Controle

Uma das coisas mais importantes em qualquer linguagem de programação é o controle de fluxo e também a modularização.

Decisão

Comando “if”

O “if” permite avaliar expressões lógicas e, se o resultado final combinado for verdadeiro (diferente de zero), executar determinados comandos. A sintaxe básica é:

```
if(<condições>)
    <comando>;

ou

if(<condições>) {
    <comando1>;
    <comando2>;
    <comandoN>;
}
```

A primeira forma permite executar apenas 01 comando, caso as condições sejam verdadeiras. Eu não recomendo o uso da primeira forma porque pode levar a problemas de manutenção do código. A segunda forma permite executar um BLOCO de comandos, caso as condições sejam verdadeiras. Um BLOCO em “C” está sempre entre chaves.

Em C, há duas tendências principais para a colocação do abre chave ({) em blocos de código, e essa escolha muitas vezes depende do estilo de codificação preferido ou das diretrizes de estilo do projeto. As duas tendências são:

1. Estilo K&R (Kernighan and Ritchie) ou One True Brace Style (1TBS)

Neste estilo, o abre chave é colocado na mesma linha da declaração da função, da instrução de controle (como `if`, `for`, `while`), ou de outra estrutura de controle. Este estilo é amplamente utilizado e recomendado por muitas convenções de codificação.

```c
#include <stdio.h>

int main() {
    int x = 10;

    if (x > 5) {
        printf("x is greater than 5\n");
    } else {
        printf("x is not greater than 5\n");
    }

    for (int i = 0; i < 10; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

2. Estilo Allman

No estilo Allman, o abre chave é colocado em uma linha separada, diretamente abaixo da declaração da função, da instrução de controle, ou de outra estrutura de controle. Este estilo é usado por alguns programadores que acham que ele melhora a legibilidade ao criar blocos de código visualmente distintos.

```c
#include <stdio.h>

int main()
{
    int x = 10;

    if (x > 5)
    {
        printf("x is greater than 5\n");
    }
    else
    {
        printf("x is not greater than 5\n");
    }

    for (int i = 0; i < 10; i++)
```

```
    {
        printf("%d ", i);
    }

    return 0;
}
```

Considerações sobre a Escolha do Estilo

- **Consistência:** Independentemente do estilo escolhido, é importante ser consistente em todo o código ou projeto. Misturar estilos pode tornar o código confuso e difícil de ler.
- **Diretrizes do Projeto:** Muitos projetos ou organizações têm diretrizes de estilo de codificação que especificam qual estilo de chaves deve ser usado. Seguir essas diretrizes ajuda a manter a uniformidade no código colaborativo.
- **Preferência Pessoal:** Alguns programadores têm fortes preferências por um estilo ou outro com base em legibilidade, hábito, ou outras razões subjetivas.

Ambos os estilos são amplamente aceitos na comunidade de desenvolvedores C, e a escolha entre eles geralmente se resume a preferências pessoais ou a diretrizes do projeto.

O importante é que você sempre use blocos de comandos, mesmo que seja um só comando. Eis alguns exemplos de “if”:

```
if(a >= 2) {
    x += 2;
    printf(“OK”);
}
...
// Condição composta
if(a >= 2 && c < 7) {
}
..
// Negação com disjunção
// Exclamação é negação
// Duplo pipe "||" é OU
// Este if será verdadeiro quando: a<=2 e c>=7:
if(!(a>2 || c<7)) {}
```

Podemos também criar bloco de comando para condições falsas, e isso é feito com o comando **else**:

```
if(c >= 2) {
    printf(“OK”);
} else {
    printf(“N”);
}
...
if(x == 4) {
    if(z > 3) {
```

```
    ...
    } else {
        ...
    }
} else if(z < 5) {
    ...
} else {
...
}
```

No **else** também há os estilos **K&R** e **Allman**:

```
// estilo K&R:
if (<condição>) {
    <comandos>;
} else {
    <outros comandos>;
}
...
// estilo Allman:
if (<condição>)
{
    <comandos>;
}
else
{
    <outros comandos>;
}
```

Conectores lógicos para condições compostas

Em C ANSI, os conectores lógicos são usados para combinar várias condições em expressões compostas. Esses conectores permitem a criação de expressões lógicas complexas que podem ser avaliadas em instruções de controle como `if`, `while`, `for`, e outras. Os conectores lógicos em C ANSI são:

1. **AND Lógico (`&&`):**
    - Usado para combinar duas condições onde ambas precisam ser verdadeiras para que a expressão inteira seja verdadeira.
    - Exemplo:

```
if (a > 0 && b < 10) {
    // Executa se 'a' for maior que 0 e 'b' for menor que 10
}
```

2. **OR Lógico (`||`):**

- Usado para combinar duas condições onde pelo menos uma delas precisa ser verdadeira para que a expressão inteira seja verdadeira.
- Exemplo:

```
if (a > 0 || b < 10) {
    // Executa se 'a' for maior que 0 ou 'b' for menor que 10
}
```

3. **NOT Lógico (`!`):**

- Usado para negar uma condição. Se a condição for verdadeira, `!` a torna falsa, e vice-versa.
- Exemplo:

```
if (!a) {
    // Executa se 'a' for falso (zero)
}
```

Precedência e Associatividade

É importante entender a precedência e associatividade dos conectores lógicos para evitar ambiguidades nas expressões compostas:

- **Precedência:** O operador `!` (NOT lógico) tem maior precedência, seguido por `&&` (AND lógico), e então `||` (OR lógico).
- **Associatividade:** Os operadores `&&` e `||` são associativos da esquerda para a direita.

Uso de Parênteses

Para garantir a correta avaliação das condições, especialmente em expressões complexas, é recomendável o uso de parênteses:

```
if ((a > 0 && b < 10) || (c == 5 && !d)) {
    // Executa se ('a' for maior que 0 e 'b' for menor que 10) ou ('c' for igual a 5 e 'd' for falso)
}
```

Exemplo Completo

Aqui está um exemplo completo que demonstra o uso de conectores lógicos em C:

```
#include <stdio.h>

int main() {
    int a = 5;
```

```
    int b = 8;
    int c = 5;
    int d = 0;

    if ((a > 0 && b < 10) || (c == 5 && !d)) {
        printf("A condição composta é verdadeira.\n");
    } else {
        printf("A condição composta é falsa.\n");
    }

    return 0;
}
```

Neste exemplo:

- `(a > 0 && b < 10)` verifica se `a` é maior que 0 e `b` é menor que 10.
- `(c == 5 && !d)` verifica se `c` é igual a 5 e `d` é falso (zero).
- A expressão inteira usa `||` para combinar essas duas condições, e a mensagem apropriada é exibida com base na avaliação da expressão composta.

Comando switch

O comando “switch” é semelhante a uma estrutura **SELECT/CASE** em outras linguagens. Elepermite a execução condicional de comandos:

```
switch(<expressão>) {
    case <constante>:
        <comandos>;
        break;
    case <outra constantes>:
        <outros comandos>;
        break;
    default:
        <comandos se nenhuma condição for aceita>;
}
```

A expressão do switch é avaliada e cada comando “case” é pesquisado para comparar com o seu valor. Quando um “case” contendo o valor correto é encontrado, a execução entra nos seus comandos e continua até encontrar um “break”. Exemplo:

```
a = 3;
switch(a) {
    case 1:
        printf(“valor 1”);
        break;
    case 2:
    case 3:
        printf(“valor 2 ou 3”);
```

```
            break;
        default:
            printf(“Ouro valor”);
    }
```

Neste caso a execução entrará no “case 3” e executará todos os comandos até encontrar o “break”. Se não houvesse um “break” a execução continuaria nos comandos “default”. Se o valor de “a” fosse “2” a execução entraria no “case 2” e continuaria pelo “case 3”.

> No **switch** a expressão avaliada tem que retornar: **int** (inteiro), **char** (caractere), **enum** (constante inteira baseada em enum). Não é possível utilizar: **strings**, **vetores** ou **structs**.

Operador ternário - "?"

Outra forma pouco comum de decisão é a expressão lógica:

```
<variável> = <condição> ? <valor se verdadeira> : <valor se falsa>;
```

Exemplos:

```
b = a<2? 3 : 5;
```

Seria o equivalente a :

```
if(a<2) {
    b = 3;
} else {
    b = 5;
}
```

Loop while

O loop “while” é o mais fácil de ententer e utilizar:

```
while(<condições>) {
    <comandos>;
}
```

O bloco de comandos será executado repetidamente até que as condições tornemse falsas. Se já forem falsas no início, o bloco não será executado. Exemplo:

```

```

```
b = 0;
while(!b) {
    a += 1;
    b = a>5 ? 1 : 0;
}
```

Saída do loop

Existem dois comandos que afetam os Loops: **break** e **continue**. O “break” força a saída prematura do Loop e o “continue” força a repetição do loop. Por exemplo:

```
while(b < c) {
    scanf(“%d”, a);
    if(c < a) {
        continue; // volta para outra interação do loop.
    } else {
        break; // interrompe o loop e sai dele.
    }
}
```

Loop for

Este loop é um pouco mais complexo mas incrementa automaticamente uma variável de controle, permitindo executar seqüências de comandos:

```
for(<expressão inicial>; <expressão de controle>; <expressão de
incremento>) {
    <comandos>;
    [break; | continue;]
}
```

- **Expressão inicial**: É a expressão executada na primeira vez que o “C” entra no “for”. Serve para incializar a variável de controle. Expressão de controle é a condição que determina o fim do loop.
- **Expressão de incremento**: É executada a cada nova interação do loop. Serve para incrementar a variável de controle.

Tanto o “break” quanto o “continue” funcionam com o “for”. Exemplos:

```
for(x = 0; x < limite; x++) {
    if(x == 5) {
        continue;
    }
    printf(“%d”, x);
    if(x == 7) {
        break;
```

