Resolução dos Exercícios do Capítulo V

1. Um programa deve ler uma sequência de números inteiros e imprimí-los. O programa deve ser interrompido quando o número lido for zero. Implemente três versões desse programa em C usando respectivamente os comandos iterativos com pré-teste, com pós-teste e um comando de escape. Discuta as soluções apresentadas em termos de redigibilidade e eficiência, indicando a melhor solução apresentada.

Pré-Teste	***Pós-Teste***	***Comando de Escape***
main() { * int n;* * scanf ("%d", &n);* * while (n != 0) {* * printf ("n: %d\n", n);* * scanf ("%d", &n);* * }* *}*	*main() {* * int n;* * do {* * scanf ("%d", &n);* * if (n)* * printf ("n: %d\n", n);* * } while (n != 0);* *}*	*main() {* * int n;* * for (;;) {* * scanf ("%d", &n);* * if (n == 0) break;* * printf ("n: %d\n", n);* * }* *}*

A melhor solução em termos de redigibilidade e eficiência é a que utiliza o comando de escape. Ela só faz uma comparação e exige a leitura do valor em apenas um ponto do programa. A solução com pré-teste é também eficiente, embora exija leitura do valor em dois pontos do programa. Portanto, é menos redigível. A solução com pós-teste é menos eficiente e redigível pois requer a realização de dois testes em dois pontos do programa a cada passo da iteração. Note que se poderia fazer uma solução com pós-teste que fosse tão eficiente como as demais. Bastaria colocar uma leitura e um teste antes da entrada na iteração. Essa solução não foi apresentada porque corresponde a um comando de pré-teste.

2. Descreva o que ocorre em cada trecho que culmina com impressões no seguinte programa em C, justificando suas afirmações.

```
#include <stdio.h>
void main () {
  int a, b, c;
  b = c = 10;
  a = b++ + b++;    //(1)
  printf("%d\n", a);
  printf("%d\n", b);
  a= ++c + ++c;    //(2)
  printf("%d\n", a);
  printf("%d\n", c);
  b = 10;
  a = b++ + b;     //(3)
  printf("%d\n", a);
  printf("%d\n", b);
```

```
  a = 10;
  b = 5;
  if (a>b || ++b>5)  //(4)
     printf("%d\n", b);
  a = 1;
  b = 5;
  if (a>b || ++b>5) //(5)
      printf("%d\n", b);
}
```

Esse programa em C é portável? O que ocorreria se um programa equivalente (isto é, usando classe e com o comando apropriado de saída) fosse implementado em JAVA? Justifique todas as suas afirmações.

(1) *b* assume o valor 12 e *a* assume o valor 21. Embora C não especifique a ordem de avaliação dos operandos de uma expressão, os operandos da operação + são os mesmos. Portanto, uma das expressões sempre retornará o valor 10 para *b* e a outra sempre retornará o valor 11.
(2) *c* assume o valor 12 e *a* assume o valor 23. Tal como na situação anterior, os operandos da operação + são os mesmos. Portanto, uma das expressões sempre retornará o valor 11 para *c* e a outra sempre retornará o valor 12.
(3) *b* assume o valor 11 e *a* pode assumir o valor 20 ou 21, dependendo da implementação do compilador. Como C não especifica a ordem de avaliação dos operandos de uma expressão, *b* pode ser avaliado antes de *b*++ ou vice-versa. No primeiro caso, *b* e *b*++ retornarão o valor 10, atribuindo 20 a *a*. No segundo caso, *b*++ retornará o valor 10 e *b* retornará 11, atribuindo 21 a *a*.
(4) Como *a*>*b* é verdadeiro, a outra expressão do ou lógico não será avaliada por causa da avaliação em curto-circuito. Assim, *b* não é incrementado, resultando em *b* = *5*.
(5) Como *a*>*b* é falso, a outra expressão do ou lógico terá de ser avaliada. Assim, *b* é incrementado, resultando em *b* = *6*.

Este programa não é portável por causa da operação na linha comentada com (3). De acordo com o compilador utilizado, o mesmo programa pode resultar no valor 20 ou 21 para *a* após esta linha de comando. É importante ressaltar que isso pode ocorrer em implementações em uma mesma plataforma, dependendo apenas do uso de dois compiladores distintos que implementem de modo diferente a ordem de avaliação daquela expressão.

