

Algoritmos e Programação

1º Ano - 1º Semestre

2. Programação numa linguagem de alto nível (C) – Parte 2

Escola Superior de Tecnologia e Gestão de Viseu

Agradecimentos a Francisco Morgado, Carlos Simões e Jorge Loureiro

2. Programação numa linguagem de alto nível (C) – Part 2

2.1 Introdução

2.2 Estrutura de um programa em linguagem C

2.3 Tipos de dados básicos

2.4 Testes e condições - expressões e operadores. Precedências

2.5 Estruturas de repetição

2.6 Funções

2.7 Tipos de dados estruturados: vetores e strings

2.8 Apontadores (Pointers)

2.9 Passagem de parâmetros de tipos estruturados

2.10 Estruturas

2.11 Memória dinâmica

2.5 Estruturas de repetição

- Instrução *for*
- Instrução *while*
- Instrução *do ... while*
- Instrução *continue*
- Ciclos infinitos

Instrução *for*

Estrutura de Repetição com um número de iterações predefinido

Esta estrutura de controlo permite a repetição de uma determinada instrução ou bloco de instruções, um número de vezes predefinido e de uma forma controlada. Cada repetição corresponde a um ciclo.

SINTAXE

```
for (inicialização ; condição ; pós-instrução)  
  Instrução;
```

ou

```
for (inicialização ; condição ; pós-instrução)  
{  
  Instrução 1;  
  ...  
  Instrução n;  
}
```

FUNCIONAMENTO

1. É executado o código presente em **inicialização** ;
2. A **condição** é avaliada ;
3. Se o resultado da condição tiver o valor **Falso**, o ciclo termina e o programa continua na instrução imediatamente a seguir à instrução (ou bloco de instruções) do ciclo;
4. Se o resultado da condição tiver o valor **Verdade**, é executada a instrução (ou bloco de instruções) do ciclo;
5. Depois de executada a instrução (ou bloco de instruções) do ciclo, é executada a pós-instrução;
6. Volta ao ponto 2.

Instrução *for*

Estrutura de Repetição com um número de iterações predefinido

SINTAXE

```
for (inicialização ; condição ; pós-instrução)  
  Instrução;
```

ou

```
for (inicialização ; condição ; pós-instrução)  
{  
  Instrução 1;  
  ...  
  Instrução n;  
}
```

Observações:

1. O ciclo *for* identifica cada componente, separando-as por (;);
2. Caso seja necessário efetuar mais do que uma inicialização ou mais do que uma pós-instrução, podem ser efetuadas, separando-as por (,).

EXEMPLO – cálculo do somatório

$$S = \sum_{i=1}^N (2 - i) \times i$$

Descrição Algorítmica

```
...  
Inst.1: soma ← 0;  
Inst.2: PARA (DE) i DESDE 1 ATÉ N FAZ  
        soma ← soma + (2-i)*i;  
        FIM-PARA (FIM-DE)  
Inst.3: Escrever ("O resultado do somatório é", soma)  
...
```

Codificação em C

```
#include <stdio.h>  
int main()  
{  
    int i, n, soma;  
    printf("Indique o n° de elementos do somatório: ");  
    scanf("%d", &n);  
    soma = 0;  
    for (i = 1; i <= n; i++)  
        soma = soma + (2-i)*i; // soma += (2-i)*i;  
    printf("O resultado do somatório é:%d", soma);  
}
```

