Programação em C/C ++ - Ciclos e Decisões

Sérgio Lopes

Programação em C/C++ - Ciclos e Decisões

Sérgio Lopes

Índice

Lista de Figuras

Lista de Tabelas

Sobre o Manual

Público Alvo

Este manual foi criado, especificamente, para o módulo *0783 - Programação em C/C++ - Ciclos e Decisões*, ministrado em formações profissionais como indicado pelo catálogo da ANQ, www.catalogo.anq.gov.pt, e faz a continuação do manual fornecido para o módulo *0782 - Programação em C/C++ - Estrutura Básica e Conceitos Fundamentais*.

Estrutura e Conteúdo

O manual começa por introduzir os operadores usados na linguagem de programação C, incluindo os operadores lógicos que fazem uso da matéria apresentada no módulo anterior, seguidos das estruturas de decisão, *if* e *switch*, e das estruturas de repetição, *for*, *while* e *do… while*. Todos os exemplos são testados no IDE apresentado no primeiro módulo, Code::Blocks.

Licença

Capítulo 1. Operadores

Temos já utilizado, em módulos anteriores, aquilo que chamamos operadores. Em cada cálculo matemático que fizemos usámos os sinais de somar, subtrair, multiplicar ou dividir, e cada um destes é um *operador* que em conjunto com os respectivos operandos (os números que somamos ou subtraímos, etc). Um operador permite definir as acções que podem ser feitas aos operados. No caso da soma, o operador + define que podemos pegar nos dois operandos, o número que estiver à esquerda e o que estiver à direita, e que o resultado é a soma dos dois.

Em qualquer linguagem de programação, a construção de programas exige o uso de operadores. Nem todos os operadores fazem operações matemáticas e nem sempre são necessários dois operandos.

Em linguagem C os operadores permitem efectuar operações sobre e com as nossas variáveis que, dependendo do tipo de operador, se podem classificar como *operador de atribuição*, *operadores aritméticos*, *operadores unários*, *operadores de incremento/decremento* e *operadores compostos*.

Atribuição

O operador de atribuição permite atribuir um valor a uma variável, é usado o sinal =, e é com este operador que colocamos valores dentro das nossas variáveis, **atribuímos valores às variáveis**.

Com um operador de atribuição a expressão é avaliada **da direita para a esquerda**, assim podemos dizer que o valor que está **à direita** é colocado dentro da variável que está **à esquerda**.

Exemplo do uso do operador de atribuição.

```
a = 5;
b = a;
c = 'A';
d = b = c = a + 1;
```

Como podemos ver na última expressão do exemplo, é possível encadear a atribuição de um valor por vários operandos, sendo que todos ficarão com o mesmo valor final.

Aritméticos

Os operadores aritméticos permite fazer contas com os valores dos operandos, tipicamente variáveis, e são usados os caracteres +, *, - e / para representar as operações de **somar**, **multiplicar**, **subtrair** e **dividir**.

```
a + b;
5 + c;
8 + 9;
8 * a;
f / g;
7 - h;
```

No caso do operador de divisão, o tipo de dados dos operandos pode afectar o resultado final. Se os operandos forem os dois inteiros então será feita uma divisão inteira onde não há parte fraccionaria, por exemplo *5 / 2 = 2*, como a divisão inteira não admite valores fraccionários, em vez do resultado ser 2,5 é apenas 2. Podemos dizer que *5 / 2* é igual a 2 com resto 1. Se a divisão for feita com dois operandos não inteiros a parte fraccionaria já é permitida, exemplo *5,0 / 2 = 2,5*.

Módulo

O operador módulo permite obter o resto de uma divisão inteira. Para este operador usamos o carácter percentagem, **%**.