A única diferença que ocorreria nos resultados desse programa é que *a* certamente assumiria o valor 21 após a execução da linha comentada com (3). Isso ocorreria porque JAVA estabelece que os operandos de uma expressão são sempre avaliados da esquerda para a direita. Assim, *b*++ seria avaliada primeiro, retornando 10 e incrementando o valor de *b*. Ao avaliar a expressão *b*, portanto, seria retornado 11, resultando na atribuição de 21 para *a*. Como consequência disso, esse programa em JAVA é portável.

3. Faça um programa em C (sem usar os escapes return ou exit) para ler um número inteiro positivo e calcular a tabuada de multiplicação de 0 a 9 de todos os números positivos inferiores ao número lido. Sempre que o resultado de uma multiplicação

for múltiplo de dez, o programa deve perguntar ao usuário se ele deseja continuar com o cálculo da tabuada. Portanto, o programa deve se encerrar ao final do cálculo da tabuada ou quando o usuário responder que não deseja continuar o cálculo. A sua solução é a mais eficiente para esse problema? Porque? Como a solução mais eficiente seria implementada em JAVA?

Existe dois modos de fazer esse programa em C obedecendo as restrições impostas. A primeira forma não é a mais eficiente pois requer o teste da variável de controle em todo passo de cada uma das repetições:

```
main() {
  int n // contém o número digitado pelo usuário
  int r, resp, saida;
  printf (“Entre com o número: “);
  scanf(“%d”, &n);
  saida = 0;
  for (i=n-1; i>0 && !saida; i--) {
     for (j=0; j<=0 && !saida; j++) {
        r = i*j;
        printf(“%d x %d = %d\n”, i, j, r );
        if (r%10==0) {
           printf("deseja continuar? \n");
           printf("sim(digite 1) / nao (digite 0)");
           scanf("%d", &resp);
           if (!resp) {
              saida = 1;
           }
        }
     }
  }
}
```

A segunda forma utiliza o comando *goto* para não requerer a realização desses testes. Por conta disso, consegue ser mais eficiente:

```
main() {
  int n // contém o número digitado pelo usuário
  int r;
  printf (“Entre com o número: “);
  scanf(“%d”, &n);
  for (i=n-1; i>0; i--) {
    for (j=0; j<=9; j++) {
       printf(“%d x %d = %d\n”, i, j, i*j );
         if ((i*j)%10==0) {
           printf("deseja continuar? \n");
           printf("sim(digite 1) / nao (digite 0)");
           scanf("%d", &resp);
           if (!resp) {
              goto saida;
           }
```

```
        }
    }
  }
}
saida:
```

Como JAVA não possui goto, para implementar a solução mais eficiente é necessário utilizar o comando de escape rotulado, tal como no trecho seguinte:

```
public static void main (String[] args) {
  int n; // contém o número digitado pelo usuário
  int r, resp;
  n = Console.readInteger();    // le um inteiro
  saida:
  for (int i=n-1; i>0; i--) {
     for (int j=0; j<=9; j++) {
          r = i * j;
         System.out.println(i+" x "+j+" = "+ r);
         if (r%10==0) {
              System.out.println ("deseja continuar? \n");
              System.out.println ("sim(digite true) / nao (digite false)");
              resp = Console.readBoolean( );  // le um booleano
              if (!resp) {
                break saida;      //  escape rotulado
              }
         }
     }
  }
}
```

4. Apresente um exemplo de expressão em C em que ocorre curto-circuito associado a efeito colateral. Analise o efeito que tal expressão pode produzir sobre a legibilidade de um programa.

```
a = 5;
b = 0;
while(--a || b++)  printf("%d\n", a + b);
```

