

TÉCNICAS DE PROGRAMAÇÃO

UNIDADE 1 - PROGRAMAÇÃO ESTRUTURADA

Fernando Cortez Sica

Introdução

Para iniciarmos a nossa conversa, é interessante falarmos sobre o que é a linguagem de programação. Sabemos que linguagem é todo sistema constituído por códigos, regido por regras, cujo objetivo é transmitir uma informação. Mas, o que vem a ser linguagem na computação? Linguagens existem apenas para desenvolver programas?

No campo da informática, a linguagem não é usada somente na criação de programas computacionais, mas sim, em diversas outras ocasiões, podemos citar as linguagens formais, que permitem, por exemplo, a especificação e teste das próprias linguagens de programação, protocolos de redes de computadores e reconhecimento de padrões em processamento de textos. Na área de *hardware*, podemos citar as linguagens de descrição de *hardware* (*HDL – Hardware Description Language*): este tipo de linguagem objetiva o projeto de circuitos eletrônicos digitais, tais como processadores. A partir do uso de uma *HDL*, pode-se, por exemplo, criar um *chip*, configurar uma *FPGA* (*Field Programmable Gate Array – Matriz de Portas Programáveis em Campo*) ou um *PSoC* (*Programmable System-on-Chip – Sistema programável em um chip*). Por fim, podemos mencionar também, as linguagens de marcação: dotadas de elementos de anotações de forma a formatar um texto a ser exibido ou a ser manipulado. Como exemplos, temos a *HTML* (*HyperText Markup Language – Linguagem de Marcação de HiperTextos*) e o *XML* (*eXtensible Markup Language – Linguagem de Marcação Extensível*).

No nosso caso, vamos nos aprofundar nas linguagens de programação, mais especificamente, nas linguagens estruturadas, utilizando para as exemplificações a linguagem C de programação. Então, o que vem a ser efetivamente uma linguagem de programação?

Neste capítulo, veremos alguns conceitos relacionados à programação estruturada assim como os primeiros passos para a construção de programas computacionais. Ao final deste você terá condições de implementar programas manipulando tipos de dados simples, *strings* e vetores.

1.1. Conceitos e características

Uma linguagem de programação tem por objetivo descrever a sequência das ações que o computador deverá executar para que um objetivo seja alcançado. (PUGA, 2016). Para tanto, a linguagem deverá descrever, também, as informações (constantes e variáveis) que serão processadas pelas instruções.

As ações a serem realizadas pelo computador são representadas, nas linguagens de programação, pelas instruções de alto nível que deverão ser codificadas em instruções de baixo nível específica do processador. A figura a seguir ilustra as etapas necessárias para que tenhamos um programa apto a ser executado em uma plataforma computacional.

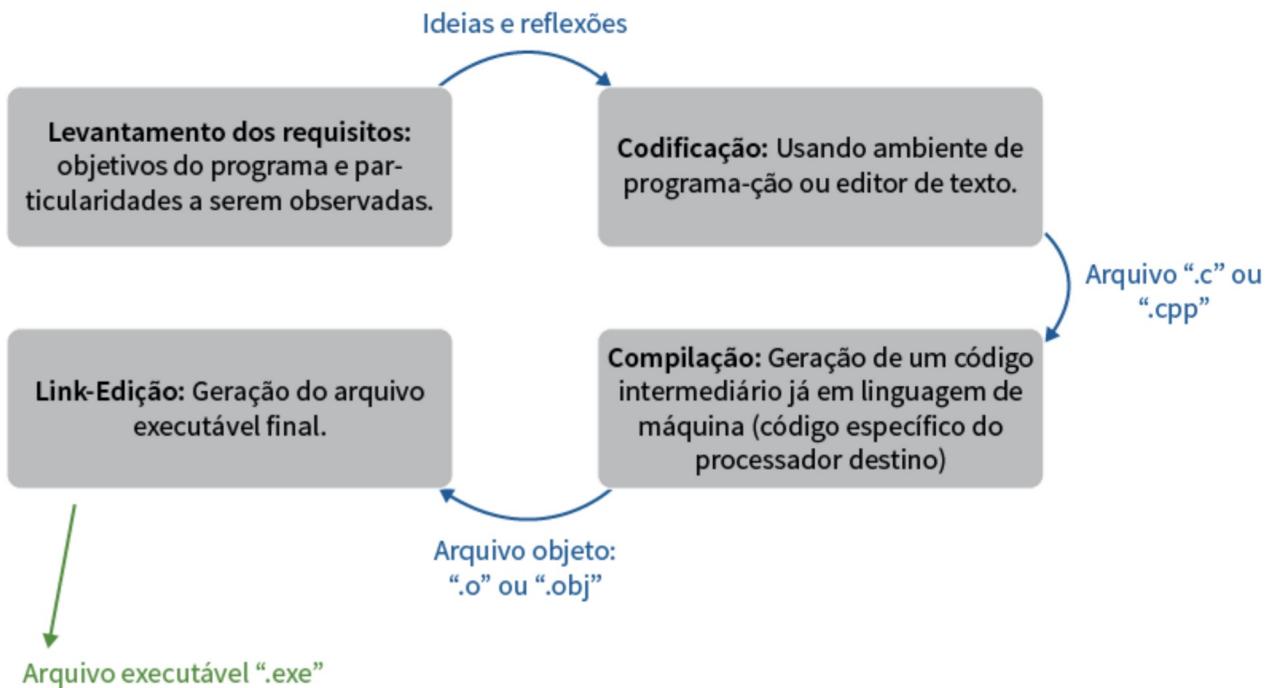


Figura 1 - Etapas para a geração de um programa (arquivo executável). Neste fluxo não estão representadas as etapas de teste e verificação de qualidade, que podem provocar uma volta a primeira etapa (processo cíclico).

Fonte: Elaborada pelo autor, 2019.

Como mostrado na figura, falou-se em “plataforma computacional”. Mas, porque não falamos diretamente “processador” ou “computador”? Para saber mais sobre o tema, clique nas abas abaixo.

