Capítulo 4: Condicionais

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

1 Tomada de decisão

Nos capítulos anteriores, foram apresentados programas nos quais o fluxo de execução seguia uma sequência linear de instruções. Todas as instruções eram executadas, uma após a outra. A chamada de uma função transferia a execução para uma outra função, mas dentro do corpo de cada função as instruções eram executadas na ordem em que foram codificadas. Em geral, precisamos ter maior controle na sequência de instruções que devem ser executadas. Muitas vezes, por exemplo, a execução de determinada instrução deve estar condicionada à satisfação de um determinado critério. É fundamental que seja possível tomar diferentes decisões baseado em condições que são avaliadas em tempo de execução. Para tanto, as linguagens de programação oferecem construções para codificar *tomadas de decisão*.

Na linguagem C, tomadas de decisão são construídas através do comando `if`. O comando `if` avalia uma *expressão booleana* e re-direciona o fluxo de execução baseado no resultado avaliado (falso ou verdadeiro). Na sua forma mais simples, toma-se a decisão de executar ou não um bloco de comandos. Se a expressão booleana resultar em verdadeiro, o bloco de comandos é executado; se resultar em falso, o bloco de comandos não é executado. De qualquer forma, a execução continua com os comandos subsequentes ao bloco do `if`. A sintaxe desta forma mais simples do comando `if` é:

```
...
if ( _expressão_booleana_ ) {
   _bloco_de_comandos_
   ...
}
...
```

Um trecho de código que usa esta construção é ilustrado a seguir:

```
...
if (nota < 5.0) {
   printf("Reprovado");
}
...
```

Outra possível construção com o comando `if` acrescenta um segundo bloco de comandos, a ser executado apenas se a expressão booleana resultar em falso. Esta construção serve para programar tomadas de decisão exclusivas: executa-se um ou outro bloco de comandos. Se a expressão resultar em verdadeiro, o primeiro bloco de comandos é executado; se a expressão resultar em falso, o segundo bloco de comandos é executado. A sintaxe desta construção é ilustrada a seguir:

```
...
```

```
if ( _expressão_booleana_ ) {
   _bloco_de_comandos_1
   ...
}
else {
   _bloco_de_comandos_2
   ...
}
...
```

O primeiro bloco de comandos é conhecido como bloco de comandos `if` e o segundo como bloco de comandos `else`. Um trecho de código que usa esta construção é ilustrado a seguir:

```
...
if (nota < 5.0) {
   printf("Reprovado");
}
else {
   printf("Aprovado");
}
...
```

A sintaxe da linguagem C permite ainda que a simples codificação em sequência de comandos `if-else` resulte na construção de seleção exclusiva dentre múltiplas condições:

```
...
if ( _condição_1_ ) {
   _bloco_de_comandos_1
   ...
}
else if ( _condição_2_ ) {
   _bloco_de_comandos_2
   ...
}
else if ( _condição_3_ ) {
   _bloco_de_comandos_3
   ...
}
...
```

Um trecho de código que usa esta construção é ilustrado a seguir:

```
...
if (nota < 3.0) {
   printf("Reprovado");
}
else if (nota >= 5.0) {
   printf("Aprovado");
}
else {
   printf("Em prova final");
}
...
```

Nestas construções, se a expressão booleana correspondente à primeira condição resultar em *verdadeiro*, apenas o primeiro bloco de comandos é executado, e as outras condições não são

sequer avaliadas. Senão, se a expressão da segunda condição resultar em *verdadeiro*, apenas o segundo bloco de comandos é executado, e assim por diante.

2 Expressões booleanas

Uma expressão booleana é uma expressão que, quando avaliada, resulta no valor *falso* ou *verdadeiro*. A linguagem C não tem um tipo de dado específico para armazenar valores booleanos. Outras linguagens de programação, como C++, definem o tipo `bool` que só pode assumir os valores `false` ou `true`. Em C, o valor booleano é representado por um valor inteiro: 0 significa *falso* e qualquer outro valor diferente de zero significa *verdadeiro*. Em geral, usa-se 1 para representar o valor verdadeiro, e qualquer expressão booleana que resulta em verdadeiro resulta no valor 1.

Uma expressão booleana é construída através da utilização de *operadores relacionais*. Em C, os operadores relacionais são: maior que (`>`), menor que (`<`), maior ou igual a (`>=`), menor ou igual a (`<=`), diferente de (`!=`), igual a (`==`). Todos estes operadores comparam dois operandos, resultando no valor 0 (falso) ou 1 (verdadeiro).