Altere este programa para que permita visualizar cada um dos termos da sucessão

EXEMPLO – cálculo da soma e do produto dos primeiros N números pares

Descrição Algorítmica



Codificação em C

```
...
produto ← 1;
soma ← 0;
par ← 2
PARA (DE) i DESDE 2 ATÉ n SALTO 2 FAZ
    soma ← soma + par;
    produto ← produto*par;
FIM-PARA (FIM-DE)
Escrever ( "Soma = ", soma,"Produto = ", produto);
...
```

```
#include <stdio.h>
int main()
{
    int i, n, soma, produto, maiorPar;
    printf("Indique o total de pares a operar: ");
    scanf("%d",&n);
    soma = 0; // elemento neutro da adição
    produto = 1; // elemento neutro da multiplicação
    maiorPar = 2*n;
    for (i = 2; i <= maiorPar; i = i+2 )
    {
        soma = soma + i;
        produto = produto*i;
    }
    printf("\nSoma= %d\nProduto= %d\n",soma,produto);
}
```

NOTAR QUE: O incremento da variável de iteração **não tem** necessariamente de ser positivo

Cálculo da soma $N + (N-1) + \dots + 3 + 2 + 1$

Descrição Algorítmica

```
...  
Inst.1: soma ← 0;  
Inst.2: PARA (DE) i DESDE N ATÉ 1 SALTO -1 FAZ  
        soma ← soma + i;  
        FIM-PARA (FIM-DE)  
Inst.3: Escrever ( "Somatório = ", Soma);  
...
```

Codificação em C

```
#include <stdio.h>  
int main()  
{  
    int i, n, soma;  
    printf("Escreva um numero inteiro positivo: ");  
    scanf("%d",&n);  
    soma = 0;  
    for (i = n; i > 0; i--)    // i = i-1  
        soma = soma + i;  
    printf("\n%d+%d+...+2+1 = %d\n", n, n-1, soma);  
}
```


Estruturas de Repetição Condicional

Instrução *while*

Esta estrutura de controlo permite a repetição de uma instrução ou bloco de instruções em ciclo, enquanto uma dada condição for verdadeira.

Sintaxe:

```
while(condição)  
    Instrução;
```

ou

```
while(condição)  
{  
    Instrução 1;  
    ...  
    Instrução n;  
}
```

Funcionamento:

1. A condição é avaliada;
2. Se o resultado da condição tiver o valor Falso, o ciclo termina e o programa continua na instrução imediatamente a seguir à instrução (ou bloco de instruções) do ciclo;
3. Se o resultado da condição tiver o valor Verdade, é executada a instrução (ou bloco de instruções) associada(s) ao ciclo;
4. Volta ao ponto 1.

Exercício 1

Determinar o *output* do programa:

1
3
5
7
9
11
13
15

```
#include <stdio.h>
int main()
{
    int i;
    i = 1;
    while (i <= 15)
    {
        printf("\n%d\n", i);
        i = i+2; //i += 2;
    }
}
```

QUESTÃO 1
Qual alteração do programa, para que mostrasse os números pares?

QUESTÃO 2
Qual seria o *resultado* se não fossem colocadas as chavetas na instrução do ... *while*?

Algumas considerações:

- ◆ O teste da condição é realizado antes da execução do bloco de instruções;
- ◆ É necessária a criação de um bloco para que sejam executadas duas (ou mais) instruções em cada iteração;
- ◆ É necessário que a variável que controla o ciclo seja alterada dentro do ciclo, para que este possa terminar.

Exercício 2

Determinar o *output* do programa

5x 1 = 5
5x 2 = 10
5x 3 = 15
5x 4 = 20
5x 5 = 25
5x 6 = 30
5x 7 = 35
5x 8 = 40
5x 9 = 45
5x10= 50

```
#include <stdio.h>
int main()
{
    int n;
    n = 1;
    printf("O que faz?\n");
    while (n <= 10)
    {
        printf("\n5x%2d = %2d", n, 5*n);
        n = n+1;
    }
}
```

O uso do formato (%**2d**) garante o alinhamento dos valores

Exercício 3

Alterar o programa anterior para que seja apresentada a “tabuada” de qualquer número (com menos de 3 dígitos) dado pelo utilizador.

```
#include <stdio.h>
int main()
{
    int n, i;
    printf("\nEscreva um número inteiro positivo, com menos de 3 dígitos: ");
    scanf("%d",&n);
    i = 1;
    printf("Tabuada do %d\n", n);
    while (i <= 10)
    {
        printf("\n%2d x %2d = %3d", n, i, i*n);
        i++; //i = i+1;
    }
}
```

