Resolução dos Exercícios do Capítulo VI

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.

```
int a, b; // variaveis globais
void troca(){
        int aux;
        aux = a;
        a = b;
        b = aux;
}
main(){
        int c, d, e, f;
        c = 0; d = 1; e = 2; f = 3;
        a = c;
        b = d;
        troca();
        c = a;
        d = b;
        a = e;
        b = f;
        troca();
        e = a;
        f = b;
}
```

Como no exemplo 6.3, a função *troca* não possui parâmetros, sendo necessário utilizar as variáveis globais *a* e *b* para lhe conferir generalidade e possibilitar o seu reuso em *main*.
Os problemas de redigibilidade, legibilidade e confiabilidade são ainda mais graves dos que os do exemplo 6.3 por causa da necessidade de modificação dos valores das variáveis globais *c*, *d*, *e* e *f* após a execução da função troca. Com isso, torna-se necessário escrever mais, a funcionalidade do programa fica ainda mais obscurecida por envolver uma maior quantidade de atribuições e aumenta-se a possibilidade do programador realizar alguma atribuição indevida ou esquecer alguma atribuição.

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 explique 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.

Não. Java optou por fazer a passagem de tipos primitivos sempre por cópia para o parâmetro formal, o que impede que uma troca entre os parâmetros formais se reflita nos parâmetros reais. Não haveria diferença significativa na resposta se a troca fosse

realizada entre parâmetros de um mesmo tipo objeto. Embora JAVA faça a passagem de parâmetros dos tipos objeto (não primitivos) por cópia de referência, os efeitos da troca continuariam sendo válidos apenas internamente a função, não se reflitindo nos parâmetros reais.

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.

Para simular o uso de TADs em C, é necessário definir um tipo de dados simples e um conjunto de operações (subprogramas) que se aplicam sobre valores desse tipo, isto é, subprogramas que têm parâmetros desse tipo.
Normalmente, a implementação das operações do TAD e quaisquer outras entidades de computação necessárias para a implementação dessas operações são definidas num arquivo de implementação (.c). No arquivo de interface (.h), o tipo da estrutura de dados é definido e os protótipos dos subprogramas correspondentes às operações do TAD são declarados, para que sejam disponibilizados para os programadores usuários. Assim, o programador usuário não necessita mais implementar o código das operações do TAD, o que torna o código mais legível e redigível, além do código usuário não precisar ser alterado caso seja necessário realizar uma alteração na implementação do tipo ou nas suas operações (desde que os cabeçalhos das operações não sejam modificados).
Os problemas com essa aproximação são que, além dela não promover o encapsulamento de dados e operações em uma única unidade sintática, ela não impede o uso indisciplinado do TAD. A operação de inicialização do TAD, por exemplo, deve ser chamada explicitamente pelo programador. Caso o programador usuário esqueça ou não chame essa operação antes de qualquer outra, o uso correto do TAD ficará comprometido. Além disso, é importante observar que ele pode realizar operações adicionais sobre o TAD além das especificadas pelos subprogramas. Por exemplo, o programador pode acessar diretamente a estrutura interna do TAD, quebrando o ocultamento de informações.

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.

Essa atribuição não produz qualquer efeito sobre o parâmetro real correspondente, só produzindo efeitos internos, uma vez que, em JAVA, as atribuições de valores completos do tipo não-primitivo ao parâmetro formal não produzem efeito no parâmetro real, sendo a passagem para tipos não-primitivos considerada, nesse caso, unidirecional de entrada.

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.