Expressões booleanas também podem ser formadas com *operadores lógicos*. Operadores lógicos combinam expressões ou valores booleanos, resultando num valor booleano (0 ou 1). Em C, os operadores lógicos são: negação (`!`), conjunção (`&&`) e disjunção (`||`). O operador de negação é unário e atua sobre um único operando. Os outros dois operadores operam sobre dois operandos. Os resultados de operações lógicas de conjunção (AND) são apresentadas a seguir:

- $true$ `&&` $true \longrightarrow true$
- $true$ `&&` $false \longrightarrow false$
- $false$ `&&` $true \longrightarrow false$
- $false$ `&&` $false \longrightarrow false$

ou seja, basta um dos operandos ser falso para o resultado ser falso. O trecho de código a seguir ilustra o uso do operador de conjunção (AND). Neste caso, se um dos critérios da condição falhar, a expressão resulta em falso:

```
...
if (media >= 5.0 && nota1 >= 3.0 && nota2 >=3.0 && nota3 >= 3.0)
{
   printf("Aprovado");
}
...
```

Os resultados de operações lógicas de disjunção (OR) são:

- $true$ `||` $true \longrightarrow true$
- $true$ `||` $false \longrightarrow true$
- $false$ `||` $true \longrightarrow true$
- $false$ `||` $false \longrightarrow false$

ou seja, basta um dos operandos ser verdadeiro para o resultado ser verdadeiro. O trecho de código a seguir ilustra o uso do operador de disjunção (OR). Se um dos critérios for satisfeito, a expressão resulta em verdadeiro:

```
...
if (media < 5.0 || nota1 < 3.0 || nota2 < 3.0 || nota3 < 3.0)
{
   printf("Em prova final");
}
...
```

Por fim, os resultados de operações lógicas de negação (NOT) são:

- ! $true \longrightarrow false$
- ! $false \longrightarrow true$

O trecho de código a seguir ilustra o uso do operador de negação (NOT):

```
...
if ( !(media < 5.0 || nota1 < 3.0 || nota2 < 3.0 || nota3 < 3.0) )
{
   printf("Aprovado");
}
...
```

Para melhor exemplificar o uso de operações booleanas e construções com o comando `if`, vamos considerar como exemplo um programa para converter o critério de avaliação de alunos em escolas brasileiras para o critério utilizado em escolas americanas. Nas escolas brasileiras, a avaliação dos alunos é reportada por uma nota que varia de 0 a 10. Nas escolas americanas, a avaliação dos alunos é feita por conceito: A, B, C, D, ou F. Podemos assumir a seguinte equivalência entre os sistemas de avaliação: A (9.0 a 10.0), B (8.0 a 8.9), C (7.0 a 7.9), D (5.0 a 6.9), e F (< 5.0). Vamos então considerar um programa que, dada um nota segundo o critério brasileiro, fornecido via teclado, exiba na tela o conceito correspondente segundo o critério americano. Uma possível implementação deste programa que só usa a forma mais simples do comando `if` é apresentada a seguir.

```
#include <stdio.h>

int main (void)
{
   float nota;
   printf("Entre com nota: ");
   scanf("%f",&nota);

   if (nota >= 9.0) {
      printf("A");
   }
   if (nota >= 8.0 && nota < 9.0) {
      printf("B");
   }
   if (nota >= 7.0 && nota < 8.0) {
      printf("C");
   }
   if (nota >= 5.0 && nota < 7.0) {
      printf("D");
   }
   if (nota < 5.0) {
      printf("F");
   }
```

```
    return 0;
}
```

Este programa, que usa apenas a forma mais simples do comando `if`, funciona da maneira esperada, reportando o resultado correto, mas poderia ser codificado de uma maneira mais estruturada e mais eficiente computacionalmente. Do jeito que está, todas as condições dos comandos `if`s são avaliadas. Assim, mesmo que o usuário entre com um valor maior do que 9.0, após imprimir o conceito A, as demais condições serão avaliadas (e logicamente resultarão em falso). Como se trata de uma seleção exclusiva, o mesmo código pode ser melhor estruturado com construções `if-else` da seguinte forma:

```
#include <stdio.h>

int main (void)
{
    float nota;
    printf("Entre com nota: ");
    scanf("%f",&nota);

    if (nota >= 9.0) {
        printf("A");
    }
    else if (nota >= 8.0) {
        printf("B");
    }
    else if (nota >= 7.0) {
        printf("C");
    }
    else if (nota >= 5.0) {
        printf("D");
    }
    else {
        printf("F");
    }

    return 0;
}
```

Neste programa, se a primeira condição falhar, isto é, se a nota fornecida não for maior ou igual a 9.0, só testamos se ela é maior ou igual a 8.0 para concluir que o conceito correspondente é o B, pois já sabemos que a nota é menor que 9.0. Este mesmo raciocínio é aplicado nos demais testes. Se todos falharem, é exibido o conceito F.

