AULA PRÁTICA 6

OUTRAS ESTRUTURAS DE REPETIÇÃO

Esta aula é também mais uma aula basicamente traduzida da nossa aula teórica, apresentando algumas pequenas diferenças dos mesmo conceitos na linguagem C. Duas outras estruturas de repetição estão disponibilizadas nesta linguagem. Uma delas é a estrutura de repetição `for`, que pode ser vista como uma outra forma de apresentação da estrutura de repetição `while`. Essas estruturas possuem exepressões lógicas iniciais que controlam o fluxo de execução de seus respectivos blocos de instruções. Uma outra estrutura apresentada aqui é a estrutura repetição `do-while`, cuja a expressão lógica de controle é posicionada no final de seu bloco de instruções.

Esta aula é baseada nas referências [9, 6].

6.1 Estrutura de repetição `for`

Retomando o programa 4.1, isto é, o primeiro exemplo com uma estrutura de repetição na linguagem C que vimos até agora, vamos refazê-lo, desta vez usando a estrutura de repetição `for`. Vejamos então o programa 6.1 a seguir.

Programa 6.1: Primeiro exemplo.

```
#include <stdio.h>

int main(void)
{
   int numero;

   for (numero = 1; numero <= 100; numero = numero + 1)
      printf("%d\n", numero);
   printf("\n");

   return 0;
}
```

A novidade neste programa é a estrutura de repetição `for`. O formato geral da estrutura de repetição `for` é dado a seguir:

```
for (inicialização; condição; passo) {
   instrução_1;
   ⋮
   instrução_n;
}
```

A estrutura de repetição `for` dispõe a inicialização, a condição e o passo todos na mesma linha de instrução. O funcionamento desta estrutura é dado da seguinte forma. Sempre que a palavra-chave `for` é encontrada, a *inicialização* é executada. Em seguida, uma *condição/expressão* é avaliada: se o resultado da avaliação é verdadeiro, então o bloco de instruções delimitado pelas chaves é executado. Ao final da execução do bloco de instruções, o *passo* é executado e o processo todo se repete. Se, em algum momento, o resultado da avaliação for falso, o fluxo de execução do programa é desviado para a primeira instrução após o bloco de instruções da estrutura de repetição.

No ponto onde nos encontramos talvez seja um bom momento para entrarmos em contato com outros dois operadores aritméticos da linguagem C. Os operadores unários de **incremento** `++` e de **decremento** `--` têm como função adicionar ou subtrair uma unidade do valor armazenado em seus operandos, respectivamente. De fato, nada mais é necessário compreender sobre esses operadores simples. No entanto, infelizmente, a compreensão das formas de uso desses operadores pode ser facilmente confundida. Isso porque, além de modificar os valores de seus operandos, `++` e `--` podem ser usados como operadores **prefixos** e também como operadores **posfixos**. Por exemplo, o trecho de código abaixo:

```
int cont;
cont = 1;

printf("cont vale %d\n", ++cont);
printf("cont vale %d\n", cont);
```

imprime `cont vale 2` e `cont vale 2` em duas linhas consecutivas na saída. Por outro lado, o trecho de código abaixo:

```
int cont;
cont = 1;

printf("cont vale %d\n", cont++);
printf("cont vale %d\n", cont);
```

imprime `cont vale 1` e `cont vale 2` em duas linhas consecutivas na saída.

Para nos ajudar, podemos pensar que a expressão `++cont` significa "incremente `cont` imediatamente" enquanto que a expressão `cont++` significa "use agora o valor de `cont` e depois o incremente". O "depois" nesse caso depende de algumas questões, mas seguramente a variável `cont` será incrementada antes da próxima sentença ser executada.

O operador de decremento `--` tem as mesmas propriedades do operador de incremento `++`, que acabamos de descrever.

É importante observar que os operadores de incremento e decremento, se usados como operadores posfixos, têm maior prioridade que os operadores unários `+`, constante positiva, e `-`, troca de sinal. Se são usados como operadores prefixos, então têm a mesma prioridade desses mesmos operadores unários (`+` e `-`).

O programa 6.2 soma os 100 primeiros números inteiros positivos e usa o operador de incremento.

Programa 6.2: Soma os 100 primeiros números inteiros positivos.

```
#include <stdio.h>

int main(void)
{
   int numero, soma;

   soma = 0;
   for (numero = 1; numero <= 100; ++numero)
      soma = soma + numero;

   printf("A soma dos 100 primeiros inteiros é %d\n", soma);

   return 0;
}
```

6.2 Estrutura de repetição `do-while`

Suponha agora que seja necessário resolver um problema simples, muito semelhante aos problemas iniciais que resolvemos usando uma estrutura de repetição. O enunciado do problema é o seguinte:

> Dada uma seqüência de números inteiros terminada com 0, calcular a soma desses números.

Com o que aprendemos sobre programação até o momento, este problema parece bem simples de ser resolvido. Vejamos agora uma solução um pouco diferente, dada no programa 6.3, contendo a estrutura de repetição `do-while` que ainda não conhecíamos.