A função *System.out.println* de Java recebe sempre uma string como argumento. Ela funciona de forma similar a *printf* por conta do uso combinado da operação de concatenação (+) de strings e da realização de conversão implícita de tipos antes da operação de concatenação. Tipos primitivos e objetos são convertidos implicitamente para strings antes da realização da concatenação (quando o outro argumento é uma string). Por exemplo, na chamada de *System.out.println* seguinte

```
int i =3;
System.out.println ("teste: " + i + ": "+ new Float (4.3) + ": "+ true);
```

o valor *3* de *i* é convertido na string *"3"* antes de ser concatenado à string *"teste"*. Observe que esse processo se repete antes da realização de cada operação de concatenação. Em particular, o objeto *Float* criado é convertido para uma string através da chamada implícita da operação *toString()* dessa classe. Note que o valor booleano (primitivo) também é convertido implicitamente na string *"true"*.
Essa abordagem de JAVA pode causar confusões e erros. Na chamada seguinte de *System.out.println*

```
int i =3;
int j =5;
System.out.println (i + j);
```

não haverá conversão de tipo e concatenação de strings. Como nenhum dos dois argumentos é string, JAVA fará primeiro a operação aritmética e depois a conversão do resultado para string, resultando na impressão da string *"8"* em vez de *"35"*, como alguém pode eventualmente esperar.
O problema da falta de lista de parâmetros variável pode ser contornado de maneira geral em JAVA através da utilização de um parâmetro formal vetor no qual não se especifica o número de elementos do vetor. Se a função recebe elementos de um mesmo tipo, isso dá flexibilidade para passar uma quantidade variável de valores para a função em cada chamada. Se a função necessita receber elementos de tipos diferentes, basta definir o parâmetro formal como um vetor de objetos. Essa solução é tão geral quanto a solução de C e C++ e é mais legível, uma vez que não é necessária a utilização de macros pré-definidas, cujo comportamento não é amplamente conhecido, como é o caso em C e C++. Por outro lado, a redigibilidade é um pouco prejudicada pela necessidade de criar o vetor de objetos com os elementos que precisam ser passados para a função antes da sua chamada. A solução mais geral de JAVA pode ser observada no trecho de código abaixo:

```
void f(Object[] x) {
  for (int i = 0; i < x.length; i++) System.out.println (x[i]);
}
void g() {
  Object[] a = new Object[] {
    new Integer(12), new Float(1.5), new Boolean(true) };
  f(a);
  a = new Object[] {
    new Integer(1), new Byte(15) };
  f(a);
}
```

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.

Em situações nas quais ocorrem alterações nos cabeçalhos (protótipos) dos subprogramas que compõem a interface, o código usuário, que utiliza esses subprogramas, tem que ser modificado. Isso também ocorre se existem atributos da estrutura do TAD que são públicos, isto é, que fazem parte da interface. Quando esses atributos são alterados, o código usuário, que utiliza esses atributos, tem que ser modificado.

7. O uso de parâmetros em um subprograma visa aumentar as possibilidades de reuso desse subprograma. Normalmente, os valores dos parâmetros correspondem a dados 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.

Exemplo em C de uma função que soma dois termos inteiros com o resultado da aplicação de uma função arbitrária sobre os mesmos.

```
#include <stdio.h>
int multiplica (int a, int b) {
  return (a * b);
}
int maluca (int a, int b) {
   return (2 * a + 5 * b);
}
int aplicacao (int a, int b, int (*f) (int, int)) {
   return (a + b + (*f) (a, b));
}
main () {
   int x = 1, y = 2, resultado;
   resultado = aplicacao (x, y, maluca);
   printf ("resultado = %d", resultado); //imprime 15
   resultado = aplicacao (x, y, multiplica);
   printf ("\nresultado = %d", resultado); //imprime 5
}
```

No programa acima, a não utilização do parâmetro ponteiro para função implicaria na necessidade da construção de diferentes funções, do mesmo formato da função *aplicacao*, porém específicas para cada funcionalidade desejada.
A utilização de ponteiros para funções como parâmetros reduz a legibilidade, devido à necessidade de se utilizar ponteiros e suas operações, porém melhora a redigibilidade e a reusabilidade, pois um mesmo subprograma pode ser utilizado para atingir

diferentes funcionalidades, como é o caso da função *aplicacao,* em um mesmo programa ou em programas diferentes. Com relação à eficiência ocorre uma pequena perda em função da necessidade de dereferenciamento do ponteiro na chamada de função dentro de *aplicacao*.