Linguagens interpretadas	O fato é que algumas linguagens, como o PHP, Ruby e o Javascript, são linguagens interpretadas, ou seja, as suas instruções são executadas por uma máquina virtual que converte, em tempo de execução, para o código do hardware específico.
Linguagens compiladas	Por sua vez, linguagens compiladas, como o C/C++ e Pascal, tem os seus códigos de baixo nível compatíveis com o processador, gerados na etapa de compilação (produzindo, por exemplo, os arquivos do tipo “.exe”).
Linguagens híbridas	Existem as linguagens híbridas, tais como o JAVA e o C#, que são compiladas, porém são executadas sobre uma máquina virtual.

Antes de continuarmos, pode vir à cabeça a seguinte dúvida: a linguagem “C ANSI” (aqui referenciada simplesmente como “linguagem C”) é diferente da linguagem C++ (C plus plus)? Existem diferenças entre os dois padrões. Rapidamente falando, a linguagem C++ é uma extensão da linguagem C incorporando, por exemplo, os conceitos de programação orientada por objetos.

VOCÊ O CONHECE?



A linguagem C foi criada nos finais dos anos de 1960 e início dos anos de 1970 por Dennis Ritchie. Além de criar essa famosa linguagem que ainda está presente em diversas plataformas e sistemas, ele também foi responsável pela criação do sistema operacional UNIX. Para saber mais sobre Dennis Ritchie, clique aqui <<http://mindbending.org/pt/um-ano-sem-dennis-ritchie>>. (MAGNUN, 2012).

Para implementar um programa, deveremos ter em mente o seu objetivo e consequentemente quais informações deverão ser manipuladas e quais informações deverão ser produzidas (variáveis de entrada e de saída) por ele. Com isso, poderemos refletir na primeira ação a ser feita para que possamos prosseguir na implementação: definição das variáveis. Na seção a seguir, falaremos sobre o que vem a ser os tipos das variáveis.

1.2. Tipos de Dados

Como mencionamos, uma linguagem de programação deve conter, além das instruções, as próprias informações a serem manipuladas, sendo assim, o sistema operacional deve reservar espaços na memória principal do computador para armazená-las. Mas, como saber a quantidade de memória a ser alocada? A resposta a essa pergunta está relacionada com o tipo das variáveis.

Em linguagens de programação, como a linguagem C/C++, o programador deve assinalar, em seu código, o tipo das variáveis a serem manipuladas. Além de possibilitar a alocação de memória, a definição do tipo servirá, também, para criar os mecanismos internos inerentes à sua manipulação. Por exemplo, se pensarmos na matemática, encontramos os tipos numéricos, cada um com as suas especificidades (números reais, inteiros, racionais dentre outros), essa diferenciação que ocorre na matemática pode ser trazida para a programação.

Vamos, inicialmente, nos deter aos tipos numéricos da linguagem C/C++. Existem diferenças nos tipos e na forma de definição entre o “*C standard*” e o *C++*? No quadro a seguir, ilustraremos alguns tipos básicos numéricos que encontramos em ambas as linguagens. (ASCENCIO, 2012).

Tipo	Descrição	Tamanho	C ANSI	C++
char	Capaz de armazenar a faixa de valores entre -128 a 127. O seu nome relaciona-se à possibilidade de seu uso para armazenar caracteres (representados pelo seu código na tabela ASCII). Porém, um char pode ser usado, também, para armazenar dados numéricos.	8 bits (1 byte)	✓	✓
int	Manipula valores inteiros na faixa de -32.768 até 32.767. Quando associado ao modificador de tipo “long” (“long int”) tem a sua faixa de -2.147.483.648 a 2.147.483.647 (4 bytes). O tipo int pode variar em função do compilador utilizado, podendo ser representado, em alguns casos, por 4 bytes.	16 bits ou 32 bits, conforme o compilador utilizado	✓	✓
float	Manipula valores reais na faixa de $\pm 3.4E^{-38}$ a $\pm 3.4E^{38}$.	32 bits	✓	✓
double	Assim como o float, o double também armazena números reais, porém na faixa de $\pm 1.7E^{-308}$ a $\pm 3.4E^{308}$.	64 bits	✓	✓
bool	Usado para valores binários “true” (verdadeiro) ou “false” (falso).	8 bits	■	✓

Quadro 1 - Tipos de dados numéricos básicos encontrados nas linguagens C/C++. Os tamanhos são relativos aos computadores padrão x86 – podem variar em outras arquiteturas.

Fonte: ASCENCIO, 2012. p.26.

Mas, como definir as variáveis? Vamos supor que temos que fazer um programa que calcule a média da idade de duas pessoas. Clique nos itens abaixo e aprenda mais sobre a definição dessas variáveis.

Qual o objetivo de nosso programa?

Calcular a média de duas idades.

Quais informações deverão ser classificadas como entradas?

As idades das duas pessoas.

As idades deverão ser de qual tipo?

Considerando que as idades são representadas por números inteiros, então, deveremos declará-las como “int”.

Qual informação que deverá ser gerada?

A média das duas idades.

A média deverá ser de qual tipo?

Como a média poderá resultar em um valor real, deveremos declará-la como “float”.

Antes de colocarmos aqui o nosso primeiro exemplo de código em C, mencionamos que a definição de uma variável segue a estrutura, de acordo com Deitel, (2011):

<tipo> nome_da_variável ;

Ou

<tipo> nome_da_variável_1, nome_da_variável_2 ;

As estruturações acima são utilizadas no código a seguir. Neste código, que objetiva o cálculo e impressão na tela, a média de duas idades. Os comentários estão precedidos por duas barras ("//") comentários também poderão aparecer entre as marcas "/*" e "*/" - usadas quando o comentário extrapolar mais do que uma linha. Os números no lado direito, colocados como comentários, representam o número das linhas.

```
//Programa para o cálculo da média de duas idades

#include <stdio.h> //1

int main() //2
{
    int idade1=20, idade2=30; //3
    float media; //4

    media = (idade1 + idade2) / 2.0; //5
    printf("Média = %f", media); //6
    return 0; //7
} //final main //8
```

Desta forma, na codificação acima, temos:

Linha 1: a diretiva "#include" referência a inclusão de um arquivo em tempo de compilação. Este arquivo é denominado: arquivo "*header*" (cabeçalho), e contém definições tais como: o formato de utilização do comando (a ser chamado oportunamente de função) "*printf*". O **stdio** denota "*standard I/O*" (*Standard Input/Output* - Entrada/Saída Padrão).