Para garantir um alinhamento à direita dos resultados.

```
#include <stdio.h>
void main()
{
    int n;
    n = 1;
    printf("Tabuada do 5\n");
    while (n <= 10)
    {
        printf("\n5x%2d = %2d", n, 5*n);
        n = n+1;
    }
}
```

Instrução *do ... while*

Nesta estrutura de repetição, contrariamente ao que acontece nos ciclos *for* e *while*, o teste da condição é feito no fim da instrução (ou bloco de instruções) e não no princípio.

Sintaxe:

```
do
    Instrução
while(condição);
```

ou

```
do
{
    Instrução 1;
    ...
    Instrução n;
}
while(condição) ;
```

Funcionamento:

1. A instrução (ou bloco de instruções) associada(s) ao ciclo é executada.
2. A condição é avaliada .
3. Se o resultado da condição tiver o valor Verdade, volta ao ponto 1 .
4. Se o resultado da condição tiver o valor Falso, o ciclo termina e o programa continua na instrução imediatamente a seguir à instrução (ou bloco de instruções) do ciclo.

➤ O ciclo **do ... while** é particularmente adequado ao processamento de menus

EXEMPLO 1 – Determinação da data da Páscoa, enquanto o utilizador pretender

```
int main()
{
    int ano, mes, dia, a, b, c, d, e;
    char opcao;
    do
    {
        do
        {
            printf("Indique um ano, entre 1900 e 2099 \n");
            scanf("%d", &ano);
        } while (ano < 1900 || ano > 2099);
        a = ano % 19;
        b = ano % 4;
        c = ano % 7;
        d = (19*a+24) % 30;
        e = (2*b+4*c+6*d+5) % 7;
        if (d+e > 9)
        {
            mes = 4;
            dia = d+e-9;
        }
        else
        {
            mes = 3;
            dia = 22+d+e;
        }
        printf("\n\nA Pascoa de %d ocorre em %d/%d \n\n", ano, dia, mes);
        printf("Pretende continuar (s/n)? ");
        scanf(" %c", &opcao); // "truque": ESPAÇO antes de %
    } while ( (opcao == 's') || (opcao == 'S') );
}
```

EXEMPLO 2 – Determinação da data da Páscoa, num intervalo de anos [início, fim]

```
int main()
{
    int inicio, fim, ano, mes, dia, a, b, c, d, e;
    char opcao;
    do
    {
        do
        {
            printf("\nIndique um intervalo de anos (primeiro e último), entre 1900 e 2099 \n");
            scanf("%d %d", &inicio, &fim);
        } while (inicio < 1900 || inicio > 2099 || fim < 1900 || fim > 2099 || inicio > fim);
        printf("\n\n ANO                PASCOA\n");
        for (ano = inicio; ano <= fim; ano++)
        {
            a = ano % 19;
            b = ano % 4;
            c = ano % 7;
            d = (19*a+24) % 30;
            e = (2*b+4*c+6*d+5) % 7;
            if (d+e > 9)
            {
                mes = 4;
                dia = d+e-9;
            }
            else
            {
                mes = 3;
                dia = 22+d+e;
            }
            printf("\n%11d %22d/%2d", ano, dia, mes);
        }
        printf("\n\nPretende continuar (s/n)? ");
        scanf(" %c", &opcao); // "truque": ESPAÇO antes de %
    } while ( (opcao == 's') || (opcao == 'S') );
}
```

Estruturas de Repetição - Síntese

Instrução *for*

Permite a **repetição** de uma determinada instrução ou de um bloco de instruções, **um número de vezes predefinido**.