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)

A abordagem adotada por C apresenta melhor redigibilidade pois em MODULA-2 é necessário criar uma função distinta para cada contexto no qual o tipo e número dos parâmetros varia. Já em C, em muitos casos, pode-se utilizar uma mesma função nestes contextos. Tomando por exemplo a função *printf* de C, tem-se que, como essa função é usada nos mais variados contextos, qualquer limitação no seu número e tipo de parâmetros a tornaria insuficiente para atender a todas as possíveis demandas. Assim, ela deveria ser reescrita, em MODULA-2, cada vez que fosse necessária em aplicações diferentes. Com relação à confiabilidade, a abordagem adotada por MODULA-2 é mais confiável, pois na abordagem adotada por C, não é possível verificar os tipos dos parâmetros em tempo de compilação. Nesse caso, é tarefa exclusiva dos programadores garantir o funcionamento correto do subprograma.
Comparando as abordagens adotadas por C e Pascal, podemos concluir que C é melhor tanto na questão da ortogonalidade, uma vez que não restringe a utilização de lista de parâmetros variáveis a alguma funções da linguagem, como o faz Pascal, quanto na questão da reusabilidade, pois permite o reúso de uma maior quantidade de funções do que Pascal.

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.

C:

A linguagem C não suporta a implementação do conceito de TADs, mas o programador pode simular o seu uso. Para simular o uso de TADs em C, é necessário definir um tipo de dados simples e um conjunto de operações (subprogramas) que se aplicam sobre valores desse tipo. A implementação das operações do TAD e quaisquer outras entidades de computação necessárias para a implementação dessas operações são definidas num arquivo de implementação (.c). No arquivo de interface

(.h), o tipo da estrutura de dados é definido e os protótipos dos subprogramas correspondentes às operações do TAD são declarados, para que sejam disponibilizados para os programadores usuários.

Exemplo em C – arquivo de interface (.h):

```
typedef struct no {
        int info;
        struct no *prox;
}No;
typedef struct fila {
        No *ini, *fim;
}Fila;
Fila Inicia ();
Fila Insere (Fila fil, int elem);
Fila Elim (Fila fil);
int Info (Fila fil);
int Vazia (Fila fil);
void Destroi (fila fil);
```

ADA:

ADA usa uma abordagem que envolve a 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 no arquivo 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 de interface.

Exemplo em ADA – unidade de interface

```
package fila_inteiros is
        type Fila is limited private;
        procedure Inicia (fil: out Fila);
        procedure Insere (fil: in out Fila; elem: in integer);
        procedure Elim (fil: in out Fila);
        function Info (fil: in Fila) return integer;
        function Vazia (fil: in Fila) return boolean;
private:
        max: constante integer := 100;
        type Fila is record
                ini: integer;
                elem: array (1 .. max) of integer;
                fim: integer;
        end record;
end fila_inteiros;
```

C++:

C++ utiliza o conceito de classe para implementar TADs. As 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 a ele. As classes

oferecem proteção dos dados do tipo utilizando especificadores de acesso. Por exemplo, em C++, os dados da classe e suas operações podem ser privados (só são visíveis para a implementação das operações do TAD) ou públicos (são visíveis para qualquer trecho do programa).
As classes tornam especiais as operações construtoras, que são identificadas pelo seu nome, que deve ser o mesmo da sua classe (resolvendo o problema da inicialização de TADs, já que, quando um objeto é criado, essa operação é chamada automaticamente), e destrutoras (também identificada pelo seu nome que é da forma *~nome_da_classe*), e oferecem suporte aos conceitos de herança e polimorfismo, fundamental para a orientação a objetos.

