinâmica de tipos. Os subprogramas *mostra1* e *mostra2* definidos inicialmente na classe *base* foram reescritos nas suas subclasses *derivada1* e *derivada2*, respectivamente.

Resultado da execução:

```
base 1
base 2
derivada 1
derivada 1
base 2
derivada 2
base 1
base 2
```

Em *main* são criados inicialmente um objeto do tipo *base* (*b*), um ponteiro para objeto do tipo *base* (**p*), um objeto do tipo *derivada1* (*dv1*) e um objeto do tipo *derivada2* (*dv2*). Com a execução da instrução *p = &b;*, *p* passa apontar para *b*. Assim, quando *prt (p);* é chamada, são executadas as funções *mostra1* e *mostra2* definidas na classe *base*, imprimindo, respectivamente, *base 1* e *base 2*. Com a execução de *dv1.mostra1();*, a função *mostra1* definida na classe *derivada1* é executada, imprimindo *derivada 1*. Com a execução de *p = &dv1;*, *p* passa a apontar para *dv1*. Na próxima linha, a execução de *prt (p);* chama o método *mostra1* de *derivada1* porque o método *mostra1* é amarrado dinamicamente em *base* (uso da palavra *virtual*) e, em seguida, chama o método *mostra2* de *base* uma vez que esse método é amarrado estaticamente (não é precedido pela palavra *virtual*). Assim, são impressos *derivada 1* e *base 2*. A instrução seguinte *dv2.mostra2 ();* imprime *derivada 2*. A execução de *p = &dv2;* faz com que *p* passe a apontar para *dv2*. A execução de *prt (p);* chama o método *mostra1* herdado de *base* por *derivada2* e depois chama o método *mostra2* de *base*, uma vez que ele é amarrado estaticamente. Por conseguinte, são impressos *base1* e *base2*.

Programa 2

Tipo de polimorfismo: Sobrecarga. Permite várias definições de funções referidas por um mesmo símbolo ou identificador. Dá a aparência de usar na chamada um mesmo trecho de código para diferentes tipos de dados dos parâmetros. Esse programa utiliza o polimorfismo de sobrecarga para implementar diferentes construtores para a classe *teste*, além de sobrecarregar o operador =.

Resultado da execução:

```
soma
8
default
0
soma
```

73
default
atribuicao 1
8
copia
0
atribuicao 2
55
copia
73
soma
24
atribuicao 1
atribuicao 1
atribuicao 1
24
24

Em *main*, a execução de *teste e1 (2,6);* imprime s*oma*, pois o construtor definido com dois parâmetros na classe *teste* é chamado. A execução de *e1.mostra ();* imprime *8*. A execução de *teste e2;* chama o construtor default (sem parâmetros) definido em *teste*, imprimindo *default.* A execução de *e2.mostra ();* imprime *0*. A execução de *teste e3 (73);* imprime novamente *soma* pois um dos parâmetros do construtor com dois parâmetros assume o valor default *0*. A execução de *e3.mostra ();* imprime *73*. A execução de *teste e4;* imprime novamente *default* e a execução de *e4 = e1* chama o método de atribuição que recebe um teste como parâmetro, imprimindo *atribuicao 1.* A execução de *e4.mostra ();* imprime *8*. A execução de *teste e5 (e2);* chama o construtor que recebe um *teste* como parâmetro e imprime *copia*, pois o construtor de cópia é chamado. A execução de *e5.mostra ();* imprime *0.* A execução de *e5 = 55;* chama o operador de atribuição que recebe um inteiro como parâmetro, imprimindo *atribuicao 2* e a execução de *e5.mostra ();* imprime *55*. A execução de *teste e6 = e3;* chama novamente o construtor de cópia, imprimindo *copia*, já que esta é uma operação de inicialização. A execução de *e6.mostra ()*; imprime *73*. A execução de *teste e7 (21, 3);* imprime novamente *soma ie* a execução de *e7.mostra ();* imprime *24*, a execução de *e1 = e3 = e5 = e7;* chama sucessivamente o operador de atribuição que recebe um *teste* como parâmetro e imprime três vezes *atribuicao 1.* A execução de *e1.mostra ();* imprime *24* e a execução de *e5.mostra ();* também imprime *24*.

Programa 3

<u>*Tipo de polimorfismo*</u>: Paramétrico. Permite a definição de estruturas de dados ou funções que contenham um ou mais parâmetros indicando o(s) tipo(s) do(s) elemento (s) que serão manipulados. O mecanismo de *template* é utilizado para permitir a implementação do tipo pilha genérica. Na execução do programa serão criadas a pilha do tipo inteiro e a pilha do tipo caractere, a partir da mesma classe pilha.

<u>*Resultado da execução*</u>:

cria 40

cria 30
emp 11
emp x
emp 22
emp y
emp 33
33
22
11
y
x
tchau
tchau

Em *main*, a instrução *pilha<int> p(40);* chama o construtor de *pilha* e imprime *cria 40*, a instrução *pilha<char> q(30);* chama o construtor de *pilha* e imprime *cria 30*, a instrução *p.empilha(11);* imprime *emp 11*, a instrução *q.empilha('x');* imprime *emp x*, a instrução *p.empilha(22)* imprime *emp 22*, a instrução *q.empilha('y');* imprime *emp y* e a instrução *p.empilha(33)* imprime *emp 33*. O *do-while* seguinte imprime inicialmente o número *33* porque é o valor retornado por *p.desempilha()*, na segunda execução imprime *22* e na terceira imprime *11*. O próximo *do-while* imprime inicialmente a letra *y* porque é o elemento retornado por *q.desempilha()* e na segunda execução imprime *x*. Antes de terminar o programa, o destrutor de *p* e *q* são chamados imprimindo *tchau* duas vezes.