Figura 1.1. Resto da Divisão

Unário

O operador unário afecta apenas uma variável e apenas altera o sinal dessa variável, passado um valor negativo a positivo e um valor positivo a negativo. Para o operador unário usam-se dois caracteres: o sinal **-** e o sinal **+**.

```
a = 5;
-a; //estamos a tornar o 5 negativo.
```

Incremento e Decremento

Os operadores de incremento e decremento permitem incrementar ou decrementar uma variável, uma unidade de cada vez. Não permite incrementar ou decrementar quantidades à escolha, apenas uma unidade. Para este operador usam-se dois sinais de **+** para incrementar, ou dois sinais de **-** para decrementar. Os dois sinais são colocados depois da variável e não podem ter espaço a separá-los.

```
a++;
b--;
```

Compostos: Atribuição + Aritméticos

É possível combinar os operadores aritméticos com o operador de atribuição, fazendo num só operador as duas operações: a operação de atribuir e a operação aritmética usada. Para estes operadores são usados os caracteres aritméticos e o operador módulo, **+**, *****, **-**, **/** e **%**, seguidos do carácter de atribuição, **=**.

```
a += 5;
b -= 2;
a *= 4;
g /= 67;
h %= 9;
```

Operadores Booleanos ou Lógicos

Os operadores lógicos permitem a comparação de expressões e devolvem sempre um de dois valores: verdadeiro ou falso.

Estes operadores podem ser organizados em: .. Operadores Relacionais: ==, >, <, >=, <=, **!=** - Comparam duas expressões ou variáveis e devolvem um valor verdadeiro ou falso de acordo com a comparação feita. .. Operadores Lógicos: **&&**, ||, **!** - Permitem aplicar as operações *boleanas* a dois valores lógicos (respectivamente: multiplicação lógica, soma lógica e negação).

Operador	**Significado**	**Exemplo**	**Descrição**
==	Igualdade	x == y	x é **igual** a y?
!=	Diferente	x != y	x é **diferente de** y?
>	Maior que	x > y	x é **maior que** y?
>=	Maior ou Igual que	x >= y	x é **maior ou igual** a y?
<	Menor que	x < y	x é **menor que** y?
<=	Menor ou Igual que	x <= y	x é **menor ou igual** a y?
&&	Multiplicação (E)	x && y	avalia x e y e **multiplica** os **seus valores lógicos**
||	Soma (OU)	x || Y	avalia x e y e **soma** os seus **valores lógicos**
!	Negação	!x	nega x

Precedência de Operadores

A precedência dos operadores determina a ordem pela qual as expressões são avaliadas. Tal como na matemática, em que as multiplicações são sempre feitas antes as somas ou subtracções, também numa linguagem de programação e nos seus operadores é importante que exista uma ordem definida. A tabela seguinte lista os operadores e a ordem pela qual são aplicados (de cima para baixo, os operadores da primeira linha têm maior precedência que os da segunda, e assim sucessivamente) bem como a direcção pela qual a expressão onde esses operadores entram é avaliada (se a avaliação é da direita para a esquerda ou da esquerda para a direita).

Tabela 1.1. Tabela de Precedências e Associatividade

Grau de Precedência	Operadores	Ordem de Associação
Maior precedência	( ) [] → .	esquerda para direita
	! ~ ++ — * (no caso de ponteiros) &	direita para esquerda
	* / %	esquerda para direita
	+ -	esquerda para direita
	<< >>	esquerda para direita
	< <= > >=	esquerda para direita
	== !=	esquerda para direita
	&	esquerda para direita
	^	esquerda para direita
	|	esquerda para direita
	&&	esquerda para direita
	||	esquerda para direita
	?:	direita para esquerda
	= += -= = / %= &= ^=	direita para esquerda
	|= <<= >>=	
Menor precedência	,	esquerda para direita

Operador de Cast

O último operador que iremos ver é o operador de **cast**, que permite a conversão de um tipo de dados para outro tipo de dados. Esta conversão nem sempre pode ser feita correctamente e é possível que a mesma resulte na perda de informação, no entanto, cabe ao programador ter em atenção se a conversão é ou não possível e o compilador apenas avisará que *pode existir* perda de informação mas fará a conversão pedida.