Exemplo em C++ - arquivo de interface (.h)

```
class Fila {
        struct No {
                int info;
                struct No *prox;
        }
        No *ini, *fim;
public:
        Fila ();
        Insere (int elem);
        Elim ();
        int Info ();
        int Vazia ();
        ~Fila ();
}
```

Enquanto a abordagem de C não atende satisfatoriamente a propriedade de encapsulamento, pois a estrutura de dados do TAD e suas operações não são definidas em uma única unidade sintática, ADA e C++ satisfazem esse requisito. De maneira análoga, ADA e C++ possibilitam o ocultamento de informação, o que não é possível em C. Com relação à confiabilidade, C++ é que mais satisfaz essa propriedade. Em ADA, eventualmente, operações podem ser chamadas sem que a operação de inicialização tenha sido realizada. Em C, além de poder ocorrer o mesmo problema de ADA, é possível que o programador usuário altere os dados do TAD sem o uso das operações definidas para o TAD. A alteração do código fonte usuário só é necessária em C++ e ADA quando os cabeçalhos das operações públicas ou atributos públicos são alterados. Em C, a alteração pode ser necessária sempre que há modificação nos atributos do TAD, uma vez que todos eles são públicos.

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.

a) Em *main*, antes da chamada da função *calculoMaluco*, tem-se *valor = 0* e *lista = {1, 3, 5, 7, 9}*. Na chamada da função *calculoMaluco*, o parâmetro formal *a* recebe *valor = 0*, ou seja, *a = 0* e o parâmetro formal *b* recebe *lista[0] = 1*, ou seja, *b = 1* (modo normal). Em *calculoMaluco*, os valores de *a* e *b* mudam para *a = 1* e *b = 2*, mas esses valores não são repassados aos parâmetros reais *valor* e *lista[0]*, pois a passagem de parâmetros é unidirecional de entrada variável e o mecanismo é por cópia. Assim, os valores das variáveis *valor* e *lista* após a execução do programa são: *valor = 0* e *lista = {1, 3, 5, 7, 9}*.
b) Em *main*, antes da chamada da função *calculoMaluco*, tem-se *valor = 0* e *lista = {1, 3, 5, 7, 9}*. Na chamada da função *calculoMaluco*, o parâmetro formal *a* passa a referenciar *valor = 0* e o parâmetro formal *b* passa a referenciar *lista[0] = 1* (modo normal e mecanismo por referência). Em *calculoMaluco*, os valores de *a* e *b* mudam para *a = 1* e *b = 2*, alterando, respectivamente, os valores de *valor* e *lista[0]* (mecanismo por referência e passagem bidirecional de entrada e saída). Assim, os valores das variáveis *valor* e *lista* após a execução do programa são: *valor = 1* e *lista = {2, 3, 5, 7, 9}*.
c) Em *main*, antes da chamada da função *calculoMaluco*, tem-se *valor = 0* e *lista = {1, 3, 5, 7, 9}*. Na primeira linha da função *calculoMaluco*, o parâmetro formal *a* passa a referenciar *valor* e o parâmetro formal *b* passa a referenciar *lista[valor]* (modo por nome e mecanismo por referência). Logo a instrução passa a ser *valor = valor + lista [valor]*. Na execução da instrução dessa linha, o parâmetro formal *valor* passa a valer *1* pois *valor* era igual a zero e *lista[0]* é igual a *1*. Na segunda linha da função *calculoMaluco*, o parâmetro formal *a* passa a retornar *1* (o valor corrente de *valor*) e o parâmetro formal *b* passa a referenciar *lista[1]*, o que vale *3* (modo por nome e mecanismo por referência). Na execução da instrução dessa linha, o parâmetro formal *b* e, consequentemente, o parâmetro real *lista[1]* tem seu valor modificado para *4*. Assim, os valores das variáveis *valor* e *lista* após a execução do programa são: *valor = 1* e *lista = {1, 4, 5, 7, 9}*.

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* c);          // cria a partir de string
struct BigInt criaBigIntN (unsigned n);       // cria a partir de unsigned
struct BigInt criaBigIntB (struct BigInt b);  // cria a partir de outro BigInt
void atribui (struct BigInt* b1, struct BigInt* b2);
struct BigInt soma (struct BigInt b1, struct BigInt b2);
void imprime (FILE* f, struct BigInt b);
void destroi (struct BigInt b);

// C++
class BigInt {
        char* digitos;
        unsigned ndigitos;
public:
        BigInt (const char* c);
        BigInt (unsigned n = 0);
        BigInt (const BigInt& b);
        void atribui (const BigInt& b);
        BigInt soma (const BigInt& b) 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.

