

Algoritmos e Estruturas de Dados I

Estruturas Condicionais e de Repetição (parte 2)

Pedro O.S. Vaz de Melo

Problema 1

- Suponha que soma (+) e subtração (-) são as únicas operações disponíveis em C. Dados dois números inteiros positivos A e B , determine o quociente e o resto da divisão de A por B .

Algoritmos Estruturados

- Para resolver o Problema 1, precisamos de um algoritmo:
- Sequência finita de instruções que, ao ser executada, chega a uma solução de um problema.

Algoritmos Estruturados

- Para resolver o Problema 1, precisamos de um algoritmo:
- Sequência finita de instruções que, ao ser executada, chega a uma solução de um problema.
- Para escrever este algoritmo, podemos usar a seguinte ideia:
 - Representar os números **A** e **B** por retângulos de larguras proporcionais aos seus valores;
 - Verificar quantas vezes **B** cabe em **A**.

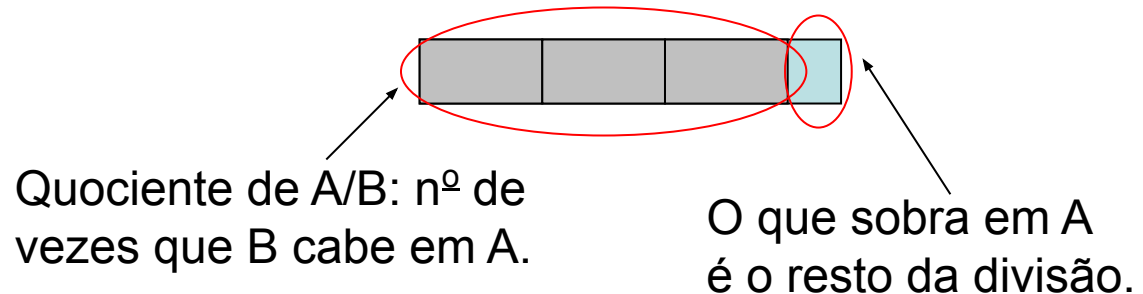
Algoritmos Estruturados

- Suponha que soma (+) e subtração (-) são as únicas operações disponíveis em C. Dados dois números inteiros positivos A e B , determine o **quociente** e o **resto** da divisão de A por B .

A 

B 

$A = 7, B = 2$



Algoritmos Estruturados

- Pode-se escrever este algoritmo como:

```
Sejam A e B os valores dados;  
Atribuir o valor 0 ao quociente (q);  
Enquanto B couber em A:  
{  
    Somar 1 ao valor de q;  
    Subtrair B do valor de A;  
}  
Atribuir o valor final de A ao resto (r).
```

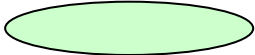
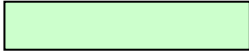
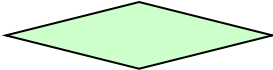



Algoritmos Estruturados

- Pode-se escrever este algoritmo como:

```
Sejam A e B os valores dados;  
Atribuir o valor 0 ao quociente (q);  
Enquanto B <= A:  
{  
    Somar 1 ao valor de q;  
    Subtrair B do valor de A;  
}  
Atribuir o valor final de A ao resto (r).
```

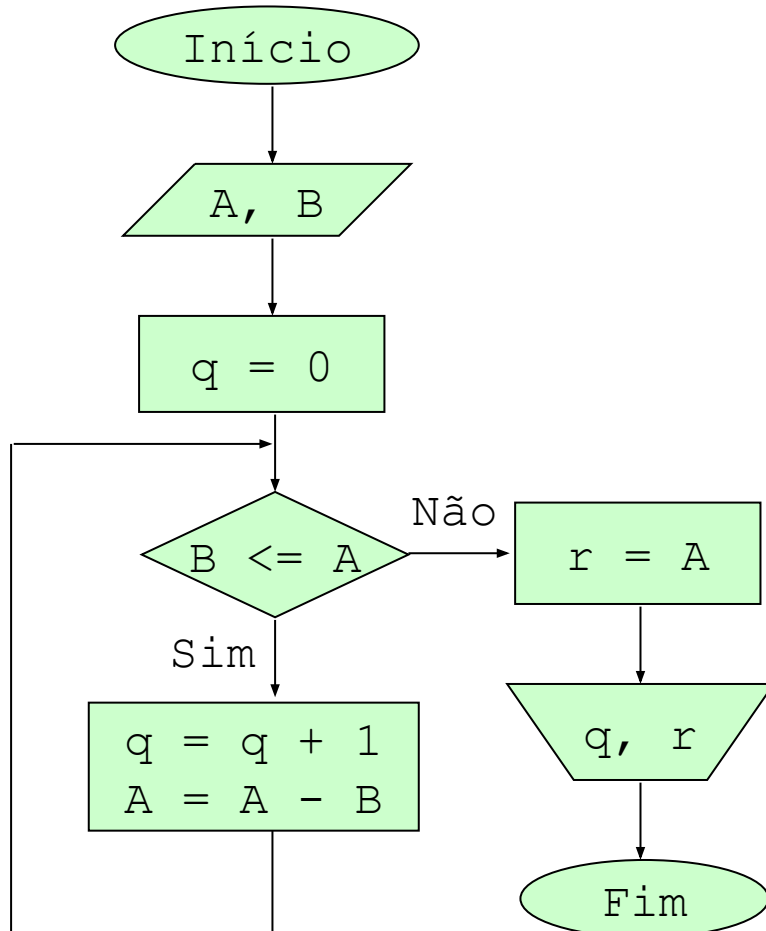
Fluxograma

- É conveniente representar algoritmos por meio de **fluxogramas** (diagrama de blocos).
- Em um fluxograma, as operações possíveis são representadas por meio de figuras:

Figura	Usada para representar
	Início ou fim.
	Atribuição.
	Condição.
	Leitura de dados.
	Apresentação de resultados.
	Fluxo de execução.