O código em C acima apresenta efeito colateral na avaliação do *while* já que durante a avaliação de *--a || b++* os valores das variáveis *a* e *b* são alterados. Também há curto circuito pois se *--a* é diferente de zero então *b++* não é avaliado.
Essa expressão produz um efeito negativo na legibilidade do programa por dificultar não apenas o conhecimento de qual valor será impresso pelo *printf*, mas também saber quantas vezes a função *printf* irá executar.
De fato, o *printf* será executado 4 vezes, pois enquanto *--a* for diferente de zero o *b* não será incrementado, e quando *--a* for igual a zero o valor de *b* será avaliado antes de ser incrementado, logo no quinto loop a condição do *while* será falsa. Por conta disso, os valores que serão impressos por *printf* serão os valores de *a*, uma vez que em todas as repetições o valor retornado por *b* será *0.*

5. O comando *goto* é empregado para realizar desvios incondicionais no fluxo de controle de programas. Com o advento das técnicas de programação estruturada, este comando foi muito criticado e, por muitas vezes, se sugeriu que linguagens de programação não deveriam incluí-lo entre seus comandos. C é uma linguagem que adota os princípios da programação estruturada. No entanto, C manteve o *goto* como um comando da linguagem. Qual a razão dessa decisão? Exemplifique, com um trecho de programa em C, uma situação na qual pode ser útil empregar o comando *goto*. Discuta como programadores da linguagem MODULA-2 e JAVA, que não incluem o *goto* entre seus comandos, lidam com esta situação.

C manteve o *goto* porque em algumas situações ele permite a construção de código mais eficiente. Por exemplo, quando se necessita sai de várias comandos de repetição aninhados:

```
for(i = 0; i < n; i++)
                    for(j = 0; j < n; j++)

if(a[i] == x) goto saida;
saida:
```

O programador de MODULA-2 tem de incluir um novo teste ao final de cada repetição para evitar a continuação das iterações. Isso torna os programas menos eficientes e menos redigíveis. Já o programador JAVA pode utilizar os comandos de escapes rotulados, obtendo assim as mesmas vantagens do uso do *goto* e, ao mesmo tempo, preservando a linguagem das inadequações proporcionadas pelo comando *goto*.

6. Diga qual o valor das variáveis *a* e *n* após cada linha do seguinte trecho de código C. Justifique suas respostas.

```
n = 3;
a = -- n ++;
a = -- n + 1;
a = -- n += 1;
```

O trecho de código desse exercício está incorreto. Do jeito que está a segunda e quarta linhas geram erros de compilação. O erro na segunda linha ocorre porque o resultado da operação de incremento é um número inteiro, o qual não pode ser argumento do operador de decremento. O erro na quarta linha ocorre porque o resultado da operação de decremento é um número inteiro, o qual não pode ser argumento do operador de atribuição composta.

A forma correta do exercício é:

```
n = 3;
a = - n ++;
a = - n + 1;
a = n += 1;
```

Assim, *n* recebe inicialmente o valor 3. Na linha seguinte, é realizada primeiramente a operação de incremento, retornando o valor 3 e incrementando *n* para 4. A operação de negação é realizada sobre o valor retornado pela operação de incremento e em seguida é feita a atribuição desse valor (-3) para *a.* Na terceira linha a operação de negação é realizada primeiro, retornando o valor –4, o qual é adicionado a 1 em

seguida. O valor –3 é atribuido para *a.* Na última linha, o valor de *n* é incrementado de 1, passando para 5. Este valor é retornado e atribuído a *a*.

7. Modifique o seguinte trecho de código para que ele realize a semântica sugerida pela sua disposição textual.

```
if ( x == 7 )
        if ( y == 11)
                z = 13;
else z = 17;
```

O trecho deve ser modificado para:

```
if ( x == 7 ) {
        if ( y == 11)
                z = 13;
                                        }
else z = 17;
```

8. Veja como funciona o exemplo 5.37. Execute-o passo a passo considerando que *n* tem valor 8 e os vetores *a* e *b* foram definidos da seguinte maneira:

```
int[] a = {1, 3, 6, 9, 10, 12, 16, 18};
int[] b = {2, 4, 5, 7, 8, 10, 11, 15};
```

Reimplemente esse exemplo em C garantindo que ele funcione exatamente como em JAVA.

O exemplo funciona da seguinte maneira:

1) se o valor do vetor *a* é maior que valor do vetor *b*, não faz nada, continuando o *for* interno;

2) se o valor do vetor *a* é menor que valor do vetor *b*, ele faz um *continue* para o *for* com rótulo *saida*, ou seja, continua o *for* mais externo;

3) se o valor do vetor *a* é igual ao valor do vetor *b*, ele faz um *break* para o *for* com rótulo *saida*, ou seja, executa a próxima linha de comando logo após o final do *for* mais externo.