A definição de C++ tem melhor encapsulamento porque os atributos e as operações de *BigInt* são definidas em uma única entidade sintática. Ela também possui maior proteção de dados porque os atributos de *BigInt* são privados, enquanto na versão de C eles são públicos. A confiabilidade das operações de inicialização e terminação também é maior em C++ porque essas operação são garantidamente chamadas no início e ao final da vida dos objetos *BigInt*. Isso não ocorre com a implementação em C pois o programador usuário deve chamá-las explicitamente, o que possibilita seu esquecimento.

Na operação *void atribui (const BigInt& b);* o parâmetro *b* tem passagem unidirecional de entrada constante por referência. Na operação *void atribui(BigInt);* o parâmetro *b* tem passagem unidirecional de entrada constante por cópia. Portanto, em termos de proteção de dados do parâmetro formal elas se equivalem (ambas são passagem unidirecional de entrada constante). O que as diferencia é a eficiência de execução, que é melhor na por referência pois não existe a necessidade de realização de cópia dos dados nem da chamada do construtor de cópia.

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++?

A recompilação do código usuário é necessária pois a alteração na estrutura de dados modifica o tamanho do espaço a ser armazenado pelo código usuário dessa estrutura. JAVA sempre aloca objetos no monte, logo o código usuário só necessita alocar uma referência para o objeto na pilha. Como a interface com o construtor não foi alterada e é ele quem aloca o espaço no monte, não haverá problemas na mudança da estrutura. C++ não incorpora o mecanismo implementado em JAVA porque a alocação no monte tem desempenho menos eficiente que a alocação na pilha.

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.

Em *main*, antes da entrada no *for*, tem-se *a = {1, 2, 3}*. Na primeira vez que o *for* é executado, os parâmetros reais de *incrementa* são *a[0] = 1* e *a[1] = 2*. Eles são passados por referência, assim, em *incrementa*, *x* e *y* são referências para *a[0] = 1* e *a [1] = 2*. Na primeira instrução, *x*, e consequentemente *a[0]*, tem seu valor alterado para *3*. Na segunda, *y*, e consequentemente *a[1]*, tem seu valor alterado para *3*. Logo, ao final da primeira chamada *a = {3, 3, 3}* e o valor impresso é *3*.
Na segunda vez que o *for* é executado, os parâmetros reais de *incrementa* são *a [1] = 3* e *a[1] =3*. Eles são passados por referência, assim, em *incrementa*, *x* e *y* são referências para *a[1] =3.* Na primeira instrução, *x*, e consequentemente *a[1]*, tem seu valor alterado para 6. Na segunda, *y*, e consequentemente *a[1]*, tem seu valor alterado para *7*. Logo, ao final da segunda chamada *a = {3, 7, 3}* e o valor impresso é *7*.
Na terceira vez que o *for* é executado, os parâmetros reais de *incrementa* são *a [2] = 3* e *a[1] = 7*. Eles são passados por referência, assim, em *incrementa*, *x* e *y* são referências para *a[2] =3* e *a[1] = 7*. Na primeira instrução, *x*, e consequentemente *a [2]*, tem seu valor alterado para *10*. Na segunda, *y*, e consequentemente *a[1]*, tem seu

valor alterado para *8*. Logo, ao final da segunda chamada *a = {3, 8, 10}* e o valor impresso é *10*.
A execução desse código produz um efeito estranho prejudicial à legibilidade. Ocorre sinonímia quando a variável *i* assume o valor *1*, no *for*, pois os dois parâmetros formais da função *incrementa* são associados ao mesmo parâmetro real *a[1]*, o que dificulta o entendimento do programa.