Fluxograma

- **Exemplo:** o algoritmo para o Problema 1 pode ser representado pelo seguinte fluxograma.



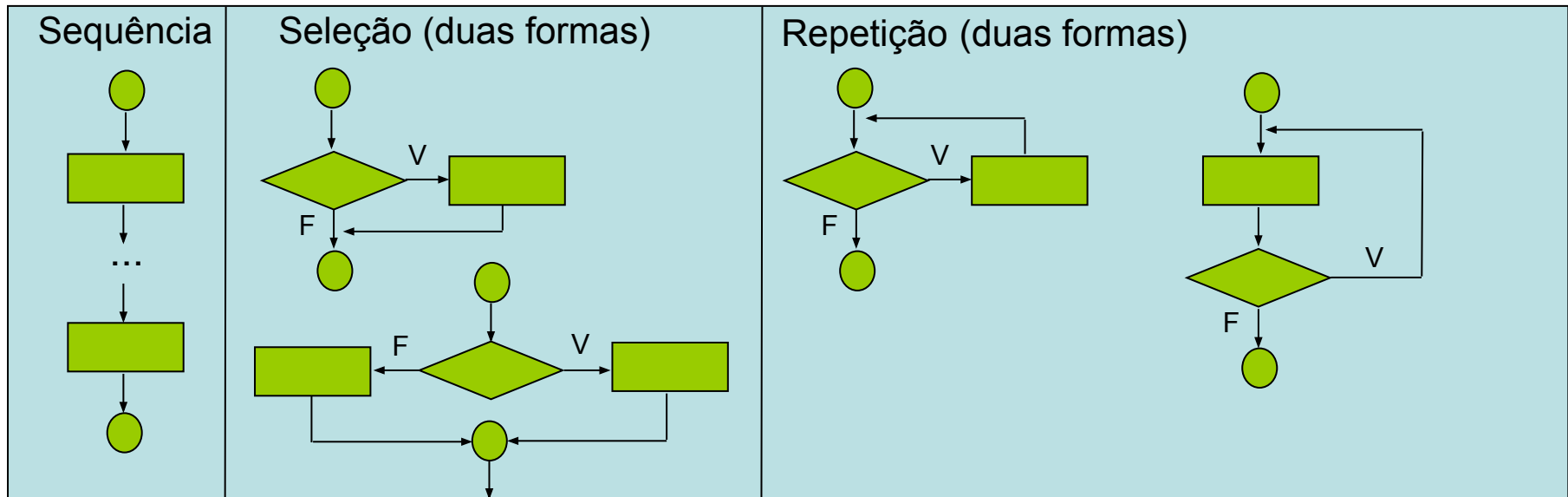
Atenção: observe que um algoritmo não inclui detalhes, tais como declaração de variáveis, mensagens a serem exibidas, etc.

Programação Estruturada

- Um algoritmo tem sempre um **único bloco início** e deve conter, pelo menos, um **bloco fim**. A execução segue **setas**
- Em geral, construir um algoritmo é mais difícil do que codificar em uma linguagem
- Porém, a construção de algoritmos pode se beneficiar de técnicas, como a **Programação Estruturada**

Programação Estruturada

- Na **Programação Estruturada**, usa-se apenas três estruturas: **sequência**, **seleção** e **repetição**.
 - Cada estrutura tem apenas um único ponto de entrada e um único ponto de saída (círculos).



Programação Estruturada

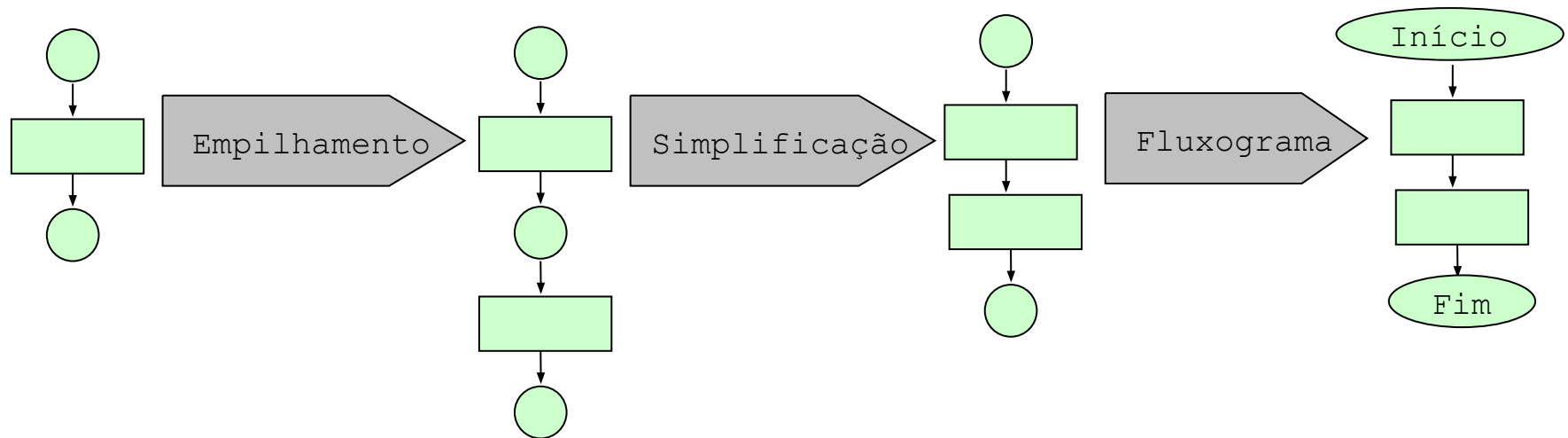
- Atenção!
- Os retângulos indicam qualquer ação, incluindo leitura de dados ou exibição de resultados
- Construir programas estruturados corresponde a combinar estas estruturas de duas maneiras:
 - **Regra do empilhamento:** o ponto de saída de uma estrutura pode ser conectado ao ponto de entrada de outra estrutura.
 - **Regra do aninhamento:** um retângulo de uma estrutura pode ser substituído por uma outra estrutura qualquer.