```
    }
}
```

A variável de controle “x” começa o Loop com zero e vai sendo incrementada (x++) equanto for menor que a variável “limite”. Porém, se o valor chegar a 5 ele não será impresso e o loop continuará. Quando chegar a 7, o loop terminará.

Comparação com o código

Agora, analise o código da aula passada e identifique esses comandos que vimos aqui.

Exercício de codificação

Muito bem. Agora você fará seu primeiro desafio de código: Criar uma função que retorne se um número é ou não primo (até 1000). Use o **Crivo de Eratóstenes** para isso.

O crivo de Eratóstenes é um algoritmo antigo e eficiente para encontrar todos os números primos até um determinado inteiro (n). Ele funciona da seguinte forma:

1. **Inicialização:**
    - Crie uma lista de números sequenciais de 2 até (n).
2. **Marcação de múltiplos:**
    - Comece com o primeiro número da lista (2). Marque todos os seus múltiplos (exceto ele mesmo) como não primos.
    - Encontre o próximo número não marcado na lista e repita o processo de marcar os seus múltiplos como não primos.
    - Continue esse processo até que não haja mais números não marcados na lista que são menores ou iguais à raiz quadrada de (n).
3. **Resultado:**
    - Todos os números que não foram marcados como não primos são primos.

Exemplo em Português Estruturado

Vamos encontrar todos os números primos até (n = 30).

1. **Inicialização:**
    - Crie uma lista de números de 2 a 30:

      ```
      [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
      20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
      ```

2. **Marcação de múltiplos:**
    - O primeiro número é 2. Marque todos os múltiplos de 2 (exceto ele mesmo):

```
[2, 3, X, 5, X, 7, X, 9, X, 11, X, 13, X, 15, X, 17, X, 19, X,
21, X, 23, X, 25, X, 27, X, 29, X]
```

- O próximo número não marcado é 3. Marque todos os múltiplos de 3 (exceto ele mesmo):

```
[2, 3, X, 5, X, 7, X, X, X, 11, X, 13, X, X, X, 17, X, 19, X, X,
X, 23, X, 25, X, X, X, 29, X]
```

- O próximo número não marcado é 5. Marque todos os múltiplos de 5 (exceto ele mesmo):

```
[2, 3, X, 5, X, 7, X, X, X, 11, X, 13, X, X, X, 17, X, 19, X, X,
X, 23, X, X, X, X, X, 29, X]
```

- O próximo número não marcado é 7. Marque todos os múltiplos de 7 (exceto ele mesmo):

```
[2, 3, X, 5, X, 7, X, X, X, 11, X, 13, X, X, X, 17, X, 19, X, X,
X, 23, X, X, X, X, X, 29, X]
```

- Continuamos até que os números a serem verificados sejam maiores que a raiz quadrada de 30 (aproximadamente 5,47). Neste ponto, todos os múltiplos de números primos até a raiz quadrada já foram marcados.

3. **Resultado:**

- Os números que não foram marcados são os números primos:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Portanto, os números primos até 30 são: 2, 3, 5, 7, 11, 13, 17, 19, 23, e 29.

Como saber se um número é primo utilizando o Crivo de Eratóstenes

Para verificar se um número (n) é primo usando o crivo de Eratóstenes, podemos seguir um processo adaptado:

1. **Crie uma lista de números primos até a raiz quadrada de (n) utilizando o crivo de Eratóstenes:**

- Se (n) não for muito grande, crie uma lista de números de 2 até (\sqrt{n}) (o teto da raiz quadrada de (n)).
- Aplique o crivo de Eratóstenes nessa lista para obter todos os números primos até (\sqrt{n}).

2. **Verifique a primalidade de (n) utilizando os números primos obtidos:**

- Divida (n) por cada um dos números primos obtidos.
- Se (n) for divisível por qualquer um desses números primos, então (n) não é primo.
- Se (n) não for divisível por nenhum desses números primos, então (n) é primo.

Exemplo

Vamos verificar se (n = 37) é primo.

1. **Calcule a raiz quadrada de 37 e arredonde para cima:**
   - (\sqrt{37} \approx 6.08 \rightarrow 7)
2. **Crie uma lista de números de 2 até 7 e aplique o crivo de Eratóstenes:**
   - Lista inicial: ([2, 3, 4, 5, 6, 7])
   - Aplicando o crivo:
     - 2: marque 4 e 6 (múltiplos de 2).
     - 3: marque 6 (múltiplo de 3).
   - Lista após aplicar o crivo: ([2, 3, 5, 7])
3. **Verifique a primalidade de 37:**
   - Divida 37 por cada um dos números primos ([2, 3, 5, 7]):
     - 37 ÷ 2 = 18.5 (não é divisível)
     - 37 ÷ 3 ≈ 12.33 (não é divisível)
     - 37 ÷ 5 = 7.4 (não é divisível)
     - 37 ÷ 7 ≈ 5.29 (não é divisível)
   - Como 37 não é divisível por nenhum desses números, concluímos que 37 é primo.

> Usar o crivo de Eratóstenes para verificar a primalidade de um número envolve gerar uma lista de números primos até a raiz quadrada do número em questão e verificar se o número é divisível por algum desses primos. Se não for divisível por nenhum, o número é primo; caso contrário, não é.

Crie uma função em C para saber se um número é primo. Veja o exercício na próxima aula e complete. Haverá uma correção nos anexos da aula seguinte.