Para usarmos o operador de **cast** devemos colocar o tipo de dados que pretendemos obter entre parêntesis seguido da variável ou valor a converter, **f = (int)x;**.

Conversões Através de Cast.

```
char x = 'D';
int a1;
float a2;
double a3;

//converter x para inteiro
a1 = (int) x;

//converter x para float
a2 = (float) x;

//converter x para double
a3 = (double) x;
```

Como é possível ver, para converter um tipo de dados para outro basta colocar o tipo de dados destino entre parêntesis seguido do valor a converter e o resultado será o valor convertido, com as perdas de informação caso tal aconteça.

Capítulo 2. Estruturas de Controlo de Fluxo de Execução

As estruturas de controlo de fluxo de execução, também chamadas de estruturas de controlo de sequência, permite controlar a ordem de execução das instruções, oferecendo métodos para determinar que instruções são executadas, no caso de *estruturas de decisão*, ou métodos que possibilitam a repetição controlada de conjuntos de instruções, no caso de *estruturas de repetição*.

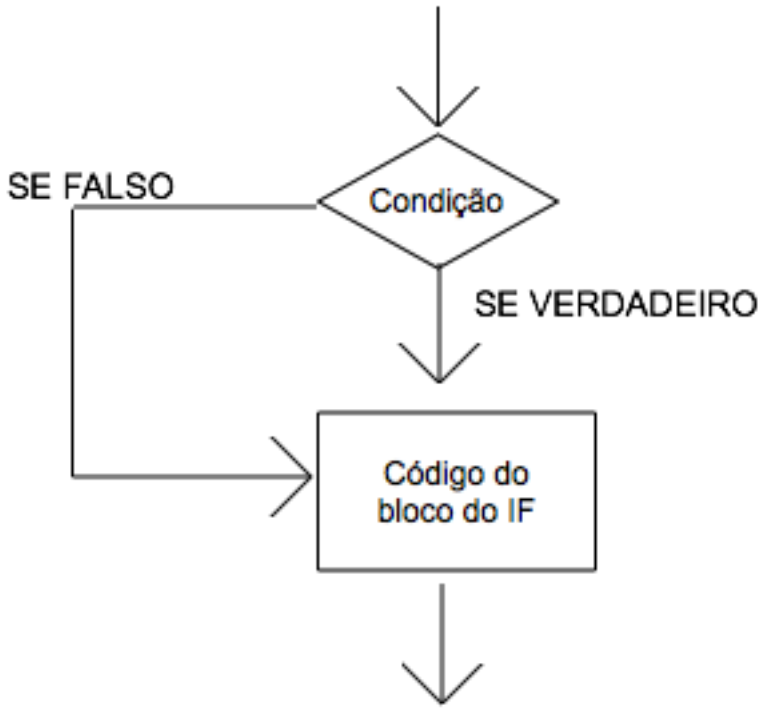
Em todos os casos as estruturas podem ou não ter duas chavetas associadas no entanto, durante todo o manual, serão usadas sempre duas chavetas e nunca uma estrutura, seja de decisão seja de repetição será apresentada sem as chavetas correspondentes. Para facilitar a escrita e leitura de código **recomenda-se que sejam usadas as duas chavetas** delimitadores de blocos de código em todas as situações onde sejam usadas estruturas de controlo de execução.

Condições

Todas as estruturas que iremos ver neste capítulo usam, pelo menos, uma condição que permite controlar a execução da estrutura. E em todas as situações, a estrutura só executa o código que contém quando a condição de controlo é verdadeira. As condições fazem uso da Álgebra de Boole, apresentada no módulo anterior, juntamente com os caracteres **&&** para *E*, || para *OU* e **!** para *Negação*.

if

Talvez a estrutura mais simples de todas, e também a mais utilizada, a estrutura de decisão **if** permite determinar se um bloco de código é executado.



Nesta estrutura, o código afectado só é executado se a instrução for verdadeira. Caso a instrução seja falsa, nenhuma linha de código associada ao **if** é executada.