O programa 6.3 é muito semelhante aos demais programas que vimos construindo até o momento. A principal e mais significativa diferença é o uso da estrutura de repetição `do-while`. Quando o programa encontra essa estrutura pela primeira vez, com uma linha contendo a palavra-chave `do`, o fluxo de execução segue para a primeira instrução do bloco de instruções dessa estrutura, envolvido por chaves. As instruções desse bloco são então executados uma a uma, até o fim do bloco. Logo em seguida, após o fim do bloco com o caractere `}`, encontramos a palavra-chave `while` e, depois, entre parênteses, uma expressão lógica. Se o resultado da

Programa 6.3: Exemplo usando a estrutura de repetição **do-while**.

```
#include <stdio.h>

int main(void)
{
   int numero, soma;

   soma = 0;
   do {
      printf("Informe um número: ");
      scanf("%d", numero);
      soma = soma + numero;
   } while (numero != 0);
   printf("A soma dos números informados é %d\n", soma);

   return 0;
}
```

avaliação dessa expressão lógica for verdadeiro, então o fluxo de execução do programa é desviado para a primeira instrução do bloco de instruções desta estrutura de repetição e todas as instruções são executadas novamente. Caso contrário, se o resultado da avaliação da expressão lógica é falso, o fluxo de execução é desviado para a próxima instrução após a linha contendo o final do bloco de execução desta estrutura de repetição.

Note ainda que o teste de continuidade desta estrutura de repetição é realizado sempre no final, após todas as instruções de seu bloco interno de instruções terem sido executadas. Isso significa que esse bloco de instruções será executado ao menos uma vez. Neste sentido, esta estrutura de repetição `do-while` difere das outras estruturas de repetição `while` e `for` que vimos anteriormente, já que nesses dois últimos casos o resultado da avaliação da expressão lógica associada a essas estruturas pode fazer com que seus blocos de instruções respectivos não sejam executados nem mesmo uma única vez.

Exercícios

6.1 Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos.

Exemplos:

1297: 12 e 97.
5314: 53 e 14.

Escreva um programa que imprima todos os números de quatro algarismos cuja raiz quadrada seja a soma das dezenas formadas pela divisão acima.

Exemplo:

$\sqrt{9801} = 99 = 98 + 01$.
Portanto, 9801 é um dos números a ser impresso.

Programa 6.4: Solução do exercício 6.1.

```
#include <stdio.h>

/* Imprime os números inteiros positivos de 4 dígitos cuja raiz quadra-
   da é igual à soma dos seus dois primeiros e dois últimos dígitos */
int main(void)
{
   int numero, DD, dd;

   for (numero = 1000; numero <= 9999; numero++) {
      DD = numero / 100;
      dd = numero % 100;
      if ( (DD + dd) * (DD + dd) == numero )
         printf("%d\n", numero);
   }

   return 0;
}
```

6.2 Dado um número inteiro não-negativo n, escreva um programa que determine quantos dígitos o número n possui.

Programa 6.5: Solução do exercício 6.8.

```
/* Recebe um inteiro e imprime a quantidade de dígitos que possui */

#include <stdio.h>

int main(void)
{
   int n, digitos;

   printf("Informe n: ");
   scanf("%d", n);

   do {
      n = n / 10;
      digitos++;
   } while (n > 0);

   printf("O número tem %d dígitos\n", digitos);

   return 0;
}
```

6.3 Dado um número natural na base binária, transformá-lo para a base decimal.

Exemplo:

Dado 10010 a saída será 18, pois $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 18$.

6.4 Dado um número natural na base decimal, transformá-lo para a base binária.

Exemplo:

Dado 18 a saída deverá ser 10010.

6.5 Dado um número inteiro positivo n que não contém um dígito 0, imprimi-lo na ordem inversa de seus dígitos.

Exemplo:

Dado 26578 a saída deverá ser 87562.

6.6 Dizemos que um número natural n com pelo menos 2 algarismos é **palíndromo** se

o primeiro algarismo de n é igual ao seu último algarismo;
o segundo algarismo de n é igual ao se penúltimo algarismo;
e assim sucessivamente.

Exemplos:

567765 é palíndromo;
32423 é palíndromo;
567675 não é palíndromo.

Dado um número natural n, $n \geqslant 10$, verificar se n é palíndromo.

6.7 Dados um número inteiro $n > 0$ e uma seqüência de n números inteiros, determinar quantos segmentos de números iguais consecutivos compõem essa seqüência.

Exemplo:

Para $n = 9$, a seqüência $\overbrace{5}, \overbrace{-2, -2}, \overbrace{4, 4, 4, 4}, \overbrace{1, 1}$ é formada por 4 segmentos de números iguais.

6.8 Dados um número inteiro $n > 0$ e uma seqüência de n números inteiros, determinar o comprimento de um segmento crescente de comprimento máximo.

Exemplos:

Na seqüência $5, 10, 6, \overbrace{2, 4, 7, 9}, 8, -3$ o comprimento do segmento crescente máximo é 4.
Na seqüência $10, 8, 7, 5, 2$ o comprimento do segmento crescente máximo é 1.