Capítulo 5: Repetições

Waldemar Celes e Roberto Ierusalimschy

29 de Fevereiro de 2012

1 Construção de laços

Uma das principais características de um computador é sua capacidade para realizar cálculo e manipular informações de forma extremamente eficiente. Quando programamos um computador, tiramos proveito deste alto poder de processamento. Diversos problemas de difícil solução podem ser resolvidos numericamente por um computador se dividido em partes. Acumulando o resultado de pequenas computações, podemos chegar à solução do problema como um todo. Para tanto, precisamos de mecanismos de programação que nos permitam requisitar que um conjunto de instruções seja repetidamente executado, até que uma determinada condição seja alcançada. A esse tipo de construção damos o nome de *repetição* e, como veremos, diversos problemas podem ser resolvidos com esta estratégia de programação.

Repetições são programadas através da construção de laços (ou ciclos). Na linguagem C, podemos construir um laço através do comando `while`. O comando `while` repete a execução de um bloco de comandos *enquanto* uma determinada condição for satisfeita. Quando a condição não é (ou passa a não ser) satisfeita, a execução do programa continua nos comandos subsequentes ao bloco do `while`. A sintaxe deste comando é:

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

Enquanto a expressão booleana resultar em *verdadeiro*, o bloco de comandos é executado; quando a expressão booleana resulta em *falso*, a execução continua nos comandos subsequentes.

Para ilustrar o uso do comando `while`, vamos retomar o clássico exemplo do cálculo do fatorial de um número. O fatorial de um número inteiro não negativo é definido pelo produtório a seguir:

$$n! = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

O fatorial de zero, consequentemente, resulta no valor neutro da multiplicação que vale 1, pois teríamos um produtório vazio.

Queremos escrever uma função que calcule o valor do fatorial de um número. Esta função receberá como parâmetro o valor do número que se deseja calcular o fatorial. Como o fatorial só é definido para números inteiros, este parâmetro é do tipo inteiro e a função tem como retorno também um valor inteiro, representando o valor do fatorial calculado.

Assumindo que o número passado como parâmetro não é negativo, a codificação desta função é simples. O produtório é calculado de forma iterativa, acumulando multiplicações simples. O valor do produtório deve ser acumulado em uma variável inicializada com 1, que é o valor neutro

da multiplicação (se tivéssemos calculando um somatório, estaríamos acumulando adições e o valor neutro seria o 0). Uma função que calcula o fatorial pode ser como mostrada a seguir:

```
int fatorial (int n)
{
   int f = 1;
   while (n > 1) {
      f = f * n;
      n = n - 1;
   }
   return f;
}
```

Quando o bloco do comando `while` for executado pela primeira vez, o valor de `n` será armazenado em `f(f = 1 * n)` e o valor de `n` será decrementado em uma unidade. Na segunda passada, o valor de `n*(n-1)` será armazenado em `f`, e assim por diante, até que o valor de `n` seja igual a 1. Neste ponto, o valor `n*(n-1)*(n-2)*...*3*2` estará armazenado em `f`, representando o valor do fatorial. Podemos verificar que o valor retornado continua correto mesmo para `n` valendo zero. Isto porque a variável `f` é inicializada com o valor 1 e o bloco do comando `while` não será executado nem uma vez, já que a expressão booleana `n > 1` resultará em falso logo no início.

Para testar nossa função, podemos escrever uma função *main* que captura um valor inteiro fornecido pelo usuário via teclado e exibe o valor do fatorial correspondente. Esta função pode ser codificada como a seguir:

```
int main (void)
{
   int x, fat;
   printf("Entre com um numero: ");
   scanf("%d", &x);
   fat = fatorial(x);
   printf("Fatorial = %d", fat);
   return 0;
}
```

A linguagem C também permite a construção de laços com o comando `for`, que é equivalente ao comando `while`. A sintaxe do comando `for` é mais concisa, mais compacta. A princípio, o comando `for` parece não ser natural, mas programadores rapidamente se acostumam com ele, sendo a forma mais utilizada para construção de laços em programas escritos em C. Na maioria das vezes, uma construção de laço tem associadas uma expressão inicial e uma expressão de atualização, além da expressão booleana que determina a execução ou não do bloco de comandos. Para ilustrar a discussão, vamos considerar um programa simples que exibe na tela os valores de 0 a 99. Com o comando `while`, este programa pode ser codificado como a seguir:

```
#include <stdio.h>

int main (void)
{
   int x = 0;
   while (x < 100) {
      printf("%d\n",x);
      x = x + 1;
   }
   return 0;
}
```