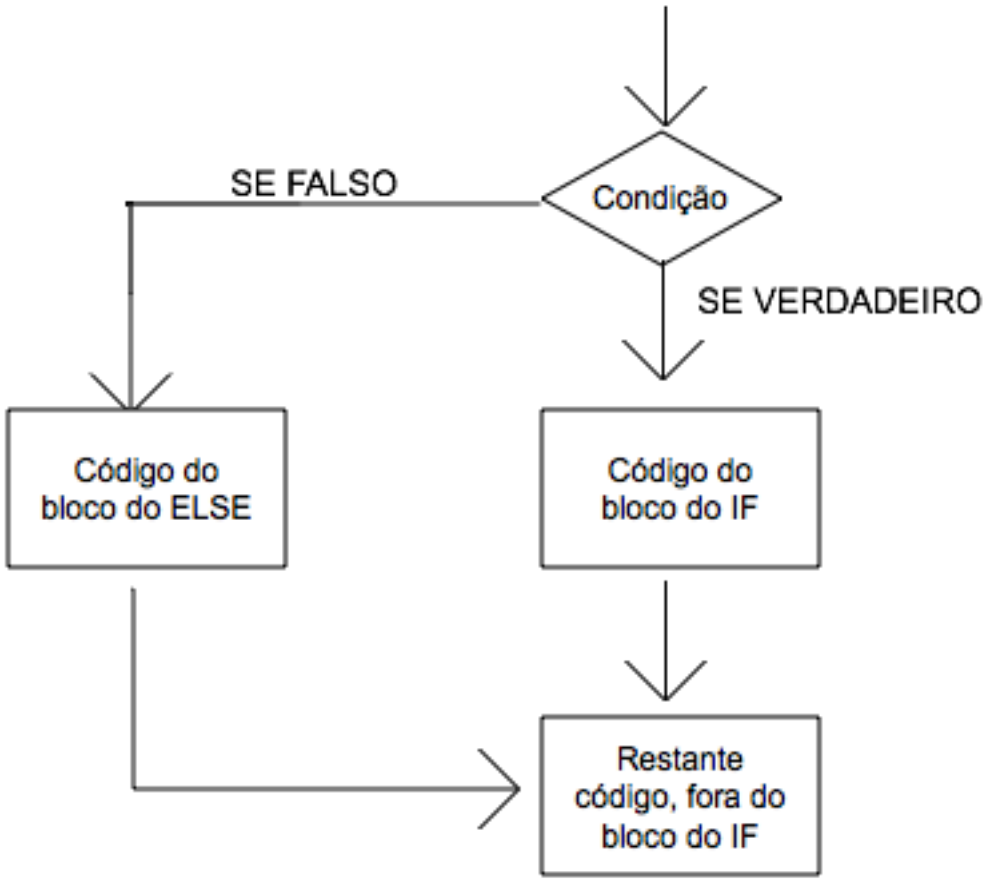
Exemplo.

```
int x = 5;

if(x > 0)
{
    //este código é executado
    //porque o x é maior que zero.
    printf("Variável positiva");
}

if(x % 2 == 0)
{
    //este código não é
    //executado porque o x não é par
    printf("Variável par");
}
```

if… else

Uma variação da estrutura **if** é incluir uma cláusula alternativa que é executada quando a condição for falsa. Temos assim a estrutura **if… else**, que tem um funcionamento igual ao da estrutura **if**, acrescentando apenas um bloco de código que é executado sempre que a condição que controla o **if** é falsa. Se usando apenas a forma simples da estrutura **if** não existe qualquer código a executar quando a condição é falsa, usando a forma completa **if… else**, **o código dentro do bloco if é executado quando a condição for verdadeira e o código dentro do bloco else é executado quando a condição for falsa**.