Instrução *while*

Possibilita a repetição de uma instrução ou bloco de instruções em ciclo, enquanto uma dada **condição**, **testada no início de cada ciclo**, for verdadeira.

Instrução *do ... while*

O **teste** da condição é feito **no fim de cada ciclo**, pelo que a instrução (ou bloco de instruções) se executa pelo menos uma vez.

Aplicação: Somar os termos de

$$S = \sum_{i=1}^N \frac{1}{(i+1)^2}$$

a) Conhecendo um número predefinido de termos a somar

b) Somando até que o valor do termo seja inferior a 0.005

Utilizar as estruturas de repetição **for**, **do ... while** e **while** e analisar as diferenças

Uso da estrutura de repetição *while*

Descrição Algorítmica

```
...  
Inst.1:  $i \leftarrow 1$ ;  
Inst.2:  $soma \leftarrow 0$ ;  
Inst.3:  $termo \leftarrow 1/((i+1)^2)$ ;  
Inst.4: ENQUANTO  $termo > 0.005$   
         $s \leftarrow soma + termo$ ;  
         $i \leftarrow i+1$ ;  
         $termo \leftarrow 1/((i+1)^2)$ ;  
FIM-ENQUANTO  
...
```



Codificação em C

```
#include <stdio.h>  
int main()  
{  
    int i;  
    float termo, soma, erro;  
    erro = 0.005;  
    soma = 0;  
    i = 1;  
    termo = 1.0/4.0;  
    while (termo >= erro)  
    {  
        soma = soma + termo;  
        i++;  
        termo = 1.0 / ((i+1) * (i+1));  
    }  
    printf("\nSoma = %.3f, com erro inferior a %.3f\n", soma,  
    erro);  
}
```

3 casas
decimais

A instrução *continue*

➤ A instrução **continue**, quando presente dentro de um ciclo, remete para a iteração seguinte

```
#include <stdio.h>
int main()
{
    int i;
    for (i=1; i<=50; i=i+1)
    {
        if (i==10)
            break;
        else
            if (i%2==0)
                continue;
            else
                printf("%3d\n", i);
    }
    printf("\n FIM DO CICLO\n");
}
```

Simulação

i	acção	output
1	i≠10 e ímpar	1
2	i≠10 e par	
3	i≠10 e ímpar	3
4	i≠10 e par	
5	i≠10 e ímpar	5
6	i≠10 e par	
7	i≠10 e ímpar	7
8	i≠10 e par	
9	i≠10 e ímpar	9
10	i=10, SAI do ciclo	
		FIM DO CICLO

➤ As instruções **break** e **continue** só têm ação no ciclo a que pertencem, mesmo que este se encontre dentro de outro(s) ciclo(s)

➤ O uso de **break** e **continue** em ciclos, nesta UC, é altamente desaconselhado e carece de justificação!

OBSERVAÇÃO: em termos de boas práticas de programação e de eficiência, a técnica usada na resolução do problema anterior não é aconselhável...

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 1; i < 10; i = i+1)
        if( i % 2 )
            printf("%3d\n", i);
    printf("\n FIM DO CICLO\n");
}
```

É preferível esta

ou esta

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 1; i < 10; i = i+2)
        printf("%3d\n", i);
    printf("\n FIM DO CICLO\n");
}
```

Ciclos infinitos

Denominam-se ciclos infinitos os ciclos que nunca terminam: apresentam condições que são sempre verdadeiras.

```
while (1)
    instrução;
```

```
for ( ; ; )
    instrução;
```

```
do
    instrução
while (1);
```

- Quando no ciclo **for** não é colocada qualquer condição, esta é substituída pela condição VERDADE
- Este tipo de ciclos utiliza-se normalmente quando não se sabe à partida qual o número de vezes que se vai iterar o ciclo
- Para terminar um ciclo infinito pode utilizar-se a instrução **break**
- Um ciclo infinito é resultado de erros de programação ou de má programação!!!

