Algoritmos 1

Conceitos Básicos

Prof. André Luiz Marasca

1 Palavras reservadas

Em todas as linguagens de programação existem palavras que são reservadas para uso da linguagem, essas palavras tem como objetivo cumprir alguma função especifica. **Essas palavras não podem ser usadas para nomear variáveis, porque são reservadas para gramática da linguagem.**

Abaixo são apresentadas algumas das palavras reservadas da linguagem C que serão utilizadas na disciplina de **Algoritmos 1**. Lembrando que existem outras palavras reservadas, serão mostradas apenas as que serão utilizadas nesta disciplina.

char: Tipo de dados utilizado para armazenar caracteres.

float: Tipo de dados de ponto flutuante (com virgula) com precisão simples.

int: É o tipo de dados mais usado para armazenar valores inteiros.

double: Tipo de dados numérico de ponto flutuante com precisão dupla.

short: É o chamado inteiro curto. Armazena valores inteiros com precisão menor (apenas 2 bytes) do que o tipo int.

long: É um tipo de dados usado para armazenar valores inteiros que possui precisão superior ao tipo int.

unsigned: Faz com que a variável apenas aceite valores positivos ou zero.

void: Comando que indica que a função não retorna nada ou que não tem parâmetros de entrada.

return: Retorna um valor dentro de uma função ou força o abandono da mesma.

sizeof: Comando usado para retornar o tamanho em bytes de um determinado tipo de dados.

if: Comando condicional que altera o fluxo do programa de acordo com uma condição que pode ser verdadeira ou falsa.

else: Indica um bloco de comandos a ser executado quando a condição do comando if for falsa.

switch: Comando de seleção usando em conjunto com o comando case, permite escolher entre várias opções.

case: Utilizado dentro do comando switch para selecionar uma constante.

break: Comando para forçar a saída imediata dos comandos switch, for, while, e , do…while.

default: É utilizado dentro de switch…case para tratar valores não definidos anteriormente nas opções case.

for: Estrutura de repetição que utiliza condições e contador.

do: Estrutura de repetição usada em conjunto com o while . O comando do…while faz com que os comandos do bloco a ser repetido sejam executados no mínimo uma vez.

while: Estrutura de repetição que executa enquanto uma condição é verdadeira.

2 Bibliotecas

Uma das funcionalidades das linguagens de programação modernas são as bibliotecas, nelas contém uma série de funções uteis já prontas para o uso do programador. Assim, o programador não precisa se preocupar com aspectos básicos do seu programa, como cálculo de raiz quadrada.

Na linguagem C, para utilizar uma biblioteca deve-se declara-las no código, isso deve ser as primeiras coisas a serem feitas no código. Ou seja, declarar cada biblioteca que será utilizada uma após a outra, cada uma em sua linha, da seguinte forma:

#include <nome_de_uma_biblioteca.h>

#include <nome_de_outra_biblioteca.h>

Algumas das bibliotecas que utilizaremos na disciplina de **Algoritmos 1** será listada abaixo, bem como as funções mais utilizadas em cada uma delas.

2.1 stdio.h

A biblioteca stdio (standard input output), tem como objetivo dispor para uso do programador as funções de entrada e saída padrão. Ou seja, funções que manipulam o teclado e a tela do computador.

Dentre as funções mais utilizadas desta biblioteca estão:

2.1.1 printf

printf: Esta função serve para imprimir algum texto na tela, é uma função que permite muitas formas de mostrar os resultados obtidos durante a execução do seu programa, bem como números inteiros, números com vírgula, frases, notação cientifica, etc.

Para utilizar essa função, deve-se chama-la no código da seguinte forma:

printf(“Olá Mundo!\n”);

Onde escreve-se o nome da função, abre-se parênteses, e dentro de aspas duplas deve-se conter a frase que será utilizada para imprimir na tela.

Porém, pode-se utilizar o printf para imprimir o conteúdo de variáveis também:

printf(“O valor eh: %d”, minha_variavel);

Observe que neste caso, temos %d na frase que queremos imprimir na tela, porém não será impresso %d na tela, mas sim o conteúdo da variável **minha_variavel**, ou seja, %d será substituído pelo valor contido dentro desta variável. Lembrando que %d é um especificador de formato, refere-se a variáveis do tipo inteiro **int**. A forma de usar um especificador de formato é a seguinte:

%[tamanho][.precisão][especificador]

Não é necessário utilizar tamanho e precisão, apenas se queremos forçar a saída com um determinado formato, por exemplo: quero 2 dígitos de pois da virgula: **%.2f** assim o tipo de dado é **float** devido ao **f** e tem 2 dígitos [.2] de precisão, neste caso não foi definido o [tamanho].

Segue abaixo uma lista dos especificadores de tipos:

Tabela 1- Lista de especificadores de tipo

Especificador	Uso	Exemplo
d	Variáveis do tipo **int**	-123456
u	Variáveis do tipo **unsigned int**	12356
f	Variáveis do tipo **float**	1.456
lf	Variáveis do tipo **double**	1.4564564654
ld	Variáveis do tipo **long int**	4294967296
lld	Variáveis do tipo **long long int**	4294967296
c	Variáveis do tipo **caractere**	a
s	Vetores de **caractere**	Brasil

[tamanho]: substituir por um número, refere-se ao número mínimo de digitos a serem impressos. Se o valor a ser impresso for menor do que esse número, o resultado será preenchido com espaços em branco. O valor não é truncado, mesmo que o resultado seja maior

[.precisão]: Para especificadores inteiros: precisão especifica o número mínimo de dígitos a serem escritos. Se o valor a ser escrito é menor do que este número, o resultado é preenchido com zeros a esquerda. O valor não é truncado, mesmo que o resultado seja maior. Uma precisão de 0 significa que nenhum caractere é escrito para o valor 0. Para especificadores float e double: este é o número de dígitos a imprimir após o ponto decimal (por padrão, isso é 6).

Comandos de tabulação e quebra linha: Para quebrar linha utiliza-se o comando **\n** e para dar uma tabulação utiliza-se o caractere **\t**.