Figura 2.1. Estrutura de Decisão IF... ELSE



Exemplo.

```
int x = 5;

if(x > 0)
{
    //este código é executado
    //porque o x é maior que zero.
    printf("Variável positiva");
}
else
{
    //este código só seria executado se
    //a variável fosse negativa ou igual a zero.
    printf("A variável é negativa ou zero");
}

if(x % 2 == 0)
{
    //este código não é
    //executado porque o x não é par
    printf("Variável par");
}
else
{
    //este código é executado porque a
    //variável é ímpar
```

```
    printf("Variável ímpar");
}
```

A estrutura **if… else** pode ser encadeada através da utilização de mais palavras reservadas **if** e **if… else**.

Exemplo.

```
int x = 3;

if(x < 0)
{
    printf("Valor menor que 0");
}
else if(x == 0)
{
    printf("Valor igual a 0");
}
else if(x == 1)
{
    printf("Valor igual a 1");
}
else if(x == 2)
{
    printf("Valor igual a 2");
}
else
{
    printf("Valor maior que 2");
}
```

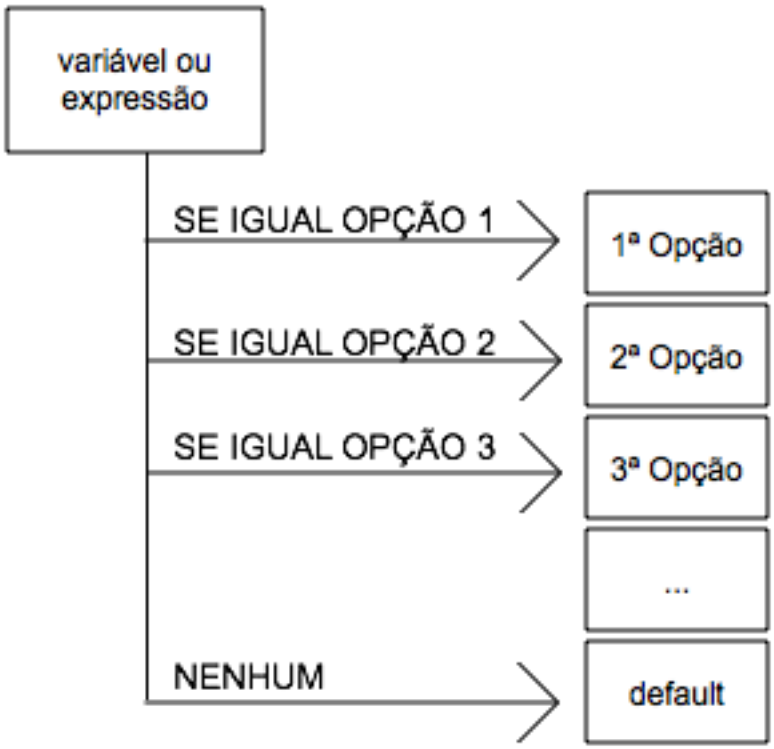
switch

A estrutura **switch** permite avaliar uma expressão ou variável e escolher a alternativa correcta de entre várias alternativas possíveis. A expressão ou variável a testar tem de ser do tipo inteiro (relembrar que os caracteres possuem uma representação interna que é inteira, e podem ser usados), e as alternativas devem estar apresentadas no corpo da estrutura através do uso da palavra reservada **case**.

O funcionamento da estrutura **switch** é simples: a expressão é avaliada e o seu valor é comparado com cada uma das opções disponíveis, assim que uma comparação for verdadeira, isto é, o valor da expressão ou variável é igual à opção indicada, o código dessa opção e de todas as seguintes é executado. Para impedirmos que o código das opções seguintes seja executado é necessário coloca a palavra reservada **break** como última linha do bloco da opção. Iremos ver o que esta palavra faz no último capítulo do manual.

Caso seja necessário colocar uma alternativa às opções indicadas basta que se coloque a secção **default** como última secção do **switch**. Esta secção permite que o código nela seja executado quando a variável ou expressão não corresponde a nenhuma das opções indicadas no **switch**.