Programação Estruturada

- Estas regras podem ser aplicadas quantas vezes forem necessárias e em qualquer ordem.
- Na construção de fluxogramas, pode-se substituir o primeiro ponto de entrada e os últimos pontos de saída por ovais (início e fim).
- Os demais pontos de entrada e saída podem ser removidos.

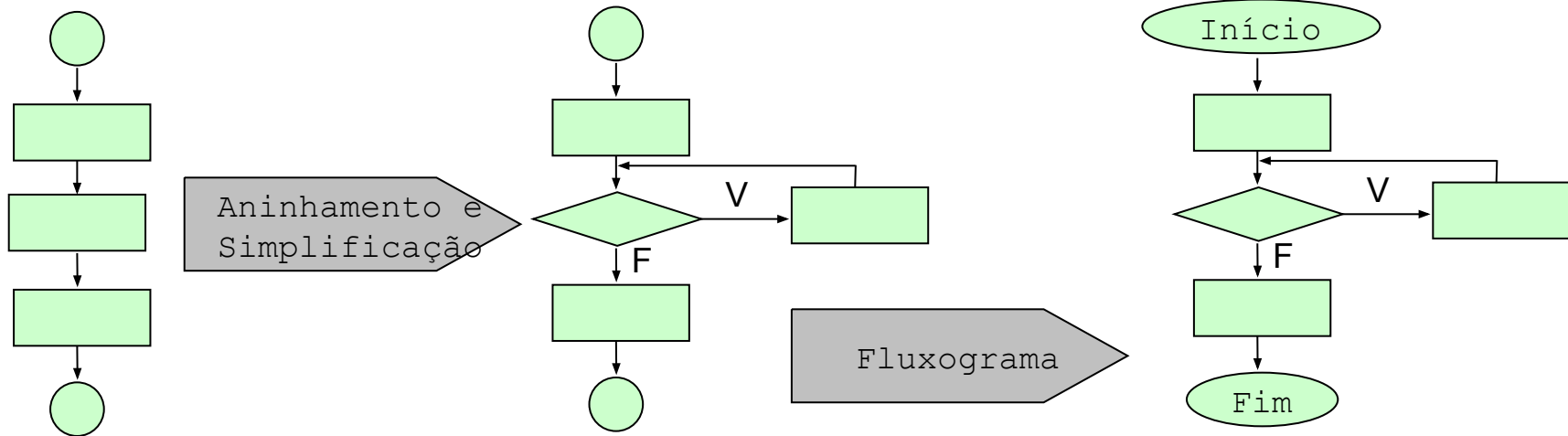
Programação Estruturada

- **Exemplo:** aplicação da regra de empilhamento.



Programação Estruturada

- Suponha que soma (+) e subtração (-) são as únicas operações disponíveis em C. Dados dois números inteiros positivos **A** e **B**, determine o quociente e o resto da divisão de **A** por **B**.



Problema 2

- Dados dois números inteiros A e B , determinar o máximo divisor comum (MDC) destes dois números.

Problema 2

- Como calcular o MDC entre dois números A e B , representado por $\text{mdc}(A,B)$?
- **Método das divisões sucessivas**: efetua-se várias divisões até chegar em uma divisão exata.

Problema 2

- Suponha que se deseja calcular $\text{mdc}(48,30)$.
 - Divide-se o número maior pelo menor:
 $48/30 = 1$ (resto 18).
 - Divide-se o divisor anterior pelo resto anterior e, assim sucessivamente:
 $30/18 = 1$ (resto 12)
 $18/12 = 1$ (resto 6)
 $12/6 = 2$ (resto 0 – divisão exata)
MDC = 6

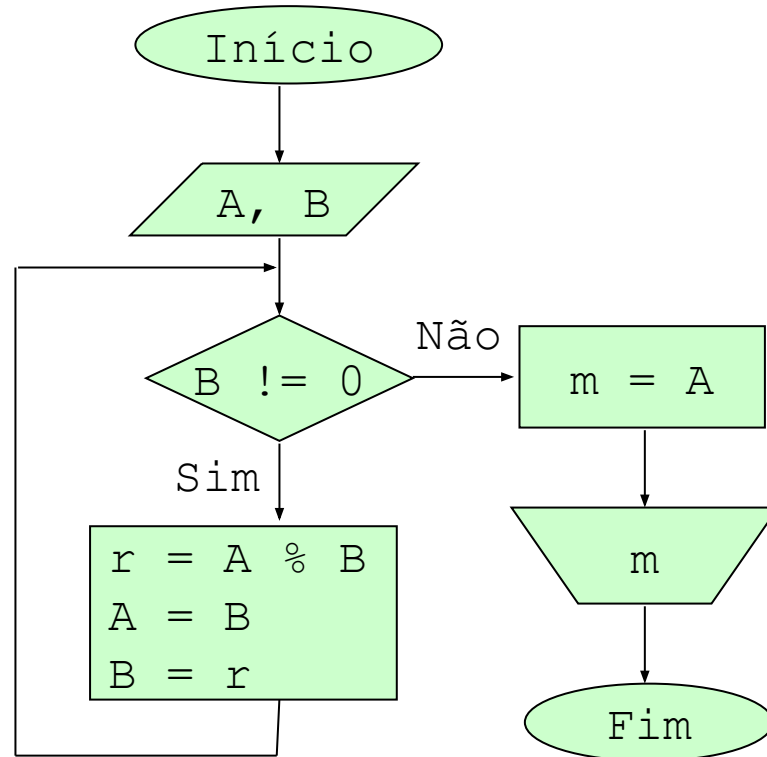
Solução - Algoritmo

- Um algoritmo para este problema pode ser escrito como:

```
Enquanto B for diferente de zero:  
{  
    r = resto da divisão de A por B;  
    A = B;  
    B = r;  
}  
mdc = A;
```