Exemplos de uso:

Comando	Saída
printf(“%d-%d-%d”,1, 2, 1999);	1-2-1999
printf(“Vm: %f”, velocidade_media);	Vm: 1.123165
printf(“%d bananas custam %.2f reais\n”, 4, 6.9874);	4 bananas custam 6.99 reais

Note que depois das aspas duplas vem uma virgula, e depois da virgula vem o primeiro valor que será substituído pelo primeiro especificador. Em sequência vem outra virgula, e o valor que será substituído pelo segundo especificador, a assim por diante.

2.1.2 scanf

A função scanf é utilizada para leitura de caracteres digitados no teclado. Essa função segue o formato do printf, porém não se deve definir tamanho nem precisão do número a ser lido, apenas o especificador, conforme a Tabela 1, o formato a ser utilizado é:

scanf("%[especificador]", &minha_variavel);

Para variáveis várias variáveis inteiras

scanf("%d %d %d %d", &var1, &var2, &var3, &var4);

Para variáveis várias de tipos diferentes

scanf("%d %ld %f %lf", &var1, &var2, &var3, &var4);

3 math.h

Essa é uma biblioteca para operações matemáticas, algumas operações de baixo nível estão implementadas nesta biblioteca, como raiz quadrada, potência de um número, funções trigonométricas, funções de arredondamento para cima e para baixo, logaritmos, etc. Na tabela abaixo são apresentadas algumas chamadas de funções da math.h:

Tabela 2 - Algumas funções da math.h

Objetivo	**Código em C**
Raiz quadrada	y = sqrt(x);
Potencia x^y	z = pow(x,y);
Arredonda para baixo	y = floor(x);
Arredonda para cima	y = ceil(x);
Arredonda	y = round(x);

4 Função principal

A função principal de programa em C (main) é onde tudo começa, ao executar um programa em C, o computador irá executar comandos em sequência, linha após linha começando na primeira linha da função principal.

A sintaxe da função principal é a seguinte:

```
int main (void)
{
      return 0;
}
```

A função é do tipo **int** porque ela retorna valor 0, isso é importante para saber se o programa terminou de executar corretamente, caso ela não retorne 0, significa que algo interrompeu a execução da função principal, indicando um erro no código.

(void) significa que ela não recebe parâmetros de entrada, ou seja, começa a executar do zero na primeira linha após as chaves.

5 Variáveis

Em linguagem C, se é necessário fazer uso da memória, então deve-se declarar variáveis. A declaração de variáveis serve para o programa saber qual é o tamanho usado na memória por esta variável, e também para computador saber qual o tipo de operação deve ser feito sobre as variáveis, se for int então o processador usará operações int, se for float o processador irá usar operações float.

A tabela abaixo contém os tipos de dados da linguagem C, comparando o tamanho de cada tipo com o intervalo de valores aceitos.

Tabela 3 - Tabela dos tipos de dados em C

Tipo de dado	Significado	Tamanho (em bytes)	Intervalo de valores aceitos
char	Caractere	1	- de 128 a 127
unsigned char	Caractere sem sinal	1	0 à 255
short int	Inteiro curto	2	-de 32.768 a 32.767
unsigned short int	Inteiro curto sem sinal	2	de 0 a 65.535
int	Inteiro	2 (no processador de 16 bits) 4 (no processador de 32 bits)	-de 32.768 a 32.767 - de 2.147.483.648 a 2.147.483.647
unsigned int	Inteiro sem sinal	2 (no processador de 16 bits) 4 (no processador de 32 bits)	de 0 a 65 535 de 0 a 4.294.967.295
long int	Inteiro longo	4	de -2.147.483.648 a 2.147.483.647
unsigned long int	Inteiro longo não assinado	4	de 0 a 4.294.967.295
float	Flutuante (real)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
double	Flutuante dupla precisão	8	de $1.7*10^{-308}$ a $1.7*10^{308}$

Para se declarar uma variável do tipo **int** como nome **x** faz-se:

int x;

Caso seja necessário mais variáveis do tipo int, pode-se colocar uma virgula e inserir outros nomes de variáveis:

int x, y, z;

Ou ainda, pode-se colocar em novas linhas:

int x;

int y;

int z;

Caso seja necessárias novas variáveis de outros tipos, deve-se necessariamente colocar em outras linhas, por exemplo:

int x, y, z;

float a, b, c;

double u, v, w;

char j, k, w, teste, minha_variavel;

5.1 Regras para declaração de variáveis

Algumas regras devem ser seguidas para declarar variáveis, caso elas não forem seguidas o compilador irá dar erro, e avisar o problema, as regras são:

- Sempre declarar variáveis linhas acima de onde a mesma está sendo usada.
- Nunca utilizar palavras reservadas da linguagem C como nome de variáveis.
- Nunca utilizar nome de funções como nome de variáveis.
- Pode usar letras, números e underscore “_” para dar nome a variáveis.
- O nome de uma variável nunca poderá começar com um número.

6 Operadores aritméticos

As operações mais básicas em C, como soma, subtração, divisão e multiplicação não precisam de função para serem realizadas, são usados operadores. A tabela abaixo lista alguns dos operadores em C:

O que faz	Operador	Como usar
Multiplicação	*	a * b
Divisão	/	a / b
Soma	+	a + b
Subtração	-	a – b
Resto da divisão	%	a % b
Soma 1 em a	++	a++ ou ++a
Decrementa 1 em a	--	a-- ou --a
Incrementa x em a	+=	a += x
Decrementa x em a	-=	a -= x
Multiplica a por x	*=	a *= x
Divide a por x	/=	a /= x

Abaixo alguns exemplos mais complexos:

x = (a + b) / c + d * e;

x = (a + b) * (c + d) * (e + f);

Lembrando que a ordem de precedência de operadores também é relevante, como na matemática, multiplicação e divisão tem a mesma precedência, e tem mais precedência que soma e subtração. O resto da divisão tem menor precedência que as quatro operações citadas anteriormente. O incremento ou decremento em uma unidade tem a menor precedência. Caso seja necessário calcular o decremento com a maior prioridade de todas utilizar --a ao invés de usar a--, isso também vale pra ++a, que tem maior prioridade enquanto a++ tem menor prioridade.

7 Começando a programar

Acima foram apresentados os conceitos iniciais de programação em linguagem C. Os quais são suficientes apenas para programação linear, sem estruturas de decisão ou repetição. Mas com isso já podemos resolver alguns problemas simples, como serão exemplificados a seguir.