Figura 2.2. Estrutura de Decisão SWITCH



Exemplo.

```
int x = 5;

switch(x)
{
    case 0:
        printf("Variável igual a 0");
        break;
    case 1:
        printf("Variável igual a 1");
        break;
    case 2:
        printf("Variável igual a 2");
        break;
    case 3:
        printf("Variável igual a 3");
        break;
    case 4:
        printf("Variável igual a 4");
        break;
    case 5:
        printf("Variável igual a 5");
        break;
    case 6:
        printf("Variável igual a 6");
        break;
    case 7:
        printf("Variável igual a 7");
        break;
    case 8:
        printf("Variável igual a 8");
        break;
```

```
    case 9:
        printf("Variável igual a 9");
        break;
    default:
        printf("Variável diferente de todas as opções");
}
```

No exemplo acima, apenas o *printf* do **case** com o valor 5 é que seria executado. Se alterarmos o valor da variável entre 0 e 9 as outras utilizações da função *printf* são executadas e se colocarmos um valor que não está em nenhuma das opções, o *printf* presente na secção **default** é executado.

É necessário notar que a secção **default** não é obrigatória, e que o uso da palavra **break** é feito para que apenas seja executado o código dentro do **case** que nos interessa e não todos os que estão depois desse.

Capítulo 3. Estruturas de Repetição

As estruturas de repetição, vulgo ciclos, permite repetir blocos de código com base numa condição de controlo. Este tipo de estruturas é muito importante em linguagens de programação uma vez que permite criar funcionalidades para situações onde a repetição de código é necessário mas a escrita desse mesmo código várias vezes não é prática. Por exemplo, se for necessário ler os dados pessoais de um cliente de uma loja será simples fazer o código de leitura, mas e se for necessário fazer a leitura dos dados de dez clientes iremos escrever o mesmo código dez vezes? E se o número de dados que queremos ler não é conhecido como é que podemos repetir o código um número indeterminado de vezes?