Linha 2: o processamento do programa deve ser iniciado por uma linha específica. Neste caso, o ponto de partida é denominado como "programa principal" – por isso o nome "**main**" (principal).

Linha 3: o símbolo "{" indica o início de um bloco funcional. Blocos funcionais envolvem as instruções que deverão ser executadas em conjunto, de forma sequencial. A finalização do bloco funcional encontra-se na linha 9 (indicada pelo caractere "}").

Linhas 4 e 5: Essas linhas indicam a definição das variáveis. Na linha 4 temos a definição de 2 variáveis do mesmo tipo (tipo "**int**") – cada variável, dentro de uma lista de um mesmo tipo, é separada por uma vírgula e na linha 5, foi realizada a definição de uma variável do tipo "**float**". O nome de cada variável não pode conter caracteres especiais (como por exemplo: "\$", "@", e acentos) e deve começar por uma letra ou com um traço "*underline*" ("_"). Em C/C++, não há a obrigatoriedade das declarações de variáveis aparecerem antes do início efetivo do código. No momento da criação da variável poderá ocorrer, também, a instanciação inicial, como por exemplo:

```
int x = 0, a = 10;
```

Linha 6: essa linha contém o processamento para o cálculo da média. Nela aparece o comando de atribuição (símbolo "=") responsável por atribuir o resultado do cálculo à variável "media". Nota-se a utilização da representação do denominador como número real (float) para que o resultado não seja arredondado para um inteiro, uma vez que as variáveis utilizadas no numerador são inteiras.

Linha 7: o comando "*printf*" realiza a impressão do resultado. No caso deste exemplo, a impressão será direcionada à tela. A mensagem a ser exibida é descrita entre aspas ("média = %f"). Mas, o que vem a ser o "%f"? O símbolo "%" indica o formato que será exibida a informação. Utilizando-se a letra "f", a variável será exibida no formato de número real (float). Caso o programador desejasse também exibir os valores das idades, poderia usar: *printf*("A média das idades %d e %d é %f", idade1, idade2, media). Neste caso, as idades seriam impressas

no formato decimal inteiro em função da utilização do “%d” na máscara de impressão. Linha 8: retorno do programa principal. Tal retorno, neste caso, é realizado pelo programa ao sistema operacional de modo que este possa saber a causa do término da execução do programa: condição normal de término ou em função de alguma falha em seu processamento.

Focando melhor no comando printf, além das opções “%d” e “%f” existem outros formatos de impressão, alguns estão resumidos no quadro a seguir:

Formato	Descrição	Exemplo de Uso	Saída na Tela
\t	Imprime um caracter de tabulação (“tab”)	printf(“%d\t%d”,x,y);	5 8
\n	Imprime uma nova linha	printf(“%c\n%c”,op1,op2);	A B
\\\	Imprime um caracter “\”	printf(“\\%d”,x);	\5
%%	Imprime um caracter “%”	printf(“%d%%”,x);	5,00%
%c	Caracter	printf(“valor=%c”,op);	valor=A
%d	Decimal (inteiro)	printf(“valor=%d”,op);	valor=65
%i	Equivalente ao %d	printf(“valor=%i”,op);	valor=65
%Ki	Inteiro com K casas à esquerda	printf(“valor=%5i”,x);	valor=5
%f	Ponto flutuante	printf(“valor=%f”,z);	valor=5.340000
.Kf	Ponto flutuante com K casas decimais	printf(“valor=%2f”,z);	valor=4331.72
%e	Notação científica	printf(“valor=%e”,v);	valor=3.3416e+3
%o	Octal	printf(“valor=%o”,k);	valor=1353;
%x	Hexadecimal (minúsculo)	printf(“valor=%x”,k);	valor=2eb
%X	Hexadecimal (maiúsculo)	printf(“valor=%X”,k);	valor=2EB

Quadro 2 - Alguns formatos de impressão na tela associados à máscara do comando “printf”.

Fonte: Elaborado pelo autor, 2019.

Vamos incrementar um pouco nosso exemplo? Veremos a seguir, algumas funcionalidades que poderão ser incorporadas aos nossos programas: tratando-se das estruturas de decisão e de repetição.

1.3. Estruturas de decisão e de repetição

Os programas nem sempre são codificados pensando-se em um fluxo de execução linear, ou seja, às vezes temos que alterar o fluxo de execução em virtude de tomadas de decisão. Essa alteração de fluxo pode ser provocada pelos comandos condicionais e pelos comandos de laços de repetição. Comandos condicionais são aqueles que resultam em uma tomada direta de decisão em função de uma condição a ser verificada. Por sua vez, comandos de laços de repetição servem para executar uma instrução ou um conjunto de instruções inúmeras vezes (de forma repetitiva) enquanto uma certa condição estiver sendo satisfeita. Vamos, a seguir, detalhar esses dois tipos de comandos.

1.3.1 Estruturas de decisão

No nosso cotidiano, temos que tomar, frequentemente, decisões para que as ações sejam consequências das escolhas compatíveis aos objetivos. No caso da programação, não é diferente, ou seja, existem comandos para tomadas de decisão.

Tomemos um exemplo de nosso cotidiano:

```
Se estiver chovendo Então
    Escolher um filme
    Fazer pipoca
    Deitar na cama para ver o filme com pipoca
Senão
    Combinar com os amigos um encontro
    Sair de casa
    Encontrar os amigos no local combinado
```

No trecho acima, aparecem elementos que poderão ser mapeados diretamente em uma linguagem de programação, por exemplo, em C, o condicionante “**Se**” é definido, na linguagem C, pelo comando “**if**” (**Se**) que tem a função de testar uma condição (“**estiver chovendo**”) para que se possa tomar a decisão de executar o bloco contendo as ações, no nosso exemplo do cotidiano, relativas à ficar em casa ou as ações relativas à sair de casa. As ações pertinentes à saída de casa são referenciadas, na linguagem C, pelo comando “**else**” (**Senão**).