Algumas notas sobre COMENTÁRIOS

Os comentários são frases não interpretadas pelo compilador, cujo papel é facilitar a compreensão do código (declaração de variáveis, expressões, instruções, etc.). Podem abranger uma linha

// Exemplo de comentário

ou diversas linhas

/* Tudo o que se encontra entre o símbolo de início e
o símbolo de fim de comentário */

Exemplos de comentários

```
#include <stdio.h>
int main()
{
    // Descubra o que faz este pequeno programa
    int i;
    for (i = 1; i <= 50; i = i+1)
        if(i % 2) /* % operador RESTO DA DIVISÃO INTEIRA */
            printf("%3d\n", i);
    printf("\nPRESSIONE QUALQUER TECLA PARA CONTINUAR ");
    getchar();
}
```

2.6 Funções

- Conceito e utilidade das funções
- Declaração e chamada
- Características de uma função
- Funcionamento de uma função
- Passagem de parâmetros de tipos básicos
- Colocação das funções no programa. Protótipos de funções
- Variáveis locais

Funções: conceito e utilidade

É frequente em linguagens de programação designarem-se **subprogramas** os conjuntos de instruções que desempenham uma tarefa específica, aos quais é atribuído um determinado nome e cuja execução acontece quando se referenciam em qualquer ponto do programa.

Distinguem-se usualmente dois tipos de *subprogramas ou rotinas*: os **procedimentos** e as **funções**. Uma **função** tem sempre um valor de retorno e um tipo associado, enquanto que um **procedimento** não devolve qualquer valor.

```
int minimo(int x, int y)
//devolve o menor de 2 números inteiros dados
{
    if(x < y)
        return x;
    else
        return y;
}
```

```
void linha(int nt)
//escreve uma linha de "comprimento" nt
{
    int i;
    for (i = 1; i <= nt; i++)
        printf("_");
    printf("\n");
}
```

Em **C** designamos por **funções** os dois tipos de subprogramas, embora usualmente se chamem procedimentos às funções que devolvem o valor **void**

Declaração e chamada

Declaração

SINTAXE

```
[tipo] identificador ([lista de argumentos formais, separados por vírgula (,)])  
{  
    bloco;  
}
```

Chamada

SINTAXE

```
identificador ([lista de argumentos reais, separados por vírgula (,)]);
```

Exemplo

```
#include <stdio.h>  
int minimo(int x, int y)  
{  
    if(x < y)  
        return x;  
    else  
        return y;  
}  
int main()  
{  
    printf("\nO menor dos números 80 e 43 é: %d\n", minimo(80,43));  
}
```

Declaração

Um programa em C tem de conter sempre no seu código a função **main()**, independentemente do número e variedade de funções existentes no programa.

Chamada

Algumas notas sobre declaração e chamada

Uma função tem sempre um tipo e um valor de retorno associado ao identificador, enquanto que um procedimento não devolve qualquer valor.

➤ Sempre que no cabeçalho de uma função não é colocado o tipo de retorno, este é considerado do tipo *int*.

```
int minimo(int x, int y)
{
    ...
}
```



```
minimo(int x, int y)
{
    ...
}
```

O tipo deve ser sempre explícito.

Uma função pode ser invocada de diferentes formas:

➤ *Dentro de uma atribuição:*

```
m = minimo(9,-2);
```

➤ *Dentro de uma função, aproveitando o valor de retorno como parâmetro:*

```
printf("Menor valor: %d", minimo(8,18) );
```

ou

```
if (minimo(7,6) > 10)
```

```
...
```

➤ *Tal como se invoca um procedimento, isto é, sem valor de retorno:*

```
getchar(); // aguardar que seja digitado um caracter
```

Características de uma função

1. Uma função tem um IDENTIFICADOR (nome) que serve para a sua invocação;
2. Uma função pode ser INVOCADA a partir de outras funções;
3. Como a sua designação indica, uma função deve realizar UMA ÚNICA TAREFA, bem definida;
4. Uma função deve comportar-se como uma CAIXA NEGRA: não importa como funciona, mas sim que dê o resultado esperado, sem efeitos colaterais;
5. O CÓDIGO de uma função deve ser tão GENÉRICO quanto possível, para poder ser reutilizado noutros projetos;
6. De forma a adaptar-se a situações distintas, uma função PODE RECEBER PARÂMETROS que alterem o seu comportamento;
7. Como resultado do seu trabalho, uma função PODE DEVOLVER UM VALOR para a entidade que a invocou.