Código que imprime na tela a mensagem **Hello World!**

```c
#include <stdio.h>

int main (void)
{
    printf("Hello World!\n");

    return 0;
}
```

Código para mostrar o resultado de uma operação matemática qualquer:

```c
#include <stdio.h>

int main (void)
{
    int y, x = 4;

    y = x * x - x - 12;

    printf("y vale %d\n", y);

    return 0;
}
```

O programa abaixo lê um número de pessoas e um número de balas e divide igualmente as balas entre as pessoas e posteriormente informa quantas balas sobraram:

```c
#include <stdio.h>

int main (void)
{
    int n_pessoas, n_balas;
    int divisao, resto;

    printf("Informe o numero de pessoas e de balas: ");
    scanf("%d %d", &n_pessoas, &n_balas);

    divisao = n_balas / n_pessoas;
    resto = n_balas % n_pessoas;

    printf("Cada pessoa ficou com %d balas\n", divisao);
    printf("Depois de dividir, sobraram %d balas\n", resto);

    return 0;
}
```

Exemplo de execução:

```
Informe o numero de pessoas e de balas: 6 9
Cada pessoa ficou com 1 balas
Depois de dividir, sobraram 3 balas

Process returned 0 (0x0)   execution time : 1.914 s
Press any key to continue.
```

8 Typecasting

Na linguagem C, ao efetuar uma operação de divisão com dois números inteiros estamos previamente dizendo ao processador que o resultado deve ser um número inteiro. Porém nem sempre isso é desejado, por exemplo, é necessário saber quanto vale 1/8, se o tipo de operação for inteiro, o processador irá retornar valor 0 para o resultado que deveria ser 0.125, ou seja, o processador irá **truncar** o resultado.

Caso desejemos realizar uma operação, seja ela qual for, e precisamos que o resultado seja convertido para outro tipo de dado (como é o caso da divisão de números **int**, em que desejamos que o resultado seja do tipo **float)**, precisamos realizar o **typecasting**, por exemplo:

```
int a, b;
float res;
res = a / (float) b;
```

Ou seja, insere-se na frente da variável desejada o nome do tipo de dado que a variável deve ser convertida, dentro de parênteses (tipo).

Caso a divisão seja utilizando uma constante, pode-se realizar outro artificio, no lugar de dividir por 10, pode-se dividir por 10.0, pois 10 é **int**, e 10.0 é **float**, por exemplo.

```
res = a / 10.0;
```

9 Conetivos lógicos

Uma estrutura de decisão tem como objetivo mudar o fluxo de um programa baseado em uma condição lógica.

Antes de entendermos como mudar o fluxo de execução de um programa usando uma estrutura de decisão é necessário entender as operações lógicas.

Na lógica proposicional, existem vários conetivos lógicos, aqui o que nos interessam são os conectivos **e** (&&) e **ou** (||).

9.1 **Conectivo** e

Assim como estamos acostumados a ouvir, no dia a dia, o **e** tem como objetivo interligar duas proposições, de maneira que as duas devem ser verdadeiras para que a sentença seja verdadeira, por exemplo:

Hoje no almoço, comi carne **e** arroz.

Caso seja mentira que esta pessoa tenha comido arroz, por exemplo, a frase não será verdadeira. Resumindo, para o && ser verdadeiro, ambas as proposições devem ser verdadeiras.

9.2 **Conectivo** ou

O conetivo ou usado na programação é ligeiramente diferente do que estamos acostumados a usar no dia a dia, aqui, o conectivo ou serve como ou\e. Ou seja, a frase só será falsa se ambas as proposições forem ao mesmo tempo falsas. Por exemplo:

Hoje no almoço, comi carne **ou** arroz.

Caso essa pessoa comeu arroz, mas não carne, a sentença ainda será verdadeira. Caso ela comeu carne, mas não arroz, a sentença ainda será verdadeira. Caso ela comeu arroz e carne, a sentença ainda será verdadeira. No entanto, se ela não comeu nem arroz, nem carne, esta sentença será falsa.

9.3 Conectivo NÃO, negação

O conectivo NÃO, ou conectivo de negação é usado para inverter o valor lógico de uma proposição.

Por exemplo: P = A porta está aberta. Então, !P, representa a porta não está aberta.

9.4 Tabela verdade

O que foi dito acima pode ser resumido utilizando o que chamamos de tabela verdade. Por exemplo, assuma que: P = Hoje fez chuva; Q = Hoje fez sol; V = verdadeiro; F = falso; && representa **e**. || representa **ou**. ! representa negação.

Tabela 4 - Tabela verdade

P	Q	P && Q	P || Q	!P
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Dessa forma, se hoje fez chuva e hoje fez sol, só é verdade quando P e Q forem verdadeiros. Da mesma forma, se hoje fez chuva ou hoje fez sol só será falso se P e Q forem falsos (O dia estava nublado, por exemplo).

9.5 Usando os conectivos && e || em conjuntos numéricos

Sabendo como funciona a tabela verdade, podemos usa-la ao nosso favor. Por exemplo caso seja necessário verificar se um número está dentro de um intervalo $0 \leq x \leq 10$. Nesse caso devemos usar o conectivo &&.

Nesse caso && está totalmente relacionado a intersecção se conjuntos. Por exemplo o conjunto dos valores no intervalo $0 \leq x \leq 10$, é a interseção dos conjuntos $x \geq 0$ e $x \leq 10$. Então se P = $x \geq 0$ e Q = $x \leq 10$, P && Q só será verdade x estiver dentro de ambos os intervalos.

De mesma forma, caso queremos identificar os valores que estão nos conjuntos numéricos x ≤ 0 juntamente com x ≥ 10, devemos usar a união de conjuntos, que pode ser feita com o conectivo ||, sendo assim, se P = x ≤ 0 e Q = x ≥ 10, logo, P || Q só será falso se P e Q forem falsos, ou seja, x = 5 por exemplo.

9.6 Operadores Lógicos e relacionais na linguagem C

Operadores lógicos e relacionais são todos aqueles que retornam valor verdade quando usados, na linguagem C se um operador lógico retorna VERDADE, então o valor retornado é 1. Caso ele retorne FALSO, o valor retornado é 0.