Para melhor exemplificar a sintaxe na programação, vamos tomar o exemplo de uma implementação para o cálculo da média de duas notas fornecidas pelo usuário, de modo a informar, posteriormente, se o cursista foi ou não aprovado:

```

//Programa para o cálculo da média de duas notas fornecidas
//pelo usuário

#include <stdio.h>

int main()
{
    float nota1, nota2, media; //1

    printf("Primeira Nota: "); scanf("%f",&nota1); //2
    printf("Segunda Nota: "); scanf("%f",&nota2); //3

    media = (nota1 + nota2) / 2; //4
    if(media >= 6) //5
    {
        printf("Cursista Aprovado!"); //6
    }
    else //7
    {
        printf("Cursista Reprovado!"); //8
    }
    return 0; //9
} //final main //10 //11

```

Na codificação acima, foi introduzido mais um comando de entrada e saída: trata-se do “**scanf**” (linhas 2 e 3). Nas referidas linhas, o programa fornece orientações ao usuário através do comando “**printf**” e espera pela digitação das notas fornecidas pelo usuário cujo formato é passado, ao “**scanf**”, através da diretiva precedida pelo símbolo “%” – no caso, a variável lida pelo teclado é armazenada no formato “**float**”. (MIZRAHI, 2008).

Por sua vez, a linha 4 corresponde ao comando condicional “**if**”, executando as instruções delimitadas pelas marcações de início de final de bloco – símbolos “{” (*begin* – início) e “}” (*end* – fim), respectivamente. Quando o bloco envolver apenas uma única instrução, a utilização dos delimitadores de bloco torna-se opcional. Nota-se, no comando condicional “**if**” a expressão de teste delimitada pelos parênteses. A expressão de teste admite os demais operadores de comparação e operadores lógicos:

- **operadores de comparação:** “>” (maior), “<” (menor), “>=” (maior ou igual), “<=” (menor ou igual), “==” (igual), “!=” (diferente)
- **operadores lógicos:** “||” (*or* – ou), “&&” (*e* – and), “!” (*not* – não). Neste caso, podemos escrever, no código, expressões condicionais compostas, como por exemplo:
 - exemplo 1: **if((a > b) && (c != d))** → neste caso, o “**if**” somente será verdadeiro caso a variável “**a**” seja maior do que “**b**” e, simultaneamente, “**c**” seja diferente em relação à variável “**d**”.
 - exemplo 2: **if((a == b) || (c))** → para esse exemplo, o “**if**” será verdadeiro caso as variáveis “**a**” e “**b**” sejam iguais entre si ou a variável “**c**” seja diferente do valor zero. Para expressar “**c**” diferente de 0, poderia-se, também, usar “**(c != 0)**”.
 - exemplo 3: **if((!a) && (c))** → por fim, neste exemplo, tem-se a execução do bloco do “**if**” somente se a variável “**a**” seja igual a 0 e a variável “**c**” diferente de 0.

Você pode estar se perguntando: Mas, qual a relação entre, por exemplo, “**!a**” com a comparação de “**a**” com o valor 0? Na linguagem C, todo valor diferente de 0 é considerado como “*verdade*” e, consequentemente, o 0 é

considerado como “*falso*”. Sendo assim, caso a variável “*a*” seja 0, a expressão “!0” pode ser tratada como “*não falso*”, ou seja, “*verdade*”.

Comandos condicionais poderão ser manipulados de formas encadeadas ou aninhadas. Neste caso, a condição do senão (“else”) resulta em um novo condicional. Como exemplo, vamos supor as alíquotas de desconto do imposto de renda sobre as faixas salariais:

- até 1.903,98 = isento
- de 1.903,99 a 2.826,65 = 7.5%
- de 2.826,66 a 3.751,05 = 15%
- de 3.751,06 a 4.664,68 = 22.5%
- acima de 4.664,68 = 27.5%

Implementando um programa, usando comandos condicionais encadeados, temos:

```
#include <stdio.h>

int main(void)
{
    float salario, desconto;
    printf("Digite o valor do salario: ");
    scanf("%f",&salario);
    if(salario<=1903.98)
        begin
            desconto=0;
        end
    else if ((salario>=1903.99)&&(salario<=2026.65))
        begin
            desconto=salario*0.075;
        end
    else if ((salario>=2026.66)&&(salario<=3751.05))
        begin
            desconto=salario*0.15;
        end
    else if ((salario>=3751.06)&&(salario<=4664.68))
        begin
            desconto=salario*0.225;
        end
    else
        begin
            desconto=salario*0.275;
        end
    printf("Salario=%.2f\tDesconto IRRF: %.2f\n",salario,desconto);
    return 0;
}
```

No caso do código acima, temos as faixas de desconto do imposto de renda sendo cobertas pelos testes condicionais. A grande vantagem de se usar testes condicionais encadeados está no fato de que quando uma condição for satisfeita, as demais comparações não serão realizadas, diminuindo-se assim, o esforço computacional e, consequentemente, o tempo de processamento. É importante salientar, que esse uso somente é aplicável quando os casos forem excludentes, ou seja, no caso de nosso exemplo, um salário se enquadrará em apenas uma faixa de descontos.

VOCÊ QUER LER?