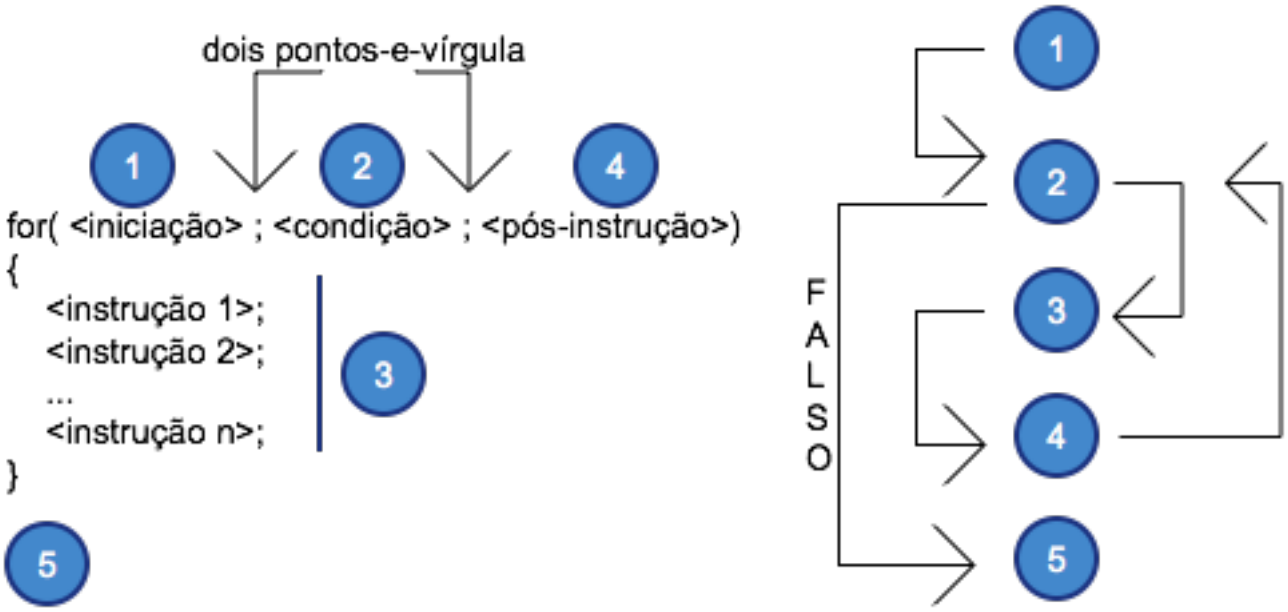
for

Esta é a primeira estrutura de repetição que iremos ver, e como todas as estruturas de repetição permite que determinado conjunto de instruções seja repetido várias vezes, dependendo de uma condição de controlo. Este ciclo é óptimo quando sabemos, à partida, quantas vezes pretendemos repetir o código, isto é, quantas iterações [1] tem o ciclo.

A estrutura do ciclo **for** tem quatro secções importantes: a secção de iniciação, secção da condição, secção de pós-instruções, e bloco de código a executar. Embora cada uma destas zonas possa ser omitida é importante que, no caso das três primeiras secções, estejam presentes os delimitadores, o **;**.

O funcionamento deste ciclo pode ser explicado da seguinte forma: **as instruções presentes na secção de iniciação (1) são executadas;** a condição é avaliada, se o resultado for falso o ciclo termina; **sendo o resultado verdadeiro, são executadas as instruções do bloco de código do ciclo;** após a última instrução do bloco de código são executadas as instruções presentes na secção de pós-instruções; ** a condição é novamente avaliada e o processo repete-se

Figura 3.1. Ciclo FOR

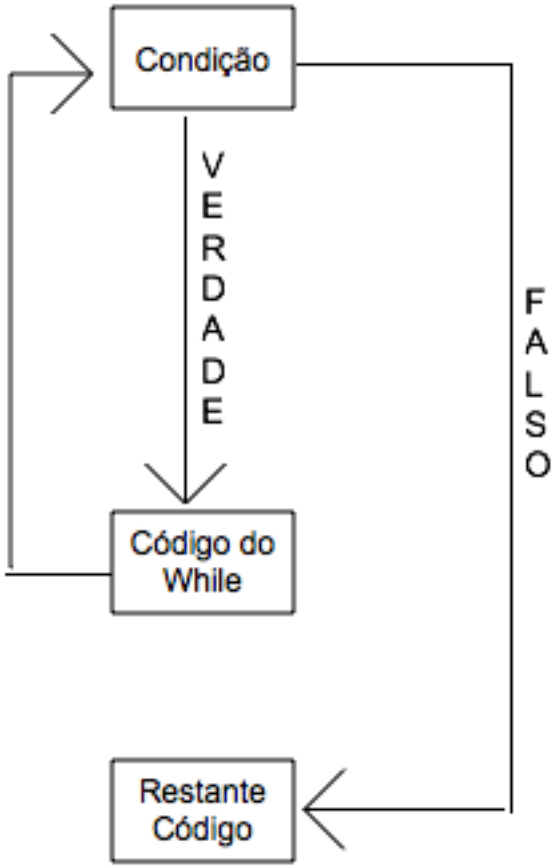


[1] uma iteração corresponde a uma execução completa do código dentro do ciclo, assim cada vez que o código executar completamente e voltarmos à condição de controlo do ciclo temos uma iteração completa

while

A estrutura **while** é usada quando queremos repetir um conjunto de instruções, delimitadas por duas chavetas como indicado no início do capítulo, enquanto determinada condição é verdadeira. A condição do ciclo é testada no início, sendo o bloco de código do **while** executado se a mesma for verdadeira, a cada execução do bloco de código a condição é novamente testada e o bloco é repetido enquanto a condição se mantiver verdadeira. Se a condição for falsa, então nenhuma instrução pertencente ao **while** é executada.

Figura 3.2. Ciclo WHILE



Exemplo.