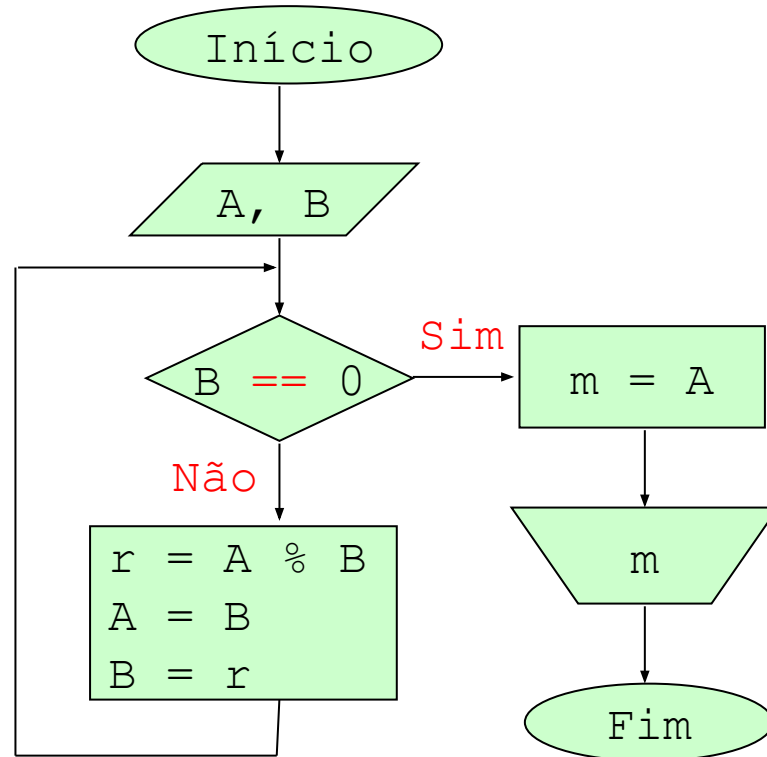
Solução - Fluxograma

- O algoritmo para calcular o máximo divisor comum pode ser representado pelo seguinte fluxograma:



Solução - Fluxograma

- O algoritmo para calcular o máximo divisor comum pode ser representado pelo seguinte fluxograma:



Solução - Código

```
int MDC(int A, int B)
{
    int r;

    while (B != 0)
    {
        r = A % B;
        A = B;
        B = r;
    }
    return A;
}
```

Problema 3

- Escrever um **programa** para ler dois inteiros do teclado, calcular o MDC entre eles e, caso o usuário deseje, repetir o processo.

Solução

```
int main(int args, char * arg[])
{
    int A, B, m;
    char c;

    do
    {
        printf("Digite os valores de A e B: ");
        scanf("%d %d", &A, &B);
        m = MDC(A,B);
        printf("MDC(%d,%d) = %d\n", A,B,m);
        printf("Continua? (S/N) ");
        do
        {
            c = getche();
            if ((c != 'S') && (c != 'N'))
            {
                putchar('\a');
                putchar('\b');
            }
        }
        while ((c != 'S') && (c != 'N'));
        printf("\n");
    }
    while ((c == 'S') || (c == 's'));
    system("PAUSE");
    return 0;
}
```


Solução

```
int main(int args, char * arg[])
{
    int A, B, m;
    char c;

    do
    {
        printf("Digite os valores de A e B: ");
        scanf("%d %d", &A, &B);
        m = MDC(A,B);
        printf("MDC(%d,%d) = %d\n", A,B,m);
        printf("Continua? (S/N) ");

        do
        {
            c = getche();
            if ((c != 'S') && (c != 'N'))
            {
                putchar('\a');
                putchar('\b');
            }
        }
        while ((c != 'S') && (c != 'N'));
        printf("\n");

        while ((c == 'S') || (c == 's'));
        system("PAUSE");
        return 0;
    }
```

Após exibir o valor do `mdc`, o programa exibe a mensagem:

Continua? (S/N)

Espera-se que o usuário digite `S` ou `N`, caracteres que serão lidos pela função `getche`.

A função `getche` retorna o código ASCII do caractere lido.

O loop será executado enquanto `c` for igual a `'S'` ou a `'s'`.

Solução

- Para evitar a comparação com letras maiúsculas e minúsculas, pode-se usar a função **toupper**:

```
do
{
    printf("Digite os valores de A e B: ");
    scanf("%d %d", &A, &B);
    m = MDC(A,B);
    printf("MDC(%d,%d) = %d\n", A,B,m);
    printf("Continua? (S/N) ");
    do
    {
        c = toupper(getche());
        if ((c != 'S') && (c != 'N'))
        {
            putchar('\a');
            putchar('\b');
        }
    }
    while ((c != 'S') && (c != 'N'));
    printf("\n");
}
while ((c == 'S') || (c == 's'));
```

Verifica se o valor de seu parâmetro corresponde ao código ASCII de uma letra minúscula.

Caso afirmativo, retorna o código da letra maiúscula correspondente.