A Tabela 5 lista os operadores lógicos da linguagem C.

Tabela 5 - Operadores Lógicos na linguagem C.

Uso	Operador
E	&&
OU	||
NEGAÇÃO	!

A Tabela 6 lista os operadores relacionais da linguagem C.

Tabela 6 - Operadores relacionais na linguagem C.

Uso	Operador
Maior do que	>
Maior ou igual a	>=
Menor do que	<
Menor ou igual a	<=
Igual a	==
Diferente de	!=

9.7 Estrutura de decisão IF, ELSE IF, ELSE em C

Em linguagens de programação no geral utilizamos dos valores lógicos de uma expressão lógica qualquer, a fim de resolver problemas.

9.7.1 IF

Para isso utilizamos a estrutura IF, caso uma condição seja verdadeira utilizando IF o fluxo de execução do código será direcionado para o interior da estrutura IF, caso contrário, o fluxo segue em frente, sem executar as instruções dentro dessa estrutura.

Como discutido anteriormente, 1 representa verdade, 0 representa falso. O código abaixo ilustra o fluxo de execução de um programa.

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(1)
    {
        printf("2 - Sempre vai entrar aqui\n");
    }

    if(0)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }

    printf("4 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

A condição if(1) sempre será satisfeita, pois 1 representa verdade, então o printf 2 sempre será executado. Em contrapartida, a condição if(0) nunca será satisfeita, pois 0 representa falso. Logo o fluxo do programa será 1, 2, 4.

```
1 - Entrou primeiro aqui
2 - Sempre vai entrar aqui
4 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Sabendo disso, podemos testar todas as possibilidades da tabela verdade, o código abaixo representa a tabela verdade do || (ou).

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(1 || 1)
    {
        printf("2 - Sempre vai entrar aqui\n");
    }

    if(1 || 0)
    {
        printf("3 - Sempre vai entrar aqui\n");
    }

    if(0 || 1)
    {
        printf("4 - Sempre vai entrar aqui\n");
    }

    if(0 || 0)
    {
        printf("5 - Nunca vai entrar aqui\n");
    }

    printf("6 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

Logo é claro imaginar que apenas 0 e 0 será falso, então o fluxo de execução do código acima será 1, 2, 3, 4 e 6.

```
1 - Entrou primeiro aqui
2 - Sempre vai entrar aqui
3 - Sempre vai entrar aqui
4 - Sempre vai entrar aqui
6 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

Seguindo o exemplo acima, o código abaixo ilustra o fluxo de execução para o operador de && (e).

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(1 && 1)
    {
        printf("2 - Sempre vai entrar aqui\n");
    }

    if(1 && 0)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }

    if(0 && 1)
    {
        printf("4 - Nunca vai entrar aqui\n");
    }

    if(0 && 0)
    {
        printf("5 - Nunca vai entrar aqui\n");
    }

    printf("6 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

Logo é claro imaginar que apenas 1 e 1 será verdadeiro, então o fluxo de execução do código acima será 1, 2 e 6.

```
1 - Entrou primeiro aqui
2 - Sempre vai entrar aqui
6 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

O último teste é utilizando o operador NOT, negação, NÃO, o que ele faz é transformar 1 em 0 e transformar 0 em 1. O código abaixo ilustra esse conceito:

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(!0)
    {
        printf("2 - Sempre vai entrar aqui\n");
    }

    if(!1)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }

    printf("4 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

Logo é claro imaginar que apenas !0 será verdadeiro, então o fluxo de execução do código acima será 1, 2 e 4.

```
1 - Entrou primeiro aqui
2 - Sempre vai entrar aqui
4 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

9.7.2 ELSE IF

É muito comum a utilização do comando ELSE IF, que em português seria “senão, se”, ou seja, caso a condição do IF seja falta, ainda queremos testar outra condição antes que algum comando seja executado. Em outras palavras, a condição ELSE IF só será testada se a condição IF falhar. Fora isso o ELSE IF funciona exatamente igual ao IF.

No exemplo abaixo, a condição IF irá falhar pois 0 representa falso, então a condição ELSE IF será testada, a condição ELSE IF irá passar pois 1 representa verdade.

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(0)
    {
        printf("2 - Nunca vai entrar aqui\n");
    }
    else if(1)
    {
        printf("3 - Sempre vai entrar aqui\n");
    }

    printf("4 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

Logo, o fluxo do programa será 1, 3 e 4.

```
1 - Entrou primeiro aqui
3 - Sempre vai entrar aqui
4 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Já no exemplo abaixo a condição do IF sempre será aceita pois 1 representa verdade, no entanto, mesmo a condição ELSE IF sendo também sempre verdade, o fluxo do programa não entrará no ELSE IF, pois já entrou no IF. Se a condição anterior for aceita não será testada nenhuma condição abaixo de ELSE IF, o código irá pular a estrutura inteira.

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(1)
    {
        printf("2 - Sempre vai entrar aqui\n");
    }
    else if(1)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }

    printf("4 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

No exemplo acima isso é bem ilustrado, sempre vai entrar no if(1), mas nunca vai entrar no else if(1). Logo o fluxo do programa será 1, 2 e 4.

```
1 - Entrou primeiro aqui
2 - Sempre vai entrar aqui
4 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

Além disso, podemos cascatear (por em sequência) uma quantidade infinita (dependendo da necessidade) de ELSE IF. Como no exemplo abaixo.

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(0)
    {
        printf("2 - Nunca vai entrar aqui\n");
    }
    else if(0)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }
    else if(0)
    {
        printf("4 - Nunca vai entrar aqui\n");
    }
    else if(1)
    {
        printf("5 - Sempre vai entrar aqui\n");
    }
    else if(1)
    {
        printf("6 - Nunca vai entrar aqui\n");
    }

    printf("7 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

Como mencionado acima, quando uma condição de IF falha, será testada a próxima condição ELSE IF, até que uma das condições seja aceita. Quando uma delas é aceita, executa o que tem dentro e pula todo o restante da estrutura. No exemplo acima, ele testa o 2, 3 e 4, todos falham, mas no 5 funciona, executa o printf e por fim pula todo o restante da estrutura condicional e vai direto para o 7. Logo o fluxo do programa será 1, 5 e 7.

```
1 - Entrou primeiro aqui
5 - Sempre vai entrar aqui
7 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

9.7.3 ELSE

O ELSE entra em ação quando todas as condições falham e você não precisa testar nada mais para que ele seja executado. Se ele for necessário, deve ser inserido sempre no fim, assim como no ELSE IF, se nenhuma condição acima for aceita, ele será executado, porém se alguma condição acima for executada ele nunca será executado. O ELSE é nada mais que o ELSE IF sem o IF.