Na construção deste laço, identificamos três expressões que o "controlam":

- uma expressão inicial (`x = 0`), avaliada antes da execução do bloco de comandos associado ao `while`;
- uma expressão booleana (`x < 100`), avaliada antes de cada execução do bloco, responsável por determinar se o bloco deve ou não ser executado;
- uma expressão de atualização (`x = x + 1`), avaliada no final de cada execução do bloco.

O comando `for` agrupa estas três expressões e constrói um laço equivalente. A sintaxe do comando `for` é:

```
...
for (_expr_inicial_; _expr_booleana_; _expr_de_atualização_) {
   _bloco_de_comandos_
   ...
}
...
```

O programa que imprime os valores de 0 a 99 pode ser escrito com o comando `for`:

```
#include <stdio.h>

int main (void)
{
   int x;
   for (x=0; x<100; x=x+1) {
      printf("%d\n",x);
   }
   return 0;
}
```

Na prática, uma das vantagens do comando `for` é que escrevemos a expressão de atualização logo no início da construção. Com o comando `while`, muitas vezes o programador acaba esquecendo de escrever a expressão de atualização, criando um *laço infinito* (isto é, um laço que fica continuamente sendo executado), resultando num programa inválido cuja execução nunca termina.

A função que faz o cálculo do fatorial pode ser re-escrita, usando o comando `for`. Uma possível codificação é mostrada a seguir:

```
int fatorial (int n)
{
   int i;
   int f = 1;
   for (i=2; i<=n; i=i+1) {
      f = f * i;
   }
   return f;
}
```

Nesta solução, o valor do fatorial é avaliado como sendo `1*2*3*...*(n-1)*n`, resultando no valor correto. Note também que se `n` for igual a zero ou um, o bloco do `for` não será executado e o valor retornado será 1, conforme esperado.

Em construções de laços, é muito comum a expressão de atualização incrementar (ou decrementar) o valor de uma variável em uma unidade. Para estes casos, a linguagem C oferece

operadores de incremento (e decremento) que podem ser usados. Podemos substituir a expressão `i = i+1` por uma que usa o operador de incremento: `i++` ou `++i`. De forma similar, se tivéssemos a expressão `i = i-1`, poderíamos substituir por `i--` ou `--i` [1].

2 Algoritmos com repetições

Um *algoritmo* é um procedimento computacional, definido por um conjunto de instruções, elaborado para realizar determinada tarefa. Dado um estado inicial (dados de entrada), um algoritmo correto deve produzir o resultado esperado (dados de saída) num tempo finito. Um algoritmo é descrito por passos que podem ser implementados num computador e, em geral, envolve repetições e tomadas de decisão. Algoritmos mais complexos podem ser elaborados pela composição de algoritmos mais simples.

Para exemplificar o conceito e a implementação de um algoritmo em C, vamos considerar o problema de determinar o *máximo divisor comum* (MDC) de dois números. O valor do máximo divisor comum de dois números inteiros positivos, MDC(x,y), pode ser calculado usando o algoritmo de Euclides. Este algoritmo é baseado no fato de que se o resto da divisão de x por y, representado por r, for igual a zero, y é o MDC. Se o resto r for diferente de zero, o MDC de x e y é igual ao MDC de y e r. O processo se repete até que o valor do resto da divisão seja zero, o que garantidamente irá acontecer pois, no caso extremo, chegaremos ao valor do MDC de um valor n e 1, que vale 1.

Conforme descrito, o algoritmo se baseia em computações simples executadas repetidas vezes. Naturalmente, teremos que implementar a repetição construindo um laço, mas antes de pensarmos na implementação do algoritmo, vamos entender como ele funciona. Vamos tomar como exemplo o cálculo do MDC de 42 e 24. Seguindo o algoritmo, $x = 42$ e $y = 24$, no primeiro passo calcula-se o valor r, resto da divisão entre x e y. No caso, o valor de r fica sendo 18. Como r não é zero, repetimos o mesmo procedimento com $x = 24$ e $y = 18$. A tabela abaixo mostra os valores computados em cada passo (iteração), até que o valor de r seja zero.