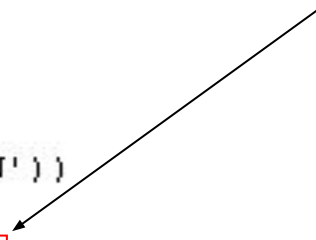
Caso negativo, retorna o próprio valor do parâmetro.

Solução

- Um sinal sonoro ('\a') pode ser usado para avisar o usuário que a tecla pressionada não é uma resposta válida.

```
do
{
    printf("Digite os valores de A e B: ");
    scanf("%d %d", &A, &B);
    m = MDC(A,B);
    printf("MDC(%d,%d) = %d\n", A,B,m);
    printf("Continua? (S/N) ");
    do
    {
        c = toupper(getche());
        if ((c != 'S') && (c != 'N'))
        {
            putchar('\a');
            putchar('\b');
        }
    }
    while ((c != 'S') && (c != 'N'));
    printf("\n");
}
while ((c == 'S') || (c == 's'));
```

Sinal sonoro



Problema 4

Escreva um programa que permita ao usuário escolher dentre as seguintes opções:

- Exibir o conteúdo da pasta;
- Modificar a hora do sistema;
- Modificar a data do sistema;
- Terminar a execução do programa.

```
int main(int args, char * arg[])
{
    char optei;
    int erro;

    do
    {
        system("CLS");
        printf("A. Exibir o conteudo da pasta\n");
        printf("B. Modificar a hora do sistema\n");
        printf("C. Modificar a data do sistema\n");
        printf("X. Terminar execucao\n");
        printf("Escolha: ");
```

Problema 4

```
do
{
    erro = 0;
    optei = toupper(getche());
    if (optei == 'A')
        system("DIR");
    else if (optei == 'B')
        system("TIME");
    else if (optei == 'C')
        system("DATE");
    else if (optei == 'X')
        ;
    else
    {
        putch('\a');
        putch('\b');
        erro = 1;
    }
}
while (erro == 1);
printf("\n");
system("PAUSE");
}
while (optei != 'X');
return 0;
}
```

Função `system`

O programa desenvolvido para o Problema 4 mostra diversas possibilidades de uso da função `system`:

Parâmetro	Usado para
CLS	Limpar a tela de execução.
DIR	Exibir o conteúdo da pasta em uso.
TIME	Exibir e permitir modificar a hora atual do sistema.
DATE	Exibir e permitir modificar a data atual do sistema.
PAUSE	Interromper a execução do programa.

Função `system`

- Os parâmetros possíveis para a função `system` dependem do `sistema operacional` sob o qual os programas serão executados.
- Assim, os parâmetros “`CLS`”, “`DIR`”, “`TIME`”, “`DATE`”, “`PAUSE`”, correspondem a comandos do sistema DOS.

Comandos **if-else** interrelacionados

Exemplo: imagine uma função que recebe como parâmetro um inteiro representando o número de um mês e retorna o número de dias deste mês (considere que fevereiro tem sempre 28 dias).

Comandos **if-else** interrelacionados

```
int dias_do_mes(int mes)
{
    int dias;
    if (mes == 1)
        dias = 31;
    else if (mes == 2)
        dias = 28;
    else if (mes == 3)
        dias = 31;
    else if (mes == 4)
        dias = 30;
    else if (mes == 5)
        dias = 31;
    else if (mes == 6)
        dias = 30;
    else if (mes == 7)
        dias = 31;
    else if (mes == 8)
        dias = 31;
    else if (mes == 9)
        dias = 30;
    else if (mes == 10)
        dias = 31;
    else if (mes == 11)
        dias = 30;
    else if (mes == 12)
        dias = 31;
    else
        dias = 0;
    return dias;
}
```

Comandos **if-else** inter-relacionados

Uma outra forma de escrever esta função, mas ainda com comandos **if-else inter-relacionados** é:

```
int dias_do_mes(int mes)
{
    int dias;
    if ((mes == 1) || (mes == 3) || (mes == 5) ||
        (mes == 7) || (mes == 8) || (mes == 10) ||
        (mes == 12))
        dias = 31;
    else if (mes == 2)
        dias = 28;
    else if ((mes == 4) || (mes == 6) || (mes == 9) ||
        (mes == 11))
        dias = 30;
    else
        dias = 0;
    return dias;
}
```

Comando **switch**

- A demanda por comandos **if-else inter-relacionados** é muito comum em programação.
- Assim, a linguagem C disponibiliza um comando especial para tais situações: **switch**. A sintaxe deste comando é a seguinte:

```
switch (expressão)
{
    case constante-1:
        comandos-1;
    case constante-2:
        comandos-2;
    ...
    default:
        comandos-n;
}
```

Comando `switch`

Com o comando `switch`, a função `dias_do_mes` pode ser reescrita como:

```
int dias_do_mes(int mes)
{
    int dias;
    switch (mes)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            dias = 31;
            break;
        case 2:
            dias = 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            dias = 30;
            break;
        default:
            dias = 0;
    }
    return dias;
}
```

Comando `switch`

Este comando permite que, de acordo com o valor de uma `expressão`, seja executado um ou mais comandos dentre uma série de alternativas.

O `caso` cuja constante for igual ao valor da expressão será selecionado para execução.

Atenção!

- Os comandos associados a este caso e `todos os comandos seguintes` serão executados em sequência até o final do comando `switch`.
- Para evitar a execução de todos os comandos seguintes, usa-se o comando `break`.

Comando `switch`

Como assim?