Identificador de uma função

1. A escolha do identificador de uma função obedece às regras anteriormente apresentadas para designação de variáveis;
2. O nome de uma função deve ser único: distinto do de qualquer variável ou de outra função;
3. Deve procurar-se que o identificador de uma função facilite a leitura e interpretação daquilo que ela realiza (Exemplo: **linha** versus **xpto**).

Cabeçalho e corpo de uma função

Cabeçalho
da função

O cabeçalho de uma
função NUNCA é seguido
de ponto e vírgula (;)

```
int minimo(int x, int y)
//devolve o menor de 2 números inteiros dados
{
    if(x < y)
        return x;
    else
        return y;
}
```

Corpo da
função

Observação: dentro do corpo de uma função pode ser escrita qualquer instrução ou conjunto de instruções. Contudo, em C, **não se podem definir funções dentro de funções.**

A instrução RETURN

Esta instrução permite terminar a execução de uma função e voltar ao programa que a invocou.

A seguir à instrução RETURN é possível colocar qualquer expressão válida em C.

```
int resto(int x, int y)
{
    return x % y;
}
```

Uma função pode ter várias instruções RETURN. Contudo, só uma delas é executada.

```
int resposta(int x)
{
    if (x > 0)
        return 1;
    else
        return 0;
}
```

Muita atenção à existência
de vários *return*!!!!

➤ Na função *main*, a instrução RETURN faz com que o programa termine

Funcionamento de uma função

1. O código de uma função só é executado quando esta é executada algures no programa a que de algum modo está ligada.
2. Quando uma função é invocada, a execução do programa é remetida para essa função. São executadas as instruções presentes no corpo da função. Depois disso, o controlo de execução volta ao ponto do programa em que a função foi invocada.
3. O programa que invoca uma função pode enviar **ARGUMENTOS**, que são recebidos pela função e armazenados em variáveis locais, automaticamente inicializadas com os valores recebidos. A estas variáveis dá-se o nome de **PARÂMETROS**
4. Uma vez concluído o seu funcionamento, uma função pode devolver um valor para o programa que a invocou.

É usual chamar parâmetros tanto aos argumentos com que se invoca uma função como aos parâmetros, designando-se os primeiros como **parâmetros reais** e os segundos como **parâmetros formais**

O tipo *void*

A utilização da palavra reservada *void* permite indicar que uma função não devolve qualquer tipo.

Assim, a função “linha” atrás apresentada pode ser escrita do seguinte modo:

```
void linha(int nt)
//escreve uma linha de "comprimento" nt
{
    int i;
    for (i=1; i<=nt; i++)
        printf("_");
    printf("\n");
}
```

Uma função que “devolve *void*” chama-se normalmente um PROCEDIMENTO.

Para terminar a execução de uma função (procedimento) que retorna *void* utiliza-se a instrução RETURN sem qualquer expressão à frente.

Colocação das funções no programa. Protótipos de funções.

Em C as funções podem ser colocadas em qualquer local do programa, antes ou depois de serem invocadas, antes ou depois da função `main()`.

Protótipo de uma função

O protótipo de uma função corresponde ao seu cabeçalho seguido de um ponto e vírgula (;)

É um bom hábito de programação colocar sempre os protótipos das funções no início dos programas, de forma a indicar ao compilador qual a estrutura das funções que vão ser utilizadas pelo programa. Desta forma o compilador pode verificar em cada chamada de uma função se ela foi ou não corretamente implementada.