3 Blocos de comandos

Na linguagem C, podemos agrupar comandos em blocos, envolvendo-os com abre e fecha chaves (`{...}`), como fizemos para delimitar o bloco de comando `if` e o bloco de comandos `else` nas construções para tomada de decisões. Na verdade, podemos criar blocos de comandos em qualquer ponto do programa, bastando envolver comandos com chaves. Uma variável declarada dentro de um bloco existe enquanto os comandos do bloco estiverem sendo executados. Quando o bloco chega ao fim, as variáveis declaradas dentro dele deixam de existir. Segundo o padrão C89 da linguagem C, uma variável só pode ser declarada no início de um bloco de comandos[1]

[1] Já no padrão C99, variáveis locais podem ser declaradas em qualquer ponto do programa.

Isso inclui o início do corpo das funções (que também são blocos) e qualquer outro bloco criado dentro da função.

Nas construções do comando `if`, os blocos são importantes para identificar o conjunto de comandos cuja execução está submetida à avaliação da expressão booleana. No entanto, se um "bloco de comandos" for constituído por apenas um único comando, as chaves podem ser omitidas. Desta forma, podemos re-escrever o código que determina o conceito correspondente a uma nota como mostrado a seguir:

```
#include <stdio.h>

int main (void)
{
   float nota;
   printf("Entre com nota: ");
   scanf("%f",&nota);

   if (nota >= 9.0)
      printf("A");
   else if (nota >= 8.0)
      printf("B");
   else if (nota >= 7.0)
      printf("C");
   else if (nota >= 5.0)
      printf("D");
   else
      printf("F");

   return 0;
}
```

Uma prática comum (e boa) usada em codificação de programas em C é escrever os blocos de comandos com margem à esquerda maior que o comando que o precede. Cada vez que abrimos um bloco de comandos, procuramos escrever os comandos deste bloco em linhas que tem um maior número de espaços (ou de tabulações) no início. Chamamos isto de *indentação*. Com uma indentação apropriada, fica fácil identificar visualmente os blocos de comandos.

4 Exemplos

Para ilustrar o uso de construções para tomada de decisão, vamos considerar alguns exemplos.

4.1 Raízes de equação do segundo grau

Como primeiro exemplo, vamos discutir a construção de um programa para calcular as raízes de uma equação do segundo grau. Sabemos que as raízes de uma equação na forma $ax^2+bx+c=0$ são dadas por:

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

A idéia é construirmos um programa que capture os coeficientes da equação, fornecidos pelo usuário via teclado, e exiba na tela os valores das raízes.

Este seria um problema de codificação direta de uma expressão matemática se não fosse pelo fato das raízes poderem não existir. Na verdade, a raiz quadrada só é definida para valores positivos. Se, dentro de um programa, tentarmos avaliar uma expressão matemática cujo resultado é indefinido, o resultado do programa certamente não será o desejado. Isto inclui ações como tentar

extrair a raiz quadrada de um número negativo, calcular o logaritmo de um número negativo, ou mesmo fazer uma divisão por zero. Por este motivo, devemos avaliar estas expressões apenas após certificarmos que os operandos são válidos.

Para o problema em questão, podemos identificar, na saída do programa exibida na tela, o número de raízes distintas reais (zero, uma ou duas). Uma possível implementação deste programa é mostrada a seguir. Note que para extrair a raiz quadrada, usamos a função `sqrt` definida na biblioteca matemática padrão de C, cuja interface é incluída. Note ainda que para evitar divisão por zero, testamos se o valor do coeficiente a fornecido é diferente de zero. Se for igual a zero, exibimos uma mensagem na tela e retornamos, finalizando a função e, por conseguinte, o programa. Este programa trabalha com números reais de dupla precisão (tipo `double`).