Para mes = 2:

Com o uso de `break`:

```
dias = 28;
```

Para mes = 2:

Sem o uso de `break`:

```
dias = 28;
```

```
dias = 30;
```

```
dias = 0;
```

```
int dias_do_mes(int mes)
{
    int dias;
    switch (mes)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            dias = 31;
            break;
        case 2:
            dias = 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            dias = 30;
            break;
        default:
            dias = 0;
    }
    return dias;
}
```

.Comandos **break** e **continue**

- Em alguns programas, durante um processamento iterativo, pode ser necessário:

- Encerrar o processamento iterativo independentemente do valor da condição do laço;
- Executar apenas parcialmente uma iteração, ou seja, executar somente algumas das instruções do laço da repetição.

- Para encerrar um processamento iterativo, independentemente do valor da condição do laço, deve-se usar o comando **break**.

Comandos **break** e **continue**

- Exemplo: dados os valores **N** (**int**) e **A** (**float**), determine a partir de qual termo o valor de:

$$s = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}$$

é maior do que **A**.

Suponha **N** = 10 e **A** = 2.

Instante	Valor de s
1 ^o termo	1.000000
2 ^o termo	1.500000
3 ^o termo	1.833333
4 ^o termo	2.083333

← A partir do quarto termo $s > A$.

Comandos `break` e `continue`

- Um programa para resolver este problema pode ser escrito como:

```
i = 1;
s = 0;
while (i <= N)
{
    s = s + 1.0/i;
    if (s > A)
    {
        printf("Numero de termos = %d\n", i);
        break;
    }
    i++;
}
```

- Neste caso, a `condição do laço` controla apenas o número de termos do somatório.
- O laço pode ser encerrado quando $s > A$, usando-se o comando `break`.

Comandos `break` e `continue`

- Para executar somente algumas das instruções do laço, **mas sem encerrar** a repetição: comando `continue`.
- Exemplo: ler a idade e o peso de **N** pessoas e determinar a soma dos pesos das pessoas com mais de 30 anos.

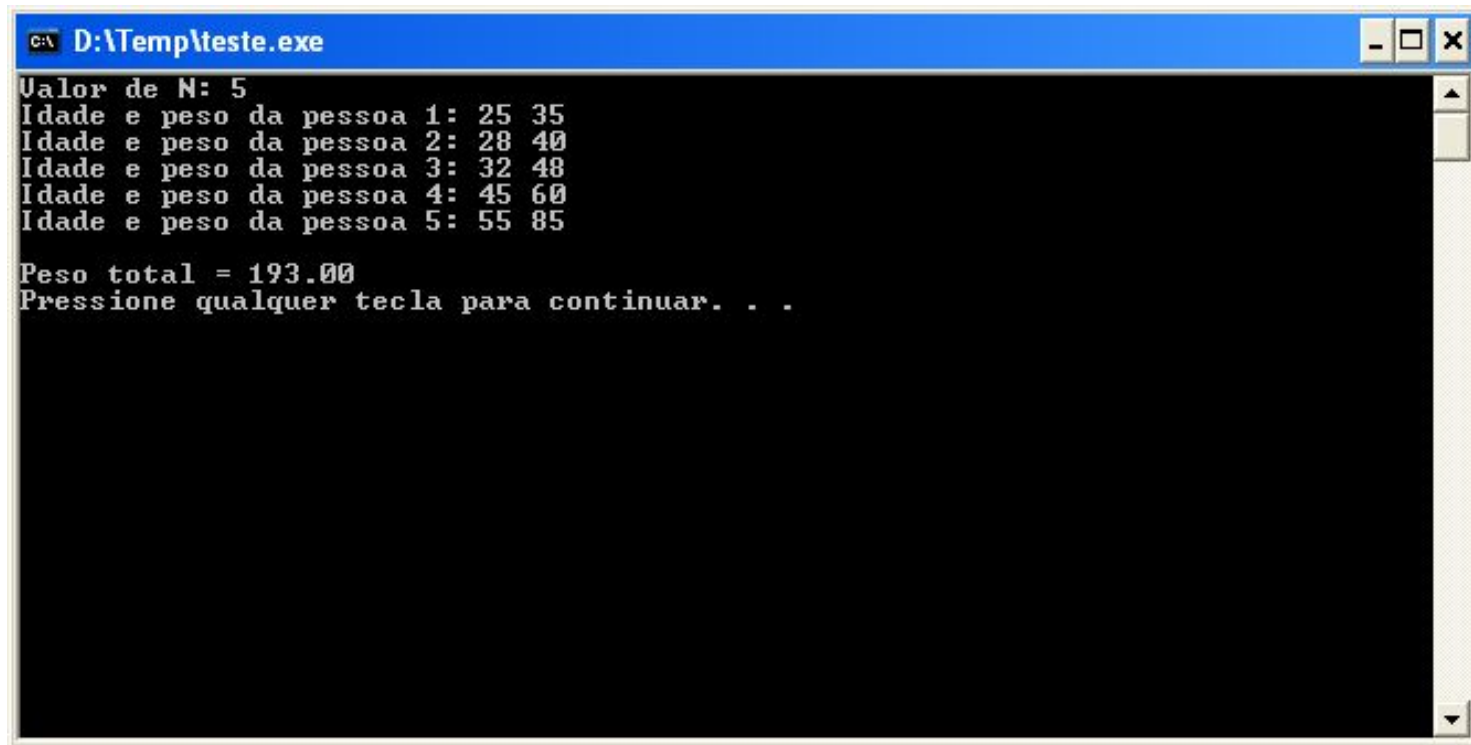
```
printf("Valor de N: ");
scanf("%d", &N);
i = 0;
s = 0;
while (i < N)
{
    i++;
    printf("Idade e peso da pessoa %d: ", i);
    scanf("%d %f", &idade, &peso);
    if (idade <= 30)
        continue;
    s = s + peso;
}
printf("Peso total = %.2f\n", s);
```

O comando `continue` faz com que a instrução `s = s + peso` não seja executada quando `idade <= 30`.