12. Qual a diferença entre as posturas adotadas por JAVA e C++ em relação ao polimorfismo de sobrecarga? Qual dessas posturas você acha melhor? Apresente argumentos justificando sua posição.

A diferença é que JAVA embute sobrecarga em operadores e em subprogramas de suas bibliotecas, mas somente subprogramas podem ser sobrecarregados pelo programador, enquanto C++ realiza e permite que programadores realizem sobrecarga tanto de programas quanto de operadores.
Os criadores de JAVA resolveram não incluir a sobrecarga de operadores por considerá-la capaz de gerar confusões e aumentar a complexidade da LP. A postura adotada por C++ é mais ampla e ortogonal, facilitando a leitura e redação dos programas, se utilizada corretamente. Em minha opinião pessoal, considero a postura de C++ mais adequada em termos de elegância. Em termos práticos, acho que o ganho conferido é pequeno se comparado a complexidade associada ao uso de sobrecarga de operadores.

13. Uma loja especializada em produtos de arte vende livros, discos e fitas de vídeo. Ela necessita montar um catálogo com as informações sobre cada produto. Todo produto possui um número único de registro, um preço e uma quantidade em estoque. Além disso, deve-se saber o número de páginas dos livros, o número de músicas dos discos e a duração da fita de vídeo. Outro aspecto importante a ser considerado é a existência de um tipo de produto de venda combinada (uma fita de vídeo combinada com um disco). Utilize C, C++ e JAVA para:

    a) Especificar tipos abstratos de dados para cada um dos produtos da loja (basta apresentar a definição do tipo e os cabeçalhos de suas operações de criação, leitura, obtenção de dados, escrita e destruição).

b) Supondo que os produtos da loja estão armazenados em uma lista genérica, fazer uma função/método para receber a lista dos produtos e uma quantidade mínima de produtos a serem mantidos em estoque, e listar quais produtos necessitam de reposição.

c) Fazer uma função/método para receber a lista dos produtos, uma quantidade de músicas e uma duração mínima de filme, e listar quais discos possuem menos músicas que a quantidade especificada e quais filmes possuem maior duração que a especificada.

AINDA A SER FEITA.

14. O Ministério da Defesa te contratou para desenvolver um protótipo de um sistema de informação em C++ que cadastre os militares existentes nas Forças Armadas Brasileiras e gere duas listagens.

a. Crie uma classe abstrata *Militar* com um atributo inteiro representando sua matrícula e outro atributo representando sua patente. Garanta que todas subclasses concretas de *Militar* implementem obrigatoriamente os métodos de leitura de dados de um militar, impressão de dados de um militar e verificação se o militar está habilitado para progredir na carreira.

   Utilize a classe seguinte para representar a patente do militar.

```
class Patente {
  private:
    string titulo;
    int tempo;  // na patente
  public:
    le() {
      cin >> titulo;
      cin >> tempo;
    }
    string retornaPatente() {
      return titulo;
    }
    int retornaTempo() {
      return tempo;
    }
    void incrementa (int t) {
       tempo+=t;
    }
    void imprime() {
      cout << titulo << " – "
      << tempo;
    }
}
```

b. Implemente uma subclasse concreta de *Militar*, denominada *MilitarAeronautica*, que será usada para representar os militares dessa divisão das forças armadas. Essa classe possui como atributo adicional o número de horas de vôo efetuadas pelo militar. Note que um militar da aeronáutica está em condições de progredir se tem mais de 2 anos naquela patente e se acumulou mais de 100 horas de vôo nesse período.

c. Considerando a existência de duas outras subclasses de *Militar* semelhantes a *MilitarAeronáutica*, denominadas *MilitarExercito* e *MilitarMarinha*, utilize uma classe *ListaMilitares* (consistindo de uma lista de ponteiros para militares) para implementar um programa que:

- solicite ao usuário o número total de militares das Forças Armadas;

- solicite ao usuário a corporação e os dados de cada militar das Forças Armadas;
- apresente os dados de todos os militares em condições de progredir na carreira;
- apresente os dados de todos os militares da Aeronáutica.

Observação: Você não deve implementar a classe *ListaMilitares*. Considere que, além de possuir o método construtor default e o destrutor, ela também possui os seguintes métodos:

```
void incluir(Militar* m);          // inclui um militar ao final da lista
Militar* retornar(int i);          // retorna o militar na i-ésima posição
```

a)

```
class Militar {
        int matricula;
        Patente p;
public:
        virtual void leitura () = 0;
        virtual void impressao () = 0;
        virtual int verificacao () = 0;
}
```

b)

```
class MilitarAeronautica: public Militar {
        float horas;
public:
        void leitura () {
                cin >> matricula;
                p.le();
                cin >> horas;
        }
        void imprime () {
                cout << matricula;
                p.imprime();
                cout << horas;
        }
        int verifica () {
                if (p.retornaTempo() > 2 && horas > 100)
                        return 1;
                else
                        return 0;
        }
}
```

c) AINDA A SER FEITA.