Os sinais usados como operadores lógicos são também encontrados na manipulação bit-a-bit. Assim, temos, por exemplo: AND Lógico = “`&&`”; AND bit-a-bit=“`&`”. Esses operadores *bit-a-bit* poderão ser encontrados e usados na implementação, por exemplo, de sistemas embarcados, *device drivers* e *kernel* (núcleo) de sistemas operacionais. Maiores detalhes sobre as operações bit-a-bit (bitwise), você pode ler aqui <<https://www.embarcados.com.br/bits-em-linguagem-c/>>. (SOUZA, 2015)

O comando condicional “`if`” também poderá escrito na forma de “*if ternário*”. Por exemplo, suponha o seguinte código:

```
if(a > b)
begin
    c = 10;
end
else
begin
    c = 15;
end
```

No trecho acima, os delimitadores “{“ e “}” (início e fim) foram utilizados porém, pelo fato de que tanto o bloco do “`if`” quanto o bloco do “`else`” conter apenas uma única instrução, poderiam ser omitidos. Podemos notar que a variável “`c`” é instanciada tanto no processamento do “`if`” quanto no processamento do “`else`”. Sendo assim, poderemos reescrever o trecho do código substituindo-o pela estrutura do “*if ternário*”:

```
c = (a > b) ? 10 : 15;
```

No “*if ternário*”, o valor da instanciação relativa à parte do “`if`” é precedida pelo símbolo “`?`” enquanto que a relativa à parte do “`else`” é precedida pelo caracter “`:`”.

VOCÊ SABIA?



Quando temos que selecionar uma ação dentre de várias opções, ao invés de utilizarmos a estrutura “**if...else if...**”, podemos usar o comando “**switch..case**”. Usando o “**switch..case**”, os nossos programas ficam mais organizados e claros para serem entendidos. Para saber mais sobre essa forma, clique aqui <<https://www.embarcados.com.br/comando-de-controle-switch-case/>>.(GATTO, 2016).

O controle de fluxo, alterando a sequência da execução das instruções não é específico aos comandos condicionais, também é pertinente às estruturas de repetição. A seguir, veremos o que são as estruturas ou laços de repetição.

1.3.2 Estruturas de repetição

Os comandos de repetição permitem que um determinado bloco de instruções seja executado por mais de uma vez. A quantidade de vezes que o bloco será executado repetitivamente é definida pela estrutura condicional. Em C/C++, encontramos 3 comandos de repetição: “**for**”, “**while**” e “**do...while**”:

- comando “**for**”: esse comando é utilizado quando se tem, de forma antecipada, a quantidade de vezes que o bloco deverá ser executado. No caso do exemplo do fatorial, sabe-se que a multiplicação “**fat = fat * i**” deverá ser executada *N* vezes. Os campos de configuração, separados por ponto e vírgula são todos opcionais. A seguir, seguem o formato para a utilização do comando “**for**” e alguns exemplos de codificação:

for(valor_inicial; condição_manutenção; variação_variável)

Exemplos:

- **for(i = 0; i < 10; i++)** → neste caso, o bloco será executado por 10 vezes, incrementando-se, automaticamente, a variável “*i*” a cada iteração.
- **for(i=0, j = 10; (i<5) && (j>0); i++, j→)** → cada campo do comando “**for**” poderá ter vários itens separados por vírgulas. Neste caso, ocorrem as instanciações das variáveis “*i*” e “*j*” para que possam ser incrementadas e decrementadas a cada interação, respectivamente. As repetições serão finalizadas quando a expressão lógica “(*i*<5)&&(*j*>0)” assumir um valor “**falso**”.
- **for(; x != y; x+=2)** → para esse caso, não existe uma instanciação de variável no início do laço de repetição, ou seja, as variáveis serão manipuladas com os seus valores atuais.
- **for (;**) → esse último caso, sem qualquer conteúdo dos campos do comando “**for**”, denota um laço infinito.
- comando “**while**”: o comando “**while**” realiza a execução do bloco de instruções a ele vinculado enquanto a sua expressão de permanência resultar em um valor “verdade”. Neste caso, utiliza-se esse comando quando não se sabe, de antemão, a quantidade de vezes que o bloco deverá ser executado. Igualmente aos

comandos “if” e “for”, a expressão pode ser composta, com os itens separados por operadores lógicos. Diferentemente ao “for”, esse comando não realiza os incrementos automáticos das variáveis ficando, neste caso, o programador responsável pelo controle do laço.

- comando “do...while”: comando semelhante do “while” porém, o teste de permanência do laço é realizado após, pelo menos, uma interação. Sendo assim, esse comando “executa e testa” enquanto que o “while” realiza a sequência “testa e executa”. A expressão de permanência do laço de repetição também pode envolver operadores lógicos.

Para uma melhor abstração, tomemos por exemplo, a figura abaixo que mostra 3 versões, para calcular o valor do factorial de um certo número; cada versão usando um comando de repetição:

```
#include <stdio.h>

int main()
{
    int N, fat=1, i=1;

    printf("N: ");
    scanf("%d", &N);

    if(N < 0)
        printf("Não existe fat de num. negativo");
    else
    {
        if(!N)
        {
            fat=1;
        }
        else
        {
            for(;i<=N;i++)
                fat = fat * i;
        }
        printf("Valor do factorial de %d é %d",
               N, fat);
    } //final else
    return 0;
}
```

(a) Usando “for”

```
#include <stdio.h>

int main()
{
    int N, fat=1, i=1;

    printf("N: ");
    scanf("%d", &N);

    if(N < 0)
        printf("Não existe fat. de num. negativo");
    else
    {
        if(!N)
            fat=1;
        else
        {
            do
            {
                fat = fat * i;
                i++;
            } while(i <= N);
        }
        printf("Valor do factorial de %d é %d",
               N, fat);
    }
    return 0;
}
```

(b) Usando “do...while”

```
#include <stdio.h>

int main()
{
    int N, fat=1, i;

    printf("N: ");
    scanf("%d", &N);

    if(N < 0)
        printf("Não existe fat. de num. negativo");
    else
    {
        i=N;
        while(i)
        {
            fat = fat * i;
            i--;
        }
        printf("Valor do factorial de %d é %d",
               N, fat);
    }
    return 0;
}
```

(c) Usando “while”

Figura 2 - Três versões de código para o cálculo do fatorial de um número: em (a), usando o comando “for”. Em (b) cálculo utilizando o comando “do..while” e, em (c) usando o “while”.

Fonte: Elaborada pelo autor, 2019.

Na figura acima podemos ver a utilização dos comandos de repetição encontrados na linguagem C: os comandos “**for**”, “**while**” e o “**do..while**”. Perceberemos, entre (b) e (c), a não necessidade do teste condicional, comparando o “N” com o valor 0, uma vez que o “**while**” realiza o teste de permanência do laço antes da primeira execução; diferente do comando “**do..while**” que realiza pelo menos uma execução para depois efetuar o teste.