No exemplo dado, ele encontra o valor 10 nos dois vetores e imprime “achou”.

A reimplementação do exemplo em C é apresentada a seguir:

```
for(i = 0 ; i < n ; i++) {
  for (j = 0 ; j < n ; j++) {
     if ( a[i ] < b[j])  goto forexterno;
     if ( a[i ] == b[j]) goto saida;
  }
  forexterno:
}
saida:
if (i<n) {
  printf(“achou!!!);
```

```
} else {
  printf("nao achou!!!");
}
```

9. Implemente e teste o seguinte programa em C e descreva o que acontece. Justifique porque isso ocorre dessa maneira (isto é, apresente o racional da decisão tomada pelos projetistas ou implementadores dessa LP).

```
void f() {
  int i = 10;
  entra:
  i++;
}
main() {
  f();
  goto entra;
}
```

Ao se compilar esse programa há erro de compilação pois o símbolo *entra* está sendo usado em *main*, mas foi definido em *f*. Isso ocorre porque os rótulos na linguagem C são visíveis apenas dentro da função na qual são definidos. Essa decisão foi tomada pelos projetistas ou implementadores da linguagem, a fim de evitar que fossem feitos desvios incondicionais para o corpo de funções do programa que não tenham sido chamadas ou que já haviam sido encerradas. Desta forma se impede que variáveis locais não alocadas (ou já desalocadas) sejam (ou continuem sendo) acessadas.

10. Analise o seguinte programa em C, identificando o que ele faz. Faça uma crítica ao estilo de programação utilizado.

```
main() {
  int i, j, k;
  k = 1;
  for (i = 0; i < 10; i++) {
       entra:
       j = 2 * i + 1;
       printf("i: %d, j: %d\n", i, j);
  }
  if (k) {
      k = 0;
      i = 7;
     goto entra;
  }
}
```

A saída para este programa será:

```
i: 0, j: 1
i: 1, j: 3
i: 2, j: 5
```

i: 3, j: 7
i: 4, j: 9
i: 5, j: 11
i: 6, j: 13
i: 7, j: 15
i: 8, j: 17
i: 9, j: 19
i: 7, j: 15
i: 8, j: 17
i: 9, j: 19

Como pode ser observado, o programador utilizou um desvio irrestrito (*goto*) para retornar ao laço de repetição com a variável de comparação igual a 7, fazendo com que o laço voltasse a ser acionado a partir deste valor. O estilo de programação utilizado fere os princípios da programação estruturada, pois um desvio incondicional existente após o laço de impressão dos números transfere o fluxo de execução para dentro do corpo do laço. Essa abordagem é prejudicial para a legibilidade do programa e pode causar efeitos indesejados, os quais podem até afetar a correção do mesmo.

11. Algumas LPs (tal como, C) consideram a operação de atribuição como sendo uma espécie de expressão (isto é, a atribuição é uma operação que retorna um valor). Dê exemplos de situações nas quais isso pode ser vantajoso. Diga também quando essa característica pode ser danosa para a qualidade da programação. Justifique sua resposta.

Essa característica permite construir código com maior redigibilidade em algumas situações, tal como no trecho seguinte:

```
void stringcopy (char* p, char* q) {
  while (*p++ = *q++);
}
```

A função *stringcopy* possibilita que seja feita a cópia de uma string. Note que o fato do comando de atribuição retornar um valor possibilita que ele seja colocado dentro da expressão de parada do *while*. Quando for atribuído o caractere nulo, a repetição se encerrará.

Essa mesma característica pode reduzir a confiabilidade dos programas em situações similares, tal como a apresentada a seguir:

```
while (i = 10) {
```

Nesse caso, o programador esqueceu de um símbolo = da operação de igualdade. Como consequência, a expressão de parada do *while* se torna uma atribuição, a qual sempre retornará um valor distinto de zero, fazendo com que esse laço nunca termine. Esse tipo de erro não será identificado pelo compilador, reduzindo portanto a confiabilidade do programa.