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++);
}
```

No modo normal (ou *eager*) a avaliação dos parâmetros reais 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.

Execução pelo modo **normal** (*eager*):

Resultado: 0
0
0

Como a avaliação dos parâmetros é normal, o valor passado para *c* será resultado da avaliação retornado pela expressão *j++* antes da execução de *avalia*. Portanto, *c = 0*.

Execução pelo modo por nome (*by name*):

Resultado: 0
1
2

Explicação: No modo de avaliação por nome, no momento em que o parâmetro formal *c* é utilizado pela primeira vez na função *avalia*, tem-se *c = 0* e *j* torna-se *1*. Na utilização de *c* pela segunda vez tem-se *c = 1* e *j* torna-se *2*. Na utilização de *c* pela terceira vez tem-se *c = 2*. O valor de *j* muda a cada avaliação do parâmetro real, o que acontece sempre que o parâmetro formal é utilizado. Isso faz com que o valor do parâmetro formal impresso seja diferente a cada impressão.

Execução pelo modo preguiçoso (*lazy*):

Resultado: 0
0

Na avaliação preguiçosa, o parâmetro real *j++* é avaliado no momento em que o parâmetro formal *c* é utilizado pela primeira vez e permanece assim até o final da execução da função. Logo, *c = 0* em todas os passos da repetição.

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.

Em C é necessário que o programador crie uma função para percorrer cada nó da lista e vá desalocando-os. Essa função deve ser chamada explicitamente quando a lista necessita ser desalocada. C++ utiliza uma função especial destrutora da classe, a qual deve ser implementada pelo programador. Essa função é chamada implicitamente quando se encerra o tempo de vida da lista. Em JAVA, a desalocação é feita automaticamente pelo coletor de lixo, não existindo necessidade de implementação de uma operação destrutora ou invocação de qualquer método.
Em termos de redigibilidade, a melhor solução é a de JAVA pois o programador não necessita redigir nada. C++ requer a implementação da função destrutora, mas não necessita da redação de código para chamada dessa função. C é a solução de menor redigibilidade pois o programador necessita implementar a função de desalocação e também chamá-la explicitamente no código usuário.
Em termos de confiabilidade, a solução de JAVA também é superior, pois o programador não necessita definir como os nós lista devem ser desalocados, nem quando isso deve ser feito. O próprio sistema se encarrega dessas tarefas. A solução de C++ é mais confiável do que a de C porque ela garante que a operação de desalocação será garantidamente chamada ao final do tempo de vida da lista.
Por outro lado, em termos de eficiência, C e C++ podem superar JAVA pois deixam a cargo do programador a função de desalocar memória quando esta não é mais necessária. Com isso, um bom programador pode explorar o contexto para dizer como e em qual momento é mais eficiente fazer a operação de desalocação.

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.

C
Direção da passagem: A linguagem 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.
Mecanismo de passagem: C oferece apenas o mecanismo de passagem por cópia.

C++
Direção da passagem: C++ oferece as passagens unidirecional de entrada variável ou constante e bidirecional de entrada e saída.
Mecanismo de passagem: C++ adota tanto o mecanismo de passagem por cópia quanto por referência. Ela usa o operador *&* para designar um parâmetro passado por referência, diferenciando-o assim dos parâmetros que usam passagem por cópia.

JAVA

Direção da passagem: 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 as 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.
Mecanismo de passagem: Quando a passagem é de tipos primitivos, ocorre cópia. Quando a atribuição é de tipos não-primitivos passa-se uma cópia da referência.

Comparando os mecanismos tem-se que, em geral, a passagem de um parâmetro que contenha um grande volume de dados, como um vetor, é mais eficiente por meio do mecanismo de referência do que o de cópia. Ao 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 valores do parâmetro real para o formal e vice-versa. Isso é contrastado com a passagem por referência, a qual demanda apenas 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 referência é a possibilidade de ocorrência de sinonímia. 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. Isso pode ser difícil de identificar, em caso de engano, e entender quando necessário, prejudicando a legibilidade e a confiabilidade no código.
Além disso, o mecanismo de cópia apresenta boa confiabilidade porque facilita 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 ao real no final da execução, uma interrupção inesperada do subprograma não afeta, em princípio, o estado das variáveis do programa.
De forma geral, pode-se dizer que C++ apresenta a maior variedade de tipos de passagem de parâmetros, oferecendo, em particular, a passagem unidirecional de entrada constante por referência que é mais confiável do que a passagem de objetos em JAVA e mais eficiente do que a passagem de tipos compostos por valor em C.