```c
#include <stdio.h>

int main (void)
{
    printf("1 - Entrou primeiro aqui\n");

    if(0)
    {
        printf("2 - Nunca vai entrar aqui\n");
    }
    else if(0)
    {
        printf("3 - Nunca vai entrar aqui\n");
    }
    else if(0)
    {
        printf("4 - Nunca vai entrar aqui\n");
    }
    else
    {
        printf("5 - Sempre vai entrar aqui\n");
    }

    printf("6 - Termina aqui, sempre executa isso\n");
    return 0;
}
```

No programa acima o fluxo de dados é o seguinte, testa 2, 3 e 4, todos falham, então executa 5 sem realizar qualquer teste. Então o fluxo é 1, 5 e 6.

```
1 - Entrou primeiro aqui
5 - Sempre vai entrar aqui
6 - Termina aqui, sempre executa isso

Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

9.8 Usando as estruturas de decisão para resolver problemas

Já vimos como é comportamento das estruturas de decisão, então agora, ao invés de deixar fixo 1 no IF, deixaremos esse parâmetro variável de acordo com nosso problema.

9.8.1 Exemplo 1 – Mostrar quem é menor de idade

O programa abaixo verifica se uma pessoa é menor de idade, se sim, imprime a mensagem “Eh menor de idade”, senão, imprime a mensagem “Não eh menor de idade”.

```c
#include <stdio.h>

int main (void)
{
    int idade = 6;
    int P;

    P = idade < 18;

    if(P)
    {
        printf("Eh menor de idade\n");
    }
    else
    {
        printf("Nao eh menor de idade");
    }

    return 0;
}
```

A variável P recebe 1 se a condição idade < 18 for verdadeira, e P recebe 0 se a condição idade < 18 for falsa. No caso apresentado acima, idade tem valor 6, logo 6 < 18 é uma condição verdadeira, então P terá valor 1, consequentemente a mensagem que será impressa na tela é:

```
Eh menor de idade

Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.
```

9.8.2 Exemplo 2 – Intervalo numérico

O programa abaixo verifica se um número está no intervalo 0 ≤ x ≤ 10. Para isso, como visto anteriormente precisaremos de um &&. Caso seja verdade que x ≤ 10 e ao mesmo tempo seja verdade que x ≥ 0, então podemos concluir que x está dentro do intervalo.

```c
#include <stdio.h>

int main (void)
{
    int x = 6;
    int P, Q;

    P = x >= 0;
    Q = x <= 10;

    if(P && Q)
    {
        printf("Dentro do intervalo\n");
    }
    else
    {
        printf("Fora do intervalo\n");
    }

    return 0;
}
```

Utilizando x = 6, P será verdadeiro e Q também será verdadeiro, logo P = 1 e Q = 1. Lembrando que && só é verdadeiro quando P e Q são verdadeiros ambos ao mesmo tempo, então a mensagem exibida será:

```
Dentro do intervalo

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Caso tentássemos com o número x = 11, P = 1 pois é verdade que x é maior que zero, porém não é verdade que x é menor que 10, então Q = 0. Sendo assim, a mensagem impressa na tela seria: Fora do intervalo.

9.8.3 Exemplo 3 – Combinando conectivos lógicos

Caso o meu intervalo numérico deseja fosse 0 ≤ x ≤ 10 união com x ≥ 100. Seria necessário realizarmos os operadores && e ||.

```c
#include <stdio.h>

int main (void)
{
    int x = 6;
    int P, Q, R;

    P = x >= 0;
    Q = x <= 10;
    R = x >= 100;

    if((P && Q) || R)
    {
        printf("Dentro do intervalo\n");
    }
    else
    {
        printf("Fora do intervalo\n");
    }

    return 0;
}
```

Lembrando que || ambos devem ser falsos para que a condição seja falsa. Caso x estiver dentro do primeiro intervalo 0 ≤ x ≤ 10, a condição será aceita. Ou ainda, se x satisfazer apenas R que é x ≥ 100, a condição também será aceita, em ambos os casos, a mensagem exibida será:

```
Dentro do intervalo

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Note que não seria necessário atribuirmos os valores das condições lógicas a variáveis, isso foi feito apenas por questões didáticas, o que um bom programador deve fazer é colocar tudo dentro do IF, pois isso gera performance no código, evitando que algumas condições sejam testadas caso seja desnecessário. O compilador, que transforma o código que escrevemos em linguagem de máquina foi desenvolvido e melhorado ao longo de anos para maximizar a performance do código desenvolvido, mas para isso, o programador deve fazer a sua parte.

```c
#include <stdio.h>

int main (void)
{
    int x = 6;

    if(((x >= 0) && (x <= 10)) || (x >= 100))
    {
        printf("Dentro do intervalo\n");
    }
    else
    {
        printf("Fora do intervalo\n");
    }

    return 0;
}
```

A explicação para o ganho de performance é a seguinte, se x < 0, a P irá falhar, logo, como a condição é P && Q, Q nem será calculado, pois no && os dois devem ser verdadeiros, se o primeiro já e falso não precisamos testar o segundo, assim será testado diretamente R, a segunda condição do ||, sendo assim, foi economizado uma verificação. Ao utilizar o código que calcula tudo e atribui em variáveis não existe economia. Em um sistema que faz milhos de vezes essas verificações, por exemplo processamento de imagens, estaremos aumentando o tempo de execução do algoritmo.

9.8.4 Exemplo 4 – IF encadeado

Dentro do IF funciona como qualquer outra parte do código, então podemos usar um IF dentro do outro.