Ou seja, ele volta a execução para o início do laço.

Comandos `break` e `continue`

Resultado da execução:



```
C:\> D:\Temp\teste.exe
Valor de N: 5
Idade e peso da pessoa 1: 25 35
Idade e peso da pessoa 2: 28 40
Idade e peso da pessoa 3: 32 48
Idade e peso da pessoa 4: 45 60
Idade e peso da pessoa 5: 55 85

Peso total = 193.00
Pressione qualquer tecla para continuar. . .
```

Peso total = ~~35~~ + ~~40~~ + 48 + 60 + 85 = 193

Comando **for**

No programa **p14.c** é preciso ler cada um dos caracteres que compõe a mensagem a ser cifrada.

Para se executar esta ação, o programa pode utilizar qualquer um dos seguintes trechos de código:

```
i = 0;
while (i < 100)
{
    scanf("%c", &texto[i]);
    if (texto[i] == '.')
        break;
    i++;
}
```

```
for (i = 0; i < 100; i++)
{
    scanf("%c", &texto[i]);
    if (texto[i] == '.')
        break;
}
```

Ou seja, os comandos **for** e **while** são equivalentes.

Problema 5

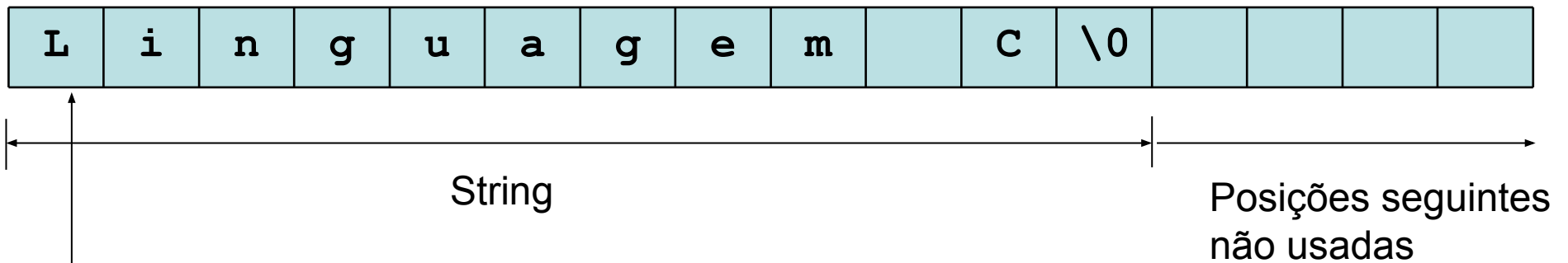
- Uma empresa quer transmitir mensagens sigilosas a seus diretores e precisa de um programa para codificar suas mensagens.
- A **regra de criptografia** deve substituir o código ASCII de cada caractere que compõe a mensagem por:
 $(5 * \text{código_ASCII} + 100) \% 256$.
- As mensagens deverão terminar com '.' (ponto).

Representação de strings

- **Strings** são conjuntos de caracteres e cada caractere é codificado como um inteiro de 8 bits (**código ASCII**).
- Se imaginarmos que o bit de sinal não é utilizado, cada caractere que compõe uma string pode ser representado por um inteiro no intervalo **[0, 255]**.
- Os inteiros de **[0, 127]** representam os caracteres do **código ASCII padrão**. Os inteiros de **[128, 255]** representam os caracteres do código ASCII **estendido**.

Representação de strings

- Uma string é armazenada em bytes consecutivos de memória.
- Para identificar o final de uma string, a linguagem C utiliza um caractere especial: `'\0'` (cód. ASCII zero).
- Exemplo: seja a string “Linguagem C”. Imagine a representação desta string na memória como:



Cada quadrado representa uma posição de memória de 8 bits.

Representação de strings

- Imagine a declaração da variável `texto` como:

```
char texto[100];
```

- Se `n` é uma constante inteira, então os símbolos `[n]` após o nome da variável indicam que ela poderá ocupar até `n` posições de memória consecutivas.
- Logo, a variável `texto` poderá ocupar até 100 posições do tipo `char` (ou seja, 100 bytes).
- Como as posições de memória são consecutivas, cada uma delas pode ser identificada por um índice.

Representação de strings

- Na linguagem C, os valores dos índices começam sempre em **zero**.
- Exemplo: considere a seguinte declaração:
- A representação de **S** pode ser imaginada como:

```
char S[20] = "Linguagem C";
```

Observe que **S** pode ocupar até 20 posições (numeradas de **0** a **19**).

L	i	n	g	u	a	g	e	m		C	\0			
S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]	S[10]	S[11]	...		S[19]

Representação de strings

- A **memória alocada** para uma variável é dada por:

(número de posições de memória) * (tamanho do tipo, em bytes)

Exemplo:

```
int a;  
short int b[15];  
float c[20];  
char d[100] = "dcc-aeds1";
```

Variável	Tipo	Nº de posições	Memória alocada
a	int (4 bytes)	1	4 bytes
b	short int (2 bytes)	15	30 bytes
c	float (4 bytes)	20	80 bytes
d	char (1 byte)	100	100 bytes

Representação de strings

- Atenção!

- Uma variável **pode ocupar menos memória** do que o total de posições alocadas.
- Exemplo:

```
char d[100] = "dcc-aeds1";
```
- Dos 100 bytes alocados, a variável está ocupando apenas 10 (lembre-se do caractere '\0').