Passo	x	y	r
1	42	24	18
2	24	18	6
3	18	$\underline{6}$	0

Neste exemplo, como no terceiro passo o valor do resto da divisão alcançou zero, tem-se que a solução do problema é o valor associado a y neste passo, isto é, 6. Note que a cada passo, os valores de x e y passam a ser, respectivamente, os valor de y e r do passo anterior. Podemos verificar, que no caso extremo, onde o MDC é igual a 1, o algoritmo também funciona. A tabela a seguir ilustra os passos para $x = 42$ e $y = 23$, resultando no valor 1.

Passo	x	y	r
1	42	23	19
2	23	19	4
3	19	4	3
4	4	3	1
5	3	$\underline{1}$	0

Devemos notar ainda que não é necessário associar o maior número a x no primeiro passo. Se inicialmente o valor de x for menor que o valor de y, o algoritmo automaticamente corrige-os no

[1] Existe uma diferença entre $i{+}{+}$ e $+{+}i$ (ou entre i-- e --i), mas quando a expressão consiste apenas no incremento (ou decremento) as duas formas se equivalem. Sem entrar em detalhes, quando o incremento (ou decremento) faz parte de uma expressão maior, a forma $i{+}{+}$ usa primeiro o valor da variável i para depois incrementar, e a forma $+{+}i$ incrementa e já usa o valor da variável i incrementada.

primeiro passo, pois o resto da divisão de um número menor por um número maior será sempre o número menor. A tabela a seguir repete o algoritmo para os valores 24 e 42, associando x ao menor deles inicialmente. Note a correção da ordem no primeiro passo.

Passo	x	y	r
1	24	42	24
2	42	24	18
3	24	18	6
4	18	6	0

Podemos então pensar na implementação deste algoritmo em C. Para tanto, vamos usar o comando `while` para construir o laço de repetição. O algoritmo pode ser implementado por uma função que recebe como parâmetros os dois números inteiros (`x` e `y`) e tem como valor de retorno também um número inteiro representando o MDC:

```
int mdc (int x, int y)
{
   int r = x%y;
   while (r != 0) {
      x = y;
      y = r;
      r = x%y;
   }
   return y;
}
```

Nesta função, inicialmente calcula-se o valor do resto da divisão, usando o operador módulo (`%`). Em seguida, executa-se o laço enquanto o valor do resto for diferente de zero. Dentro do laço, atualiza-se os valores de `x`, `y` e `r`. Quando o valor do resto alcançar zero, o último valor atribuído a `y` representa a solução, e é então retornado. Note que a ordem das atribuições dentro do laço é fundamental: devemos atualizar o valor de `x` *antes* de atualizar o valor de `y`; caso contrário, `x` receberia o valor de `y` já atualizado. O mesmo vale para `r`: só pode ser atualizado após atualizar `y`. Fica como exercício a adaptação desta implementação para usar o comando `for`. Fica também como exercício a codificação de uma função *main* para testar a implementação do cálculo do MDC: pode-se capturar dois valores inteiros fornecidos pelo usuário via teclado e exibir o MDC computado.

Como segundo exemplo de um algoritmo, vamos considerar o problema de determinar se um dado número inteiro positivo, n, é ou não um *número primo*. Como se sabe, um número é dito primo se for divisível apenas pelo número 1 e pelo próprio número, sendo que o número 1 não é primo.

Para resolver este problema, podemos primeiramente pensar num algoritmo muito simples, que testa todas as possibilidades. Se o número fornecido for maior que 1, podemos testar se o número é divisível por algum outro número entre 2 e $n-1$. Para testar se o número é divisível, basta verificar o valor do resto da divisão. Podemos escrever uma função que implementa este algoritmo. A função deve ter como valor de retorno um valor booleano: falso, se o número fornecido não for primo; verdadeiro, se o número for primo. Em C, fazemos a função ter como retorno um valor inteiro: 0 para falso e 1 para verdadeiro. Uma possível implementação deste algoritmo é apresentada a seguir:

```
int primo (int n)
{
   int i;
   if (n < 2) {
      return 0;
```

```
    }
    for (i=2; i<n; i++) {
        if (n%i == 0) {
            return 0;
        }
    }
    return 1;
}
```

Este algoritmo segue um padrão muito comum em programação. Após testar o caso básico (`n < 2`), o algoritmo busca a existência de uma determinada condição (`n%i == 0`). Caso a condição seja satisfeita, tem-se a resposta do problema e a execução da função pode ser interrompida (`return 0`). Se a condição não for encontrada, *após testar todas as possibilidades*, isto é, se o laço executar todos os passos previstos, conclui-se que o número é primo (`return 1`).