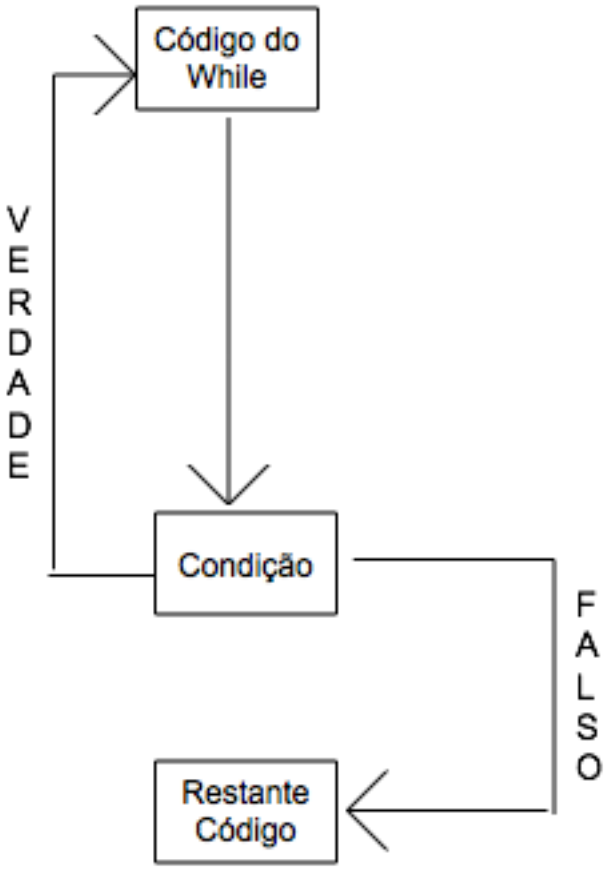
```
int x;

while(x > 0)
{
    printf("Valor de x: %d", x);
    x = x - 1;
}
```

do… while

Esta estrutura é idêntica à estrutura **while**, a diferença, além da sintaxe, reside na posição onde a condição de controlo é verificada. Enquanto que para o ciclo **while** a condição é verificada logo no início, no

ciclo **do… while** a condição é testada no fim. A consequência desta alteração é que no ciclo **while** o o código é executado apenas se a condição for verdadeira mas no ciclo **do… while** o código é sempre executado uma vez antes da condição ser testada, se a condição for falsa o ciclo não executa mais, mas pelo menos uma vez o código foi executado.

Figura 3.3. Ciclo DO... WHILE



Exemplo.

```
int x;

do
{
    printf("Valor de x: %d", x);
    x = x - 1;
}
while(x > 0); //notar o uso de ;
```

Terminar Ciclos: break e continue

Estas duas instruções permitem terminar um ciclo ou a estrutura **switch** (não têm qualquer efeito no caso da estrura **if** ou **if… else**). Sendo que a instrução **break** termina o ciclo imediatamente, sem que seja verifica da condição de controlo, e a instrução **continue** permite saltar para a próxima iteração sem executar o código que possa, eventualmente, existir depois da instrução **continue**.

Dentro de um **swich** a instrução **continue** não terá qualquer efeito uma vez que esta estrutura não executa várias iterações, não é um ciclo, mas como vimos, a instrução **break** permite que apenas seja executado o código nas opções que nos interessam, isto porque o **break** interrompe a execução do **switch**. Assim, ao colocarmos como última instrução dos nossos blocos de código de cada **case** garantirmos que se esse **case** é escolhido, os seguintes não são executados.

Exemplo.

```
int i;

for(i = 0; i < 100; i++)
{
    if (i % 2 == 0)
    {
        //se o i for par passamos
        //para a próxima iteração e não
        //executamos as linhas abaixo
        continue;
    }

    printf("Vamos em: %d", i);

    if(i == 51)
    {
        //assim que chegamos a 51
        //terminamos o ciclo for
        break;
    }
}
```

Bibliografia

- Richard L. Petersen. *Introductory C, Second Edition: Pointers, Functions, and Files*. Morgan Kaufmann. 7 de Novembro, 1996. ISBN 978-0125521420.