- Uma variável **jamais poderá ocupar mais memória** do que o total de memória alocada! Exemplo:

`v[50] = 7.9;` representa **invasão de memória!!**

```
double v[50];
```

→ Memória alocada: 50*8 bytes = 400 bytes.
Posições variam de 0 a 49

Análise do programa

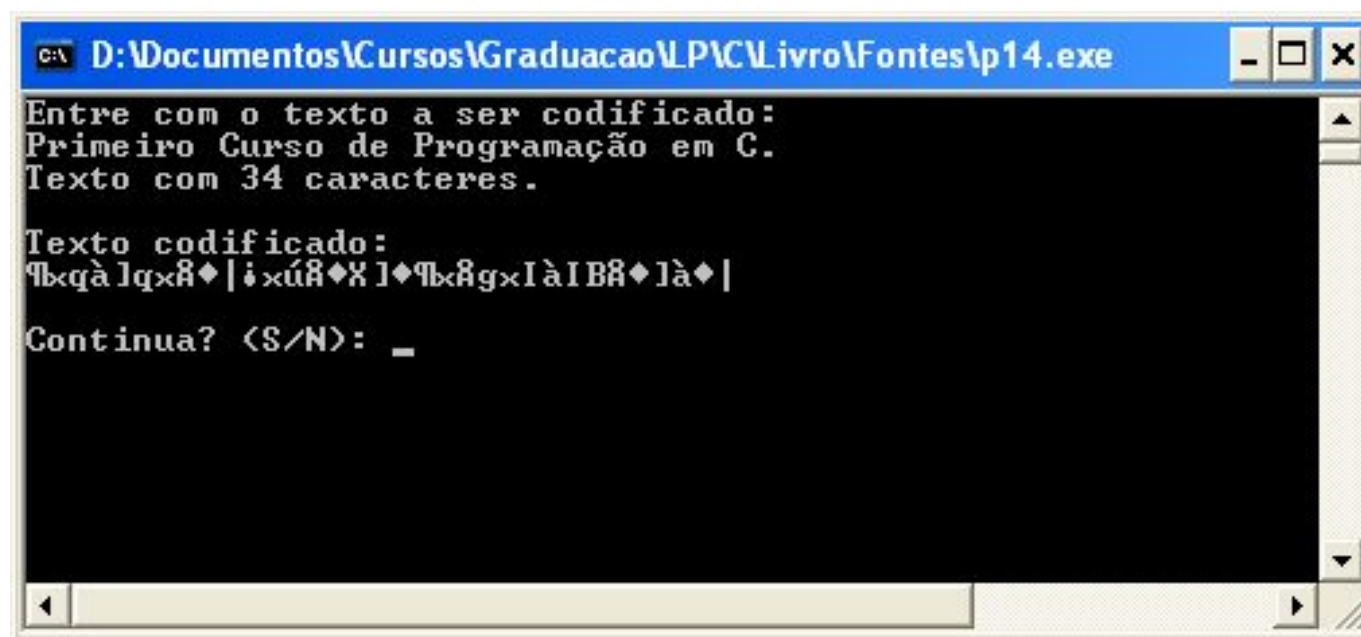
```
// Programa p14.c
#include <stdio.h>
#include <stdlib.h>

int main(int args, char * arg[])
{
    int i,n;
    char optei;
    char texto[100];

    do
    {
        system("CLS");
        printf("Entre com o texto a ser codificado:\n");
        for (i = 0; i < 100; i++)
        {
            scanf("%c",&texto[i]);
            if (texto[i] == '.')
                break;
        }
        n = i;
        printf("Texto com %d caracteres.\n\n", n);
        for (i = 0; i < n; i++)
        {
            texto[i] = (5*texto[i] + 100) % 256;
        }
    }
```

Análise do programa

```
    printf("Texto codificado:\n");  
    for (i = 0; i < n; i++)  
        printf("%c", texto[i]);  
    printf("\n\n");  
    printf("Continua? (S/N): ");  
    optei = toupper(getche());  
}  
while (optei == 'S');  
printf("\n");  
system("PAUSE");  
return 0;  
}
```



```
C:\ D:\Documentos\Cursos\Graduacao\LP\CLivro\Fontes\p14.exe  
Entre com o texto a ser codificado:  
Primeiro Curso de Programação em C.  
Texto com 34 caracteres.  
  
Texto codificado:  
Ŧb<qàlq>8♦|îxú8♦Xl♦Ŧb8g>IàIB8♦lâ♦|  
  
Continua? <S/N>: _
```

Endereços de Strings

```
#include <stdio.h>
#include <ctype.h>
void main ()
{
    char str[]="Test String.\n";
    int i=0;

    printf("\n%s", str);

    printf("\n%c-%c-%c-%c\n", str[0], *str, *str+1, *(str+1));

    for (i=0; str[i] != '\0'; i++)
        printf("\nEndereco de str[%d]: %p ou %p", i, &str[i], str+i);
}
```

Endereços de Strings

Test String.

T-T-U-e

Endereco de str[0]:	0028FEFE	ou	0028FEFE
Endereco de str[1]:	0028FEFF	ou	0028FEFF
Endereco de str[2]:	0028FF00	ou	0028FF00
Endereco de str[3]:	0028FF01	ou	0028FF01
Endereco de str[4]:	0028FF02	ou	0028FF02
Endereco de str[5]:	0028FF03	ou	0028FF03
Endereco de str[6]:	0028FF04	ou	0028FF04
Endereco de str[7]:	0028FF05	ou	0028FF05
Endereco de str[8]:	0028FF06	ou	0028FF06
Endereco de str[9]:	0028FF07	ou	0028FF07
Endereco de str[10]:	0028FF08	ou	0028FF08
Endereco de str[11]:	0028FF09	ou	0028FF09
Endereco de str[12]:	0028FF0A	ou	0028FF0A