O exercício abaixo ilustra o funcionamento de um carro, se o carro estiver ligado e a pessoa usar cinto de segurança, ele mostra a mensagem “Boa viagem”, caso a pessoa ligar o carro e não usar cinto de segurança, o código mostra a mensagem “Aviso, coloque o cinto de segurança”, caso o carro esteja desligado, ele mostra a mensagem “Sem avisos”:

```c
#include <stdio.h>

int main (void)
{
    int carro_ligado = 1;
    int uso_do_cinto = 0;

    if(carro_ligado)
    {
        printf("Carro Ligado\n");
        if(uso_do_cinto == 0)
        {
            printf("Aviso, coloque o cinto de seguranca\n");
        }
        else
        {
            printf("Boa viagem\n");
        }
    }
    else
    {
        printf("Sem avisos\n");
    }

    return 0;
}
```

O código acima mostra o uso de IFs aninhados, ou seja, um IF dentro do outro. Isso pode ser feito pois o código dentro do IF funciona exatamente como outra parte do código. Assim torna-se evidente que podemos colocar infinitos IFs dentro do outro.

9.8.5 Exemplo 5 - Escopo das variáveis no IF

A diferença entre o código dentro de um IF e de um código externo a um IF, é que, se declararmos variáveis dentro do IF, elas não porão ser vistas pelo código fora desse IF. No entanto, variáveis declaradas fora do IF podem ser vistas dentro do IF.

O exemplo abaixo ilustra bem o escopo das variáveis em C, x foi declarada dentro do IF (na linha 7), então fora do IF (na linha 9), não conseguiremos acessar a variável x.

```c
#include <stdio.h>

int main (void)
{
    if(1)
    {
        int x = 3;
    }
    printf("X = %d\n", x);

    return 0;
}

```

A resposta do compilador a esse erro é que x não foi declarado, e estou usando pela primeira vez nessa na função main, isso porque todo o código fora do IF não consegue enxergar as variáveis declaradas dentro do IF.

```
File          L... Message
                   === Build file: "no target" in "no project" (compiler: unknown) ===
C:\Users\...       In function 'main':
C:\Users\... 9     error: 'x' undeclared (first use in this function)
C:\Users\... 9     note: each undeclared identifier is reported only once for each function it appears in
                   === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Contudo, é importante ressaltar que, ao declarar uma variável dentro do IF, tudo dentro do IF consegue enxergar essa variável, até mesmo os IFs internos ao IF em questão.

```c
#include <stdio.h>

int main (void)
{
    if(1)
    {
        int x = 3;
        printf("X = %d\n", x);
        if(1)
        {
            printf("X = %d\n", x);
        }
    }

    return 0;
}
```

Logo, o código acima é valido, pois x foi declarado dentro do mesmo escopo em que é usado.

9.8.6 Exemplo 6 – Calcular as raízes de polinômios de primeira e segunda ordem

O programa abaixo tem como objetivo calcular as raízes de um polinômio de primeira ou segunda ordem, do tipo: $a * x^2 + b * x + c = 0$.

Nesse caso, se delta der negativo, deve-se calcular raízes complexas, se a = 0, então o polinômio é de primeira ordem, então será encontrada apenas uma raiz (Não podendo utilizar a fórmula de Bhaskara).

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  int main (void)
 5  {
 6      double a = 1, b = 2, c = 3;
 7      double delta, x1, x2;
 8
 9      printf("%lf * x^2 + %lf * x + %lf = 0\n", a, b, c);
10      if(a == 0)
11      {
12          printf("Polinomio de primeira ordem\n");
13          x1 = -c / b;
14          printf("x1 = %lf\n", x1);
15      }
16      else
17      {
18          printf("Polinomio de segunda ordem\n");
19          delta = b * b - 4 * a * c;
20          if(delta < 0)
21          {
22              printf("Raizes imaginarias\n");
23              delta = -delta;
24              double real, imaginario;
25              real = (-b)/(2*a);
26              imaginario = (sqrt(delta))/(2*a);
27              printf("x1 = %lf + i * %lf\n", real, imaginario);
28              printf("x2 = %lf - i * %lf\n", real, imaginario);
29          }
30          else
31          {
32              printf("Raizes Reais\n");
33              x1 = (-b+sqrt(delta))/(2*a);
34              x2 = (-b-sqrt(delta))/(2*a);
35              printf("x1 = %lf\n", x1);
36              printf("x2 = %lf\n", x2);
37          }
38      }
39      return 0;
40  }
```

O exemplo acima foi feito com o objetivo de ser um exemplo completo da utilização de IF encadeado e escopo de variáveis. As variáveis real e imaginário só serão declaradas se o polinômio informado for um polinômio imaginário, pois delta < 0.

9.9 Estrutura de decisão switch case em C

A estrutura switch case é uma simplificação do IF, funciona no caso particular em que uma variável é testada com constantes.

```c
#include <stdio.h>

int main (void)
{
    int dia = 5;

    switch (dia)
    {
    case 1 :
        printf ("Domingo\n");
        break;

    case 2 :
        printf ("Segunda\n");
        break;

    case 3 :
        printf ("Terça\n");
        break;

    case 4 :
        printf ("Quarta\n");
        break;

    case 5 :
        printf ("Quinta\n");
        break;

    case 6 :
        printf ("Sexta\n");
        break;

    case 7 :
        printf ("Sabado\n");
        break;

    default :
        printf ("Valor invalido!\n");
    }

    return 0;
}
```

É muito utilizado para construção de menu, o conteúdo de uma variável é comparado com um valor constante, e caso a comparação seja verdadeira, um determinado comando é executado.

O switch assim como no IF, testa as condições até que uma seja verdadeira, então o break força o código terminar os testes. Se não usar break, após a execução das instruções desejadas, o programa continua testando.

O comando default é equivalente ao ELSE, uma vez que só entra no default caso nenhuma condição testada foi verdadeira

Não são aceitas expressões condicionais no comando switch case, somente são aceitos valores constantes.

10 Estrutura de repetição

Imagine o seguinte caso: Leia um milhão de valores e calcule a média desses valores. É de se imaginar que, o que temos até agora, não é o suficiente para resolver esse problema. Pois seria necessário um código com um milhão de scanf, sendo assim, seria muito trabalhoso, humanamente inviável.

Sendo assim, surge um conceito muito útil: Estruturas de repetição.