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

int main (void)
{
   double a, b, c;  /* coeficientes */
   double x1, x2;   /* raízes */
   double delta;

   printf("Entre com os coeficientes (a b c):");
   scanf("%lf", &a);
   scanf("%lf", &b);
   scanf("%lf", &c);

   if (a == 0.0) {
      printf("Valor de 'a' nao pode ser zero.");
      return 1;
   }

   delta = b*b - 4*a*c;
   if (delta < 0) {
      printf("Raizes reais inexistentes.");
   }
   else if (delta == 0.0) {
      x1 = -b / (2*a);
      printf("Uma raiz real: %f", x1);
   }
   else {
      delta = sqrt(delta);
      x1 = (-b + delta) / (2*a);
      x2 = (-b - delta) / (2*a);
      printf("Duas raizes reais: %f e %f", x1, x2);
   }

   return 0;
}
```

Este mesmo programa funcionaria com reais de precisão simples (tipo `float`); provavelmente, a precisão dos resultados já seria plenamente satisfatória. As funções da biblioteca matemática são em geral definidas usando o tipo com maior precisão, justamente para poder atender às aplicações que necessitam trabalhar com precisão dupla. Assim, o protótipo da função `sqrt`, por exemplo, é definido como `double sqrt (double x);`. Isto é, o valor de entrada esperado é um real de dupla precisão e o valor de retorno também é de dupla precisão. Isso, no entanto, não impede chamarmos a função passando para ela um valor do tipo `float`: o valor passado será

convertido para o valor correspondente de dupla precisão (logicamente, não há nenhuma perda nesta conversão). Também podemos atribuir o valor retornado a uma variável do tipo `float`: o valor é convertido para o valor correspondente na precisão simples (neste caso, há uma perda de precisão, em geral insignificante, uma vez que a aplicação em si optou por trabalhar com tipo de precisão simples). Conforme mencionado, alguns compiladores reportam uma advertência quando há perda de precisão na conversão entre tipos. Se quisermos evitar esta eventual mensagem de advertência, basta usarmos o operador de conversão explícita de tipo. Supondo, por exemplo, que `x` e `y` são variáveis do tipo `float`, podemos invocar a função `sqrt` da seguinte forma: `y = (float) sqrt(x)`.

4.2 Cálculo de volumes

Como último exemplo, vamos retomar o problema de calcular volumes de objetos geométricos. Ao invés de fazer um programa específico para cada tipo de objeto, vamos construir um programa que permita calcular o volume de vários tipos de objetos diferentes. A idéia é apresentar um menu para o usuário com os tipos de objetos suportados. O usuário então escolhe a opção desejada, entra com os dados correspondentes e o programa exibe o volume computado.

Para este nosso exemplo, vamos considerar o cálculo de volume dos seguintes objetos geométricos:

- caixa de lados a, b e c: abc
- esfera de raio r: $\frac{4}{3}\pi r^3$
- cilindro de raio r e altura h: $\pi r^2 h$
- cone de raio r e altura h: $\frac{1}{3}\pi r^2 h$

A função *main* deste programa pode ser definida por:

```
#include <stdio.h>

int main (void)
{
   int escolha;

   /* exibe menu na tela */
   printf("Escolha uma opcao:\n");
   printf("1 - Caixa\n");
   printf("2 - Esfera\n");
   printf("3 - Cilindro\n");
   printf("4 - Cone\n");

   /* lê opção escolhida e chama função correspondente */
   scanf("%d", &escolha);
   if (escolha == 1) {
      calcula_caixa();
   }
   else if (escolha == 2) {
      calcula_esfera();
   }
   else if (escolha == 3) {
      calcula_cilindro();
   }
   else if (escolha == 4) {
      calcula_cone();
   }
   else {
```

```
        printf("Opcao invalida.");
    }
    return 0;
}
```

As funções auxiliares seguem uma mesma estrutura. Inicialmente, os valores das dimensões dos objetos são capturados. Em seguida, o volume é calculado e o resultado é exibido na tela. Para ilustrar, apresentamos abaixo a função relativa ao cálculo do volume de um cone. As demais ficam como exercício. Note que estas funções não recebem nem retornam valores, pois capturam os dados de entrada fornecidos via teclado e exibem o resultado na tela.