CASO

Atualmente, para o desenvolvimento de sistemas, encontramos dois grandes grupos: as linguagens puramente estruturadas (como por exemplo, o *C Standard*) e as linguagens orientadas a objetos (como o *C++* e o *Java*). Um certo desenvolvedor de sistemas embarcados (sistemas onde a programação é, geralmente, voltada para um micro controlador) se deparou com o dilema na escolha do paradigma. Começou a ponderar alguns pontos, tais como: tamanho de código gerado (sistemas embarcados possuem grande limitação de memória) e necessidade de ter o máximo controle sobre a manipulação de memória e ter a possibilidade de otimizar ao máximo o código em função das particularidades do dispositivo a ser programado (a orientação por objetos, em seu encapsulamento de código, torna difícil atender a esses requisitos). Analisou que, mesmo com as facilidades de desenvolvimento, manutenção e portabilidade do paradigma orientado a objetos, os dois primeiros pontos foram decisivos na adoção de programação estruturada utilizando-se, para tal, a linguagem *C*. Moral da história: mesmo com todas as facilidades proporcionadas pelas linguagens, por exemplo, *C++* e *Java*, juntando com as facilidades dos ambientes integrados de desenvolvimento, é sempre bom sabermos uma linguagem que temos maior controle de como as operações estão sendo feitas, principalmente nos níveis mais baixos de abstração (próximos ao *hardware*). Sendo assim, é conveniente que se saiba linguagens do tipo “*C Standard*” ou “*C ANSI*”.

Os comandos de laço de repetição e condicionais são a base da lógica de programação. Sendo assim, um programa é uma sequência de instanciações e controles de fluxo de execução. Particularmente, os comandos de repetição são extremamente úteis quando manipulamos variáveis cujas estruturas são denominadas como vetores. É o que veremos a seguir.

1.4. Vetores

Em algumas situações, temos que manipular um grande número de informações do mesmo tipo, por exemplo, supondo que devemos armazenar as notas de uma certa prova de uma turma inteira de cursistas. Vamos voltar ao exemplo da média, abordado anteriormente, havíamos manipulado apenas duas notas, gerando, para isso, duas variáveis. Você deve se perguntar: Se para duas notas fornecidas pelo usuário usamos duas variáveis, então, para, por exemplo, 40 notas, deveremos usar 40 variáveis? A resposta tornará a solução mais simples: usaremos um vetor para armazenar as notas. Mas, então, o que é um vetor?

Vetor é uma estrutura computacional para o armazenamento de dados homogêneos, ou seja, todas as informações do vetor possuem a mesma tipagem. Para definirmos um vetor, em C/C++, usaremos a seguinte sintaxe:

```
<tipo> nome_da_variável [tamanho_do_vetor];
```

Por exemplo, se desejarmos, no caso da nota, a manipulação de 8 itens, teremos:

```
float notas [8];
```

Para manipularmos um vetor, teremos que usar um índice que indicará qual a célula a ser acessada. A figura a seguir ilustra a estruturação do vetor de oito posições, cujas células armazenarão valores do tipo “**float**”.

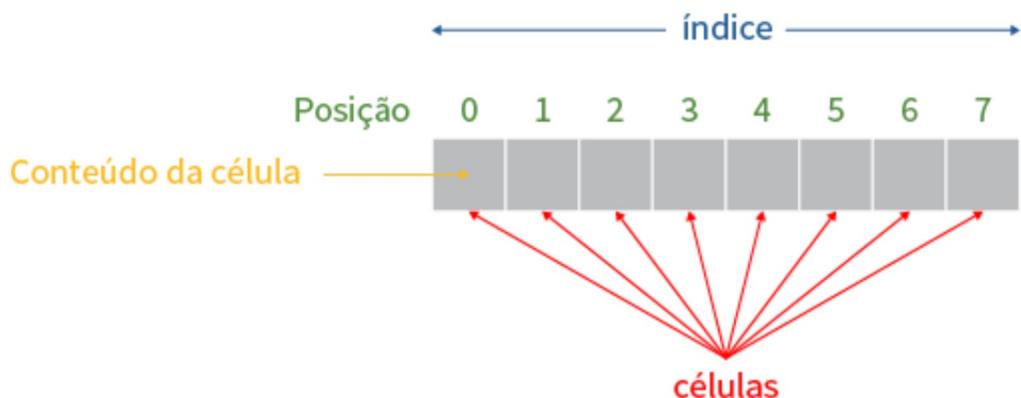


Figura 3 - Estruturação de um vetor de 8 posições. As posições são acessadas por um índice que varia da posição 0 até a posição (N-1) – no caso deste exemplo, até a posição 7.

Fonte: Elaborada pelo autor, 2019.

Na figura acima, temos a estruturação de um vetor. Mas, como efetivamente declarar um vetor, preenchê-lo e utilizarmos as informações de suas células? Vamos adaptar o exemplo da média das notas para que a média seja calculada sobre um conjunto de 8 notas:

```

//Programa para o cálculo da média de 8 notas
//fornecidas pelo usuário - usando vetor

#include <stdio.h>

int main()
{
    float notas[8], media=0; //1
    int indice; //1

    for(indice = 0; indice < 8; indice++) //2
    {
        printf("Nota %d: ", indice); //3
        scanf("%f", &notas[indice]); //4
    }

    for(indice = 0; indice < 8; indice++) media += notas[indice]; //5

    media = media / 8; //6
    printf("Media final: %.2f", media); //7
    return 0;
}

```

Na codificação acima, para o cálculo da média de 8 notas, temos:

- Linha 1: criação do vetor de notas com 8 posições de “**float**” e também, criação da variável que receberá o cálculo da média já a atribuindo o valor 0 devido ao fato de que ela acumulará a somatória das notas em cada iteração do laço de repetição.
- Linha 2: definição do laço de repetição manipulando a variável de índice para o acesso das células do vetor. O índice varia de 0 até 7 em função do tamanho do vetor criado.
- Linha 3: apenas uma mensagem de orientação ao usuário de forma que ele forneça a nota da iteração corrente. A impressão na tela se dará na forma, por exemplo: “Nota 3:”.
- Linha 4: linha responsável pelo recebimento do valor através da digitação do usuário. Nota-se que será preenchida a célula do vetor cuja posição é apontada pelo índice. Desta forma, tem-se genericamente, que a atribuição do valor à uma célula do vetor é feita na forma “**vetor[i] = valor**”, onde a variável “*i*” desempenha a função de índice do vetor. O caractere “&” faz parte do uso da função “**scanf**”.
- Linha 5: responsável pela somatória das notas. Em função da complexidade do programa, essa somatória poderia ser colocada no mesmo laço de repetição da entrada, porém, foi colocada em um laço à parte para permitir uma melhor abstração. Para realizar a somatória, é acessada a célula correspondente à interação. Desta forma, tem-se genericamente, que o acesso ao valor de uma célula do vetor é feito na forma “**variável = vetor[i]**”. Na mesma linha, aparece a expressão:

```
media +=notas[indice];
```

Que equivale à:

```
media = media + notas[indice];
```

- Linha 6: essa linha finaliza o cálculo da média dividindo-se a somatória obtida no laço de repetição pelo número de valores fornecidos.
- Linha 7: nesta linha, existe um elemento novo na formatação do “**printf**”: o uso do “.*2*” entre o símbolo “%” e o caractere que indica o formato de impressão “*f*” (float). Com isso, formata-se a impressão para dois dígitos após o ponto, realizando um arredondamento do valor.

Caso haja a necessidade de usar estruturas com mais de uma dimensão, por exemplo, matrizes, como proceder? A forma de manipulação de matrizes é análoga à dos vetores, adicionando mais uma dimensão e, consequentemente, trabalhando com mais um índice: um para percorrer as linhas e outro para percorrer as colunas. O trecho de código a seguir exemplifica uma criação e instanciação de uma matriz de dimensões 5x4:

```
    . . .
float matriz[5][4];
int i,j;
    . . .
for(i = 0; i < 5; i++)
    for(j = 0; j < 4; j++)
        {
            matriz[i][j] = 0;
        }
    . . .

```

Existe um tipo especial de vetor que é representado pelas “strings”. Sendo assim, uma “string” nada mais é do que uma sequência de caracteres. Veremos, a seguir, um pouco de manipulação de “strings”.

1.5. *Strings*

mo mencionado anteriormente, as *strings* são uma sequência de caracteres, ou seja, um vetor onde cada célula armazenará uma letra (tipo **char** – caractere). A seguir, são relacionadas algumas das principais funcionalidades para a manipulação de *strings* da linguagem *C*.

- **Criação, leitura e impressão de string:** A criação é feita através de um vetor de caracteres (tipo **char**).

No código a seguir, na função “**scanf**” não foi utilizado o “&” junto à variável “**nome**” pelo fato da *string* ser tratada como uma posição de memória.

Exemplo:

```
#include <stdio.h>
    . . .
char nome[20];
    . . .
printf("Digite um nome: ");
scanf("%s", nome);
printf("%s", nome);
    . . .

```

- **Tamanho da String:** o tamando de uma *string* é retornado através da função **strlen** (*C Standard*). Neste caso e, também, no caso da utilização de outras funções que manipulam *strings*, faz-se necessário incluir o arquivo *header* “*string.h*”.

Exemplo:

```
#include <stdio.h>
#include <string.h>
.
.
.
char nome[20];
int tam;
.
.
.
tam = strlen(nome);
printf("Tamanho: %d", tam);
.
.
```

- **Concatenação:** concatenar significa unir duas *strings*. No caso da linguagem *C*, essa funcionalidade é feita pela função **strcat**. O resultado da concatenação é atribuído ao primeiro parâmetro da função, no caso do código exemplo, à variável “**texto**”.

Exemplo:

```
#include <stdio.h>
#include <string.h>
.
.
.
char texto[20] = "O nome eh",
      nome[20] = "abcdefg";
.
.
.
strcat(texto, nome);
printf("==> %s", texto);
.
```

- **Concatenação com quantidade específica de caracteres:** semelhante à função “**strcat**” porém concatena, ao destino, os *N* primeiros caracteres da variável fonte. No caso do exemplo ao lado, os 5 primeiros caracteres de “**nome**” são concatenados à variável “**texto**”.

Exemplo:

```

#include <stdio.h>
#include <string.h>
.
.
.
char texto[20] = "O nome eh",
      nome[20] = "abcdefgh";
.
.
.
strncat(texto,nome,5);
printf("==> %s",texto);
.
.
.
```

- **Cópia:** a cópia de *strings*, em *C*, é feita pela função **strcpy**. Na função “**strcat**”, o primeiro parâmetro (destino) representa a *string* que receberá o resultado da cópia da informação contida no segundo parâmetro (fonte).

Exemplo:

```

#include <stdio.h>
#include <string.h>
.
.
.
char texto[20], nome[20];
.
.
.
strcpy(texto,nome);
printf("==> %s",texto);
.
.
```

- **Cópia com quantidade específica de caracteres:** análogo à função “**strcpy**”, porém realiza a cópia dos *N* primeiros caracteres da variável fonte. No exemplo ao lado, a variável “**texto**” será instanciada com os 5 primeiros caracteres da variável “**nome**”.

Exemplo:

```

#include <stdio.h>
#include <string.h>
.
.
.
char texto[20], nome[20];
.
.
.
strncpy(texto,nome,5);
printf("==> %s",texto);
.
.
```

- **Comparação:** para se comparar duas *strings*, em *C* usa-se a função **strcmp**. Quando as duas *strings* forem iguais, a função “**strcmp**” retorna o valor 0.

Exemplo:

```

#include <stdio.h>
#include <string.h>
...
char txt1[10] = "abcdef",
     txt2[10] = "abcdef";
...
if(!strcmp(txt1,txt2))
    printf("Iguais!");
else
    printf("Diferentes!");
...