Basicamente, uma estrutura de repetição serve para que um determinado trecho de código seja executado várias vezes, de acordo com o nosso objetivo.

Se nosso objetivo é repetir um número X de vezes o mesmo código, por exemplo, ler um milhão de números inteiros, podemos pedir para o programa repetir a instrução scanf um milhão de vezes.

Na linguagem C existem três estruturas de repetição, FOR, WHILE e DO WHILE, a única que não pode ser utilizada em todos os casos é a DO WHILE, pois a primeira execução é feita sem teste algum.

10.1 Estruturas de repetição WHILE

A estrutura de repetição WHILE é comumente utilizada quando o objetivo é realizar uma repetição até que uma condição seja satisfeita. O WHILE funciona de forma muito similar a um IF, na verdade, a única diferença entre o WHILE e o IF, é que ao fim da última instrução dentro do WHILE, o fluxo de execução do programa volta para antes do WHILE.

```c
#include <stdio.h>

int main (void)
{
    while(1)
    {
        printf("Ola Mundo!\n");
    }

    return 0;
}

```

O código acima ilustra a estrutura do WHILE. Na linha 5, será verificada a condição entre parênteses, porém 1 representa verdade, então o fluxo do programa será executar o código dentro do WHILE (entre as chaves da linha 6 até a linha 8), nesse exemplo, o código é mostrar a mensagem “Ola Mundo!”. Ao fim da execução da linha 7, o programa irá saltar novamente para a linha 5, aí que vem a diferença entre WHILE e IF. Se fosse um IF, o programa iria continuar na linha 9, mas o WHILE faz com que o fluxo do programa volte no início depois de terminar.

No exemplo do código acima, a condição será sempre verdadeira, logo a mensagem “Ola Mundo!”, será executada infinitas vezes.

10.1.1 Exemplo 1 – Mostrar os números de 0 a 100 usando WHILE

A primeira tarefa a se aprender ao utilizar um WHILE é, imprimir na tela os números de 0 a 100, está é uma tarefa bem simples, precisamos apenas declarar uma variável, atribuir valor 0, e a cada vez que executar o WHILE adicionarmos 1 nesta variável, sendo assim, de 1 em 1 o valor dessa variável chegará em 100, então basta deixarmos a condição do WHILE executando enquanto a variável for menor ou igual a 100.

O código abaixo, na linha 5 foi declarada a variável x e atribuído a ela o valor 0, em sequência na linha 6, foi testada a condição x <= 100, e então a condição foi satisfeita, pois 0 <= 100 é verdadeiro, logo irá entrar no WHILE. Entrando no WHILE, é executado primeiro a linha 8, onde é mostrado o valor 0 na tela, em seguida a variável x recebe 0 + 1, ou seja, o novo valor de x é 1, ao fim da última linha desse WHILE (linha 9), o programa volta para linha 6, e todo o processo é reiniciado.

Na linha 6, foi testada a condição x <= 100, e então a condição foi satisfeita, pois 1 <= 100 é verdadeiro, logo irá entrar no WHILE. Entrando no WHILE, é executado primeiro a linha 8, onde é mostrado o valor 1 na tela, em seguida a variável x recebe 1 + 1, ou seja, o novo valor de x é 2, ao fim da última linha desse WHILE (linha 9), o programa volta para linha 6, e todo o processo é reiniciado.

Após fazer isso muitas vezes, digamos que agora x é 100. Na linha 6, foi testada a condição x <= 100, e então a condição foi satisfeita, pois 100 <= 100 é verdadeiro, logo irá entrar no WHILE. Entrando no WHILE, é executado primeiro a linha 8, onde é mostrado o valor 100 na tela, em seguida a variável x recebe 100 + 1, ou seja, o novo valor de x é 101, ao fim da última linha desse WHILE (linha 9), o programa volta para linha 6, e todo o processo é reiniciado.

Nesse caso, na linha 6, foi testada a condição x <= 100, e então a condição não foi satisfeita, pois 101 <= 100 é falso, então o programa salta para linha 11, e o programa é finalizado.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5      int x = 0;
6      while(x <= 100)
7      {
8          printf("%d\n", x);
9          x = x + 1;
10     }
11
12     return 0;
13 }
```

10.1.2 Exemplo 2 – Calcular a raiz quadrada de um número qualquer

Uma das técnicas de calcular a raiz quadrada de um número é através de uma fórmula recursiva (significa que o valor atual depende do valor anterior) simples. Ou seja, se queremos calcular raiz quadrada de X, precisamos estimar vários valores, até o erro entre a raiz quadrada estimada elevada ao quadrado e o valor de X for pequeno o quanto desejamos (erro = X - raiz_estimada^2).

Por exemplo, “chuta-se” inicialmente um valor para raiz quadrada do número X. Depois calcula-se o valor da próxima raiz baseado nesse número. O código abaixo realiza esse procedimento:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    double raiz_ant, raiz_atual, x = 2;
    double erro = 1;


    raiz_ant = x / 2.0; // chutar valor inicial
    while(erro > 0.0000001 || erro < -0.0000001)
    {
        raiz_atual = (raiz_ant + x / raiz_ant) / 2.0; // calculo recursivo
        erro = x - raiz_atual * raiz_atual; // calculando o erro

        printf("R ant: %.10lf\t", raiz_ant);
        printf("R atual: %.10lf\t", raiz_atual);
        printf("R atual^2: %.10lf\t", raiz_atual*raiz_atual);
        printf("erro: %.10lf\n", erro);

        raiz_ant = raiz_atual; // na proxima iteracao o atual é anterior
    }

    return 0;
}
```

Como o erro pode ser negativo então é necessário dizer que o erro deve ser maior que o desejado negativo. Ou seja, deve-se continuar enquanto o erro não estiver dentro do intervalo desejado.