Note que a função também está correta para n igual a 2, pois o bloco do `for` não será executado nem uma vez, retornando que o número é primo.

Podemos facilmente perceber que o algoritmo apresentado, embora correto, não é muito eficiente. Em programação de computadores, buscamos soluções eficientes, isto é, buscamos soluções que minimizam (ou diminuem) o número de instruções executadas para se encontrar a resposta do problema. No caso, estamos fazendo diversos testes desnecessários. Uma forma simples de aumentar a eficiência do nosso algoritmo é diminuir o número de potenciais divisores. Isto pode ser feito sabendo-se que: o único número primo par é o 2; e se um número não é primo, pelo menos um de seus divisores é menor ou igual a sua raiz quadrada. Assim, nosso algoritmo modificado pode ser codificado como mostrado a seguir:

```
int primo (int n)
{
    if (n == 2) {
        return 1;
    }
    else if (n<2 || (n%2)==0) {
        return 0;
    }
    else {
        int i;
        int lim = (int) sqrt(n);
        for (i=3; i<=lim; i=i+2) {
            if (n%i == 0) {
                return 0;
            }
        }
        return 1;
    }
}
```

A rigor, esta segunda versão da função só tende a ser mais eficiente que a primeira para valores de n grandes, pois temos um custo computacional adicional associado à avaliação da raiz quadrada.

3 Avaliação de séries

A possibilidade de construir laços de repetição possibilita o uso de computadores para a avaliação de séries. Uma série é representada por uma sequência de termos. Podemos facilmente construir funções que avaliam uma determinada série. Como primeiro exemplo de avaliação de série,

vamos considerar a *série de Fibonacci*. A série de Fibonacci é constituída dos seguintes valores:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \ldots$$

isto é, o primeiro termo vale 0, o segundo vale 1 e os termos subsequentes representam a soma dos dois termos anteriores.

Vamos considerar a implementação de um algoritmo que calcula o n-ésimo termo da série. Se o valor fornecido de n for 1 ou 2, temos a resposta de imediato, 0 ou 1, respectivamente. Caso o valor de n seja maior que 2, podemos calcular o resultado avaliando a série termo a termo, até que se alcance o número de termos especificado. A implementação desta função pode ser como ilustrada a seguir:

```
int fibonacci (int n)
{
   if (n <= 2) {
      return n-1;   /* retorna 0 ou 1, respectivamente */
   }
   else {
      int i;
      int a = 0;        /* primeiro termo */
      int b = 1;        /* segundo termo */
      int c;            /* termo corrente */
      for (i=3; i<=n; i++) {
         c = a + b;
         a = b;
         b = c;
      }
      return c;
   }
}
```

Como um exemplo adicional de avaliação de séries, vamos considerar um algoritmo para avaliar o valor de π. O valor do número π pode ser avaliado a partir da seguinte série:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}\ldots\right)$$

Nosso objetivo é implementar uma função para fornecer uma aproximação do valor de π, segundo a série apresentada. Nossa função pode receber como parâmetro o número de termos da série, n, que se deseja usar na avaliação. Para melhor visualizar a implementação da função, podemos re-escrever a série acima como um somatório:

$$\pi = 4\sum_{i=0}^{n-1} \frac{-1^i}{2i+1}$$

Agora, fica mais simples escrever a função, lembrando que um somatório é implementado por uma repetição onde se acumula adições. A variável que armazena o resultado do somatório deve então ser inicializada com o valor zero (valor neutro da adição). Para avaliar a expressão -1^i não é razoável usarmos a função de potenciação da biblioteca matemática (`pow`), pois podemos escrever uma função auxiliar mais eficiente (afinal, é necessário apenas avaliar se o expoente é par ou ímpar). Uma possível implementação desta função auxiliar e da função que calcula uma aproximação do valor de π são mostradas a seguir:

```
float pow_1 (int n)
{
   if (n%2 == 0)
```

```
        return  1.0;
    else
        return -1.0;
}

float valor_pi (int n)
{
    int i;
    float soma = 0.0;
    for (i=0; i<n; i++) {
        soma = soma + ( pow_1(i) / (2*i+1) );
    }
    return 4*soma;
}
```