Do ponto de vista da correção sintática, basta que o protótipo de uma função seja colocado antes da sua invocação.

O objetivo da escrita do protótipo de uma função é indicar ao compilador qual o tipo de **retorno** da função e não quais os seus parâmetros.

Variáveis locais

As variáveis declaradas dentro do próprio corpo de uma função apenas são *visíveis* (conhecidas) dentro dessa função. São por isso denominadas **VARIÁVEIS LOCAIS**.

A declaração de variáveis dentro de uma função deve ser realizada antes de qualquer instrução.

```
função (.....)
{
    declaração de variáveis
    instruções
}
```

```
#include <stdio.h>
void linha(int nt);
void main()
{
    int i=11;
    printf("\nBoa tarde!\n");
    linha(i);
}
void linha(int nt)
{
    int i;
    for (i=1; i<=nt; i++)
        printf("_");
    printf("\n");
}
```

Protótipo da função linha

Depois de terminada a execução de uma determinada função, todas as suas variáveis locais são destruídas.

Algumas Funções de Uso Frequente

int isdigit(char c)

```
int isdigit(char c) // Devolve Verdade quando c é um dígito e Falso c.c.
{
    return (c>='0' && c<='9');
}
```

int isalpha(char c)

```
int isalpha(char c) // Devolve Verdade quando c é uma letra e Falso c.c.
{
    return (c>='a' && c<='z' || c>='A' && c<='Z'); // note que && tem maior precedência que ||
}
```

int isalnum(char c)

```
int isalnum(char c) // função que devolve Verdade quando c é um caracter alfanumérico e Falso c.c.
{
    return isalpha(c) || isdigit(c);
}
```

char tolower(char c)

```
char tolower(char c) // função que devolve c transformado na minúscula correspondente
{
    if (c>='A' && c<='Z')
        return c+'a'-'A';
    else
        return c;
}
```

Obtém-se acesso a estas funções através da diretiva
`#include <ctype.h>`
// Funções sobre o tipo
char (*ctype -> char type*)

Exemplo 1: dadas as funções “toma_la” e “da_ca”, abaixo escritas, determinar o *output* das seguintes chamadas:

a) toma_la(1); b) toma_la(3); c) da_ca(2); d) da_ca(4); e) da_ca(5);

void da_ca(int x); //protótipo da função "da_ca"

```
void toma_la(int x)
{
    switch (x)
    {
        case 1: printf("\nEste exercício é uma brincadeira!");
        case 2: printf("\nBem interessante..."); return;
        case 3: printf("\nEstou no toma_la e x=%d", x);
                da_ca(x++);
                break;
        default: printf("\nEntrei pelo default");
                da_ca(x);
    }
}
```

```
void da_ca(int x)
{
    switch (x)
    {
        case 2: return; printf("\nOlá a todos!");
        case 3: printf("\nEu sou um CRAQUE em C"); break;
                printf("\nSei mesmo disto!!!"); break;
        case 4: printf("\nNão percebo patavina!");
                da_ca(2); return;
        default: printf("\nCá estamos de novo");
                toma_la(x--);
    }
}
```

Simulação

--> a) toma_la(1)
Este exercício é uma brincadeira!
Bem interessante ...
--> b) toma_la(3)
Estou no toma_la e x=3
da_ca(3) Eu sou um CRAQUE em C
--> c) da_ca(2)
--> d) da_ca(4)
Não percebo patavina!
--> e) da_ca(5)
Cá estamos de novo
toma_la(5) Entrei pelo default
da_ca(5) Cá estamos de novo
toma_la(5) Entrei pelo default
(...ciclo infinito, repetindo as 2 últimas linhas)
--> e se na última linha do da_ca fosse toma_la(--x)?