```
#define PI 3.14159

void calcula_cone (void)
{
    float r, h;
     printf("Entre com o raio e altura do cone: ");
     scanf("%f", &r);
     scanf("%f", &h);
     printf("Volume calculado: %f", PI*r*r*h/3.0);
}
```

4.3 Jogo de par ou ímpar (Opcional)

Neste exemplo, vamos criar um jogo muito simples, a fim de introduzir uma discussão relacionada com o desenvolvimento de qualquer jogo de computador: como fazer o computador apresentar um comportamento aleatório ou, mais precisamente, apresentar um comportamento aparentemente aleatório. Qualquer jogo de computador, para se tornar interessante, deve introduzir um fator de aleatoriedade. O fator de aleatoriedade pode ser aplicado, por exemplo, para determinar quando e como um adversário controlado pelo próprio computador age. Com isso, tornamos o jogo mais dinâmico, menos previsível. A construção de jogos sofisticados que usam algoritmos de inteligência artificial naturalmente foge do escopo deste texto, mas podemos criar comportamentos aleatórios de forma bastante simples. Vamos considerar o exemplo de construir um "jogo" de par ou ímpar.

O objetivo do programa que queremos construir é simular um desafio de par ou ímpar: o usuário *versus* o computador. Na execução deste jogo, o usuário definiria sua escolha, par ou ímpar, e em seguida forneceria um número inteiro. O computador geraria de forma aleatória um outro número inteiro e avaliaria se o resultado da soma dos dois números é par ou ímpar, reportando o ganhador do desafio.

A novidade deste exemplo é a geração aleatória (ou randômica) de um número. Para esta implementação, vamos considerar que o computador gerará um número aleatório entre 0 e 9. A construção de um algoritmo para a geração de uma seqüência de números aleatória pode ser muito complicada, mas para muitas aplicações é suficiente usar a função `rand` da biblioteca padrão da linguagem C (*stdlib.h*). A função `rand` não recebe parâmetro de entrada e retorna um número aleatório entre 0 e `RAND_MAX`, onde `RAND_MAX` representa uma constante simbólica também definida pela biblioteca padrão.

Para adaptar o uso da função da biblioteca padrão para o nosso exemplo, onde queremos números apenas entre 0 e 9, podemos criar uma função auxiliar que usa a função da biblioteca mas retorna um número entre 0 e *n - 1*, onde *n* é um valor especificado como parâmetro:

```c
#include <stdio.h>

int main (void)
{
   printf("%d",aleatotio(10));
   printf("%d",aleatorio(10));
   printf("%d",aleatorio(10));
   printf("%d",aleatorio(10));
   return 0;
}
```

Se executado, este programa de teste deve exibir na tela 4 valores gerados aleatoriamente. No entanto, se re-executarmos este mesmo programa outras vezes, vamos verificar que a sequência de números aleatórios é sempre a mesma. Isto porque o algoritmo para geração de números aleatórios da biblioteca padrão gera uma sequência de números, distribuídos uniformemente no intervalo definido, baseado num valor *semente*. A semente de um gerador de números aleatórios define uma determinada sequência de números a ser gerada. Se queremos gerar diferentes sequências, devemos passar para o gerador diferentes sementes. A função srand da biblioteca padrão deve ser usada para definir uma semente. Esta função recebe como parâmetro um número inteiro, representando a semente a ser usada. Logicamente, para se ter uma sequência diferente a cada execução do programa, devemos fornecer sementes diferentes. O valor da semente também não pode ser fornecido pelo usuário, pois eliminaria o efeito aleatório que queremos simular. Um truque simples para resolver esta questão é usar o valor retornado pela função time, definida na biblioteca padrão `time.h`. A função time tem como valor de retorno o número total de segundos decorridos desde $1.^{o}$ de janeiro de 1970. Esta função espera um parâmetro de entrada, mas como estamos interessados apenas no valor de retorno da função, podemos especificar a constante simbólica `NULL`, também definida na biblioteca padrão. Podemos então criar outra função auxiliar para inicializar nosso gerador de número aleatório. Nesta função, além de especificar uma semente, descartamos o primeiro número da sequência (isto porque é comum a função `rand`, que tem suas limitações, repetir o mesmo primeiro número, mesmo com sementes diferentes):

```c
#include <stdio.h>

void semente ( void)
{
  srand(time(NULL));
  rand();
}
```

Agora, no código completo do programa, devemos lembrar de incluir a biblioteca `time.h`, pois estamos usando uma função definida nela. Podemos então re-escrever nosso teste, e o resultado deve ser agora diferente a cada execução (desde que o intervalo entre execuções sucessivas seja maior que 1 segundo):