4 Repetição com teste no final

Como vimos, os comandos `while` e `for` são equivalentes, e ambos avaliam a expressão booleana que controla a execução do bloco de comandos no *início* do laço. Se a expressão booleana resultar inicialmente em falso, o bloco de comandos não é executado nem uma vez. A linguagem C oferece ainda uma terceira construção de laços através do comando `do-while`. Neste caso, a expressão booleana é avaliada no *final* do laço. Isto significa que o bloco de comandos é executado pelo menos uma vez. A sintaxe do comando `do-while` é mostrada a seguir:

```
...
do {
    _bloco_de_comandos_
    ...
} while (_expressão_booleana_);
...
```

A construção de laços com teste no final é especialmente útil quando precisamos executar um passo da iteração antes de avaliar a condição de repetição. Esta situação ocorre, por exemplo, quando queremos garantir que o erro cometido numa aproximação está dentro de uma determinada tolerância. Para ilustrar o uso da construção `do-while`, vamos retomar o exemplo da avaliação do valor de π dada pela série apresentada anteriormente.

Queremos implementar uma função para avaliar o valor de π dentro de uma determinada tolerância numérica. Em vez de indicar explicitamente o número de termos da série que deve ser usado, a função pode receber o valor de uma tolerância numérica, e todos os termos da série com valores absolutos maiores que esta tolerância devem ser considerados. Assim que encontrarmos um termo cujo valor absoluto seja menor que a tolerância especificada, truncamos a série, isto é, não consideramos os termos subsequentes na avaliação de π.

Uma possível implementação desta segunda versão da função que avalia o valor de π é mostrada a seguir. Esta implementação faz uso da função que retorna o absoluto de um número, `fabs`, definida na biblioteca matemática padrão de C.

```
float valor_pi_tol (float tol)
{
    int i = 0;
    float s = 0.0;
    float termo;
    do {
        termo = pow_1(i) / (2*i+1);
```

```
        s = s + termo;
        i++;
    } while (fabs(termo) > tol);
    return 4*s;
}
```

Exercícios

1. O máximo divisor comum de três números inteiros positivos, MDC(x,y,z), é igual a MDC(MDC(x,y),z). Escreva um programa que capture três números inteiros fornecidos via teclado e imprima o MDC deles, usando a função MDC apresentada no texto.

2. Considerando a função que determina se um dado número é ou não primo, escreva um programa para:

   (a) Imprimir todos os números primos menores que um valor x, fornecido via teclado.

   (b) Imprimir os primeiros n números primos, onde n é fornecido via teclado.

3. Considerando a função que avalia o valor do n-ésimo termo da série de Fibonacci, escreva um programa para:

   (a) Imprimir os primeiros n termos da série, onde n é fornecido via teclado.

   (b) Determinar se um dado valor x, fornecido via teclado, pertence ou não à série.

4. A raiz quadrada de um número, $\sqrt{n}$, pode ser avaliada através do seguinte algoritmo: escolhe-se inicialmente um valor inicial qualquer, x_0. O valor escolhido não é relevante. Podemos, por exemplo, escolher $x_0 = 1$. A partir do valor inicial, avaliamos repetidas vezes um próximo valor através da expressão abaixo, até que o valor de x_i aproxime o valor de $\sqrt{n}$ dentro de uma certa tolerância.

   $$x_{i+1} = \frac{1}{2}\left(x_i + \frac{n}{x_i}\right)$$

   Usando a construção `do-while`, implemente uma função para calcular a raiz quadrada de um número real dentro de uma determinada tolerância. Para verificar se o erro na avaliação da raiz quadrada está dentro da tolerância, basta verificar se $|(x_i \times x_i) - n| < tol$.