```

- **String para número:** Em C, a conversão é feita mediante a função `atoi` (*ASC TO Integer*). Neste caso, faz-se necessário incluir o arquivo header “`stdlib.h`”.

Exemplo:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
...
char texto[5];
int valor;
...
valor = atoi(texto);
printf("==> %d", valor);
...

```

- **Número para string:** Em C, a conversão é feita pela função `itoa` (*Integer TO ASC*). Como parâmetros da função “`itoa`”, temos, na ordem da esquerda para a direita: o valor numérico a ser convertido; a *string* que receberá a conversão; por último, a base numérica sobre a qual será a conversão. No caso do exemplo, o valor 10 denota a base decimal – caso tivesse, por exemplo, o valor 2, a conversão seria baseada na base binária.

Exemplo:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

...
char texto[5];
int valor=123;

...
itoa(valor, texto, 10);
printf("==> %s", texto);
...

```

Quando definir uma *string* por meio do vetor de caracteres, deve-se sempre dimensionar a variável com um espaço a mais em relação à demanda do programa. Esse espaço a mais é usado para colocar o caractere terminador de *string* (carácter '\x0' – carácter nulo ou caractere 0 – zero). O carácter terminador de *string* serve para indicar o término de uma *string*. Neste caso, por exemplo, a função “**printf**” executa a impressão dos caracteres de uma *string* até encontrar o carácter nulo. Sendo assim, aproveitando o fato de que toda *string* é finalizada com esse carácter, para limpar o conteúdo de uma *string*, basta:

```
texto[0] = '\x0'; // texto[0]=0;
```

Na linha de código acima, temos duas formas de referenciar o carácter nulo: através de seu valor na tabela ASCII ou através da simbologia que o representa. A figura a seguir ilustra o formato de uma *string* sendo finalizada pelo carácter '\x0'.

```

char str[6]  A  E  I  O  U  '\x0'

```

Figura 4 - Formato de uma string de 6 posições destacando que a sua última posição deve conter, necessariamente, o carácter finalizador de string ('\x0').

Fonte: Elaborada pelo autor, 2019.

Analizando-se a figura acima, convém então, reforçar que uma *string* deve necessariamente ser finalizada com o carácter '\x0'.

VOCÊ QUER VER?



Saber manipular *strings* é de suma importância no contexto de desenvolvimento de sistemas computacionais, pois as encontraremos em diversas situações. Por exemplo, uma simples falta do caractere '\x0' pode acarretar em travamento do programa quando em execução. Você pode acessar este link <<https://www.youtube.com/watch?v=L4IYcJSoH3M>>, para ver um pouco de manipulação de *strings* na prática. (LOBATO, 2012).

Até o momento, tanto nos vetores quanto nas *strings*, falamos sobre o tipo homogêneo, capaz de armazenar, em sua estrutura, valores de apenas um tipo. Mas, será possível armazenar, em uma variável, dados heterogêneos, ou seja, haver vários tipos de informações centradas em apenas uma variável? Sim, isso é possível através da utilização das estruturas de dados, em C esse tipo é representado pela "struct".

Síntese

Chegamos ao fim de nossa primeira conversa sobre técnicas de programação. Tivemos a oportunidade de conhecer alguns conceitos iniciais e comandos básicos de uma linguagem estruturada, com estas informações já será possível o desenvolvimento de programas computacionais escritos em C. Porém, para desenvolver um programa, não basta apenas entender a linguagem de programação, mas também, treinar bastante o raciocínio lógico de forma a facilitar a otimização dos códigos a serem criados. A partir das informações contidas neste capítulo, esperamos que você continue treinando e incrementando os seus programas computacionais de forma a deixá-los mais funcionais, corretos e eficientes.

Nesta unidade, você teve a oportunidade de:

- ter contato com conceitos de linguagens de programação estruturadas;
- utilizar os tipos de dados de forma apropriada;
- analisar e empregar corretamente aos comandos condicionais e de repetição;
- estruturar as informações com vetores.

Bibliografia

ASCENCIO, A. F. G. **Fundamentos de Programação de Computadores**: Algoritmos, PASCAL, C/C++ (Padrão ANSI) e Java. 3. ed. São Paulo: Pearson Education do Brasil, 2012. Disponível em: <<https://laureatebrasil.blackboard.com/webapps/ga-bibliotecaSSO-BBLEARN/homePearson>>. Acesso em: 27/06/2019.

DEITEL, P. J.; DEITEL, H. C. **Como Programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011. Disponível em: <<https://laureatebrasil.blackboard.com/webapps/ga-bibliotecaSSO-BBLEARN/homePearson>>. Acesso em: 27/06/2019.

GATTO, E. C. **Comando de Controle Switch Case**. 2016. Disponível em: <<https://www.embarcados.com.br/comando-de-controle-switch-case/>>. Acesso em: 05/07/2019.

- LOBATO, P. **Programação C** – Strings (char, strlen, strcmp) – Inverso, Palíndromo. Aula 11. 2012. 18 min. Disponível em: <<https://www.youtube.com/watch?v=L4IYcjSoH3M>>. Acesso em: 05/07/2019.
- MAGNUN. **Um Ano sem Dennis Ritchie**. 2012. Disponível em: <<http://mindbending.org/pt/um-ano-sem-dennis-ritchie>>. Acesso em: 01/07/2019.
- MIZRAHI, V. V. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008. Disponível em: <<https://laureatebrasil.blackboard.com/webapps/ga-bibliotecaSSO-BBLEARN/homePearson>>. Acesso em: 27/06/2019.
- PUGA, S.; RISSETTI, G. **Lógica de Programação e Estruturas de Dados – com Aplicações em Java**. 3 ed. São Paulo: Pearson Education do Brasil, 2016. Disponível em: <<https://laureatebrasil.blackboard.com/webapps/ga-bibliotecaSSO-BBLEARN/homePearson>>. Acesso em: 27/06/2019.
- SOUZA, F. **Bits em Linguagem C – Conceito e Aplicação**. 2015. Disponível em: <<https://www.embarcados.com.br/bits-em-linguagem-c/>>. Acesso em: 01/07/2019.