Abaixo o resultado da execução desse código:

```
R ant: 1.0000000000     R atual: 1.5000000000   R atual^2: 2.2500000000 erro: -0.2500000000
R ant: 1.5000000000     R atual: 1.4166666667   R atual^2: 2.0069444444 erro: -0.0069444444
R ant: 1.4166666667     R atual: 1.4142156863   R atual^2: 2.0000060073 erro: -0.0000060073
R ant: 1.4142156863     R atual: 1.4142135624   R atual^2: 2.0000000000 erro: -0.0000000000

Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

10.2 Estruturas de repetição FOR

A estrutura de repetição for é muito utilizada para realizar um número de repetições fixas. O exemplo de mostrar os números de 0 a 100 seria perfeito para se realizar utilizando FOR, na verdade, tudo que é feito usando WHILE, pode ser feito usando FOR. Pois este é apenas, em alguns casos, uma forma mais organizada de fazer o WHILE.

10.2.1 Exemplo 1 - Mostrar os números de 0 a 100 usando FOR

O exemplo abaixo mostra os números de 0 a 100 usando FOR. Dentro dos parênteses que vem depois da palavra FOR, tem 3 partes, a primeira parte é sempre executada apenas uma vez, seria o mesmo que colocar x = 0 na linha 6. Já a segunda parte x <= 100, é executada igual no exemplo do WHILE, ele vai fazer enquanto x <= 100. A última parte, x++, é executada depois que termina de executar o conteúdo do FOR, ou seja, nesse exemplo, poderia ser colocado o x++ na linha 10, depois do printf, isso deve ficar claro, pois se utilizado o valor do x, ele terá o valor obtido da iteração passada, o incremento só ocorre no fim.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x;
6
7      for (x = 0; x <= 100; x++)
8      {
9          printf("%d\n", x);
10     }
11
12     return 0;
13 }
14
```

10.3 Estruturas de repetição DO WHILE

A estrutura de repetição DO... WHILE serve para os casos onde não é necessário executar verificações na primeira iteração, como foi o caso do cálculo da raiz quadrada de X, onde foi feita uma “gambiarra”, atribuindo-se valor erro = 1, para que ele entrasse no WHILE no primeiro caso.

Então a diferença entre WHILE e DO...WHILE é que o WHILE sempre verifica uma condição antes de entrar em sua estrutura, e o DO...WHILE permite entrar na primeira vez sem realizar testes.

Abaixo, como ficaria o código da raiz quadrada de X usando DO...WHILE

10.3.1 Exemplo 1 – Calcular a raiz quadrada de um número qualquer

Note que a execução o programa é a seguinte, entra na linha 14 sem verificações, ao terminar de executar a linha 22 o programa faz a verificação na linha 24 e se for verdadeira ele pula novamente para linha 14. Esse ciclo será realizado enquanto a condição do WHILE na linha 24 for verdadeira.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      double raiz_ant, raiz_atual, x = 2;
7      double erro;
8
9
10     raiz_ant = x / 2.0; // chutar valor inicial
11
12     do
13     {
14         raiz_atual = (raiz_ant + x / raiz_ant) / 2.0; // calculo recursivo
15         erro = x - raiz_atual * raiz_atual; // calculando o erro
16
17         printf("R ant: %.10lf\t", raiz_ant);
18         printf("R atual: %.10lf\t", raiz_atual);
19         printf("R atual^2: %.10lf\t", raiz_atual*raiz_atual);
20         printf("erro: %.10lf\n", erro);
21
22         raiz_ant = raiz_atual; // na proxima iteracao o atual é anterior
23     }
24     while(erro > 0.0000001 || erro < -0.0000001);
25
26     return 0;
27 }
28
```

10.4 – Exemplo úteis utilizando estruturas de repetição

10.4.1 Exemplo 1 – Fazendo um programa executar infinitamente.

Se quisermos que um programa reinicie após terminar seu processamento e faça isso infinitamente, podemos utilizar o WHILE com uma condição sempre verdadeira.

O programa abaixo lê dois valores da tela e calcula a soma deles, imprime na tela e em seguida reinicia, sem apagar o conteúdo já impresso na tela.

```c
#include <stdio.h>

int main (void)
{
    int a, b, soma;

    while(1)
    {
        printf("Informe dois numeros: ");
        scanf("%d %d", &a, &b);
        soma = a + b;
        printf("%d + %d = %d\n", a, b, soma);
    }
    return 0;
}
```

Exemplo de execução:

```
Informe dois numeros: 1 2
1 + 2 = 3
Informe dois numeros: 3 4
3 + 4 = 7
Informe dois numeros: 5 6
5 + 6 = 11
Informe dois numeros: 7 8
7 + 8 = 15
Informe dois numeros: 9 10
9 + 10 = 19
Informe dois numeros: _
```

10.4.2 Exemplo 2 – Calcular toda a tabuada

Podemos utilizar uma estrutura de repetição dentro da outra, dessa forma, quando i = 1, entra no FOR do j, e executa 10 vezes, e por último pula linha, em seguida i = 2, executa o FOR do j mais 10 vezes, por último pula linha, até que i = 11, então não entra mais no FOR do i.

```c
#include <stdio.h>

int main (void)
{
    int i, j;

    for (i = 1; i <= 10; i++)
    {
        for (j = 1; j <= 10; j++)
        {
            printf("%d * %d = %d\n", i, j, i * j);
        }
        printf("\n");
    }

    return 0;
}
```

10.4.3 Exemplo 3 – Calcular todos os divisores de X

Sabendo o valor de X, podemos encontrar todos os seus divisores. Para isso verificamos todos os números desde 1 até X/2.

```c
#include <stdio.h>

int main (void)
{
    int i, x, fim;
    scanf("%d", &x);
    // para melhor desempenho
    fim = x / 2; // fazer a conta fora da verificacao
    for (i = 1; i <= fim; i++)
    {
        if (x % i == 0)
        {
            printf("%d ", i);
        }
    }
    printf("\n");
    return 0;
}
```

10.4.4 Exemplo 4 – Calcular os múltiplos dos números de 1 a 100

Tendo resolvido o exemplo 3, este problema está praticamente resolvido, basta ao invés de realizarmos a leitura de x, fazermos um FOR de 1 a 100.

```c
#include <stdio.h>

int main (void)
{
    int i, x, fim;
    for (x = 1; x <= 100; x++)
    {
        fim = x / 2;
        printf("Para %d: ", x);
        for (i = 1; i <= fim; i++)
        {
            if (x % i == 0)
            {
                printf("%d ", i);
            }
        }
        printf("\n");
    }
    return 0;
}
```