```c
#include <stdio.h>

int main (void)
{
  semente();
  printf("%d",aleatotio(10));
  printf("%d",aleatorio(10));
  printf("%d",aleatorio(10));
  printf("%d",aleatorio(10));
  return 0;
```

```
}
```

Temos agora as funções auxiliares necessárias para a implementação do nosso jogo de par ou ímpar. Para o usuário escolher entre par ou ímpar, podemos pedir que seja fornecido um valor numérico: 0 significando par e 1 significando ímpar. Em seguida, o usuário fornece o número dele, o computador gera um número aleatório e a soma dos dois é verificada: se o resto da divisão da soma por 2 for 0, o número é par; se for 1, o número é ímpar. Este valor é comparado com a escolha do usuário para definir o ganhador. A função `main` que implementa este jogo pode ser dada por:

```
#include <stdio.h>

int main (void)
{
   int escolha;      /* armazena a escolha do usuário: 0=par, 1=ímpar */
   int usuario;      /* número fornecido pelo usuário */
   int computador;   /* número gerado pelo computador */

   printf("Entre com sua escolha: 0 (par) ou 1 (impar): ");
   scanf("%d", &escolha);
   printf("Entre com seu numero: ");
   scanf("%d", &usuario);

   semente();
   computador = aleatorio(10);
   printf("Computador: %d \n", computador); /* exibe valor gerado pelo computador */

   /* testa se soma é par ou ímpar, comparando com escolha do usuário */
   if (((usuario+computador) % 2) == escolha)
      printf("Usuario ganhou!");
   else
      printf("Computador ganhou!");

   return 0;
}
```

Exercícios

1. Considere uma disciplina que adota o seguinte critério de aprovação: os alunos fazem duas provas (P1 e P2) iniciais; se a média nessas duas provas for maior ou igual a 5.0, e se nenhuma das duas notas for inferior a 3.0, o aluno passa direto. Caso contrário, o aluno faz uma terceira prova (P3) e a média é calculada considerando-se essa terceira nota e a maior das notas entre P1 e P2. Neste caso, o aluno é aprovado se a média final for maior ou igual a 5.0.

   Escreva um programa completo que leia inicialmente as duas notas de um aluno, fornecidas pelo usuário via teclado. Se as notas não forem suficiente para o aluno passar direto, o programa deve capturar a nota da terceira prova, também fornecida via o teclado. Como saída, o programa deve imprimir a média final do aluno, seguida da mensagem "Aprovado" ou "Reprovado", conforme o critério descrito acima.

2. Escreva um programa para fazer conversões entre diferentes unidades. As opções do programa devem ser exibidas em forma de um menu apresentado na tela, em dois níveis. No primeiro nível o usuário escolhe a classe de unidade; no segundo nível o usuário escolhe a conversão que deseja, fornecendo então o valor a ser convertido. Por fim, o programa exibe o valor resultante na tela. As opções apresentadas no menu podem ser:

   1. Peso
      1. Libra → Quilograma
      2. Quilograma → Libra
      3. Onça → Grama
      4. Grama → Onça

   2. Volume
      1. Galão → Litro
      2. Litro → Galão
      3. Onça → Mililitro
      4. Mililitro → Onça

   3. Comprimento
      1. Milha → Quilômetro
      2. Quilômetro → Milha
      3. Jardas → Metro
      4. Metro → Jardas

   Sabe-se que 1 libra equivale a $0.4536Kg$, 1 onça a $28.3495g$, 1 galão a $3.7854l$, 1 onça fluido a $29.5735ml$, 1 milha a $1.6093Km$ e 1 jarda a $0.9144m$.

3. (Opcional) Modifique o programa do jogo "par ou ímpar" apresentado para que, em vez do usuário, o computador escolha, de forma aleatória, se quer par ou ímpar. O usuário apenas fornece seu número.

4. (Opcional) Escreva um programa que implemente o jogo conhecido como *pedra, papel, tesoura*. Neste jogo, o usuário e o computador escolhem entre *pedra*, *papel* ou *tesoura*. Sabendo que *pedra* ganha de *tesoura*, *papel* ganha de *pedra* e *tesoura* ganha de *papel*, exiba na tela o ganhador: usuário ou computador. Para esta implementação, assuma que o número 0 representa *pedra*, 1 representa *papel* e 2 representa *tesoura*.