Exemplo 2: Verificar se uma dada letra é vogal

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
int isvogal(char c)
{
    switch (tolower(c))
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': return 1;
        default : return 0;
    }
}
```

Escrever um função equivalente, usando uma instrução de atribuição em vez da estrutura *switch*

```
int isvogal_atribuicao(char c)
{
    char cc = tolower(c);
    return (cc=='a' || cc=='e' || cc=='i' || cc=='o' || cc=='u');
}
```

```
void main()
{
    char letra;
    printf("\n\nDigite uma letra: ");
    scanf(" %c",&letra);
    printf("\nA letra lida, %c, ", letra);
    if (isvogal_atribuicao(letra))
        printf(" é uma vogal\n");
    else
        printf(" não é uma vogal\n");
    system ("pause");
    printf("\nConversao da letra %c em maiúscula: %c\n", letra, toupper(tolower(letra)));
    system ("pause");
}
```

Mais funções...

int resto (int a, int b) // Devolve o resto da divisão de a por b

```
int resto(int a, int b)
{
    return a % b;
}
```

int impar (int x) // Devolve Verdade se x for impar e Falso c.c.

```
int impar(int x)
{
    return resto(x,2) !=0;
}
```

int minimo (int a, int b) // Devolve Mínimo{a,b}

```
int minimo(int a, int b)
{
    return a<b ? a : b;
}
```

int abs (int a) // Devolve |a|

```
int abs(int a)
{
    return a>=0 ? a : -a;
}
```

int perfeito (int n) // Devolve Verdade se n for “perfeito” (igual à soma dos divisores de n, inferiores a n) e Falso c.c.

```
int perfeito(int n)
{
    int div, soma, meio;
    soma = 0;
    meio = n/2;
    for (div=1; div<=meio; div++)
        if (resto(n,div)==0) //se div é divisor de n
            soma += div;
    return n==soma;
}
```

Exercícios

1. Um número diz-se primo quando apenas é divisível por si próprio e pela unidade.
Escrever uma função que devolva Verdade se n é primo e Falso c.c.

```
int primo (int n)
{
    int div, meio, sai;
    div=1;
    meio=n/2;
    do
    {
        div++; // neste contexto, equivalente a div = div + 1;
        sai=(n%div)==0;
    } while (sai==0 && div<meio);
    return sai==0; // devolve um valor LÓGICO
}
```

2. No século I d.C. os números naturais dividiam-se em três categorias:

REDUZIDOS: os superiores à soma dos seus divisores

ABUNDANTES: os inferiores à soma dos seus divisores

PERFEITOS: os que são iguais à soma dos seus divisores

- a) Escrever uma função que liste os inteiros entre a e b, $a > b$, classificando-os de acordo com esse critério.
- b) Aperfeiçoe a função anterior para que escreva também o total de cada uma das categorias.

NOTA: na definição supra, exclui-se o próprio número do conjunto dos seus divisores.

Solução - alínea a)

```
#include <stdio.h>
int resto (int a, int b);
void classifica (int na, int nb)
{
    int n, div, soma, meio;

    for(n = na; n <= nb; n++)
    {
        soma = 0;
        meio = n/2;
        for (div=1; div<=meio; div++)
            if (resto(n,div)==0)
                soma += div;
                if (n==soma)
                    printf("\n\t%d\t PERFEITO",n);
                else if (n>soma)
                    printf("\n\t%d\t REDUZIDO",n);
                else printf("\n\t%d\t ABUNDANTE",n);
    }
}

void main()
{
    int limInf, limSup;
    do
    {
        printf("\nDigite dois numeros inteiros positivos entre 1 e 500: ");
        scanf("%d %d",&limInf, &limSup);
    } while (limInf < 1 || limInf > limSup || limSup > 500);
    classifica(limInf, limSup);
    printf("\n");
}
```

Aperfeiçoe o programa (2 instruções novas e 2 modificadas) para melhorar o layout:

<u>Numero</u>	<u>Tipo</u>	.